MSPlayer: Multi-Source and Multi-Path Video Streaming
Yung-Chih Chen, Don Towsley Fellow, IEEE, and Ramin Khalili

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% over the past two years. Moreover, people are constantly seeking to stream high quality videos for better experience while often suffering from limited bandwidth. With the rapid deployment of content delivery networks (CDNs), popular videos are now replicated at different sites, and users can stream over-the-top videos from close-by sources with low latency. Aggregating bandwidth for high definition video streaming has become possible as mobile devices nowadays are equipped with multiple wireless interfaces (e.g., WiFi and 3G/4G). We propose a client-based video streaming solution, MSPlayer, that takes advantage of multiple video sources and leverages multiple network paths through different interfaces. MSPlayer reduces start-up latency and provides robust data transport with high video quality in mobile scenarios. We experimentally demonstrate our solution on a testbed and through the YouTube video service.

Index Terms—Video Streaming; HTTP; MPTCP; CDN; LTE; YouTube

1 INTRODUCTION
With the high demand for online video streaming, video content providers are offering better technology to satisfy customers’ desires for streaming high quality videos. However, streaming video to a user nowadays still encounters the following challenges. First, people from time to time experience insufficient bandwidth when streaming videos. Research has shown that viewers are not patient enough to wait if the start-up delay is longer than a few seconds [23] and measurements from [24] also confirmed that users very often suffer video re-buffering or more than five seconds start-up delay. As a result, users tend to drop videos if they frequently stop, freeze, or experience quality changes during the playout [10]. Also, connections to a particular network can break down temporarily due to mobility and re-establishing a connection introduces additional delays. Last, as network bandwidth is highly variable, commercial video players have experimented with video rate adaption, which in turn can result in unstable performance such as variable video quality, unfairness to other players, and low bandwidth utilization [5], [6], [19], [20], [25].

As mobile devices are now equipped with multiple wireless interfaces connected to different networks (WiFi or cellular 3G/4G), one possible solution to the above challenges is to use multi-path TCP (MPTCP) [12]. However, since MPTCP requires kernel modifications at both the client and server sides [27], and many network operators do not allow MPTCP traffic to pass their middleboxes [17], [18], MPTCP has been slow to deploy globally. Furthermore, although MPTCP provides a means for balancing loads over different paths to a single server, it does not utilize source diversity present in networks to facilitate content availability. Therefore, there is a potential to develop a solution to stream high quality videos to end users without overloading the video servers and to better utilize network resources.

In this paper, we ask the following question: Is it possible for a mobile device to leverage both path diversity and source diversity to provide robust video delivery and to reduce video start-up latency?

We take a first step to answering this question. We show that one can utilize both of the available WiFi and 4G interfaces simultaneously on a mobile device to aggregate bandwidth for higher quality video streaming. This video streaming solution does not require modifications in the kernel stacks and is not hindered by network middleboxes. Moreover, it does not require changes at the server side or in the network and is more suitable than MPTCP for

This research was partially sponsored by US Army Research Laboratory and the UK Ministry of Defense under Agreement Number W911NF-06-3-0001. This material is also supported by the National Science Foundation under Grants No. CNS-1040781 and CNS-1413998.
video streaming over today’s Internet. We instantiate these designs in our video player, MSPlayer. By investigating the streaming mechanisms of YouTube, we further show how to simultaneously leverage the existence of multiple video sources in different networks. We then experimentally evaluate the performance of different MSPlayer schedulers as well as the performance of MSPlayer through the YouTube service.

The remainder of this paper is organized as follows: Sec. 2 introduces the design principles of MSPlayer. We overview the architecture of MSPlayer in Sec. 3. Implementation details of MSPlayer are presented in Sec. 4 and Sec. 5. We evaluate MSPlayer’s performance in our testbed and over YouTube infrastructure in Sec. 6 and 7. We discuss future work and conclude this paper in Sec. 8.

2 DESIGN PRINCIPLES

In this paper, we present MSPlayer, a client-based approach for video streaming that requires no changes in either the server or the client’s kernel stacks. 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.

Just-in-time with High Quality: Since viewers often prematurely stop watching videos [11], [23], streaming the entire video to a viewer at once can waste bandwidth and network resources. This, along with the rise of adaptive streaming over HTTP [30], has drawn attention to just-in-time video delivery, which has been exploited by most large scale video streaming services such as YouTube and Netflix. A video is partitioned into many small file segments called video chunks. The video server maintains multiple profiles of a video for different bitrates and quality levels. Clients then periodically request video chunks and adapt video bitrates.

Just-in-time video delivery avoids a waste of resources if a user drops the video during its playback. Dynamically adapting video bitrates, however, can result in performance problems such as low link utilization [5], unfairness to other TCP users [5], [19], and unstable video quality [6], [25]. In our design, we share the just-in-time concept for video delivery. However, we do not investigate rate adaption and instead focus on how to stream videos with a fixed bitrate by exploiting network diversity.

Robust Data Transport: When a mobile user streams a video, his connection (typically WiFi) can break and the downloaded video will be abandoned. In order to resume the video, the user then needs to switch to another available interface connected to another network, establish a new TCP connection, and move/skip to the break point. In the worst case, one has to wait until reaching the next WiFi hotspot and repeat the entire process mentioned above.

One possible solution is to use Multi-Path TCP (MPTCP) [12], which has been standardized by the IETF, aimed at providing robust data transport. However, MPTCP still faces several deployment challenges: First, MPTCP requires kernel modifications at both the server and client [12]. Second, it relies on the TCP option field to exchange path and interface information. In the latter case, research has shown that MPTCP suffers significantly from network middleboxes as they very often strip away unknown options [17], [18], forcing MPTCP connections to fall back to legacy single-path TCP. In our measurements, all three major US cellular carriers we examined do not allow MPTCP traffic to pass through the default HTTP 80 port (two of them supported it until early 2014), which is a potential problem for video streaming to popular sites.

There are several other proposals that aim to utilize multiple wireless interfaces simultaneously. OpenFlow Wireless [31] proposes an architecture to separate control plane from data plane for bandwidth aggregation, but it requires existing network platforms to be virtualized. MicroCast [21], on the other hand, requires no changes to existing infrastructure but needs a group of close-by users to form a peer-to-peer network to enable data and bandwidth sharing. Similarly, Netup [29] claims to speed
up mobile Internet, but it relies on a VPN architecture that requires both WiFi and LTE interfaces to connect to a proxy. This VPN-based proxy solution might not scale well and the user could eventually retrieve video content from a farther server that is only close to the proxy [3]. As WiFi/cellular networks exhibit very different characteristics [8], having such a proxy can introduce an unnecessary delay and vulnerability to the underlying data transport.

We design a client-based multi-path solution to provide robust data transport for high quality video streaming. Furthermore, each path runs legacy TCP and is therefore guaranteed to successfully work with network middleboxes and the modern Internet.

**Content Source Diversity:** Current MPTCP [12] 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 server demand surges. This high demand, particularly for high quality videos, can overload the server and congest shared bottleneck links and 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 [24] for large scale video streaming services such as YouTube, Netflix, and Hulu [1], [2]. As part of our design is 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 [16], 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 among 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 all networks simultaneously to avoid overwhelming particular video servers and to balance the load across the servers.

**Chunk Scheduler:** MSPlayer relies on HTTP 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 latencies [8], efficient scheduling of chunks over different paths is challenging. Therefore, it is desirable to have a good scheduler that estimates path quality over time and efficiently assigns chunks to each path.

To satisfy just-in-time video delivery, the scheduler pauses chunk retrieval when the playout buffer is full and resumes chunk retrieval when the amount of buffered video falls below a certain level (that is referred to as periodic downloading or ON/OFF cycles [28]). To reduce memory usage of out-of-order chunks from different paths, the MSPlayer scheduler attempts to complete the transfer of a chunk over each path at the same time, and allows at most one out-of-order chunk to be stored.

### 3 MSPLAYER OVERVIEW

We now overview the MSPlayer architecture. Fig. 1 illustrates standard operating procedures of streaming YouTube videos with MSPlayer. When a video is requested by a user, MSPlayer first conducts a domain name system (DNS) lookup for the requested video URL. Afterwards, MSPlayer contacts the YouTube portal (referred to as web proxy server) based on the resolved IP address over a network and waits for an access token from that path. The access token contains detailed information of the server that hosts the video (referred to as video server) in that network. Upon receiving access tokens from different paths, MSPlayer contacts different video servers and fetches partial content of the requested video over each path. In the following sections, we describe in detail how YouTube video streaming works, the mechanism of just-in-time video delivery, and each of MSPlayer’s design components: multi-source, multi-path, and chunk scheduler.

#### 3.1 YouTube Video Streaming

When users want to watch a video from YouTube, they either go to the YouTube website and choose a video to watch, or they click an URL on a web page of the following form \(http://www.youtube.com/watch?v=qjT4T2gU9sM\). Each YouTube video is identified by an 11-literal video ID after \(watch?v=\) in the URL [2]. With this URL, users then watch the video through their browsers using a built-in Adobe Flash player [4] or HTML5 player [32]. The video player first performs a DNS lookup to resolve the IP address of the domain name www.youtube.com and the user’s video request is directed to one of YouTube web proxy servers. The YouTube web proxy server processes the request and returns the
related video information and a new URL to the user in JavaScript Object Notation (JSON) format, indicating where the associated and available YouTube video servers are. The player then establishes another connection to one of the dedicated video servers and starts to stream the YouTube video using HTTP range requests.

The streaming process starts with a **prefetch phase** followed by a periodic re-buffering phase [28]. The prefetch phase takes place at the beginning of the video streaming session and aims at retrieving enough video data into the playout buffer for the initial video playout. After the player consumes the video content for a while and the amount of video in the playout buffer falls below a certain level, the player enters the re-buffering phase, and makes new requests periodically to refill the playout buffer. This periodic re-buffering repeats until the video is completely watched or dropped.

### 3.2 Multi-Source and Multi-Path

Before describing our scheme of streaming videos with multiple sources and multiple paths, we first describe how each path establishes a connection to the YouTube web proxy server and the associated video server. Fig. 2 illustrates a flow diagram when a user contacts YouTube’s web proxy server to retrieve video information. The connection starts with a TCP 3-way handshake (3WHS). Afterwards, the client initiates a secure connection handshake message at time $t_1$. It takes the server times $\Delta_1$ and $\Delta_2$ to verify the key and complete the key exchange process. The first HTTP request is made at time $t_3$, and the first JSON packet from the web proxy server arrives at $t_4$. Note that these JSON packets are delivered within two round trips (slightly less than 20 packets), and the secure connection ends at $t_5$ followed by a TCP FIN. If we denote by $R_0$ and $R_1$ the RTTs of the first and the second paths (assuming $R_0 \leq R_1$, i.e., the first path is a fast path), it takes time $\eta_i = 4R_i + \Delta_1 + \Delta_2$ to establish a secure connection and receive the first response to the HTTP request over path $i$ (from $t_0$ to $t_4$), and time $\psi_i = 6R_i + \Delta_1 + \Delta_2$ to receive all the JSON packets related to video information (from $t_0$ to $t_5$).

If the YouTube web proxy server is close to the video server, and both servers have similar capabilities for key verification, it takes approximately $\pi_i \approx \psi_i + \eta_i$ seconds for path $i$ to get all the JSON packets from the web proxy server, establish a secure connection to the video server, and eventually receive the first HTTP response from the video server.

If MSPlayer only starts to fetch video content from both paths when both required tokens are received, MSPlayer needs to delay the data transport over the first path until the second path is ready. That is, before the second path starts to retrieve video packets from its associated video server, the first path will be idle for a duration of $\pi_1 - \pi_0 \approx 10 \cdot (R_1 - R_0)$.

Therefore, in MSPlayer, the processes of fetching video chunks over each path are executed by independent threads under the management of the chunk scheduler (described in the next section). When fetching video content from YouTube, MSPlayer contacts a video server over one path as soon as the IP address of the video server associated with that path is decoded, and does not wait for the decoding process over other paths to finish. This mechanism reduces unnecessary video start-up delay and allows the first path to contribute more in the prefetch phase (details in Sec. 7).

### 3.3 Chunk Scheduler

In order to reduce memory used for out-of-order chunks and their wait time in receive buffer for the late video chunk, our goal is to schedule chunk transfers over different paths to complete at roughly

---
1. As of July 2014, YouTube has applied algorithms to encode copyrighted video signatures. As these signatures are needed to contact the video servers, for copyrighted videos, an additional operation is required in order to fetch the video web page containing a decoder to decipher the video signature.
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 dynamics.

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 [28]. Since different mobile devices have prefetch periods of different lengths (ranging from 20 seconds to 1 minute) [28], we also investigate the performance of different schedulers (namely Ratio, EWMA, and Harmonic as described below) when applying different chunk sizes and prefetch periods.

We denote by $S_i(t)$ the chunk size of path $i$ at time $t$ ($i = 0, 1$ labels the first and the second path), 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 $\hat{w}_i(t) = S_i(t)/T_i$.

We first showcase a baseline scheduler called Ratio and then propose two different chunk size schedulers that adjust chunk sizes according to network bandwidth changes, namely the exponential weighted moving average (EWMA) and the harmonic mean (Harmonic). MSPlayer chunk size selection should adapt to path quality variations over time, and the bandwidth estimator of MSPlayer thus plays a critical role in the chunk size selection process. In this paper, we label the chunk scheduler according to the bandwidth estimator used. We compare and evaluate the performance of these three schedulers in our testbed.

Baseline Scheduler: Suppose $w_i(t) \leq w_{1-i}(t)$, 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., $S_{1-i}(t+1) = w_{1-i}(t)/w_i(t) \cdot B$).

Dynamic Chunk Adjustment Scheduler:

When path bandwidth estimates are available, the chunk size of each path is adjusted according to Algorithm 1. We denote by $\delta$ the throughput variation parameter. If the current bandwidth measurement of the slow path is $(1 + \delta)\hat{w}_i$ times larger than the estimated value, then the chunk size is halved. Similarly, if the current value is $(1 - \delta)$ times smaller than the estimated value, the chunk size is halved.

\begin{algorithm}
\caption{Dynamic chunk size adjustment}
\begin{algorithmic}[1]
\Procedure{DCSA}{$i, \hat{w}_0, \hat{w}_1, \delta, B$}
\label{alg:DCSA}
\If{$\hat{w}_i$ not available then} \Comment{i = 0,1}
\State $S_i \gets B$ \Comment{initial chunk size}
\ElsIf{$\hat{w}_i < \hat{w}_{1-i}$ then} \Comment{slow path}
\State $\hat{w}_i \gets 2 \times \hat{w}_i$
\ElseIf{$\hat{w}_i < (1-\delta) w_i$ then}
\State $S_i \gets \max \{\lfloor S_i/2 \rfloor, \text{16KB}\}$
\Else
\State $S_i$ unchanged
\EndIf
\EndIf
\Else \Comment{fast path}
\State $\gamma = \lceil \hat{w}_i / \hat{w}_{1-i} \rceil$
\State $S_i \gets \gamma \times S_{1-i}$
\EndIf
\State return $S_i$ \Comment{final chunk size}
\EndProcedure
\end{algorithmic}
\end{algorithm}

```
\begin{equation}
\hat{w}_i(t+1) = \alpha \cdot \hat{w}_i(t) + (1 - \alpha) \cdot w_i(t).
\end{equation}
```

EWMA smooths out the fluctuation of the bandwidth measurements and the weight is given to the latest data.

However, since network bandwidth can vary quickly, extreme measurement values can bias the EWMA bandwidth estimation. Hence, we introduce another bandwidth estimator called harmonic mean.

The benefit to estimating path bandwidth by harmonic mean is that it tends to mitigate the impact of large outliers due to network variations [20].

Given a series of bandwidth measurements, $w_i(t)$, where $t = 0, 1, 2, \cdots, n - 1$ and $w_i(t) > 0$, the harmonic mean is

\begin{equation}
\hat{w}_i(n) = \frac{n}{n \sum_{t=0}^{n-1} \frac{1}{w_i(t)}}.
\end{equation}

The harmonic mean can be computed by maintaining all or a window of the most recent measurements [20]. However, to reduce memory usage and computational cost, one can compute the current harmonic mean without maintaining all previous observations. 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 $i$, $w_i(n)$, and the previous harmonic mean $\hat{w}_i(n)$. That is,

\begin{equation}
\hat{w}_i(n+1) = \frac{n+1}{\sum_{t=0}^{n} \frac{1}{w_i(t)}} = \frac{n+1}{\frac{n}{\hat{w}_i(n)} + \frac{1}{w_i(n+1)}}.
\end{equation}

In the following sections, we present MSPlayer implementation details and evaluate MSPlayer’s performance.
4 MSPlayer Implementation

In order to exploit both available wireless interfaces simultaneously, we pass additional interface information to the socket API to bind each interface to an IP address and packets can thus be scheduled to a desired interface. Moreover, we configure an independent routing table for each interface 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 [14], MSPlayer leverages source and path diversity in the network for YouTube video streaming by interacting with Google APIs through Python.

As depicted in Fig. 1, 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 [14]. When the requested operations are granted, the web proxy server resolves the user’s public IP address and checks to see which video server is suitable and available to this user [3]. Afterwards, the web proxy server generates an access token (valid for an hour) that matches the video server’s IP address and 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, video server domain names, etc) in JavaScript Object Notation (JSON) format and returns these objects to the user through the requested interface. Upon receiving the data, MSPlayer 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 can now be retrieved by making HTTP range requests to different video servers using persistent connections through both interfaces. The chunk scheduler in MSPlayer then coordinates the thread process of each path, monitors path quality for chunk assignments, and keeps track of the allocated byte ranges. When the retrieved video chunks are in order, the scheduler pushes the video data to the application for playback. Note that YouTube has been gradually replacing insecure HTTP connections with secure ones. To be compatible to YouTube’s data service, we follow YouTube’s latest HTTPs connection policy with both web proxy and video servers.

As part of the just-in-time video delivery principle, MSPlayer uses the following streaming strategy similar to commercial YouTube players such as Adobe Flash player or HTML5: a prefetch phase followed by a steady-state re-buffering phase [28]. MSPlayer leaves the prefetch phase when more than 40-second video data is received. It then consumes the video data until the playout buffer contains less than 10-second video. MSPlayer resumes requesting chunks from both video servers and refills the playout buffer until 20-second video data is retrieved.

5 DNS selection

As MSPlayer sends requests to retrieve video content over interfaces associated with different networks, it is important to understand the impact of using different DNS services on its performance. In this section, we seek to understand these performance differences. This is critical for MSPlayer as how DNS responds to hostname resolution determines which video sources are selected and this might in turn influence MSPlayer’s performance.

To understand these performance differences, we examined MSPlayer’s performance when streaming YouTube’s most viewed 100 videos of all time [33] (as of September 1, 2014) from four major US cellular/wireless carriers (i.e., cellular Verizon, Sprint, AT&T, and Comcast WiFi networks) while using three different DNS services (i.e., local DNS of each carrier, Google public DNS [15], and Open DNS [26]). We configured MSPlayer to query one particular DNS and stream one video over a network at a time, and iterated through all the configuration combinations. When conducting the experiments, we randomized the order of wireless carriers, DNS services, and the video IDs for each configuration and repeated the same experiments in the morning, afternoon, and evening of a day. Throughout this
paper, we use box and whisker plots to summarize our measurement results. The line inside each box is the median, the left/right sides of each box are the first/third quartile (25% and 75%), and the ends of the whiskers are the min/max values.

For each configuration, when an access token is delivered from the YouTube web proxy server in a network, we collected the URL of the video server (as described in Sec. 4) and resolved its IP address with the same DNS. Fig. 3 demonstrates how much time MSPlayer takes to fetch video information and the access token from a YouTube web proxy server when using different types of DNS servers (corresponding to steps 2 and 3 in Fig. 1). Although in some cases fetching the same video information from the IP addresses resolved by the local DNS servers is slightly faster, there are no significant differences in the download times among the three DNS services.

We further examined the URLs of the video servers returned from different web proxy servers. We resolved the video server hostnames to IP addresses using the same DNS server and check to see if different DNS services might return very different answers (corresponding to step 4 in Fig. 1). Fig. 4 presents a distribution of video server domains (of /24 subnets) where MSPlayer actually streams the top 100 videos from when using different DNS services over different wireless technologies.

From our observations, using different DNS services to stream YouTube videos does not make much difference from a carrier’s perspective. When making requests over a network, MSPlayer streams the videos from certain network domains regardless of the DNS used. We hypothesize that YouTube aims to optimize its video server allocation to users based on their network affiliation rather than the DNS used. This can easily be achieved by the web proxy server since resolving a user’s public IP address is part of its task of resource allocation. Therefore, in the following evaluations, we use Google’s public DNS as MSPlayer’s default DNS.

### 6 Testbed Experimentation

We first evaluate the performance of each component of MSPlayer on a testbed in a controlled environment that emulates YouTube’s video streaming mechanisms. The final performance evaluation is carried out on the YouTube infrastructure and service (see Sec. 7). Two types of servers are emulated in our testbed: web proxy servers (responsible for authentication and video object information delivery) and video servers. Both types of servers use the standard Linux 3.5 kernel with CUBIC congestion control [13] coupled with Apache service. Each type of server is hosted in two different UMass subnets for source diversity.

The client running MSPlayer is a Lenovo X220 laptop equipped with a built-in 802.11 a/b/g/n WiFi interface connecting to a home WiFi network and an LTE dongle connecting to one of the major US cellular carriers. Video requests are sent over both interfaces simultaneously to two different YouTube 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 in our testbed) and fetches video chunks from the video servers. In our testbed, the videos are pre-downloaded in the servers from YouTube with MP4 format of HD (720p) video quality and 44,100 Hz audio quality.

#### 6.1 Multi-Source and Multi-Path

Fig. 5 demonstrates the initial video prefetching download time using single-path WiFi, single-path
LTE, and MSPlayer for HD videos in our emulated testbed. Since the content needed for this prefetch period usually involves more than one video chunk, when the slower path becomes ready for data transport, MSPlayer leverages both paths to further reduce video start-up delay. Note that a 40-sec prefetch period is presented here as this is YouTube servers’ default prefetch size for Flash videos [28]. The median download time of MSPlayer is 6.9 seconds while that of the best single-path over WiFi is 10.9 seconds, a 37% delay reduction in the prefetch phase.

As MSPlayer leverages multiple video sources and interfaces/paths, packet scheduling across each path to each server can significantly affect performance. The MSPlayer results in Fig. 5 are based on the Ratio scheduler with an initial chunk size of 1 MB. Next, we investigate different MSPlayer schedulers and evaluate their performance.

6.2 Chunk Scheduler

We examine the performance of the following three schedulers: Harmonic, EWMA, and Ratio (the baseline). We first examine download times for different prefetch durations (for 20/40/60 seconds). For each prefetch duration, we further inspect each scheduler’s performance with respect to different initial chunk sizes (from 16 KB to 1 MB). Throughout the experiments, we randomize the order in which the configurations are tested and repeat the same experiments 20 times over the course of 12 hours. We use the throughput variation parameter $\delta = 5\%$ and EWMA weight $\alpha = 0.9$.

As shown in Fig. 6, for each prefetch duration, download time decreases as chunk size increases. For small chunk sizes, MSPlayer requires more range requests to accumulate the same amount of video in the prefetch phase. For larger chunk sizes, fewer requests are made and hence less overhead is included to retrieve the same amount of video.

The baseline scheduler does not perform well and exhibits greater variability as it fails to quickly respond to bandwidth changes. Dynamic chunk size adjustment schedulers (EWMA and Harmonic), on the other hand, vary path chunk sizes according to estimated bandwidth and exhibit better performance. More specifically, the scheduler using the harmonic mean estimator outperforms the others in most cases as this estimator mitigates outliers such as large bursts. In our experiments, we use the harmonic mean estimator as the default. Since the performance of the harmonic mean scheduler is similar for chunk sizes 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 [13].

7 Evaluation over YouTube System

We evaluate MSPlayer performance over the YouTube video infrastructure by comparing the download times of MSPlayer and the commercial YouTube player settings during both the prefetch phase and the re-buffering phase. We first focus on the prefetch phase (where commercial players accumulate a specified amount of video data of as one large chunk) and check on how MSPlayer reduces start-up latency. Fig. 7 shows that MSPlayer outperforms both single-path TCP over WiFi and LTE for different specified amounts of prefetched video. In comparison to the best single-path technology used, MSPlayer reduces start-up delay by 12%, 21%, 28% for 20, 40, and 60 seconds of prefetched video, respectively. When MSPlayer enters the re-buffering phase, we investigate how quickly it refills the play-out buffer and compare its performance to that of
other commercial players using default chunk sizes of 64 KB (Adobe Flash) and of 256 KB (HTML5) over single path WiFi and LTE [28]. Similarly, we look at re-buffering sizes for 20, 40, and 60 seconds video.

Fig. 8 presents download times when streaming YouTube videos over single-path WiFi/LTE with HTTP byte ranges of sizes 64/256 KB for different re-buffering sizes. All of the players refill the playout buffer quickly when using larger chunks. This is because more requests are required for smaller chunks and introduces more overhead. MSPlayer, on the other hand, efficiently estimates network bandwidth and adjusts the chunk size accordingly. It outperforms the single-path schemes and significantly reduces the time to refill the playout buffer.

In order to understand how the MSPlayer chunk scheduler distributes traffic over paths, we investigate the fraction of traffic carried by each path. Table 1 lists the fraction of traffic carried by WiFi for both of the prefetch and re-buffering phases with an initial chunk size of 256 KB. We observe that the WiFi path on average carries more than 60% of traffic in the prefetch phase. This is mainly due to the fact that our design allows the fast path to start fetching video chunks as soon as it decodes necessary information from YouTube’s web proxy server.

Although we do not have control of the YouTube servers to accurately keep track of packet RTTs over each path, we seek to understand how quickly MSPlayer utilize the first available path during the prefetch phase by investigating the time difference of the first chunk arrival times of each path. We observe that the first chunk over the first path arrives at the client at an average time of 254±31 ms (150±16 ms) prior to the arrival of the first chunk over the second path when the initial chunk size is 256 KB (64 KB).

For the case of prefetching 20-second video content, this asynchronized initialization of chunk retrieval processes saves a period of 254 ms that represents a 7.54% reduction in video start-up latency.

When entering the re-buffering phase, the WiFi path slightly dominates packet delivery (as shown in Table 1). This is because each path needs to wait for one RTT before receiving the first packet from the associated video source for each range request. As WiFi exhibits much smaller RTTs in our experiments, the WiFi path saves a time of length \( R_1 - R_0 \) for each range request and introduces less overhead when compared with the LTE path in the re-buffering phase. Note that in some occasions (e.g., 60-second re-buffering in Fig. 8), when the very last chunk is assigned to LTE with a relatively large chunk size (8 or 16 MB), packet delivery over LTE can suffer from cellular network’s excess packet buffering and result in significant high latency (up to seconds) and large performance variation [9]. Since this issue of delivering large last chunk over cellular network is more related to cellular network characteristics, we leave this for future work.

### 8 Conclusion and Future Work

We proposed a client-based video streaming 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 does not require kernel modifications at either the server or the client side. Moreover, it provides robust data transport and does not suffer from middleboxes in the networks as does MPTCP.

So far we have taken an initial step to demonstrate the possibility of leveraging multiple video sources with different interfaces/paths in a real video service network using a constant video bitrate. As dynamic adaptive streaming over HTTP (DASH) [30] is now widely used, exploring how rate adaption and efficient chunk encoding can be integrated with MSPlayer, and how MSPlayer can be used for other streaming services are future works.

We also plan to extend MSPlayer to serve more general HTTP and web traffic [22] over today’s Internet.
REFERENCES


[26] openDNS. https://www.opendns.com/


[33] YouTube top 100 videos. https://www.youtube.com/playlist?list=PLirAqAtl_h2r5g8xGajEwxdXd3x1szHshC

Yung-Chih Chen received a B.Sc. degree in computer science from National Tsing Hua University, Taiwan and a Ph.D. degree in computer science from the University of Massachusetts at Amherst in 2004 and 2015, respectively. During 2006-2007, he worked as a research assistant at the Institute of Information Science of Academia Sinica, Taiwan. He is currently associated with Akamai Technologies.

Don Towsley (M’78 – SM’93 – F’95) received a B.A. degree in physics and a Ph.D. degree in computer science from the University of Texas at Austin in 1971 and 1975, respectively. He is currently a Distinguished University Professor in the Department of Computer Science at the University of Massachusetts at Amherst. He has twice received IBM Faculty Fellowship Awards, and is a Fellow of the IEEE and the ACM.

Ramin Khalili received a B.Sc. degree in electrical engineering from University of Shiraz and a M.Sc. degree in Telecommunications from Sharif University of Technology, Iran in 1999 and 2001, respectively. He received his Ph.D. degree in Computer Networks and Distributed Systems from LIP6, France in 2006. He was associated with UMass Amherst, FPFL, Tu-Berlin, and is now with Huawei Research Munich.