Performance of *On-Off* Traffic Stemming From Live Adaptive Segmented HTTP Video Streaming

Tomas Kupka
University of Oslo
Simula Research Laboratory
tomasku@ifi.uio.no

Pål Halvorsen
University of Oslo
Simula Research Laboratory
paalh@ifi.uio.no

Carsten Griwodz
University of Oslo
Simula Research Laboratory
griff@ifi.uio.no

***Abstract*—A large number of live segmented adaptive HTTP video streaming services exist in the Internet today. These quasi-live solutions have been shown to scale to a large number of concurrent users, but the characteristic *on-off traffic pattern* makes TCP behave differently compared to the bulk transfers the protocol is designed for. In this paper, we analyze the TCP performance of such live on-off sources, and we investigate possible improvements in order to increase the resource utilization on the server side. We observe that the problem is the bandwidth wastage because of the synchronization of the *on* period. We investigate four different techniques to mitigate this problem. We first evaluate the techniques on pure *on-off* traffic using a fixed quality and then repeat the experiments with quality adaptation.**

***Index Terms*—live HTTP streaming, adaptation strategy, TCP**

I. Introduction

Both the amount of video data and the number of video streaming services in the Internet are rapidly increasing. The number of videos streamed by these services is on the order of tens of billions per month where YouTube alone delivers more than four billion video views globally every day [25]. Furthermore, many major (sports) events like European soccer leagues, NBA basketball and NFL football are streamed *live* with only a few seconds delay. For example, such services streamed events like the 2010 Winter Olympics [26], 2010 FIFA World Cup [13] and NFL Super Bowl [13] to millions of concurrent users over the Internet, supporting a wide range of devices ranging from mobile phones to HD displays.

As a video delivery technology, streaming over HTTP has become popular for various reasons, e.g., it runs on top of TCP, provides NAT friendliness and is allowed by most firewalls. Furthermore, it has also been shown that video streaming over TCP is possible as long as congestion is avoided [23] and that adaptive HTTP streaming can scale to millions of users [1]. The solution used for example by Microsoft's Smooth Streaming [26], Move Networks [1], Adobe's HTTP Dynamic Streaming [2] and Apple's live HTTP streaming [17] is to split the original stream into segments and upload these to web-servers; such an approach, which is known as MPEG Dynamic Adaptive Streaming over HTTP (MPEG-DASH) [21], has recently been ratified for international standard by ISO/IEC's joint committee known as MPEG. The video segments can then be downloaded like traditional web objects. To allow adaptation to available resources and client devices, each segment may be encoded in multiple bit rates (and thus qualities), and the adaptation is then just a matter of downloading the segment in a different quality. In the live streaming scenario, the segments are produced periodically. A new segment becomes available as soon as it has been recorded, completely encoded and uploaded to a web-server, possibly located in a content distribution network (CDNs) like Akamai or Level 3 [11]. Because the segment must be first fully encoded and uploaded to the server, live adaptive HTTP segment streaming is considered to be only quasi-live.

The nature of network traffic generated by live segment streaming is very different from the traditional bulk transfer traffic stemming from progressive video download and file transfer. In the latter case, the entire file is retrieved using a single request, i.e, the type of operation TCP is designed and optimized for. Segmented HTTP video streaming is characterized by many successive download-and-wait operations creating an *on-off traffic pattern* of periodic HTTP GET requests. This behavior makes TCP behave differently from the traditional bulk operations [5], [24]. In this work, we investigate the system performance of *regular* on-off traffic generated from adaptive HTTP segment streaming.

In this paper, we analyze a multi-user server-side scenario, and present our simulation results describing the behavior of TCP *cubic*, the default Linux TCP variant, in a live HTTP streaming scenario. The observation is that the performance of default TCP cubic is reduced for on-off traffic compared to bulk transfers. Based on the results, we further investigate ways of increasing the overall system performance by looking at protocol alternatives and different streaming options like the segment duration and segment request timing. Based on our experimental results, we outline different ways or recommendations for improving the resource utilization on the sender (server) side: 1) use another TCP congestion control; 2) use longer video segments; 3) avoid that the clients request a segment that became recently available at the same time; and 4) limit the TCP congestion window.

The outline of the paper is as follows: Section II discusses the related work, and Section III describes our simulation setup. We present and discuss our results for single quality HTTP streaming in Section IV before we test our results with a common quality adaptation strategy in Section V and Section VI. In Section VII, we discuss our results, and in

Section VIII we conclude the paper.

II. Related work

On-off sources using TCP for delivering data have been studied in the literature a couple of times. Analytical models were provided for example for simplified TCP Tahoe [5], [12], Vegas [24] and Reno [6]. However, most of these models assume exponentially (randomly) distributed *on* and *off* periods whereas in the segmented HTTP streaming scenario the *on* and *off* periods are distributed differently. Different is also that an HTTP segment streaming *on* period is first over when all bytes of a segment have been downloaded, not when a certain time has elapsed. Furthermore, others [20] assume perfect bandwidth sharing between TCP connections. This is of course a valid assumption if the *on* periods are adequately long to establish bandwidth sharing fairness, but in our scenario, the stream segments are too small to make this assumption valid [7].

Adaptive HTTP segment streaming has been a hot topic over the past few years, but most of the research has focused on the observation and the improvement of a single connection [8], [14], [18]. However, in huge multi-user scenarios like the ones listed in Section 1, the server will be the bottleneck [9], and we therefore need to look at how multiple connections behave when sharing a server.

III. Simulation setup

Because a network consisting of hundreds of computers is practically not feasible in our research lab, we used the ns-2 network simulator [3] to analyze the performance of periodic on-off video streams in a multi-user server scenario. Our simulation setup is based on the observation that network bottlenecks are moving from the access network closer to the server [9]. Figure 1 shows the network we used for our simulations. The links between the client machines and the R2 router are provisioned for more than the highest media bitrate (Table I). The link between the HTTP server and the router R1 is overprovisioned and can support the highest media bitrate for all clients. The link between R1 and R2 is the simulated bottleneck link at the server side with a capacity of 100 Mbit/s (we have also tested with a 1 Gbps bottleneck link, but the trends are the same, i.e., only the total numbers of clients are scaled up). The delays between the clients and the server are normally distributed with an average of $\mu = 55ms$ and a variation of $\sigma = 5ms$. Distributing the delays prevents phase effects in the simulation, and it also seems to be a reasonable assumption for an ADSL access network [16]. The router queue is limited by 1375 packets which corresponds to the rule of thumb sizing of bandwidth delay product. Furthermore, we modeled client arrivals as a Poisson process with an average inter-arrival time $\lambda = \frac{number\ of\ clients}{segment\ duration}$, i.e., all clients join the stream within the first segment duration. This models the case when people are waiting in front of a 'Your stream starts in . . .' screen for the start of the game or TV show so they do not miss the beginning of the show. Finally, to have a realistic set of data, we used the soccer stream used in [19], encoded with variable bitrate (Table I) for a real HTTP segment streaming scenario.

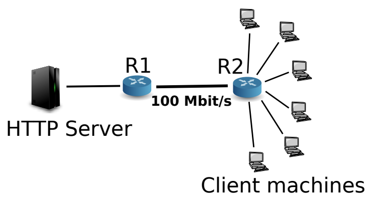


Fig. 1: Simulation setup

Quality	Avg. Bitrate
0	250 kbit/s
1	500 kbit/s
2	750 kbit/s
3	1000 kbit/s
4	1500 kbit/s
5	3000 kbit/s

TABLE I: Video qualities

IV. System effects of HTTP segment streaming

In order to get more insight into the behavior of live adaptive HTTP segment streaming, we first considered a simplified case using only a single quality stream with an average bitrate of 500 Kbit/s (quality 1 from Table I). We used a 20 minutes long video stream of 600 segments with the duration of 2 seconds, as recommended by research [15] and for example Microsoft [26], [13]. We configured the HTTP server to use ns-2's Linux implementation of the *cubic* congestion control algorithm, because *cubic* is used as the default congestion control algorithm in the Linux kernel, which again is used by more than 30% of the websites whose operating system is known [22].

On the client side, we allowed the clients to buffer only one segment. This forces the clients to download a segment, wait for the next segment to become available, download it, wait for the next one etc. There are two real world cases where clients show this on-off behavior. The first case is a scenario where all clients try to stay only one segment duration behind the live stream. The second case occurs when all clients caught up with the live stream and start waiting for the next segment to become available and so enter the download-and-wait cycle.

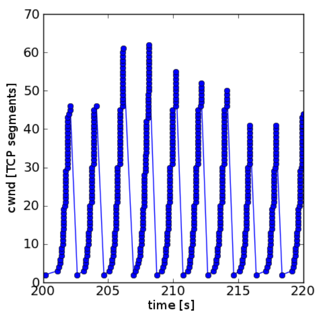
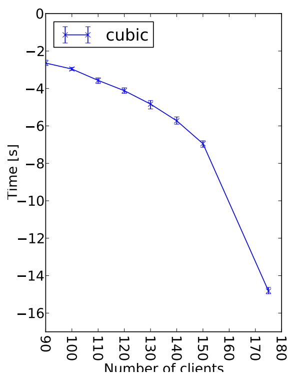
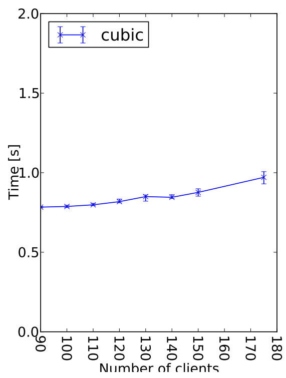
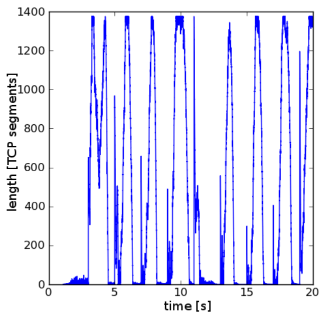


Fig. 2: Observed TCP congestion window.

A typical development of TCP's congestion window during the download-and-wait cycle is shown in Figure 2. The congestion window opens during the download period and collapses while the client is waiting for the next 2-second segment, i.e., the download period always starts with a slow start, which leads to bandwidth not being fully utilized. We also want to


(a) Liveness


(b) Startup


(c) R1 router queue - 150 clients

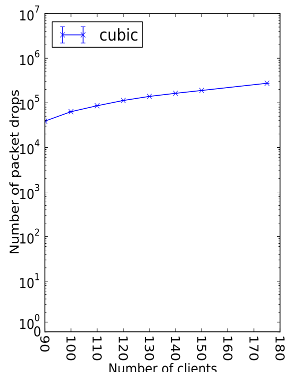
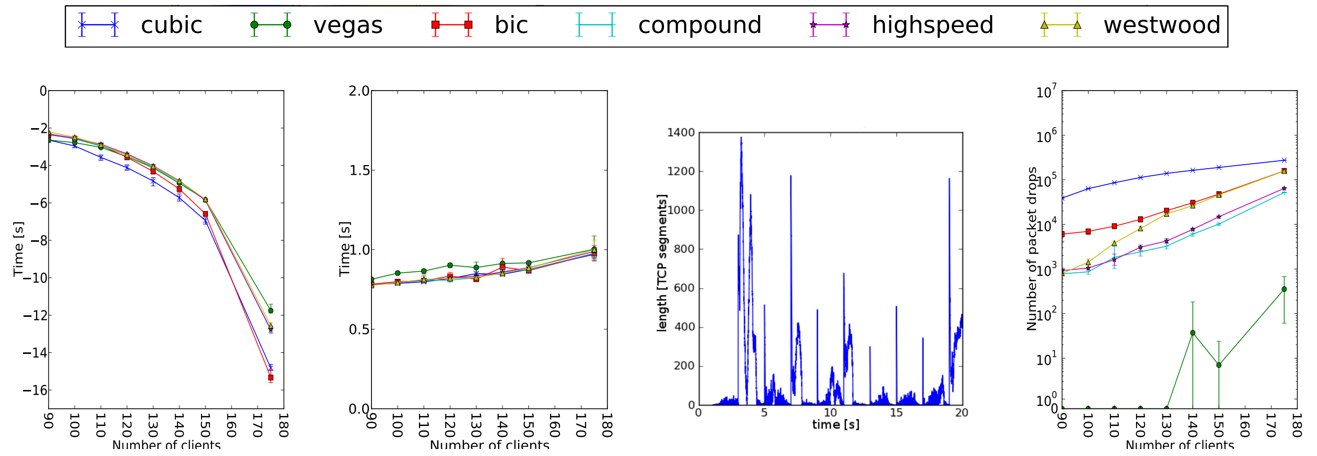

(d) Number of dropped packets

Fig. 3: Performance of TCP *cubic*



(b) Liveness (c) Startup (d) R1 router queue: 150 clients (*vegas*) (e) Number of dropped packets

Fig. 4: Performance of alternative TCP congestion control algorithms

note that the wait (*off*) period ends for all clients (that received the previous segment in time) at about the same time when the next segment becomes available.

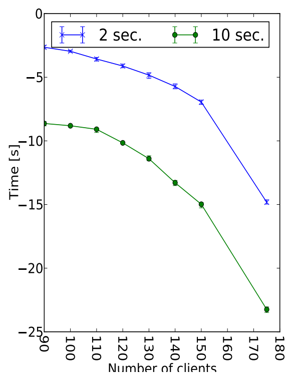
We looked at two important measures for live HTTP segment streaming - startup time and liveness. The *startup time* is the time needed to receive the first segment, after which the playout can begin. The *liveness* measures the time that the client lags behind the live stream at the end of the stream, e.g., a liveness of -4 seconds means that a client will play the last segment 4 seconds after the segment has been made available on the server. There are three main factors that contribute to liveness: client arrival time, client startup time and client playout stalls (buffer underruns) when segments do not arrive in time for playout.

The average liveness and startup time for different numbers of clients for the configuration described above are plotted in Figures 3(a) and 3(b). Every experiment was run with different inter-arrival times 10 times. The plots show the average value and the error bars the min and max value. The server bandwidth of 100 Mbit/s should provide enough bandwidth for smooth playout without pauses for around 200 clients. However, it is clear from the liveness graph that as the number of clients increases the clients start lagging more and more behind the live stream. Furthermore, the figure also shows that the average startup time does not increase dramatically with the number of clients.

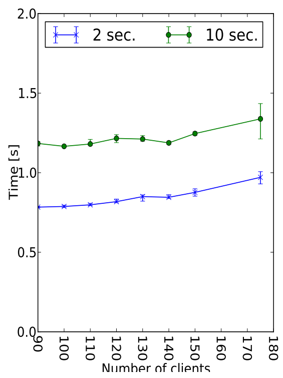
Since the clients arrive in all cases within one segment duration, the change in liveness must be contributed by **playout stalls**. Any playout stalls certainly decrease the user experience and should therefore be prevented. The reason for playout stalls is the inefficient use of bandwidth by the server. Figure 3(c) shows a sample of router R1 queue development over time. We observe that the queue overruns approximately every segment duration when a segment becomes available (the end of a waiting period). The accumulated number of dropped packets is shown in Figure 3(d). We see that there must be a lot of retransmissions reducing the achieved *goodput* of the server. In the following sections, we analyze different ways for improving the system's performance.

A. *Alternative congestion control algorithms*

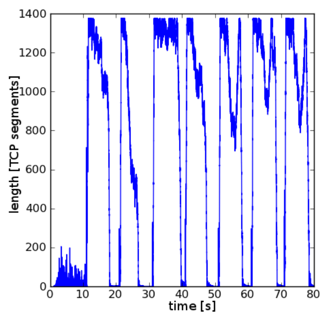
A number of alternative TCP congestion control mechanisms exist that aim at different scenarios. These can be easily interchanged, e.g., in Linux, you may set the `/proc`-variables or use `sysctl`. We repeated the experiments above replacing *cubic* with other alternatives, and the results are plotted in



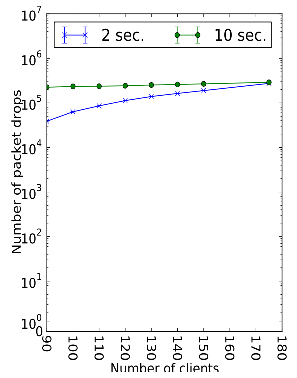
(a) Liveness



(b) Startup



(c) R1 router queue - 150 clients



(d) Number of dropped packets

Fig. 5: Performance of longer segments: 10-second segments (*cubic*)

Figure 4. From the plots, we can observe that BIC and its successor CUBIC give the worst liveness, and thus, most playout hiccups. Using another congestion control algorithm can therefore improve the performance.

An interesting alternative is the *vegas* congestion control algorithm which is designed to back off before a bottleneck queue is overrun. Average liveness and startup time are shown in Figures 4(b) and 4(c), respectively. We observe that *vegas* performs much better. The reason is that this congestion control algorithm manages to avoid overflowing the router queue R1 (at least most of the time) and therefore can cope with the increased number of clients better. Figure 4(d) shows a sample development of the router queue over time for 150 clients in case *vegas* is used. It is apparent from Figures 4(d) and 4(e) that *vegas* induces very few packet losses due to queue overruns.

B. Increased segment duration

Recent congestion control algorithms were optimized for bulk transfer. However, the segmented approach of HTTP streaming creates a short interval *on-off* traffic rather than bulk traffic (commercial systems use standard segment lengths from 2 seconds [26] to 10 seconds [17]). To see the effects of the segment length, we also changed this parameter.

Figure 5 plots the results using different segment lengths. For example, Figure 5(a) shows that the startup times are slightly higher for 10-second segments because they take a longer time to download than 2-second segments. Please note that 10-second segments are 5 times longer in duration than 2-second segments, but not necessarily 5 times larger in size. Quality switching between segments requires that every segment starts with an intra encoded frame (I-frame), but the rest of the frames in a segment can be lightweight P- and/or B-frames that require a much smaller fraction of the segment than an I-frame. Nevertheless, longer-duration segments larger than shorter ones and therefore require more time to download, prolonging the download (*on*) periods. This gives TCP more time to reach its operating state.

The liveness is lower than in the case of 2-second segments for multiple reasons. Firstly, we distributed the client arrivals also here across one segment duration, i.e., 10 seconds. Therefore the client arrival time is in general higher and so decreases the liveness. However, note that the clients still synchronize when they request the second segment. Therefore, in this respect, this scenario is similar to the 2-second segment scenario, but instead of synchronizing their requests at the third second, clients synchronize at the eleventh second.

We can observe that client liveness changes from about -8 seconds to -24 seconds for 175 clients. The drop of liveness with the number of clients is very much comparable to the 2-second case.

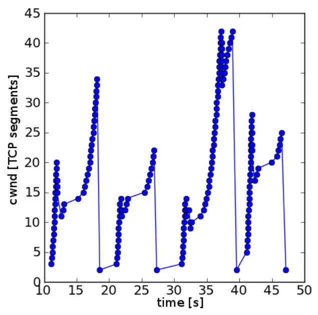
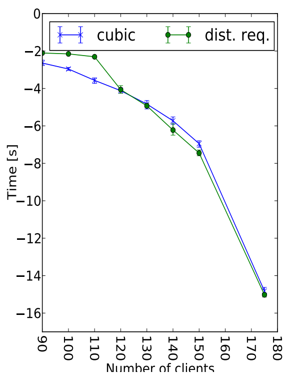


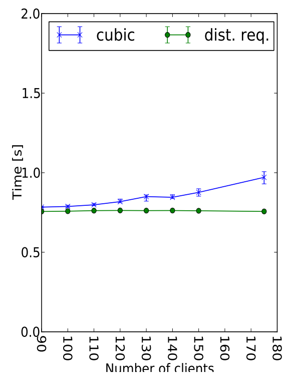
Fig. 6: Sample TCP congestion window for 10 second segments

A trace of a congestion window using 10-second segments is plotted in Figure 6. Compared to the 2-second segment scenario in Figure 2, we see that the window size oscillates with a lower frequency, i.e., relative to the segment size. One can also see the characteristic curve of *cubic* when it probes for available bandwidth. There is no time to do that in the 2-second segment scenario.

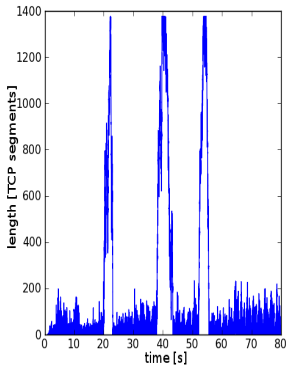
The probing for bandwidth leads to queue overruns as shown in Figure 5(c). These overruns trigger almost constant packet loss independent of the number of clients (Figure 5(d)). Based on this observation we would rather lean towards using 2-second segments because they give us better liveness and give the adaptation algorithm 5 times more chances to adapt stream's quality.



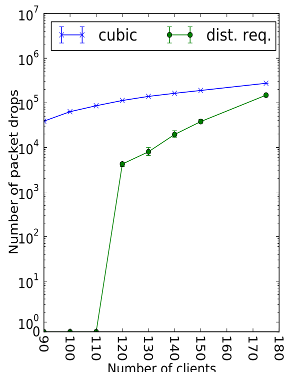
(a) Liveness



(b) Startup



(c) R1 router queue: 150 clients



(d) Number of dropped packets

Fig. 7: Performance of regular vs. distributed requests (*cubic*)

C. *Requests distributed over time*

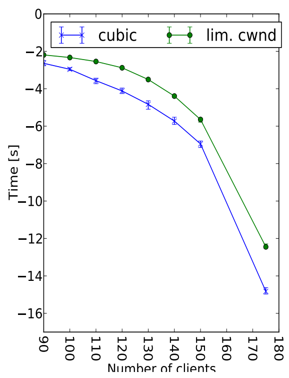
In a live streaming scenario, the clients usually download the new segment as soon as it becomes available. Thus, the server will experience large flash crowd effects with a huge competition for the server resources. However, we showed in [10] that such segment request synchronization leads to reduced performance in terms of quality and liveness. The negative effects can be seen in the default configuration of the server where the router queue fills up (Figure 3(c)), and packets are dropped (Figure 3(d)), i.e., the performance degrades.

To avoid this synchronization, we propose to distribute the requests across the segment duration. For example, upon arrival, the clients make a request every segment duration. This algorithm requires the clients to check the media presentation description only in the beginning of the streaming to find out which segment is the most recent. After that they can assume that a new segment is going to be produced every segment duration. This way the requests stay distributed over time also beyond the first segment.

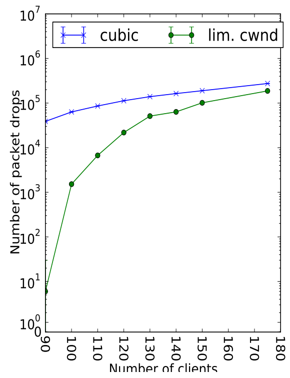
In our experiment, the requests are exponentially distributed over the entire segment duration. The results show that distributed requests increase the liveness if the client number is small (Figure 7(a)). If the number of clients is high, clients that get more than one segment duration behind the stream implicitly build up the possibility for buffering, i.e., they break the download-and-wait cycle and reduce the effect of distributed requests until they catch up with the live stream again. We also see that distributed requests lead to less pressure on the router queue (Figure 7(c)) which can possibly leave more space for other traffic, and that the number of packet losses is greatly reduced (Figure 7(d)).

D. *Limited congestion window*

In Video-On-Demand scenarios, clients can download the whole stream as fast as their download bandwidth and their playout buffers permit. On the other hand, in live streaming scenarios, clients are additionally limited by segment availability. In the download-and-wait cycle, a fast download of a segment prolongs only the wait period. Hence, there is no need for the client to download a segment as quickly as possible as long as it is downloaded in time for playout.



(a) Liveness



(b) Number of dropped packets

Fig. 8: Performance of a limited TCP congestion window (*cubic*)

TCP's bandwidth sharing is fair for long running data transfers. However, for short transfers, the sharing is in many cases very unfair. To reduce this unfairness, we experiment with a limited TCP congestion window. The limited congestion window can lead to longer download times, resulting in a behavior similar to TCP pacing [4]. To avoid playout stalls due to congestion window limitation, we chose the congestion window so that a segment can easily be downloaded in one segment duration. We set the congestion window to 20 TCP segments, which equals to a bandwidth 3 times bigger than the average bitrate of the stream (to account for bitrate variance). The startup time is low as in the other scenarios tested above, yet the average liveness is improved (Figure 8(a)). Furthermore, from Figure 8(b), we observe a significantly reduced number of dropped packets which also indicates a lighter load on the bottleneck router queue, resulting in a more efficient resource utilization.

V. Application effects using quality adaptation

In the previous section, we showed how different changes of default settings have a beneficial impact on HTTP segment streaming without adaptation. In this section, we look at the impact of these changes on the adaptive HTTP segment

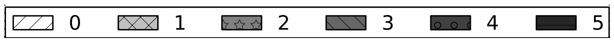


Fig. 9: Quality coding in figures from low (0) to high (5)

streaming. We chose a classic adaptation strategy that is very similar to the strategy used by Adobe's HTTP Dynamic Streaming [2], [11]. This strategy is known to follow the available bandwidth very closely [14], [18]. We use the following formula to estimate the available bandwidth:

$$b_s = \alpha * b_{s-1} + (1 - \alpha) * \frac{segment\_size_s}{download\_time_s}$$

The estimated bandwidth b_s after the download of a segment s is the weighted sum with aging (α was 0.1) of the estimated bandwidth after segment $s - 1$ and the quotient of the size of segment s and its download time $download\_time_s$. After the bandwidth is estimated, the strategy finds the sizes of the next segment using a HTTP HEAD request and chooses to download the segment in the highest bitrate (quality) possible so that the estimated download time does not exceed one segment duration.

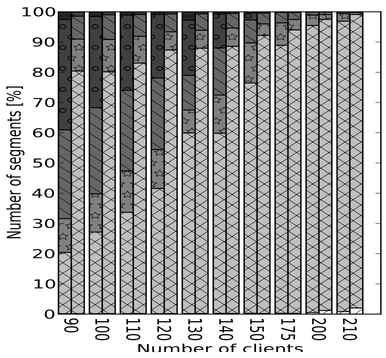
In the following subsections, we compare a quality adaptive scenario using a system with the default settings from section IV (*unmodified*) with systems that include modified settings over a 30 minute playback. Here, the startup times are in general lower than in the previous section since the first segment is downloaded in a low quality to ensure a fast startup like in most of the commercial systems. We focus therefore only on the user experience metrics liveness (smooth playout) and segment bitrate (video quality). Note that a segment can only be encoded when all (video) data in it is available, i.e., the best achievable liveness is one segment duration.

A. Alternative congestion control algorithms

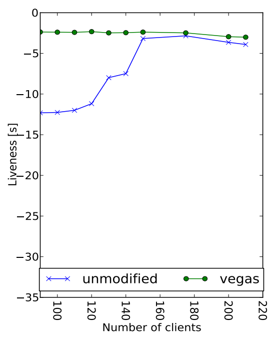
In the previous sections, we saw that the *vegas* congestion control is slowing down connections' throughput to reduce packet losses, which leads in general, as observed in Figure 10(a), to much lower quality than that achieved with *cubic* (for quality coding see Figure 9). However, because of its very careful congestion control, *vegas* is not causing the clients to have many late segments. The liveness (and thus video playout) is therefore almost perfect without any hiccups. On average, clients lag only about 2 seconds behind the live stream independent of their number. In case of *cubic*, the clients need to buffer at least 12 seconds for a smooth playout for smaller numbers of clients. When the number of clients rises, the competition for the resources increases, forcing every client to select a lower quality, thereby improving liveness. Thus, *vegas* should be considered if liveness is the goal, and the quality is secondary or if clients cannot afford buffering.

B. Increased segment duration

In section IV-B, we compared short segments with long segments from the system perspective where longer segments give TCP more time to reach its operating state. However, the results showed that 2-second segments lead in general to better performance than 10-second segments. The question answered



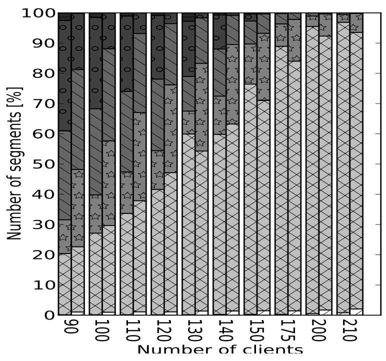
(a) Segment quality (regular left bar)



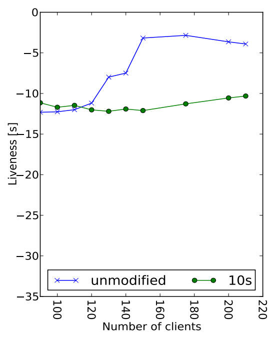
(b) Liveness

Fig. 10: Alternative congestion control: *cubic* vs. *vegas*

in this section is how this influences the segment video quality. In this respect, Figure 11(a) shows that the relative number of high quality segments is lower compared to the 2-second case. The liveness seems also worse, but it is actually quite good for 10-second segments. The reason is that the best liveness achievable is always one segment duration, i.e., 10 seconds, as we explained in the beginning of this section. This means that the clients need to buffer only one segment, even though it is 10 seconds long. Nevertheless, it seems to be more efficient to use 2-second segments – both from the system and the user perspective. This raises the question if even shorter segments should be used, but this would negatively influence the perceived user quality [15].



(a) Segment quality (regular left bar)

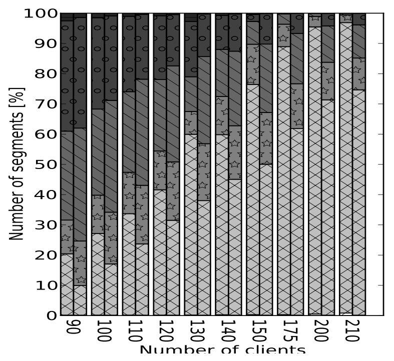


(b) Liveness

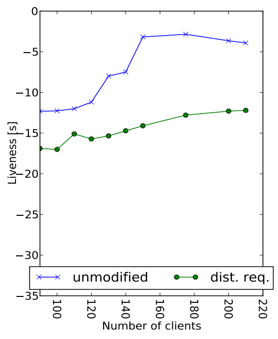
Fig. 11: Segment lengths: 2 vs. 10 seconds

C. Requests distributed over time

Section IV-C showed a better system performance when distributing the requests for a segment over the entire segment period. Figure 12(a) shows the respective quality improvement when the requests are distributed as in the previous section. The quality improvement is obvious especially when the number of clients is high. However, we need to point out that the liveness suffers. Figure 12(b) shows that because of the poor quality, the strategy without any modifications is able to provide the clients with a more 'live' stream.

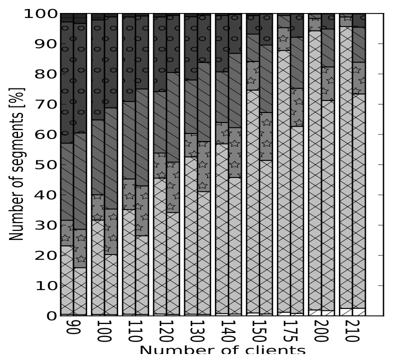


(a) Segment quality (regular left bar)

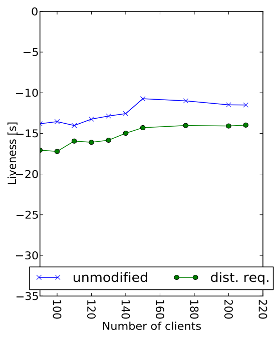


(b) Liveness

Fig. 12: Request distribution (1 segment buffer)

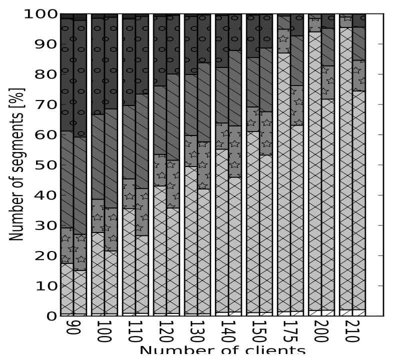


(a) Segment quality (regular left bar)

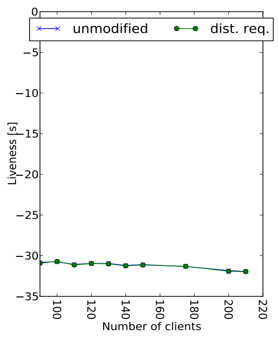


(b) Liveness

Fig. 13: Request distribution (5 segment buffer)



(a) Segment quality (regular left bar)



(b) Liveness

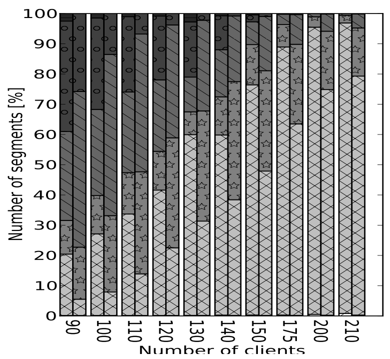
Fig. 14: Request distribution (15 segment buffer)

In practice, however, this would be handled by allowing buffering, which does affect liveness, but fixes the interruptions in playout. Thus, the observed liveness of -16 seconds means the client would require at least a 14 seconds buffer in order to play the stream smoothly without any pauses. For live streaming, this means that a client cannot request the most recent segment from the server when it joins the stream, but has to request an older segment to have a chance to fill its buffer before catching up with the live stream and so entering the download-and-wait cycle.

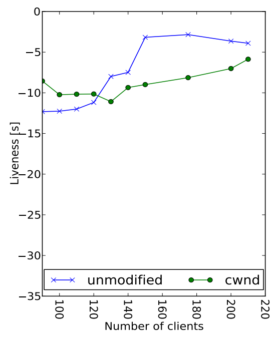
Figure 13 shows the quality improvement and liveness when clients use 5-segment (10-second) buffers. We observe that the quality gain of distribution is preserved while also improving the liveness. Moreover, Figure 14 shows the trend of improved quality, yet comparable liveness continues when the size of buffers is further increased to 15 segments (30 seconds). We conclude therefore that distributing requests is important especially when client buffers are already in place.

D. Limited congestion window

Restricting the client's TCP congestion window and so distributing the download over a longer time has positive effects not only on the server and bottleneck resources (Section IV-D), but also on quality as shown in Figure 15(a). The interesting thing about reducing the congestion window is that for small numbers of clients both the liveness and the quality are better. As the number of clients grows, the quality of an unmodified adaptation strategy degrades quickly, improving the liveness as shown in Figure 15(b). The modified settings keep the liveness about the same irrespective of the number of clients, which is a good thing for practical implementation reasons when the client buffer size is to be chosen.



(a) Segment quality (regular left bar)



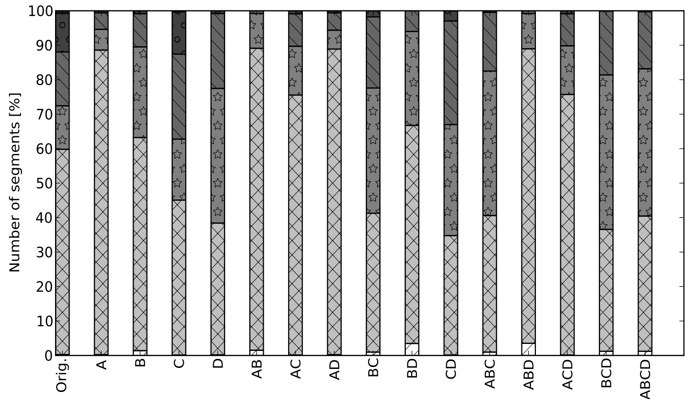
(b) Liveness

Fig. 15: Limiting the congestion window (*cubic*)

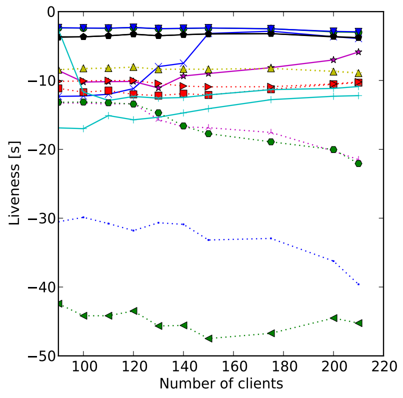
VI. Combination of alternative settings

In this section, we briefly outline and compare the effect of the 16 different setting combinations (see Figure 16) using the 140 client scenario as a representative example. It is again important to note that the graphs must be evaluated together, i.e., a gain in quality (Figure 16(a)) often means reduced liveness (Figure 16(b)) and increased packet loss (Figure 16(c)).

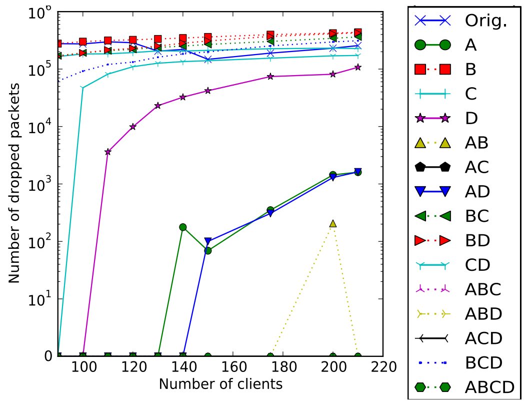
As seen above, changing congestion control, i.e., replacing *cubic* with *vegas*, results in no or very mild loss which leads



(a) Quality - 140 clients



(b) Liveness



(c) Drops

Fig. 16: Performance of combined settings (A=*vegas*, B=10 second segments, C=distributed requests, D=cwnd limitation)

to much better liveness, also in combinations. On the other hand, it has implications on quality, which is lower compared to the default setting, *cubic*. Furthermore, 10-second segment combinations show lower quality than the same combinations with 2-second segments. The only exception is *vegas* used with distributed requests, but it is still worse than other combinations. The liveness is naturally smaller using longer segments. Moreover, in general, combinations with distributed requests results in higher quality, however, it decreases the liveness in most of the cases. Finally, a congestion window limitation does not seem to have influence on liveness or quality apart from the case when it is the only setting modified, or when used with distributed requests where it performs better in terms of liveness especially for small client numbers. Thus, there are several combinations that can be used, but it is usually a trade-off between liveness (which can be fixed by buffering and a startup latency) and video quality.

VII. Discussion and Future Work

The importance of resource utilization is often a question of the number of supported clients. The beauty of adaptive streaming is that the system adapts to resource availability by downgrading the quality when the number of clients increases, e.g., we do see in our experiments that we can support more or less the same number of clients due to the adaption. However, an improvement in resource utilization does increase the users' perceived quality, i.e., it reduces delayed packets and playout interrupts and increases video quality. Furthermore, at a given lower quality threshold, the experiments shows that the number of clients can be increased, i.e., the client number per (CDN) machine can be higher using the tested configurations.

In this paper, we have investigated various techniques to improve the server resource utilization in the HTTP streaming scenario, but the proposed changes do also have potential positive effects outside the single server machine. For example, distributing the requests over a short time period (like the segment duration) gives the web proxy caches time to buffer the data meaning that a lot of the requests can be served by machines in the network, offloading the CDNs. The research to quantify such broader gains is, however, out of the scope of this paper and left for future work.

We would like to point out that the *on-off* traffic pattern does not only apply to live streaming but also to Video-On-Demand, when the client filled its buffer and is waiting for the playout to consume a segment from it. In this case, however, the requests from clients are not synchronized and so rather resemble our distributed requests.

An interesting investigation is what happens if the congestion window is not truncated when the TCP connection is idle. How much does it influence the clients competition when the connections resume after an *off* period with full speed and a big burst of data? Avoiding a colapse of the window is a potential improvement, and we will look into this in the next step of our investigation.

We would also like to mention that we are working on estimating the influence of cross traffic on our results. This might be interesting especially for *vegas* since it is known to loose against other more aggressive TCPs. Cross traffic would also help to estimate the impact of our modifications if the bottleneck is closer to the clients. In the future we plan to evaluate the impact of other possible modifications on the investigated scenario and we also plan to evaluate our results with real large scale systems in a collaboration with a Norwegian video provider.

VIII. Conclusion

We have first studied the performance of HTTP segment streaming in a multi-user server-side scenario. It turns out that because of the different traffic pattern of HTTP segment streaming, the server is able to handle much fewer clients than its network bandwidth would allow. We proposed and tested four different methods (and their combinations) ranging from network-layer to application-layer changes to improve the situation. Additionally, we re-evaluated these methods also in a scenario where clients used a quality adaptation strategy.

Our results show that other congestion control algorithms (like *vegas*) avoid queue overruns, also in the case of short (up to 2 seconds) connections. Therefore, its use leads to very few late segments which means an increased liveness with less buffer underruns. On the other hand, *vegas*'s careful approach reduces the quality. Moreover, we do not find a persuasive reason for using 10-second segments, i.e., in a single-user scenario, it uses the congestion window better, but in a multi-user scenario with competition for the resources, neither the video quality nor the liveness are improved over 2-second segments. We further observe that the distribution of requests over time leads to very good results in terms of quality and in case of client side buffers also liveness. Limiting the congestion window contributes to better fairness among the connections and gives a better bandwidth estimation. This results in good quality and stable liveness for different numbers of clients, but only if used as the only modification.

Acknowledgment

This work is supported by the iAD (Information Access Disruptions) center for Research-based Innovation funded by Norwegian Research Council, project number 174867.

References

[1] "Move Networks," 2009. [Online]. Available: http://www.movenetworks.com

[2] "HTTP dynamic streaming on the Adobe Flash platform," 2010. [Online]. Available: http://www.adobe.com/products/-httpdynamicstreaming/pdfs/httpdynamicstreaming_wp_ue.pdf

[3] "The Network Simulator ns-2," 2012. [Online]. Available: http://www.isi.edu/nsnam/ns/

[4] A. Aggarwal, S. Savage, and T. Anderson, "Understanding the performance of tcp pacing," in *Proc. of IEEE INFOCOM*, 2000, pp. 1157 – 1165.

[5] C. Casetti and M. Meo, "A new approach to model the stationary behavior of tcp connections," in *Proc. of IEEE INFOCOM*, 2000, pp. 367–375.

[6] W. Elkilani, "An analytical model of non-persistent tcp sessions sharing a congested internet link," in *Proc. of ICNM*, 2009, pp. 76 –82.

[7] S. Ha, I. Rhee, and L. Xu, "Cubic: a new tcp-friendly high-speed tcp variant," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 64–74, Jul. 2008.

[8] I. Hofmann, N. Farber, and H. Fuchs, "A study of network performance with application to adaptive http streaming," in *Proc. of BMSB*, june 2011, pp. 1–6.

[9] L. Kontothanassis, "Content Delivery Considerations for Different Types of Internet Video," 2012. [Online]. Available: http://www.mmsys.org/?q=node/64

[10] T. Kupka, P. Halvorsen, and C. Griwodz, "An evaluation of live adaptive HTTP segment streaming request strategies," in *Proc. of IEEE LCN*, 2011, pp. 604–612.

[11] T. Lohmar, T. Einarsson, P. Frojdh, F. Gabin, and M. Kampmann, "Dynamic adaptive http streaming of live content," in *Proc. of WoWMoM*, 2011, pp. 1–8.

[12] M. A. Marsan, C. Casetti, R. Gaeta, and M. Meo, "Performance analysis of tcp connections sharing a congested internet link," *Perform. Eval.*, vol. 42, no. 2-3, pp. 109–127, Oct. 2000.

[13] Move Networks, "Internet television: Challenges and opportunities," Move Networks, Inc., Tech. Rep., November 2008.

[14] C. Müller, S. Lederer, and C. Timmerer, "An evaluation of dynamic adaptive streaming over http in vehicular environments," in *Proc. of MoVid*, 2012, pp. 37–42.

[15] P. Ni, A. Eichhorn, C. Griwodz, and P. Halvorsen, "Fine-grained scalable streaming from coarse-grained videos," in *Proc. of NOSSDAV*, Jun. 2009.

[16] L. Plissonneau and E. Biersack, "A longitudinal view of HTTP video streaming performance," in *Proc. of MMSys*, 2012.

[17] R. Pantos (ed), "HTTP Live Streaming," 2009. [Online]. Available: http://www.ietf.org/internet-drafts/draft-pantos-http-live-streaming-00.txt

[18] H. Riiser, H. S. Bergsaker, P. Vigmostad, P. Halvorsen, and C. Griwodz, "A comparison of quality scheduling in commercial adaptive http streaming solutions on a 3g network," in *Proc. of MoVid*, 2012, pp. 25–30.

[19] H. Riiser, T. Endestad, P. Vigmostad, C. Griwodz, and P. Halvorsen, "Video streaming using a location-based bandwidth-lookup service for bitrate planning," *ACM TOMCCAP*, accepted 2011.

[20] H.-P. Schwefel, "Behavior of tcp-like elastic traffic at a buffered bottleneck router," in *Proc. of INFOCOM*, 2001, pp. 1698–1705.

[21] T. Stockhammer, "Dynamic adaptive streaming over HTTP - standards and design principles," in *Proc. of MMSys*, 2011, pp. 133–144.

[22] W3techs, "Usage statistics and market share of unix for websites - april 2012," http://w3techs.com/technologies/details/os-unix/all/all, 2011.

[23] B. Wang, J. Kurose, P. Shenoy, and D. Towsley, "Multimedia streaming via TCP: an analytic performance study," in *Proc. of ACM MM*, 2004.

[24] A. Wierman, T. Osogami, and J. Olsen, "A unified framework for modeling tcp-vegas, tcp-sack, and tcp-reno," in *Proc. of MASCOTS*, 2003, pp. 269–278.

[25] YouTube, http://youtube-global.blogspot.com/2012/01/holy-nyans-60-hours-per-minute-and-4.html, Jan. 2012.

[26] A. Zambelli, "Smooth streaming technical overview," http://learn.iis.net/page.aspx/626/smooth-streaming-technical-overview/, 2009.