Investigating Redundant Internet Video Streaming Traffic on iOS Devices: Causes and Solutions

Yao Liu, Qi Wei, Lei Guo, Bo Shen, Songqing Chen, and Yingjie Lan

***Abstract*—The Internet has witnessed rapidly increasing streaming traffic to various mobile devices. In this paper, through analysis of a server-side workload and experiments in a controlled lab environment, we find that current practice has introduced a significant amount of redundant traffic. In particular, for the popular iOS based mobile devices, accessing popular Internet streaming services typically involves about 10% - 70% redundant traffic. Such a practice not only over-utilizes and wastes resources on the server side and the network (cellular or Internet), but also consumes additional battery power on user's mobile devices and leads to possible monetary cost. To alleviate such a situation without changing the server side or the client side, we design and implement CStreamer that can transparently work between existing mobile clients and servers. We have implemented a prototype and installed on Amazon EC2. Experiments conducted based on this prototype show that CStreamer can completely eliminate the redundant traffic without degrading user's QoS.**

***Index Terms*—Internet Mobile Streaming, iOS, redundant traffic**

I. Introduction

Today mobile devices, such as iPhone and iPad, are becoming more and more popular. As of June 2013, iOS holds 57% market share of mobile devices including smartphones and tablets [1]. Besides the general web surfing on the Internet, these days more and more accesses from mobile devices are directed to all kinds of Internet streaming services. For example, popular video sharing websites such as YouTube [2], Dailymotion [3], and Veoh [4], and service providers such as Hulu [5] and Netflix [6], all allow mobile users to access their services via mobile browsers and/or applications. As a result, today mobile video traffic dominates the Internet mobile traffic. According to Cisco's report [7], mobile video traffic accounts for about 51% of total mobile data traffic in 2012, and is predicted to exceed two thirds by 2017. It has been found that 80% of the mobile video accesses take place on iOS devices [8].

Y. Liu is with the Department of Computer Science, Binghamton University, State University of New York. E-mail: yaoliu@cs.binghamton.edu.

Q. Wei is with the Department of Bioengineering, George Mason University. E-mail: qwei2@gmu.edu.

L. Guo is with the Department of Computer Science and Engineering, The Ohio State University. E-mail: lguo@cse.ohio-state.edu.

B. Shen is with Vuclip, XinLab. Inc.. E-mail: bshen@vuclip.com.

S. Chen is with the Department of Computer Science, George Mason Unviersity. E-mail: sqchen@cs.gmu.edu.

Y. Lan is with the Guanghua School of Management, Peking University. E-mail: ylan@gsm.pku.edu.cn.

However, compared to streaming services to traditional desktops, Internet streaming to mobile devices is still challenging. Mobile devices typically use wireless connections. Be it WiFi or 3G/4G, typically, the capacity of a wireless connection still could not keep up with its wired counterpart, while streaming applications often involve bulk data transmission in a continuous fashion. This constrains the video quality (e.g., video encoding rate bits/second) that could be effectively delivered to mobile users. In addition to the quality, this could cause additional monetary cost to mobile users if cellular network connections have been used, because today the cellular data plan usually uses a tiered billing model, e.g., [9]. Watching videos online can not only quickly generate a significant amount of traffic, but also result in the usage tier to be reached sooner, and extra monetary cost for subsequent data usage.

Moreover, the limited battery supply is still the Achilles' heel of any mobile device. Receiving, decoding, and displaying a bulk amount of streaming data inevitably depletes the limited battery power supply at a fast pace [10].

Therefore, for mobile devices and mobile users, it is very important that the streaming data should be delivered in a precise fashion without unnecessary traffic or extra monetary cost. However, in this paper, through server-side workload analysis and client-side measurements and experiments in a controlled lab environment, we find that current Internet mobile streaming practices introduce a significant amount of redundant traffic. In particular, for the popular iOS based mobile devices, accessing streaming services typically involves about 10% to 70% redundant traffic if a user watches the requested video from the beginning to the end. That is, such redundant traffic is not due to the early termination of the client access. Through experiments and analysis, we further investigate why such a significant amount of redundant traffic is transmitted. Our results show that: (1) to improve user's experience of potentially re-watching the video, the MediaPlayer on iOS devices constantly re-downloads the beginning part of the video again after finishing downloading the entire file; (2) when the downloading speed is fast, the MediaPlayer frequently aborts the HTTP connection and then sends the request again, causing data in flow to be wasted; and (3) when the downloading speed is slow, the MediaPlayer continuously sends additional and overlapping requests to smooth the playback.

Such a significant amount of redundant traffic not only wastes network bandwidth, but also over-utilizes server-side resources. A streaming server is often short of bandwidth and processing power today due to the rapid increase of video files

and requests [11]. Moreover, even if such redundant traffic is for the sake of user's perceived streaming performance, it is detrimental to the mobile device's interest (in terms of battery power consumption) and the mobile user's interest (in terms of potentially extra monetary cost).

Motivated by our measurement results, we examine the potential causes for such un-necessary traffic in normal mobile streaming accesses. We find these problems are mainly due to the limited available memory and the too fast/slow network connections. These findings motivate us to seek effective solutions to alleviate and minimize such redundant traffic without modifying the server side or the client side. For this purpose, we design and implement CStreamer that can transparently work between the client and the server. CStreamer partitions the video content into small segments. To eliminate the re-downloaded traffic, CStreamer synchronizes the MediaPlayer's downloading with the playback progress. To refrain from sending too fast, it serves the segments periodically, instead of all at once. To deal with slow connections, CStreamer allows the MediaPlayer to seamlessly switch to a lower quality version of the same video provided by the server during playback.

To evaluate the effectiveness of CStreamer in minimizing the redundant traffic, we have implemented a prototype of CStreamer and have deployed it on Amazon EC2. Different iOS devices are instructed to access various streaming services via this prototype. Our experimental results show that CStreamer can completely eliminate the redundant traffic without affecting user's perceived streaming experience. In summary, this paper makes the following contributions:

- Through server-side workload analysis and client-side measurements, we find that the current Internet streaming services to iOS mobile devices often generate 10% to 70% redundant traffic that is detrimental to the server (for delivery), the network (for transmission), the mobile device (for battery consumption), and the mobile user (for money).
- Conducting experiments in controlled environments, we investigate the potential causes of such redundant traffic. We find it is mainly attributed to the limited available memory and too fast or too slow network connections.
- Motivated by our findings, we design and implement CStreamer that transparently works between the client and the server. We evaluate our CStreamer prototype with various popular Internet streaming services, and show that CStreamer can completely eliminate redundant traffic without degrading the user's QoS.

The rest of the paper is organized as follows. We present some background about HTTP range requests and iOS streaming in Section II. Both the server-side and client-side measurements are presented in Section III. We present our analysis in Section IV. Our design and implementation of CStreamer is presented in Section V and we evaluate its performance in Section VI. Some related work is presented in Section VII and we make concluding remarks in Section VIII.

II. HTTP Range Request and Streaming to iOS

Among the popular mobile devices, iOS based devices are leading the market [1]. According to Freewheel, 80% of wireless video views take place on iOS devices [8]. iOS supports two streaming protocols: Pseudo Streaming [12] and HTTP Live Streaming (HLS) [13]. Pseudo Streaming today carries more mobile traffic than HLS, as it is often used by video streaming services like YouTube [2] and DailyMotion [3], and YouTube alone contributes 27% of mobile traffic in North America [14].

With Pseudo Streaming, the client can download the media content from an HTTP server. The playback can start before the entire file is downloaded. For this reason, Pseudo Streaming is also referred to as progressive downloading. In order to support VCR-like control, such as fast forward and rewind, the client can also use HTTP range requests to request part of the video file. An HTTP range request, or range request in short, is an HTTP request with ranges specified in the header of the request, indicating the desired data range of the requested file. The server only needs to respond with that part of file instead of the entire file. However, the entire file can be requested with the range specified from 0 to filesize-1.

The iOS MediaPlayer identifies itself with the user agents (e.g., `AppleCoreMedia/1.0.0`). Upon receiving a Pseudo Streaming request, it first checks the cache on its storage (i.e., flash memory). If it could not find the requested video in the cache, it asks the server for meta-data information about the video file, including file size, last modified time, etc. This is achieved by sending out an HTTP GET request specifying a range of `0-1`. Then, the MediaPlayer sends multiple HTTP requests for the file, and specifies a range to download in each request.

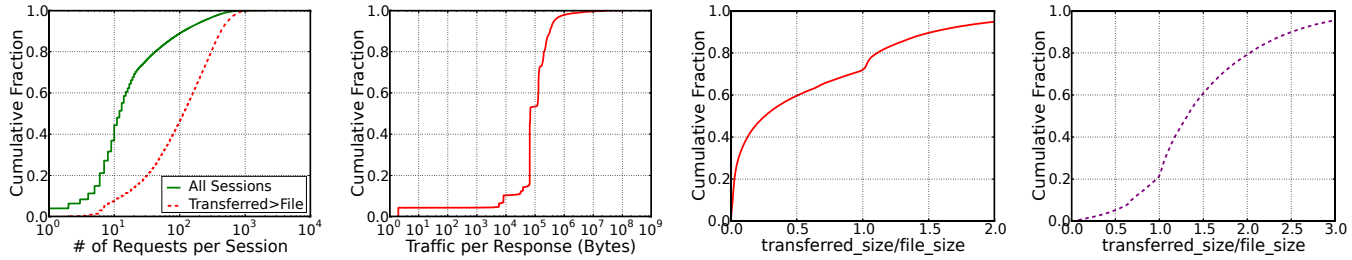
If the requested video is cached, the MediaPlayer sends a `If-Modified-Since` request to the server. If the server replies with HTTP `304 Not Modified`, the MediaPlayer starts to play the cached content. However, due to the limited storage and the potentially large video file size, the MediaPlayer does not cache the entire file. Instead, it only caches the beginning part (e.g., several hundred KBytes) of the video file. So it issues multiple consecutive HTTP range requests to request the non-cached part of the file. Correspondingly, the server replies with HTTP `206 Partial Content` for each range request.

III. Internet and Local Measurements

In order to investigate the Internet mobile streaming delivery to iOS, we conduct measurement and analysis from both a server side and a client side.

A. A Server-Side View

To gain a collective view of the current Internet mobile streaming practice, we are allowed to access server-side logs from a top Internet mobile streaming service provider, Vuclip [15]. Vuclip allows users to access the streaming service via an application installed on their mobile devices. Users can use either cellular or WiFi connections to search and play videos on mobile devices. Pseudo Streaming is available from this service. We collected one-month server log from Feb 1st to Feb 28th, 2011. While accesses from various mobile devices are logged, we extract accesses from iOS devices based on



(a) # of HTTP Requests per Session (CDF) (b) Traffic (Bytes) per Response (CDF) (c) Ratio Between Traffic and Filesize (CDF) (d) When Session Length $> \frac{3}{4}$ Video Duration

Fig. 1. iOS Traffic from Server-side Log

`User-Agent String` in HTTP requests, and compare the logged traffic amount with the actual file size.

In the one-month server-side workload, we extract 397,940 unique video sessions from iOS devices accessing the Vuclip during February 2011. Our following analysis focuses on these iOS sessions only. Note that in each session the video is not necessarily watched from the beginning to the end, as a user may find the video not interesting and terminate the session early.

Each viewing session consists of multiple HTTP (range) requests. Figure 1(a) shows the distribution of # of HTTP requests that are served by the server per viewing session. For about 50% of the sessions, more than 12 requests are used. More than 20% of the sessions have issued more than 40 requests.

Figure 1(b) shows the distribution of HTTP response size. Note here when we count the size of a response, we have excluded the response header information and only count the size of the response body. In general, the response size is determined by two factors: (1) if the request is successfully served, the response size is the same as the requested range; (2) if the request was aborted by the client, then the response size is smaller than the specified range. As shown in the figure, more than 35% responses are about 64 KBytes (reasons to be discussed in Section IV-C).

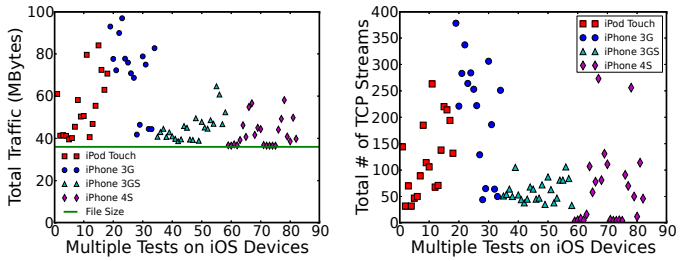
For each unique session, we sum up the body size of HTTP responses, termed as `transferred size`, and compare it with the actual size of the requested file. Figure 1(c) shows the distribution of the ratio between the transferred size against the actual file size. Overall, more than 28% sessions received more data than the actual file size. Besides the extra data received, Figure 1(a) shows these sessions also initiated more HTTP requests to download the video. Given that we are considering all sessions here, such a number would be much larger if we only consider sessions that had watched the entire video. Figure 1(d) further shows that for sessions that last longer than three quarters of the video duration, 78% of them received more data than the video file size. 39% of them even received over 50% more traffic than the actual file size.

B. A Client-Side View

In addition to server-side workload analysis, we also conduct experiments in our lab in order to investigate the redundant traffic. We have conducted experiments with four devices running different versions of iOS as shown in Table I.

TABLE I
DEVICES USED IN EXPERIMENTS

Name	iOS version	Memory Size
iPod Touch	3.1.2	128 MB
iPhone 3G	4.2.1	128 MB
iPhone 3GS	5.0.1	256 MB
iPhone 4S	5.1	512 MB



(a) Traffic received by iOS devices (MBytes) (b) Total # of TCP Streams Per Watching Session

Fig. 2. Statistics of iOS devices watching YouTube

These devices are instructed to access two types of services. On the one hand, they are used to access the streaming service of YouTube [2], Dailymotion [3], and Veoh [4] via the default web browser on iOS, MobileSafari. On the other hand, we also setup our own HTTP server in our lab running Nginx 0.7.65 [16] to do controlled experiments in order to further investigate why and how the redundant traffic is delivered.

During the experiments, to record all the incoming and outgoing packets, we setup Wireshark [17] to listen on the same channel as the testing device in promiscuous mode and capture all packets received/delivered from our testing device. We analyze the traffic that is received by these testing devices, with a focus on the amount of redundant traffic that is received.

Our first set of experiments is to investigate whether such a redundant traffic phenomenon is unique to Vuclip or it exists for other services as well. For this purpose, we first use our iOS devices to watch a same YouTube video repeatedly at different times. We capture all the incoming and outgoing packets in the streaming sessions, and compare the actual size of the video file (36.7 MBytes) with the total number of bytes in the HTTP responses that are received by our testing devices. While the server-side workload may have included early terminated viewing sessions, in the client-side experiments, all viewing sessions are normal sessions without early termination.

TABLE II
AVERAGE TRANSFERRED SIZE VS. FILE SIZE (BYTES)

	YouTube	Dailymotion	Veoh
Duration (sec)	480	478	484
File Size	38,517,389	27,517,003	21,645,105
iPod Touch	57,176,659	Not Playable	41,397,294
iPhone 3G	74,442,375	35,584,935	46,122,323
iPhone 3GS	47,460,396	43,231,408	30,335,657
iPhone 4S	44,538,836	30,999,255	34,320,877

TABLE III
EXTRA TRAFFIC DELIVERED AFTER DOWNLOADING IS FINISHED (BYTES)

	Ratio	Average Re-Downloaded Traffic
iPod Touch	14/18	23,964,478
iPhone 3G	14/16	50,057,285
iPhone 3GS	19/24	1,998,859
iPhone 4S	14/24	5,063,157

Figure 2(a) shows the results of our iOS devices watching the same YouTube video at different times of a day. Most of the streaming sessions received more than 40 MBytes of responses, which is 10% more than the file size. It is noticeable that iPhone 3G even received more than 74 MBytes of traffic in some sessions, which is more than doubled the size of the video file. Note that we only count the payload (i.e., video data) of HTTP responses here, without taking into account the HTTP/TCP/IP headers.

We further conduct experiments with all three different video streaming services, accessing 3 different video files from our testing devices. Table II shows the average received traffic of our 4 testing iOS devices compared to the actual file size from 10 tests each. We find that on average mobile devices' received traffic is consistently more than 110% of the actual file size across all three video files and four different devices.

The impact of such an extra amount of redundant traffic is multi-fold. First, this increases the traffic on the Internet. More importantly, this adds additional load on the server while a server is constantly busy with serving multiple clients. Besides unnecessarily over-utilizing the server and Internet resources, such traffic is also detrimental to the user and mobile devices' interests. On one hand, receiving more data would make the wireless network interface card (WNIC) on the mobile device to work longer and thus consume more battery power. On the other hand, if the video is downloaded using a cellular network connection, it would lead to the data plan tier be reached sooner than expected and generate more monetary cost because cellular data plans today often use a tiered billing model.

IV. ANALYSIS OF REDUNDANT STREAMING TRAFFIC

To find out why such redundant traffic is transmitted, we closely study the captured workloads and further conduct experiments to validate our findings.

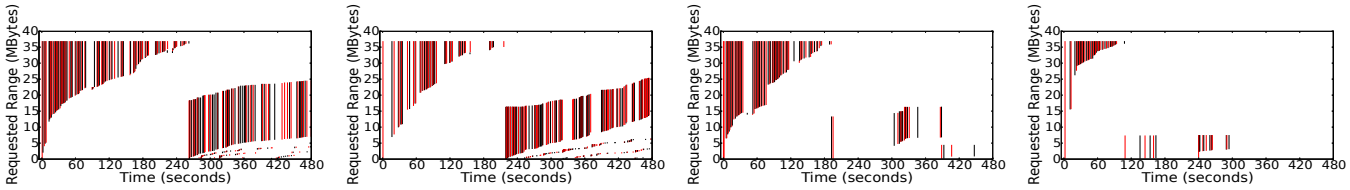
A. MediaPlayer RE-REQUESTs downloaded data after the entire file is downloaded

We have discussed in Section II that to request the video file for playback, the MediaPlayer sends out multiple HTTP range requests for the file. We thus examine how the requested range changes as playback proceeds. Figure 3 shows the `Byte-Range` of consequent HTTP requests during one typical YouTube experiment for our 4 iOS devices, respectively. In all experiments, we instruct our testing devices to watch the same 480-second long video on YouTube. The file size of the video is 38,517,389 Bytes (i.e., the YouTube video in Table II).

Figure 3(a) shows two phases, one before 262 seconds and one after 262 seconds. While the entire video file is fully downloaded in the first phase, we notice that there are quite some range requests afterwards (the second phase). Note that the video plays for 480 seconds. For iPhone 3G, Figure 3(b) also shows similar two phases. The first one before 220 seconds, and the second one after 220 seconds. For iPhone 3GS shown in Figure 3(c), the two phases are before 190 seconds and after 190 seconds. For iPhone 4S shown in Figure 3(d), the two phases are before 109 seconds and after 109 seconds. However, this is not due to re-watching or rewinding, because we watched the entire video without seeking backwards during the playback or clicking on the replay button. That is, after the video file has been completely downloaded, the MediaPlayer automatically starts to request an earlier portion of the file again. Such a behavior is consistently observed in not only YouTube but also other Internet streaming services and our HTTP server.

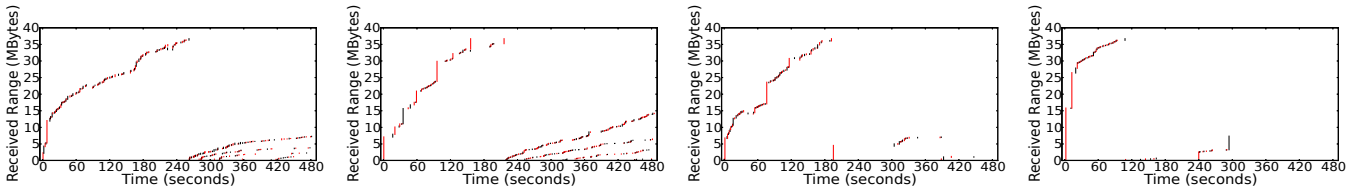
To study these re-downloading, we further examine the repeated YouTube tests. In these viewing sessions, we study the amount of extra traffic as being transmitted after the entire file has been downloaded. The results are shown in Table III. For 14 out of 18 tests on iPod Touch, the MediaPlayer downloaded the beginning part of the video again after finishing downloading the entire file, and the average amount of extra traffic is 22.9 MBytes. For iPhone 3G, such traffic is seen in a higher percentage of tests (14 out of 16), with an average of 47.7 MBytes, which is even larger than the file size. The first such re-request requests from an early point of the file (`Range_Start`, file content after this point is no longer cached in memory) to a certain point in the middle of the file (`Range_End`). In succeeding requests, the `Range_Start` increases with an interval of 64 KBytes as would `Range_End`.

After multiple experiments, our conjecture on this behavior is that mobile devices have limited amount of memory and do not use swap to expand memory size. Given the limited available memory and big video file size, in order to download unplayed portion of the file, some played portion of the video file has to be evicted from the memory. At a later time after the entire file has been downloaded, in order to accommodate user's potential request to re-watch the video, the MediaPlayer downloads the data from beginning again and requests the missing part of the video that has been evicted from the memory, causing re-requests. However, at this point, the memory is mostly occupied by downloaded but not yet



(a) iPod Touch (b) iPhone 3G (c) iPhone 3GS (d) iPhone 4S

Fig. 3. HTTP Range Requests in one YouTube Experiment



(a) iPod Touch (b) iPhone 3G (c) iPhone 3GS (d) iPhone 4S

Fig. 4. Received Range per Request in one YouTube Experiment

played data, and thus the available free memory space is small. In addition, the downloading speed is so fast that it is *about* to fill up the available memory (Section IV-B). Then the MediaPlayer decides to abort that connection so as not to replace the un-played data, causing multiple re-requests to be sent.

In general, mobile devices have a smaller memory size compared to desktop/laptops. For example, the iPod Touch and iPhone 3G we used for experiments have only 128 MBytes memory, and the system daemons use up more than 2/3 of the memory, leaving about only 30 MBytes memory for applications running on the mobile device. Because of the small memory, sometimes it is not feasible for a mobile device to fit the entire video in the memory. One may wonder if by increasing the memory size would solve the problem. However, as we have shown in Figures 3(c) (d) and Table III, with memory size increased to 256 MBytes in iPhone 3GS and 512 MBytes in iPhone 4S, such re-downloading behavior still persists.

While the iOS devices are mobile devices, we further use Safari 5.1.1 to watch the same video on mobile YouTube [2] (instead of the www.youtube.com site) from a MacBook Pro running Mac OS 10.7 Lion with 4 GBytes of memory. While multiple repeated experiments have been conducted, with no exceptions, no re-requests are issued for the downloaded data after the entire file has been downloaded. Clearly, MacBook Pro has a much larger memory size, allowing it to cache the entire video in the memory and serve directly from the memory if the user wants to re-watch the video.

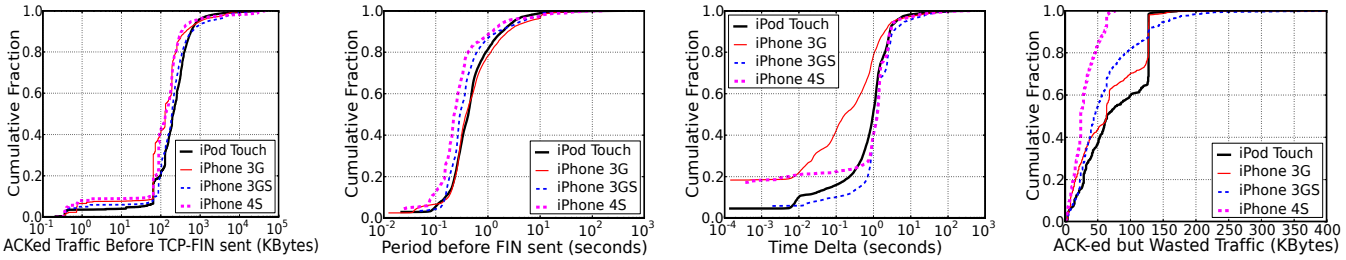
On iOS mobile devices, however, with the increased video quality, higher screen resolutions, and multi-tasking capability, the available memory would always be limited. And as a result, to better accommodate the user's potential replay request, the MediaPlayer's decision to download the video again has potentially contributed a significant amount of redundant traffic. Even worse, the user may decide not to re-watch the video, and the re-downloaded data is completely wasted.

B. The client frequently ABORTs the connection, causing data in flow useless

While Figure 3 shows the range of the requests sent from the mobile devices, Figure 4 shows the corresponding responses to these requests. As we can observe by comparing both figures, while the MediaPlayer often sends out an HTTP range request from the start or a later point to the end of the file, the MediaPlayer also often aborts the connection before it receives all the requested data, and starts a new connection after that. A new HTTP range request is sent to the server through a new connection, with `Byte-Range` from the last successfully received byte in the previous connection all the way to the end of the file. Figure 4(a) shows that 113 connections were set up and terminated during the first 262 seconds when the MediaPlayer was actively downloading the file. For iPhone 3G/iPhone 3GS/iPhone 4S as shown in Figures 4(b) (c) and (d), the received ranges are also consistently smaller than the requested ranges as shown in Figures 3(b) (c) and (d).

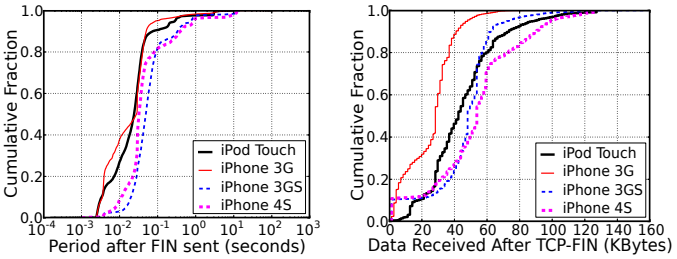
Figure 2(b) shows the total number of TCP connections used to download the entire video file from the YouTube server. On average, iPod Touch uses 114 TCP connections to watch the video, while iPhone 3G uses 208, iPhone 3GS uses 61, and iPhone 4S uses 64. These numbers are also consistent with what we have observed from the server-side log shown in Figure 1(a): among the sessions that received more traffic than the file size, 47% of them sent out more than 100 HTTP requests.

Figure 5(a) shows the distribution of traffic amount that has been successfully received and acknowledged at the TCP layer of the client for each connection. More than 95% of the TCP connections received less than 1 MB data, much smaller than the `Byte-Range` specified in the HTTP requests as shown in Figure 3. Given the small amount of traffic transferred, it is not hard to imagine that the TCP connections do not last long either. Figure 5(b) shows the distribution of time elapsed from the client sends out TCP-SYN to start a TCP stream till it decides to terminate the connection and sends out TCP-FIN. It is shown that about 78%-88% TCP



(a) ACK-ed Traffic of TCP Streams Before TCP-FIN is Sent (b) Connection Time of TCP Streams Before TCP-FIN is Sent (c) Time Delta Between Two Consecutive TCP Stream Connections (d) ACKed but requested AGAIN in Subsequent Connections

Fig. 5. TCP Stream Connections



(a) TCP Stream Session Time After FIN is Sent (b) Data Delivered After TCP-FIN is Sent

Fig. 6. Time and Traffic Delivered During Half-closed TCP

streaming sessions of our testing iOS devices lasted less than 1 second before they were terminated. A subsequent TCP connection was immediately started after that. As shown in Figure 5(c), the mean time difference between the transmission of TCP-FIN of the previous connection and the TCP-SYN of a new connection is about 1 second for iPod Touch/iPhone 3GS/iPhone 4S, and 200 ms for iPhone 3G.

Such abnormal aborts apparently can cause the data in flow to be wasted as typically it takes at least a round-trip time for the server to respond to the termination. One may wonder such frequent aborts and new requests may be due to the slow connection speed. However, these experiments were conducted in our lab with dedicated AP and at different times. As shown later, we have validated that this is not the reason. Before we try to look for answers, we first show how much traffic has been wasted due to such abnormal aborts.

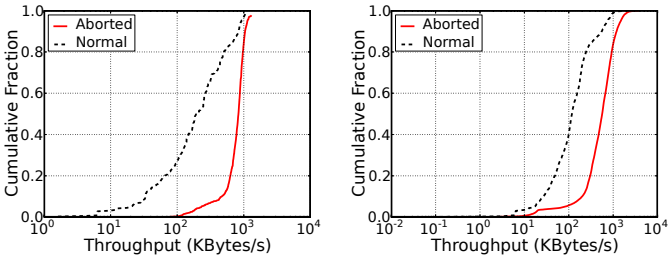
First, in our experiments, we observe that some of the packets received and acknowledged at TCP layer are requested again in the subsequent connection. For example, suppose the MediaPlayer requested the range from 100 KBytes to the end of file in a request, and 500 KBytes are received and acknowledged at TCP layer before TCP-FIN is sent out. Ideally, the subsequent request should be from 600 KBytes to the end of file if the ACK-ed packets are delivered to the application. However, we consistently observe the subsequent request to request from anywhere between 100 KBytes and 600 KBytes, to the end of the file, causing duplicated traffic transmission. Figure 5(d) shows the distribution of the wasted traffic amount. In about 36% and 27% of all aborted connections for iPod Touch and iPhone 3G, more than 120 KBytes data are ACK-ed at TCP layer, but requested again in the subsequent connection. For iPhone 3GS and iPhone 4S, 20% aborted connections wasted more than 93 KBytes and 46 KBytes data, respectively.

Second, besides ACK-ed packets that are not successfully delivered to the application layer, more data is wasted in half-closed connections. When the MediaPlayer decides to terminate a connection, it sends out a TCP-FIN, and expects the server to reply with TCP-FIN. The HTTP server, however, interprets this as a half-closed TCP connection in which there is nothing to transmit from the client side. The server continues to send out the response that has already been in the TCP buffer before replying with a TCP-FIN, given that the TCP window size at the client side is often set to 131,072 Bytes (128 KB) in the last segment. However, as we observe from the traces, for each subsequent none TCP-FIN packet received at the client side, the client sends out a TCP-RST, asking the server to stop sending to this connection. Moreover, most video servers today use asynchronous I/O, which might cause more packets to be sent out before the TCP-RST is processed.

Because of the mismatch between the client and the server, the server continues to send packets after receiving TCP-FIN. We analyze the RST-ed traffic of traces from our experiments. Figure 6 depicts the distribution of time between when TCP-FIN is sent out and when the last packet of the TCP connection is received at the client side. For most TCP stream sessions, it takes the YouTube server about 10 ms to 100 ms to stop sending packets to the closed TCP connection. As a result, as shown in Figure 6(b), a median of 28 KBytes to 54 KBytes data is wasted per closed connection for 4 testing devices. Recall that we have shown in Figure 2(b) that more than 60 TCP connections on average are used for a single viewing session in our experiments, wasting 54 KBytes data per connection would lead to more than 3 MBytes redundant traffic in total for these RST-ed packets.

1) Reasons for frequent connection aborts: As we observe a high number of aborted connections, we set to investigate why such TCP behaviors happen. We first examine whether the serving speed impacts the MediaPlayer's decision to abort a connection. We categorize all TCP connections observed in our experiments into two categories: TCP connections that are aborted before the full HTTP response is received (termed as `Aborted` connections), and normal TCP connections that received the full HTTP response (termed as `Normal` connections).

Figure 7(a) shows the average throughput per connection



(a) Local Nginx HTTP Server (b) YouTube

Fig. 7. Throughput (KBytes/s) of Aborted Connections (CDF)

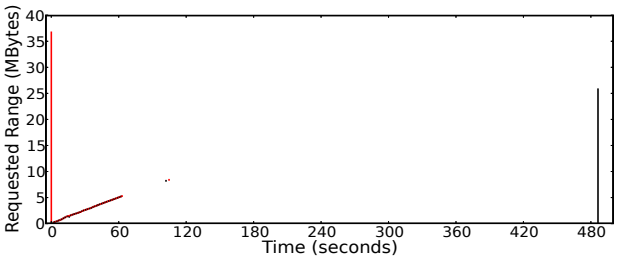


Fig. 8. Byte-Range of Requests in Slow Connections

in our experiments with our local Nginx HTTP server. In these experiments, we set different bandwidth limits to the server, and examine the connections that are aborted or closed normally. We calculate average throughput as the traffic amount that has been received and acknowledged divided by the TCP connection time between TCP-SYN and TCP-FIN. As shown in the figure, about 80% of `Normal` TCP connections have a throughput smaller than 600 KBytes/s, while more than 85% of `Aborted` connections' throughput is more than 600 KBytes/s. Overall, although there is no clear distinction, `Aborted` connections generally have higher throughput than `Normal` connections. The results from Internet (YouTube) experiments as shown in Figure 7(b) are also similar. This confirms that such frequent connection abortions are actually not caused by slow connections, but rather fast connections.

Fast connections can impact the downloading behavior from two aspects. On one hand, when the MediaPlayer finds the downloading speed is so fast that the downloaded but unplayed part of the video has nearly filled up all the available memory, it may decide to abort the connection, and let the playback buffer be consumed before resuming the downloading. On the other hand, with fast connections, the MediaPlayer finds the downloading speed is too fast, but it does not know if the user would continue to watch the video. If not, continuing to download at such a high speed would waste both traffic and battery power on the mobile device. So it decides to abort the current connection, and start a new one afterwards if the user is still watching.

C. Additional OVERLAPPING requests are sent to compensate slow connections

While the fast serving/downloading speed from the server can cause redundant traffic, we find that a slow serving/downloading speed causes problems as well. This happens when the MediaPlayer finds the downloading speed of the current connection is not fast enough to keep up with the playback progress. Such a slow downloading speed may be caused by either server-side bandwidth constraint or client-side connection limit. Server-side bandwidth constraint is potentially caused by: (1) the server is serving too many connections; and (2) the server is throttling the per connection serving speed. On the client side, the mobile device's slow connection speed can be caused by competition for wireless channel, be it either cellular or WiFi. When the connection speed is slow, the MediaPlayer starts a new connection to request data in the unit of 64 KBytes. This helps smoothly play the video by fetching desired data directly. However, the original "slow" connection is not terminated, continues to download the file.

This pattern is very common in the server log we have collected. As shown in Figure 1(b), more than 35% HTTP responses are around 64 KBytes. Close inspection of the log reveals that 29% of HTTP responses we collected from the log are exactly 65,536 Bytes. Among streaming sessions that receive at least one 65,536 Bytes HTTP response, 30% are slow sessions whose average throughput is smaller than the video encoding rate.

We also observe this phenomenon in our HTTP server when we set the serving speed limit. The requested range and the served size as logged are usually 65,536 Bytes (64 KB). For example, Figure 8 shows the `Byte-Range` as specified in each HTTP request when we limit the serving speed to 82 KBytes/s and instruct iPhone 4S to view a video of 480 seconds and 38,517,389 Bytes (80 KBytes/s). The initial request was from the beginning all the way to the end of the file, and was not aborted throughout the session. It took the server 469 seconds to finish delivering the data. During the 469 seconds when the response was being sent, the MediaPlayer further sent out 79 requests, each with a `Byte-Range` of 65,536 Bytes. As a result, 5,117,344 Bytes were received to *compensate* the slow connection. After the entire file was downloaded, the MediaPlayer started to download the file again from the beginning, as shown at 487 seconds in the figure. Because these requests were aborted before the MediaPlayer received the full response, they only led to a total of 86,271 Bytes of traffic. Overall, the MediaPlayer received 13.7% more traffic than the actual file size.

D. Summary

Through experiments and analysis, we find the redundant streaming traffic can be mainly attributed to the following: (1) the limited available memory causes the MediaPlayer to re-download the previously downloaded data in order to accommodate potential replay requests from the user; (2) the limited available memory and the fast connection speed cause HTTP connections to be frequently aborted, wasting a lot of data in flow; (3) a slow connection can also cause the MediaPlayer to issue overlapping requests to provide better experience to end users.

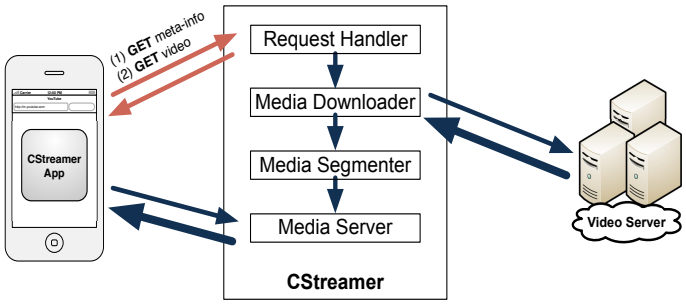


Fig. 9. Overview of CStreamer

V. Design and Implementation of CStreamer

The redundant traffic is mainly caused by the limited available memory on mobile devices and the mismatch between the client and the server for connection aborts. Such redundant streaming traffic not only over-utilizes the Internet and server resources, but also ultimately incurs extra battery power consumption and potentially monetary cost to users.

Unlike desktop operating systems, mobile operating systems today do not use swap/virtual memory to extend memory size. Moreover, as we have shown, even if the physical memory size is increased from 128 MBytes in iPhone 3G to 512 MBytes in iPhone 4S, the problem persists. This is likely due to the increased screen resolution of iPhone 4S that uses more memory for display, and the increased degree of multitasking on iPhone 4S. As the quality level of mobile videos also keeps increasing, the limited memory size is likely to continue as a bottleneck for Internet mobile streaming.

Furthermore, iOS is a closed system, which makes it difficult for users to modify the iOS MediaPlayer. One may argue that such a problem is due to design pitfall or a software bug, and can be fixed by software updates. However, such a problem is seen in different iOS versions from 3.1.2 to 5.1 with millions of devices installed. Updating existing software may not be easy and quick.

With these considerations in mind, we have built a middleware system, which we call CStreamer. With CStreamer, redundant traffic can be eliminated without changing either the iOS MediaPlayer or the many media sites which serve videos via Pseudo Streaming.

A. CStreamer Design

While Pseudo Streaming to iOS generates redundant traffic due to three reasons as discussed in Section IV, we find that such phenomenon does not happen when videos are delivered with HTTP Live Streaming (HLS) [13]. This is because in HLS, a big video file is segmented into small segments, each containing typically 10 seconds of streaming content. Given the small size of these segments, the iOS MediaPlayer can download each segment in one HTTP request and store the downloaded segment fully in memory. This is different from Pseudo Streaming, where the big video file being downloaded cannot be stored in full given the limited available memory.

This suggests a straightforward solution for mitigating the redundant traffic in Pseudo Streaming: convert Pseudo Streaming into HLS. The challenge here, however, is how such conversions can be done in a transparent approach.

Figure 9 shows the architecture of the CStreamer. CStreamer combines an iOS App with a proxy-like CStreamer server. The iOS App works with the CStreamer server to rewrite Pseudo Streaming video links so that the MediaPlayer requests streaming data using HLS from the CStreamer server. When the CStreamer server receives such a video request with the rewritten URL, it downloads the desired video from the video server using a single HTTP GET request. Then it segments the video according to HLS, and transmits the segments to the iOS devices for playback. Converting Pseudo Streaming to HLS with CStreamer brings the following benefits:

1) When downloading speed is fast: With Pseudo Streaming, the MediaPlayer requests the media file aggressively. In CStreamer, however, the MediaPlayer requests file segments sequentially and periodically. That is, a subsequent request is not sent out immediately following the current one. Rather, it waits for its turn until the playback progress has reached its scheduled time.

This allows the MediaPlayer to take into consideration the playback progress when issuing requests. Depending on the memory available at the client side and the connection speed, the MediaPlayer requests at least 1, at most 5 segments ahead of the current playback. On one hand, this reduces the unwatched data if the user stops watching in the middle. On the other hand, when the available memory size is small, the request rate is not as aggressive as in Pseudo Streaming, and therefore HTTP requests would not be aborted. Moreover, the MediaPlayer does not re-download the beginning portion of the video after finishing downloading the entire video.

2) When downloading speed is slow: While downloading a video using Pseudo Streaming under a slow connection, the iOS MediaPlayer issues parallel, overlapping requests for video ranges, leading to redundant traffic and even slower effective downloading speeds. When the MediaPlayer goes through CStreamer transparently, however, it always waits to receive the full response of the current request, without sending out any additional overlapping requests.

To deal with various connection speeds, many video service providers today, including YouTube, offer different versions of the same video encoded in different rates. This allows a user to switch to a lower quality version when the downloading speed is slow compared to the streaming rate. To adaptively deliver the video when the connection speed is slow, CStreamer requests a high quality version and a low quality version of the same video, segments both versions, and puts the meta-information of both versions in the same playlist, allowing the user to seamlessly switch between different versions.

As a middleware solution, CStreamer can be deployed at various places on the Internet. For example, users can set up CStreamer on their desktop computers at home, streaming service providers can set up CStreamer at the server-side as a reverse proxy, cellular service providers can set up CStreamer at their base stations, Internet companies can also set up CStreamer as an add-on service for their users.

B. CStreamer Implementation

Our CStreamer prototype consists of four major components:

1) Request Handler: The Request Handler processes two types of requests sent by the mobile device: meta-info requests and video requests. For *meta-info request* (e.g., requesting a file containing video name, duration, and video link), the Request Handler requests the desired content from the video server. However, before it delivers the response, it rewrites the Pseudo Streaming link in the response to a new URL: the `CStreamer URL`. This URL is an HLS URL that points to a new playlist file on the CStreamer Media Server. After the mobile device receives the response containing the CStreamer URL, if the user decides to watch the video, the MediaPlayer sends out a *video request* directing for the CStreamer URL. When the Request Handler receives such a video request, it calls the Media Downloader.

2) Media Downloader: The Media Downloader receives the request from the Request Handler. It extracts the original Pseudo Streaming link from the CStreamer URL, and starts immediately to download the requested video at the highest speed. As the video is being downloaded, the Media Downloader pipelines content to the Media Segmenter, which segments video without waiting for the download to complete. This pipelining procedure results in a minimal user perceived start-up delay.

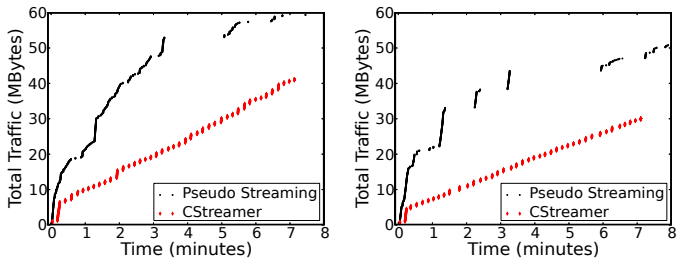
3) Media Segmenter: The Media Segmenter consists of two parts: Container Changer, and Segmenter. Videos deliverable to iOS devices via Pseudo Streaming today, are often put into either MP4 or 3GP format other than MPEG2-TS used by HLS. The video file must be put into MPEG2-TS container format to be segmented. However, unlike video transcoding which is CPU intensive and slow, changing only the container format does not require changing the audio/video encoding and is fast enough to be conducted at real-time.

The Media Segmenter receives pipelined output from the Media Downloader, feeds the data into the Container Changer to change the container format. The Container Changer further pipelines its output to the Segmenter, which segments the video into segments according to the HLS specification [13]. The pipelined execution of the Media Downloader and the Media Segmenter makes CStreamer very fast to prepare the video content.

After the requested video has been processed, the Media Downloader and the Media Segmenter can move on to process another version of the same video, either in higher quality or lower quality.

4) Media Server: While the Media Downloader and Media Segmenter are still processing, the Media Server allows the user to download and watch the first segment. Without an `EXT-X-ENDLIST` tag in the playlist file, the MediaPlayer waits and retrieves the playlist again later from the Media Server, which contains updated playback meta-information.

To efficiently utilize the storage at the Media Server, and save the downloading bandwidth cost, the Media Server also maintains a database with information about the original video file (e.g., web service, video id, video link, etc.) and its corresponding segmented files (e.g., location, playlist file, etc.). This allows more requests for the same video to be served directly from the Media Server, without repeating the downloading and segmenting processes.



(a) YouTube (36.7 MBytes) (b) DailyMotion (26.2 MBytes)

Fig. 10. CStreamer Eliminates Redundant Traffic

C. CStreamer iOS App

For an iOS device to use CStreamer, the end user can set the CStreamer server as an HTTP proxy to handle the requests. However, manually configuring the iOS device is inconvenient for end users, and proxying all traffic through CStreamer puts a lot of burden on the CStreamer server. To mitigate such drawbacks, we have also implemented a CStreamer App. To end users, the CStreamer App is a web browser. However, it monitors all requests, and identifies meta-info requests. For example, the request URL for a YouTube video meta-info starts with `http://m.youtube.com/watch?ajax=1`. The response to this request contains a json file with video's Pseudo Streaming link in it. The CStreamer App redirects such video meta-info requests to the CStreamer server, where the response is rewritten by the Request Handler.

VI. Performance Evaluation

To evaluate the effectiveness of CStreamer, we deploy our prototype on Amazon EC2 [18]. We run CStreamer on an EC2 `Micro Instance`, and instruct our iOS devices to access the video services of YouTube and Dailymotion via CStreamer. For each access, we repeat the experiments 10 times consecutively. In our experiments, we focus on whether CStreamer can serve user's requests in a timely manner, so we do not consider the case when the video can be directly served from CStreamer cache. After each experiment with CStreamer, we empty the media server's storage.

Figure 10(a) shows the traffic patterns of two consecutive experiments we conducted to watch a 480-second YouTube video on iPhone 3GS using Pseudo Streaming and CStreamer respectively. With Pseudo Streaming, over 59 MBytes of traffic is delivered. With CStreamer, the 480-second video is segmented into 48 segments. In the current implementation, the segments are delivered periodically. Thus each segment is delivered every 10 seconds, except for the first 5 segments, which are requested aggressively by the MediaPlayer. Figure 10(a) shows that with CStreamer: (1) each segment is downloaded only once and no RE-REQUEST is observed, even if the last segment finishes downloading 40 seconds earlier before the end of playback; (2) The MediaPlayer on the iPhone

TABLE IV
ESTIMATED START-UP DELAY (SECONDS)

Name	Pseudo Streaming	CStreamer
YouTube	1.78	1.75
Dailymotion	2.42	2.87

TABLE V
AVERAGE WNIC SLEEP TIME (%)

Name	Pseudo Streaming	CStreamer
YouTube	80.9	87.7
Dailymotion	79.8	90.5

3GS does not abort any connections, and each segment is downloaded in only one connection. As a result, no redundant traffic is transmitted during the entire streaming session. As a result, about 31% of traffic is saved compared to using Pseudo streaming.

Similarly, Figure 10(b) shows the traffic patterns of watch a 478-second video on Dailymotion. More than 50 MBytes of traffic is transmitted using Pseudo Streaming, while CStreamer does not cause any redundant traffic.

While CStreamer can eliminate redundant traffic as shown in Figures 10(a) and 10(b), one may wonder if the user perceived streaming quality could be affected due to additional processing between the client and the server. A vital metric here is the start-up delay. We thus examine how long does it take from the user choosing to watch a video to the MediaPlayer starts playback.

We estimate the start-up delay of Pseudo Streaming by examining the period between when the HTTP request for the video is sent and when 10 seconds of streaming data is received. For CStreamer, we examine the period between when the video request is sent and when the first segment was downloaded. To make the comparison more meaningful, we compare a pair of experiments that are conducted sequentially. Table IV shows the results. For YouTube, we find that the video server is close to our testing location. So with Pseudo Streaming, it took only 1.78 seconds to download the initial 10 seconds of playback data. With CStreamer, despite the communication between our client and CStreamer server as well as the processing delay, it took only 1.75 seconds to download the first 10-second segment. This is potentially because the EC2 instance we used to run CStreamer is also close to the YouTube server on the Internet, and therefore it can download the video at very fast speed. Similar to YouTube, we find that Dailymotion does not experience much additional delay either. This indicates the start-up delay, which is important to user's perceived QoS is not affected by using CStreamer.

Using CStreamer also brings another benefit. It allows the wireless network interface card (WNIC) on the mobile device to spend more time in low-power sleep mode, and thus saves battery power consumption. The battery saving comes from two aspects: the reduced total traffic amount and the bursty traffic delivery. For example, for the YouTube experiment presented in Figure 10(a), the WNIC is able to sleep 86.8% (416 seconds) of time during the 480 second playback via CStreamer; while it only sleeps 85.0% of time in our succeeding test when watching the same video via Pseudo Streaming. For Dailymotion, using CStreamer allows the WNIC to sleep 91.7% (439 seconds) of the time over 478 seconds, while it can only sleep 83.6% time when using Pseudo Streaming. The average of the 10 experiments is shown in Table V.

VII. RELATED WORK

In recent years, the Internet streaming traffic has increased dramatically. Plenty of previous work had mainly focused on characterization, measurement, and analysis of all kinds of VoD and P2P-assisted streaming services. For example, different user access patterns, session lengths, video popularity, and other content properties have been studied in traditional and live VoD systems [19], [20]. Krishnappa et al. examined Hulu traffic, focusing on the potential caching and prefetching at the edge networks [21]. For P2P-assisted streaming systems, there are also lots of studies that aimed to characterize and improve the existing performance [22], [23], [24], [25].

Similar studies have been conducted on popular video sites, such as YouTube. For example, user behaviors and video popularity of YouTube were studied and compared with non-UGC content from Netflix [26]. The video properties and access patterns of YouTube were analyzed in [27]. Gill et al. reported the traffic characteristics of YouTube at a campus edge network [28].

With the increase of Internet mobile streaming services, Xiao et al. studied the power consumption of mobile YouTube [29]. Finamore et al. collected traffic from several edge locations and studied the potential reasons for the inferior streaming experience of mobile YouTube users [30]. Rao et al. characterized and compared the traffic pattern of YouTube and Netflix on desktops and mobile devices [31]. Erman et al. examined mobile video traffic over cellular networks from the ISP's perspective [32]. Li et al. examined an iOS-based mobile TV based on server-side logs [33]. In this work, focusing on the dominant iOS devices, we find that Pseudo Streaming to these iOS devices has introduced a significant amount of redundant traffic in the current practice. This is different from Android devices and Windows Phone devices that do not incur redundant streaming traffic [34] [35]. Such redundant traffic is detrimental to both the server and the client, as well as the resource utilization on the Internet. Our proposed solution can effectively address this problem without requiring changes at either the client side or the server side.

VIII. CONCLUSION

Internet mobile streaming traffic has started to dominate the Internet mobile data traffic, and it continues to increase with wider adoption of all kinds of mobile devices. Precisely delivering streaming traffic to mobile devices is not only important to the service providers and the Internet, but also important to mobile devices (battery power wise) and mobile

users (monetary cost wise). In this paper, through measurement and analysis, we find that there is non-trivial redundant traffic delivered when existing mobile streaming services are accessed on iOS devices. Motivated by the analysis results, we design a middleware that can transparently reduce such redundant traffic. Having evaluated with a prototype installed on Amazon EC2, we find that our solution can completely eliminate such redundant traffic without degrading end users' performance.

IX. Acknowledgement

We appreciate constructive comments from anonymous referees. The work is partially supported by NSF under grants CNS-0746649 and CNS-1117300. An earlier version [36] of this manuscript is published in the proceedings of INFOCOM 2013 mini-conference.

References

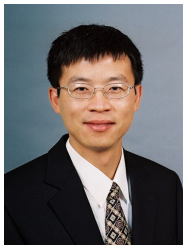
[1] "iOS,Android Market Share on Mobile/Tablet," http://www.netmarketshare.com/mobile-phones.aspx?qprid=9&qpcustomb=1&qpcustom=iOS,Android.
[2] "YouTube," http://m.youtube.com/.
[3] "Dailymotion," http://touch.dailymotion.com/.
[4] "Veoh," http://www.veoh.com/iphone/.
[5] "Hulu," http://www.hulu.com.
[6] "Netflix," http://www.netflix.com.
[7] "Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2012-2017 ," http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-520862.pdf.
[8] "Freewheel Video Monetization Report," http://www.freewheel.tv/docs/FreeWheelMonetizationReport_Q1_2011.pdf.
[9] "AT&T Data Plan," http://www.att.com/shop/wireless/plans/data-plans.jsp.
[10] Y. Liu, F. Li, L. Guo, and S. Chen, "A Measurement Study of Resource Utilization in Internet Mobile Streaming," in *Proc. of ACM NOSSDAV*, 2011.
[11] Y. Liu, F. Li, L. Guo, B.Shen, S. Chen, and Y. Lan, "Measurement and Analysis of an Internet Streaming Service to Mobile Devices," in *IEEE Transactions on Parallel and Distributed Systems*, to appear.
[12] L. Guo, S. Chen, Z. Xiao, and X. Zhang, "Analysis of Multimedia Workloads with Implications for Internet Streaming," in *Proc. of ACM WWW*, 2005.
[13] "Apple HTTP Live Streaming," http://tools.ietf.org/html/draft-pantos-http-live-streaming.
[14] "Sandvine Global Internet Phenomena Report," http://www.sandvine.com/downloads/documents/Phenomena_1H_2012/Sandvine_Global_Internet_Phenomena_Report_1H_2012.pdf.
[15] "Vuclip-Chinese Cinema," http://www.vuclip.com/.
[16] "Nginx," http://www.nginx.org/.
[17] "Wireshark," http://www.wireshark.org.
[18] "Amazon EC2," http://aws.amazon.com/ec2/.
[19] H. Yu, D. Zheng, B.Y. Zhao, and W. Zheng, "Understanding User Behavior in Large-Scale Video-on-Demand Systems," in *Proc. of EuroSys*, 2006.
[20] H. Yin, X. Liu, F. Qiu, N. Xia, C. Lin, H. Zhang, V. Sekar, and G. Min, "Inside the Bird's Nest: Measurements of Large-Scale Live VoD from the 2008 Olympics," in *Proc. of ACM IMC*, 2009.
[21] D. K. Krishnappa, S. Khemmarat, L. Gao, and M. Zink, "On the Feasibility of Prefetching and Caching for Online TV Services: A Measurement Study on Hulu," in *Proc. of PAM*, 2011.
[22] X. Hei, C. Liang, J. Liang, Y. Liu, and K.W. Ross, "A Measurement Study of a Large-Scale P2P IPTV System," in *IEEE Transactions on Multimedia*, December 2007.
[23] S. Ali, A. Matur, and H. Zhang, "Measurement of Commercial Peer-To-Peer Live Video Streaming," in *Proc. of the Workshop on Recent Advances in Peer-to-Peer Streaming*, 2006.
[24] C. Wu, B. Li, and S. Zhao, "Exploring Large-Scale Peer-to-Peer Live Streaming," in *IEEE Transactions on Parallel and Distributed Systems*, June 2008.
[25] Y. Huang, T. Z.J Fu, D.-M. Chiu, J. C.S. Lui, and C. Huang, "Challenges, Design and Analysis of a Large-scale P2P-VoD System," in *Proc. of ACM SIGCOMM*, 2008.
[26] M. Cha, H. Kwak, P. Rodriguez, Y.-Y. Ahn, and S. Moon, "I Tube, You Tube, Everybody Tubes: Analyzing The World's Largest User Generated Content Video System," in *Proc. of ACM IMC*, 2007.
[27] X. Cheng, C. Dale, and J. Liu, "Statistics and Social Network of YouTube Videos," in *Proc. of IEEE IWQoS*, 2008.
[28] P. Gill, M. Arlitt, Z. Li, and A. Manhanti, "YouTube Traffic Characterization: A View From the Edge," in *Proc. of ACM IMC*, 2007.
[29] Y. Xiao, R. Kalyanaraman, and A. Yla-Jaaski, "Energy Consumption of Mobile YouTube: Quantitative Measurement and Analysis," in *Proc. of NGMAST*, 2008.
[30] A. Finamore, M. Mellia, M. Munafo, R. Torres, and S. G. Rao, "YouTube Everywhere: Impact of Device and Infrastructure Synergies on User Experience," in *Proc. of ACM IMC*, 2011.
[31] A. Rao, A. Legout, Y.-S. Lim, D. Towsley, C. Barakat, and W. Dabbous, "Network Characteristics of Video Streaming Traffic," in *Proc. of ACM CoNext*, 2011.
[32] J. Erman, A. Gerber, K.K. Ramakrishnan, S. Sen, and O. Spatscheck, "Over The Top Video: The Gorilla in Cellular Networks," in *Proc. of ACM IMC*, 2011.
[33] Y. Li, Y. Zhang, and R. Yuan, "Measurement and Analysis of a Large Scale Commercial Mobile Internet TV System," in *Proc. of ACM IMC*, 2011.
[34] Y. Liu, F. Li, L. Guo, B.Shen, and S. Chen, "A Comparative Study of Android and iOS for Accessing Internet Streaming Services," in *Proc. of PAM*, 2013.
[35] M. Siekkinen, M. A. Hoque, J. K. Nurminen, and M. Aalto, "Streaming over 3G and LTE: How to Save Smartphone Energy in Radio Access Network-Friendly Way," in *Proc. of MoVid*, 2013.
[36] Y. Liu, F. Li, L. Guo, B.Shen, and S. Chen, "Effectively Minimizing Redundant Internet Streaming Traffic to iOS Devices," in *Proc. of IEEE INFOCOM mini-conference*, 2013.

Yao Liu received the BS degree in computer science from Nanjing University and the PhD degree in computer science from George Mason University. She is an assistant professor in the Department of Computer Science at Binghamton University, State University of New York. Her research interests include Internet mobile streaming, multimedia computing, Internet measurement and content delivery, and cloud computing.

Qi Wei is an Assistant Professor in the Department of Bioengineering and an Affiliate Professor in the Department of Computer Science at George Mason University. Dr. Wei had her postdoctoral training in the Department of Physiology at Northwestern University. Dr. Wei received her Ph.D. from Rutgers University in 2010 and M.Sc. from The University of British Columbia in 2004, both in Computer Science. She had a B.E. degree in Computer Engineering from Beijing Institute of Technology. Dr. Wei's research interests include biomechanical modeling and simulation, medical imaging, computer graphics and networks.

Lei Guo received the BS degree in space physics and the MS degree in computer science from the University of Science and Technology of China in 1996 and 2002, respectively, and the PhD degree in computer science and engineering from the Ohio State University in 2007. After that, he worked in Yahoo! and Microsoft as a senior member of technical staff on the systems and algorithms of social search and social networks. He is a research scientist in Ohio State University. His research interests include big data processing and clouding computing, multimedia systems, mobile computing, social networks, P2P networks, and Internet measurement and modeling.

Bo Shen received the BS degree in computer science from Nanjing University of Aeronautics and Astronautics, China and the PhD degree in computer science from Wayne State University, Detroit MI. He is now the vice president of engineering at Vuclip. Before that, he was a Senior Research Scientist with Hewlett-Packard Laboratories. His research interests include multimedia signal processing, multimedia networking and content distribution systems. He has published over 50 papers in prestigious technical journals and conferences. He holds seven U.S. patents with many pending. Dr. Shen has been on the Editorial Board for IEEE TRANSACTIONS ON MULTIMEDIA from 2006 to 2008. He served as the Lead Guest Editor for IEEE TMM Special Section on Multimedia Applications in Mobile/Wireless Context. He also served on Program Committee for a number of technical conferences including SIGMM.



Songqing Chen received BS and MS degrees in computer science from Huazhong University of Science and Technology in 1997 and 1999, respectively, and the PhD degree in computer science from the College of William and Mary in 2004. He is currently an associate professor of computer science at George Mason University. His research interests include the Internet content delivery systems, Internet measurement and modeling, operating systems and system security, and distributed systems and high performance computing. He is a recipient of the US NSF CAREER Award and the AFOSR YIP Award.

Yingjie Lan is an Assistant Professor at Peking Univeristy. His research interest includes robust networks, competitive analysis of online algorithms, operations management, revenue management and supply chain management.