ABSTRACT

Online video streaming through mobile devices has become extremely popular nowadays. Youtube, for example, reported that the percentage of its traffic streaming to mobile devices has soared from 6% to more than 40% for the past two years. Moreover, people are constantly seeking to stream high quality videos for better experience while often suffering from limited bandwidth. Thanks to the rapid deployment of content delivery networks (CDNs), popular videos now have multiple replicas at different sites, and users can stream videos from a close-by location with low latency. As mobile devices nowadays are equipped with multiple wireless interfaces (e.g., WiFi and 3G/4G), aggregating bandwidth for high definition video streaming has become possible.

We propose a client-based video streaming solution, called MSPlayer, that takes advantage of multiple video sources as well as network paths through different interfaces. MSPlayer reduces start-up latency, and aims to provide high quality video streaming and robust data transport in mobile scenarios. We experimentally demonstrate our solution on a testbed and through the Youtube video service.

1. INTRODUCTION

With the high demand for online video streaming, video content providers are offering better technology to satisfy customers’ desire of streaming high quality videos. However, streaming video to a user nowadays still suffers from the following issues. First, people from time to time encounter insufficient bandwidth. Research has shown that viewers are not patient enough to wait for a start-up delay longer than a few seconds [20] and video quality has a huge impact on user engagement [9]. Furthermore, since network bandwidth is highly variable, to cope with bandwidth variation, commercial video players have experimented with video rate adaptation, which in turn results in performance issues such as variable video quality, unfairness to other players, and low bandwidth utilization [5,6,18,21].

As mobile devices are now equipped with multiple wireless interfaces connecting to different networks (WiFi or 3G/4G), aggregating available bandwidth from these networks can reduce video start-up delays and can provide users with better streaming experience and higher video quality. Moreover, in mobile scenarios, path quality varies quickly and connections to a particular network can break down temporarily. Re-establishing a connection introduces additional delays. Hence, leveraging multiple interfaces simultaneously provides robust video transport in rapidly changing environments.

However, as servers hosting popular content and high quality videos of high quality might quickly become overloaded, it is critical to stream high quality videos without overloading the video servers. With popular content replicated at multiple locations in the content delivery networks (CDNs), users can stream one particular video from different video servers belonging to different networks or data centers.

In this paper, we make the following contributions. We first inspect YouTube’s video service architecture and its streaming mechanism. We show how one can utilize both of the available WiFi and 4G interfaces simultaneously to aggregate bandwidth for higher quality video streaming without being hindered by middle-boxes (as is the case with MPTCP) and instantiate this in our YouTube player, MSPlayer. By understanding YouTube’s streaming mechanism, we further demonstrate how to utilize the existence of multiple video sources in different networks at the same time. We then experimentally evaluate the performance of different MSPlayer schedulers as well as the performance of MSPlayer through the YouTube service.

2. YOUTUBE BACKGROUND

People usually go to the YouTube website and choose a video to watch, or just click on an URL of the following form http://www.youtube.com/watch?v=qjT4T2gU9sM on a web page. Users can then watch the video through their browsers using built-in Adobe Flash player [1]. Each YouTube video is identified by an URL and the 11-literal string after watch?v= is the video ID [8].

With this URL, the video player (e.g., Adobe Flash) first performs a DNS lookup to resolve the IP address of www.youtube.com and the user’s video request will be directed to one of YouTube’s web proxy servers. The
YouTube web proxy server then processes the request and returns the related video information and a new URL to the user, indicating where the associated and available YouTube video servers are. The player henceforth establishes another connection to one of the dedicated video servers and starts to stream the YouTube video with HTTP byte range requests (see Sec 5 for details about interactions with YouTube servers).

3. DESIGN PRINCIPLES

Our solution, MSPlayer, is a client-based approach which requires no changes at either the server or the client’s TCP stack. It leverages diversity in the network and performs load balancing at the client side. MSPlayer also supports user mobility and provides robust data transport. In order to be fair to other TCP users, MSPlayer limits the number of paths to two (one over WiFi and one over 3G/4G) and leverages HTTP range requests to stream videos. It has the following design features for video streaming.

Just-in-time with High Quality: Since viewers often prematurely stop watching videos [10, 20], streaming the entire video to a viewer at once can result in network bandwidth and resource waste. That, along with the rise of adaptive streaming over HTTP [24], has drawn attention to just-in-time video delivery, which has been exploited by most large scale video streaming services such as YouTube, Netflix, and Hulu. A video is partitioned into many small file segments, call video chunks. The video server maintains multiple profiles of the same video for different bitrates and video quality levels. Clients hence periodically request video chunks and adapt video bitrates.

Just-in-time video delivery prevents potential resource wastes if a user drops the video during its playback. Dynamically adapting video bitrates, however, results in performance issues such as low link utilization [3], unfairness to other players/TCP users [6, 18], and instability in video quality [6, 21]. In our design, we share the same just-in-time concept of delivering videos. However, we do not investigate rate adaption and focus on how to stream videos users with a fixed bitrate supported by exploiting network diversity.

Robust Data Transport: When a user streams a video while moving, his connection (mostly WiFi) might break and the downloaded video is thus abandoned. In order to resume the video, the user then needs to switch to another interface and another network, establish a new TCP connection, and move/skip to the break point. In the worst case, one will need to wait until reaching the next WiFi hotspot and repeat the above process.

Multi-Path TCP (MPTCP) [11] has been standardized by the IETF, aiming to provide robust data transport. However, MPTCP requires kernel modifications at both the server and the client side [22] and thus is infeasible for wide deployment at the moment. Moreover, MPTCP relies on the TCP option field to exchange path and interface information while it has been reported that MPTCP could suffer significantly from network middleboxes as they very often strip away unknown options [15, 16], forcing MPTCP connections to fall back to legacy single-path TCP. In our measurements, two out of three major US cellular carriers do not allow MPTCP traffic to pass through the default HTTP 80 port, which is henceforth a potential problem for video streaming to popular sites such as YouTube or Netflix.

In our proposal, we design a client-based multi-path solution to provide robust data transport for high quality video streaming. Furthermore, each of the paths runs legacy TCP and is therefore guaranteed to successfully pass middleboxes in the networks.

Content Source Diversity: The current MPTCP design [11] and other similar approaches, such as [7], only allow flows or paths to be established between a client and a single server. If the current YouTube infrastructure were to support MPTCP, users streaming videos from one server with high aggregate bandwidth through multiple paths could quickly incur demand surges. This high demand, particularly for high quality videos, can overload the sever and congest the bottleneck links. The outcome of this could directly or implicitly affect other viewers’ experience.

As popular content is now replicated at multiple locations or data centers, content delivery networks (CDNs) are responsible for handling video replicas and delivering videos across different data centers for large scale video streaming services such as YouTube, Netflix, and Hulu [1, 2]. As part of design to provide robustness, MSPlayer, at the initial phase, collects a list of YouTube servers’ addresses in each network exploited. If a server in a network fails or is overloaded, MSPlayer switches to another server in that network and resumes video streaming. Other proposals, such as [14], aim to emulate the use of multiple paths in a controlled environment by setting up multiple connections to the servers connected by a switch with only one single interface. Although this approach can potentially distribute the load on the connected servers, having multiple connections over one interface could quickly saturate the bottleneck link.

As wireless interfaces are associated with different networks, MSPlayer requests partial content from video servers in both networks simultaneously to avoid overwhelming particular video servers and to balance the load across the servers. In this work, we use Google’s public DNS service in both networks to resolve the IP addresses of YouTube servers.
**Chunk Scheduler:** MSPlayer relies on range requests to retrieve video chunks over different paths. As making a range request incurs additional overhead (packets start to arrive one RTT after the request is sent) and different paths usually exhibit diverse latency [8], how to schedule chunks over different paths efficiently is challenging. Therefore, having a good scheduler that estimates path quality over time and assigns chunks to each path is highly desired.

To satisfy the just-in-time video delivery, the scheduler pauses chunk retrieval when the playout buffer is full and resumes chunk requesting when there is not enough data in the buffer (referred to as periodic downloading or ON/OFF cycles [23]). To reduce the memory usage of out-of-order chunks from different paths, the MSPlayer scheduler manages to have each assigned chunk over each path completing at the same time when they arrive at the client, and allows at most one out-of-order chunk to be stored in the memory.

### 4. TESTBED AND YOUTUBE IN THE WILD

In MSPlayer, the design components are first evaluated on a testbed in a controlled environment which emulates YouTube’s video streaming mechanism. The final performance evaluation is carried out using the real YouTube service. Two types of servers are emulated in our testbed: web proxy servers (responsible for authentication and video object information) and video servers. Both types of servers use the standard Linux 3.5 kernel with CUBIC congestion control [12] with Apache service. Each type of server is hosted in two different UMass subnets for source diversity.

The client running MSPlayer is equipped with two wireless interfaces, WiFi and LTE, connecting to a home WiFi network and one of the major US cellular carriers. Video requests are sent through both interfaces simultaneously to two different video web proxy servers. Upon receiving packets from the web server, MSPlayer decodes the associated JSON objects (with a pre-loaded video server’s IP address) and fetches video chunks from the video servers. The videos are pre-downloaded from YouTube with MP4 format of HD (720p) video quality and 44100 Hz audio quality.

### 5. MSPLAYER IMPLEMENTATION

In order to exploit both available wireless interfaces simultaneously, when accessing the sockets, we pass additional interface information to the socket API to bind each interface to an IP address so that packets can be scheduled to a particular interface. Moreover, for each interface, an independent routing table is configured so that when a source IP address is specified, instead of using the default interface and gateway, the desired interface and gateway are used. Since video players can access YouTube videos through Google’s Data APIs [13], MSPlayer is hence developed to leverage source and path diversity in the network for YouTube video streaming by interacting with Google’s APIs.

First, when the desired video object is chosen, the player contacts the web proxy server with the URL containing the 11-literal video ID. The web proxy server then authenticates the user (player type and/or the user account) with OAuth 2.0 and verifies the video operations requested by the user [13]. When the requested operations are granted, the web proxy server resolves the user’s public IP address and check to see which video server is suitable and available to this user. Afterwards, the web proxy server generates an access token (valid for an hour) that matches the video server’s IP address as well as the operations requested.

The web proxy server then encodes the token, together with the user’s public IP address and the video’s information (i.e., available video formats and quality, title, author, file size, view counts, timestamp, video server address, ..., etc) in JavaScript Object Notation (JSON) format and returns these objects to the user through the requested interface. MSPlayer hence decodes the JSON objects received on each interface and synthesizes a new URL (with the required information, video server address, and a valid token) to contact the corresponding video server in the associated network. Video content is then retrieved by making HTTP range requests to different video servers through both interfaces. Note that YouTube has been gradually replacing insecure HTTP connections with secure ones. To be compatible to the current and future YouTube’s data service, we follow YouTube’s latest HTTPS connection policy with both web proxy servers and video servers.

As part of the just-in-time video delivery principle, MSPlayer has the following streaming strategy similar to commercial YouTube players such as Adobe Flash player or HTML5: a pre-buffering phase followed by a steady-state re-buffering phase [23]. The pre-buffering phase takes place at the beginning of the streaming, aiming to retrieve enough video data in the playout buffer. MSPlayer leaves the pre-buffering phase when more than 40-second video data is received. It then waits for the player to consume the video data until the playout buffer contains less than 10-second video. At this point in time, MSPlayer resumes requesting chunks from both YouTube servers and refill the playout buffer. This periodic re-buffering repeats until the video is completely watched or dropped.

#### 5.1 Multi-Source and Multi-Path

Before exploring our scheme with multi-source and multi-path, we first investigate the details of how each

---

1As YouTube’s client libraries are mostly in web languages, MSPlayer, however is programmed in python rather than JavaScript or PHP.
path establishes a connection to the YouTube web proxy server and the associated video server. Fig. 1 illustrates a flow diagram for the contact with YouTube’s web proxy server to retrieve video information. The connection starts with TCP 3-way handshake (3WHS), and at time $t_1$, the client initiates the secure connection handshake message. It takes the server time durations of $\Delta_1$ and $\Delta_2$ to verify the key and the key exchange process.

The first HTTP request is made at time $t_4$, and the first JSON packet from the web proxy server arrives at $t_5$. Note that these JSON packets are delivered within two round trips (slightly less than 20 packets), and the secure connection ends at $t_6$ followed by TCP’s FIN.

If we denote by $R_1$ and $R_2$ the RTTs of the first and the second paths, and the RTT ratio $\theta = R_2/R_1$ (assuming $R_1 \leq R_2$, hence first path is a fast path), it takes $\eta_i = 4R_i + \Delta_1 + \Delta_2$ to establish a secure HTTP connection over path $i$, and $\psi_i = 6R_i + \Delta_1 + \Delta_2$ to receive complete video information before contacting the video server. If the YouTube’s web proxy server is close to the video server, and both servers have similar capability for key verification, it takes approximately $\pi_i \approx \psi_i + \eta_i$ for path $i$ to receive the first video packet from the video server.

Therefore, before the second path starts to retrieve video packets from its contacted video server, the first path will receive video packets for a duration of $\pi_2 - \pi_1 \approx 10 \cdot (R_2 - R_1) = 10 \cdot (\theta - 1)R_1$.

In MSPlayer, the process of fetching video chunks of each path is an independent thread, and the threads are under the management of the chunk scheduler (described in the next section). As part of chunk scheduling in MSPlayer, the first path is designed to contact its video server as soon as its video server’s IP address is decoded, without waiting for the second path to finish its decoding process. The first path can hence utilize the duration of $\pi_2 - \pi_1$ to further reduce the start-up delay (i.e., the first few chunks are scheduled to the first path before the second path is available).

Fig. 2 demonstrates the initial video per-buffering download time using single-path WiFi, single-path LTE, and MSPlayer for HD videos in our emulated testbed.

Note that a 40-sec pre-buffering period is presented here as it is YouTube servers’ default pre-buffering size for Flash videos [23]. The median download time of MSPlayer is 6.9 seconds while that of the best single-path over WiFi is 10.9 seconds, with around 37% delay time reduction in the pre-buffering phase.

As MSPlayer leverages multiple video sources and interfaces/paths, how packets are scheduled through each path to each server can significantly affect the performance. The MSPlayer result in Fig. 2 uses a baseline scheduler called Ratio with initial chunk size 1 MB (details see Sec. 5.2). Next, we will investigate different MSPlayer schedulers and evaluate their performance.

### 5.2 Chunk Scheduler

In order to reduce out-of-order delay for video streaming, and to reduce memory usage to store out-of-order chunks, a rule-of-thumb design is to schedule chunks (of different sizes) over both paths so that the transfers complete at roughly the same time.

To optimize video streaming performance with MSPlayer, chunk size selection for each path is critical and should be adapted over time in response to network dynamism. A previous measurement study shows that YouTube players, such as Adobe Flash or HTML5, use 64 KB and 256 KB as their default chunk sizes, while Netflix player (silverlight) uses larger chunk sizes that range from 2 MB to 4 MB [23]. We examine the performance of different schedulers with different initial chunk sizes for each path. Since different mobile devices have pre-buffering periods of different lengths (ranging from 20 seconds to 1 minute) [22], we also investigate the performance of different schedulers when applying different chunk sizes and pre-buffering periods.

We denote by $S_i(t)$ the chunk size of path $i$ at time $t$, by $B$ the base chunk size, and by $T_i$ the time required to download chunk $S_i(t)$. The estimated throughput to download $S_i(t)$ is denoted by $w_i(t) = S_i(t)/T_i$.

We first investigate a baseline scheduler called Ratio. We then compare the baseline with two different schedulers that adjust chunk sizes according to network...
bandwidth changes, namely the exponential weighted moving average (EWMA) and Harmonic.

Baseline Scheduler: suppose \( w_i(t) \leq w_{i-1}(t) \), the baseline Ratio scheduler assigns a fixed chunk size to the path with lower throughput such that \( S_i(t + 1) = B \) and adjusts the chunk size of the path with higher throughput based on throughput ratio (i.e., \( S_{i-1}(t + 1) = w_{i-1}(t)/w_i(t) \cdot B \) and \( i = 0,1 \).

Dynamic Chunk Adjustment: MSPlayer chunk size selection should adapt to path quality variations over time. The bandwidth estimator hence plays a critical role in the chunk size selection process.

When path bandwidth estimates are available, the chunk size of each path is adjusted according to Algorithm 1.

We denote by \( \delta \) the performance fraction threshold. For the slow path, if its current bandwidth measurement is \((1 + \delta)\) times better than the estimated value, the chunk size is doubled. Similarly, if the current value is \((1 - \delta)\) times worse than the estimated value, the chunk size is halved. For the fast path, its chunk size is adjusted based on the throughput ratio.

We focus on two bandwidth estimators: exponential weighted moving average (EWMA) and Harmonic. The weighted moving average is defined as:

\[
\hat{w}_i(t + 1) = \alpha \cdot \hat{w}_i(t) + (1 - \alpha) \cdot w_i(t) \tag{1}
\]

As different \( \alpha \) might exhibit different results, we only report \( \alpha = 0.9 \) for comparison (details see next section).

Given a series of bandwidth measurements, \( w_i(t) \), where \( t = 0,1,2,\ldots,n-1 \) and \( w_i(t) > 0 \), the harmonic mean is \( \hat{w}_i(n) = n / \sum_{t=0}^{n-1} 1/w_i(t) \). A key factor of computing harmonic mean is to maintain a number of past measurements \( \lceil \log_2 B \rceil \). However, to reduce memory usage and computational cost, one can compute the current harmonic mean without maintaining all the previous states. Statistics from the past can be recovered simply by recording an additional parameter, \( n \), the total number of past measurements. The harmonic mean can be updated with the most recent measurement of path \( i \),

\[
\hat{w}_i(n+1) = \frac{n + 1}{\sum_{t=0}^{n} 1/w_i(t)} = \frac{n + 1}{\hat{w}_i(n) + w_i(n+1)} \tag{2}
\]

There are several benefits to estimating path bandwidth by harmonic mean. First, harmonic mean is an appropriate measure for average rates. Second, it tends to mitigate the impact of large outliers that might occur due to network variation \([10]\).

5.3 Evaluation of Chunk Scheduler

We examine scheduler performance with the following three estimators: Harmonic, EWMA, and Ratio (the baseline). We first examine the download time performance for various pre-buffering times (for 20/40/60 seconds). For each pre-buffering duration, we further inspect each scheduler’s performance with respect to different initial chunk sizes (from 16 KB to 1 MB). For all the measurements, we randomize the order of the configurations and repeat for 20 times over the course of 12 hours. Note that the performance fraction threshold \( \delta = 5\% \), and the EWMA weight \( \alpha = 0.9 \).

As shown in Fig. 3 for each pre-buffering period, the download time improves as chunk size increases. For small chunks sizes, MSPlayer requires more range requests to accumulate the same amount of video in the pre-buffering phase. For larger chunk sizes, fewer requests are made and hence less overhead is included in the download time.

The baseline scheduler does not perform well and exhibits higher variability as it fails to respond to bandwidth changes quickly. Dynamic chunk adjustment schedulers (EWMA and Harmonic), on the other hand, vary path chunk sizes according to estimated bandwidth and

\[
w_i(n), \text{ and the previous harmonic mean } \hat{w}_i(n). \text{ That is,}
\]

\[
\hat{w}_i(n+1) = \frac{n + 1}{\sum_{t=0}^{n} 1/w_i(t)} = \frac{n + 1}{\hat{w}_i(n) + w_i(n+1)} \tag{2}
\]

\[
\text{Figure 3: Download times of three schedulers: Harmonic/EWMA/Ratio (top to down order) for different pre-buffering periods (right Y-axis) and initial unit chunk sizes (left Y-axis).}
\]

\[
\text{Download Time (sec)}
\]

\[
\text{Harmonic}
\]

\[
\text{EWMA}
\]

\[
\text{Ratio}
\]

\[
\text{16KB}
\]

\[
\text{64KB}
\]

\[
\text{256KB}
\]

\[
\text{1MB}
\]

\[
\text{20 sec}
\]

\[
\text{40 sec}
\]

\[
\text{60 sec}
\]

\[
\text{Download Time (sec)}
\]

\[
\text{Harmonic}
\]

\[
\text{EWMA}
\]

\[
\text{Ratio}
\]

\[
\text{16KB}
\]

\[
\text{64KB}
\]

\[
\text{256KB}
\]

\[
\text{1MB}
\]

\[
\text{20 sec}
\]

\[
\text{40 sec}
\]

\[
\text{60 sec}
\]
hence exhibit better performance. More specifically, scheduler with harmonic mean estimator outperforms all the others in most cases as it is designed to mitigate large outliers such as large bursts. In our experiments, we use harmonic mean estimator as our default estimator in our scheduler. Since the performance of harmonic mean scheduler is similar with initial chunk size of 256 KB and 1 MB, we use a default initial chunk size of 256 KB as smaller chunk sizes are preferable to reduce network bursts [12].

6. EVALUATION ON YOUTUBE SERVICE

We evaluate MSPlayer performance over the YouTube video infrastructure by comparing the download time of MSPlayer and the streaming schemes of the commercial YouTube players in both the pre-buffering phase and the re-buffering phase. We first look at the pre-buffering phase (where commercial players accumulate video data of a specified amount as one large chunk) and check to see how MSPlayer can reduce the start-up latency. Fig.4 shows that MSPlayer outperforms both single-path TCP over WiFi and LTE for different specified amounts of pre-buffered video data. When comparing to the best single-path technology used, MSPlayer reduces the start-up delay by 12%, 21%, 28%, respectively, for 20/40/60 second buffers.

When MSPlayer enters the periodic re-buffering phase, we investigate how quickly MSPlayer can refill the playout buffer and compare that to the performance of other commercial players with HTTP range requests using default chunk sizes of 64 KB (Adobe Flash) and of 256 KB (HTML5) over single path WiFi and LTE [23]. Similarly, we also look at the different re-buffering sizes for 20/40/60 seconds.

Fig.5 presents the performance when streaming YouTube videos over single-path WiFi/LTE with HTTP byte ranges of sizes 64 KB and 256 KB for different re-buffering sizes. All the players can refill the playout buffer more quickly when large chunk sizes are used. This is mainly because more requests are required for smaller chunk sizes and hence introduces more overhead over time. MSPlayer, on the other hand, with its scheduler estimating network bandwidth efficiently and adjusts the chunk size accordingly and hence outperforms all the single-path schemes and can significantly reduce the time to refill the playout buffer.

7. CONCLUSION AND FUTURE WORKS

We propose a client-based multi-source and multi-path solution, MSPlayer, that streams videos from multiple YouTube video servers via two interfaces (WiFi and LTE) simultaneously. MSPlayer manages to reduce video start-up delay and can quickly refill the video playout buffer for just-in-time high quality video delivery. It requires no kernel modification at both the server and client side. Moreover, it provides robust data transport and does not suffer from middleboxes as is in MPTCP. However, due to time constraint, we do not report the results on how MSPlayer provides robustness for video delivery in mobile scenarios.

As for future works, our scheduler currently does not take into account budget and energy constraints for potential performance issues when leveraging multiple interfaces on mobile devices [17]. Also, we use a simple periodic downloading mechanism for playout re-buffering. A more careful investigation of periodic downloading and ON/OFF mechanisms will be explored. Last, as we take the initial step to demonstrate the possibility of leveraging multiple video sources with different interfaces/paths in a real video service network, we only focus on using a constant video bit-rate. As dynamic adaptive streaming over HTTP (DASH) [24] is now widely used, exploring how rate adaption can be integrated with MSPlayer and how MSPlayer can be used for other streaming services are also our future works.
Acknowledgments

This research was sponsored by US Army Research laboratory and the UK Ministry of Defense under Agreement Number W911NF-06-3-0001. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the US Army Research Laboratory, the U.S. Government, the UK Ministry of Defense, or the UK Government. The US and UK Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

8. REFERENCES


