A Comparative Study of Android and iOS for Accessing Internet Streaming Services

Yao Liu[1], Fei Li[1], Lei Guo[2], Bo Shen[3], and Songqing Chen[1]

[1]Dept. of Computer Science, George Mason University,
{yliud, lifei, sqchen}@cs.gmu.edu
[2]Dept. of CSE, Ohio State University, lguo@cse.ohio-state.edu
[3]Vuclip, bshen@vuclip.com

Abstract. Android and iOS devices are leading the mobile device market. While various user experiences have been reported from the general user community about their differences, such as battery lifetime, display, and touchpad control, few in-depth reports can be found about their comparative performance when receiving the increasingly popular Internet streaming services.
Today, video traffic starts to dominate the Internet mobile data traffic. In this work, focusing on Internet streaming accesses, we set to analyze and compare the performance when Android and iOS devices are accessing Internet streaming services. Starting from the analysis of a server-side workload collected from a top mobile streaming service provider, we find Android and iOS use different approaches to request media content, leading to different amounts of received traffic on Android and iOS devices when a same video clip is accessed. Further studies on the client side show that different data requesting approaches (standard HTTP request vs. HTTP range request) and different buffer management methods (static vs. dynamic) are used in Android and iOS mediaplayers, and their interplay has led to our observations. Our empirical results and analysis provide some insights for the current Android and iOS users, streaming service providers, and mobile mediaplayer developers.

1 Introduction

Mobile devices are gaining increasing popularity among common users. While the market competition between different devices has been intense, iOS devices (such as iPhone, iPad, and iPod Touch) and Android devices (such as Galaxy Nexus, Motorola Droid, and Kindle Fire) are most popular today. It is reported that iOS and Android devices comprise more than 79% of all existing mobile devices [1].

Today more and more mobile users use their devices for Internet streaming accesses. While various streaming protocols are supported, Pseudo Streaming [2] is the most popular among mobile devices. Both iOS and Android have native support for Pseudo Streaming from the very beginning. YouTube [3], Dailymotion [4], and Veoh [5] all support Pseudo Streaming for mobile devices to access their video content.

As streaming accesses typically involve a large amount of data transferring in a continuous fashion for a relatively long duration, two aspects are of particular concerns to a mobile device user. The first is about the battery power consumption. Today the limited battery power supply is still the Achilles' heel of all mobile devices, and a breakthrough of the battery technology is still not on the horizon yet. On the other hand, for most common mobile users, their mobile traffic amount is closely related to the monetary cost that they need to pay to the cellular service provider. Streaming accesses often involve bulk data transmission, resulting in more traffic than other routine activities. Thus it is of a user's greatest interest if a less amount of traffic is delivered while the service quality remains unchanged.

In this work, focusing on Internet streaming accesses, we set to analyze and compare the performance when Android and iOS devices are accessing Internet streaming services. We start with the analysis of a server-side workload collected from a top mobile streaming service provider. In this workload, about 26,713,708 HTTP requests were observed to access 15,725 video clips in 28 days, generating a total of 27.4 TB video traffic. Analyzing this workload, we find that Android and iOS devices use different approaches to request media content, leading to a different amount of received traffic on Android and iOS devices when a same video clip is accessed.

To figure out the underlying causes, we further conduct client-side experiments with the state-of-the-art iOS and Android devices. Through extensive experiments and by delving into the source code of the Android mediaplayer, we find that the current Android and iOS mediaplayers employ different data requesting approaches (standard HTTP request vs. HTTP range request) and different playout buffer management methods (static vs. dynamic). These contrasting approaches and methods lead to a significant amount of redundant traffic received on iOS devices but not on Android devices. Intuitively, this causes more battery power consumption on iOS devices and potentially results in more monetary cost to iOS users.

Our study provides some insights for common users when they access online streaming services. In addition, our experiments and analysis show that different mediaplayer frameworks have been used in Android and iOS with different media content requesting approaches and playout buffer management methods. These insights can help the future mediaplayer development as well as streaming service providers. The client-side trace is available for download at [6].

2 Server-side Observations

The server log we have collected is from a top mobile streaming video site, Vuclip, which serves worldwide mobile users. The workload is collected from Feb 1st to Feb 28th, 2011. In this workload, about 26,713,708 HTTP requests are observed to access 15,725 video clips in 28 days, generating a total of 27.4 TB video traffic.

Vuclip supports both iOS and Android. Users can install an application [7] on their mobile devices from iOS AppStore or Google Play. The application provides the same user interface to both iOS and Android users, and allows them to access the same pool of videos via WiFi or cellular connections. Thus, it provides a good base for our study.

Vuclip allows users to watch videos on their mobile devices using Pseudo Streaming. With Pseudo Streaming, a client can download the video file via HTTP requests, and can start video playback without waiting for the file being completely downloaded. It can also support a user's request to jump to a certain position for playback by downloading the desired part of the file directly via HTTP range requests – HTTP requests with properly specified range headers. In order to provision for the variance of network speed during playback, Pseudo Streaming usually requires a buffer, often referred to as playout buffer, on the client side to store video data to be played. Typically, downloading should be faster than the playback for good user experience, and it is very common that the entire video file has been downloaded while the playback just proceeds to an earlier part of the video.

We use the *User-Agent* string to examine whether a request comes from an iOS device or an Android device. For example, when sending HTTP requests, iOS devices use `AppleCoreMedia/1.0.0` for its User-Agent string, while Android devices identify themselves with `stagefright/1.x (Linux;Android x.x.x)`. In the workload, we extract 397,940 unique video sessions from iOS devices and 884,648 unique video sessions from Android devices. Each session may consist of multiple HTTP requests. In these sessions, the users do not necessarily watch the entire video from the beginning to the end. Users may find the video uninteresting, and terminate the playback in the middle.

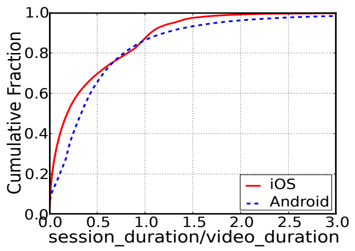


Fig. 1: Ratio Between Session Duration and Video Duration (CDF)

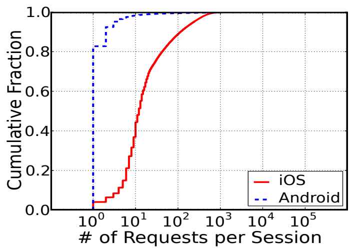


Fig. 2: # of HTTP Requests per Session (CDF)

Figure 1 shows the distribution of downloading session duration for both iOS and Android accesses[1]. Note that the downloading session duration may be shorter than the user's actual viewing duration, because in Pseudo Streaming, the downloading is often faster than the playback. Comparing the accesses from iOS with these from Android devices, we find that the patterns of session duration as opposed to the video duration are quite similar (although Android devices generally have a slightly longer session duration than that of iOS devices). This indicates similar accessing behaviors of Android and iOS devices to this streaming service.

More requests are sent out by iOS devices. Figure 2 shows the distribution of the # of HTTP requests that were sent to the server from mobile devices in these sessions. We find that more than 80% of Android sessions consist of only one single HTTP request, and only less than 2% sessions consist of more than 10 HTTP requests. On the contrary, iOS devices always send more HTTP requests. The median number is 13 HTTP requests per iOS session. This is quite surprising because intuitively, only one HTTP request is needed, which happened to most Android sessions. We are interested in why so many more HTTP requests have been used in iOS sessions.

Based on the log, we find that the MediaPlayer on a mobile device can request the video file in two ways: (1) it requests the entire video file with a standard HTTP request, and the server responds with `HTTP 200 OK`, or (2) it requests a portion of the video file using an HTTP range request, and the server responds with `HTTP 206 Partial Content`. Typically an HTTP range request is used when a user wants to skip part of the video, and jump to the desired content directly. However, in this

[1] iOS and Android accesses (or sessions) refer to accesses (or sessions) originated from an iOS or Android device. They are used for brevity.

server log, we find that iOS devices always use HTTP range requests, even for the first request. But Android devices always use standard HTTP requests, and only use HTTP range requests to fetch desired content directly if the user decides to jump to another part of the video. Table 1 shows the percentage of different types of HTTP requests that have been used by iOS and Android devices, respectively. As shown in the table, more than 80% Android traffic is delivered using standard HTTP responses (200), while almost all iOS traffic is delivered using HTTP partial content response (206). Note that although the percentage of HTTP range requests in Android sessions seems also high, it is mainly because once a user starts to use interactive functions, a sequence of range requests often have to be used. Nevertheless, over 80% of Android traffic is delivered via standard HTTP connections.

More traffic is received at iOS devices. We further sum up the size of HTTP responses that belong to the same video session, and examine if such different content requesting approaches on Android and iOS devices have any impact on the traffic delivered to them. Figure 3 shows the result. As we can observe from this figure, for Android devices, about 55% Android sessions downloaded the same amount of traffic as the video file size, and only a very small percentage of the sessions downloaded more data than the video file size. This could be caused by user re-watching the video. The rest (about 43%) only downloaded partial video content and terminated earlier.

On the other hand, for iOS devices, about 72% iOS sessions terminated earlier before the entire file is downloaded. But the most surprising result is that for about 28% iOS sessions, the downloaded traffic is larger than the video file size. Because we are comparing the requests of Android and iOS devices from a same streaming service, it is reasonable to assume that the users' interest and access patterns are similar. Thus, among the 28% sessions that iOS devices downloaded more data than the actual video file size, only a very smaller portion is likely due to users' real re-watching activities. We are interested in about 28% iOS sessions that have received extra traffic (than the actual file size).

Table 1: HTTP Request/Reply (Number and Traffic Amount)

HTTP 200		
Name	#Requests	Traffic Amount
iOS	0.01%	0.001%
Android	27.30%	80.594%

HTTP 206		
Name	#Requests	Traffic Amount
iOS	99.99%	99.999%
Android	72.70%	19.406%

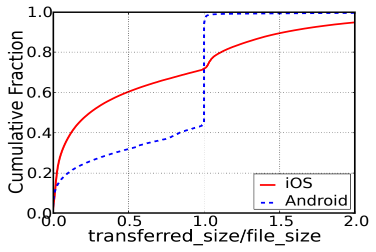


Fig. 3: Ratio Between Received Traffic and File Size (CDF)

3 Analysis of Android and iOS Mediaplayers

While the server-side workload has provided us a high-level overview of different content requesting approaches of iOS and Android devices when accessing Internet streaming services as well as different amounts of traffic received, the workload cannot provide more details for us to explore the underlying reasons. Thus, in this section, we further investigate these observations using the state-of-the-art Android and iOS devices.

For iOS, because we cannot access its source code, we mainly conduct client-side experiments in a controlled environment to infer how it works by analyzing the captured

Table 2: Devices Used

Name	OS version	Memory Size
iPod Touch	iOS 3.1.2	128 MB
iPhone 3G	iOS 4.2.1	128 MB
iPhone 3GS	iOS 5.0.1	256 MB
iPhone 4S	iOS 5.1	512 MB
Nexus One	Android 2.3.4	256 MB
Kindle Fire	Android 2.3.4	512 MB

Table 3: iOS Devices Accessing a 36.7 MB YouTube Video

Name	# of HTTP Connections	Received Traffic (Bytes)	Re-downloaded (Bytes)
iPod Touch	261	83,410,351	26,450,851
iPhone 3G	301	82,616,828	37,449,911
iPhone 3GS	105	63,713,281	11,523,915
iPhone 4S	67	51,625,429	9,292,410

traffic. For Android, in addition to the client-side experiments, we are able to get a better idea of how it works by accessing the source code of its mediaplayer.

The client-side experiments are conducted in our lab with a dedicated 802.11 b/g access point (AP). We use six different mobile devices running different mobile operating systems and different versions of the mobile OS. Table 2 lists these devices. We use 4 different iOS devices and 2 different Android devices. Note that although Kindle Fire uses a customized version of Android, it uses the same mediaplayer framework as other Android devices including the Nexus One we use in our experiments.

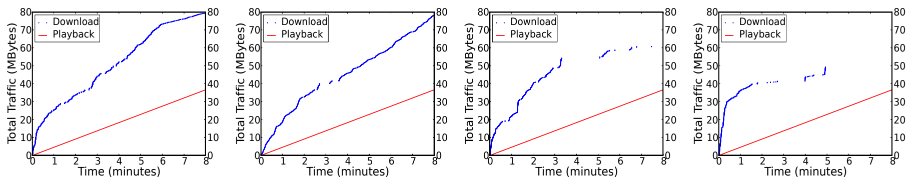
In order to examine all the incoming and outgoing traffic to/from our testing devices, we set up Wireshark [8] running on a laptop computer to listen on the same channel as the AP in promiscuous mode. Packets are captured in real-time and processed offline.

3.1 iOS and AppleCoreMedia

The mediaplayer in iOS is called AppleCoreMedia. When Pseudo Streaming is used to access a video file, AppleCoreMedia will send out HTTP requests for the video file. On the server's side, it can be identified with User-Agent of `AppleCoreMedia/1.0.0`. On iOS devices, a mobile user may access the video streaming service in various ways, e.g., from the mobile browser of MobileSafari, or a third party streaming application installed on the iOS device. AppleCoreMedia will be called when the mobile browser or the application has to handle a streaming request. AppleCoreMedia usually specifies a range in its HTTP requests. For example, if it is requesting the entire video file, it will send out an HTTP request with the range specified from 0 to filesize-1.

To study the behavior of AppleCoreMedia in downloading media content, we use our testing devices to access a same 480-second YouTube video via their mobile browsers. The file size of that video is 38,517,389 Bytes. In each experiment, we let an iOS device watch the entire video (8 minutes) from the beginning to the end without any manual activities. Figure 4 shows the accumulative traffic pattern of 4 different iOS devices accessing this video along time as well as the playback progress. Note the total traffic in this figure only includes the media content. That is, protocol headers are all excluded. We find that during the first 30 seconds of each session, AppleCoreMedia downloads with a high speed, and slows down afterwards. Clearly, this is the initial buffering phase of a video streaming session, which is also called *fast start* [9]. More interestingly, we notice that the amount of received traffic by iOS devices is larger than the video file size (36.7 MBytes). For iPod Touch and iPhone 3G, the total received traffic amount is even more than twice of the actual video file size.

Table 3 summarizes the amount of total traffic received during these sessions by 4 iOS devices. Note that these sessions are normal sessions without early terminations or any replays. Analyzing the corresponding packet-level workload we have captured, we



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

Fig. 4: Traffic Pattern of iOS Devices Accessing a YouTube Video

find that multiple HTTP range requests are issued to download the streaming content. That is, instead of using a standard HTTP request, iOS devices always issue multiple range requests to download media content. This is consistent with what we have observed from the server-side workload shown in Figure 2. It is noticeable that iPhone 3G even issued more than 300 HTTP requests to download the video file. For devices with an increased memory size, such as iPhone 3GS and iPhone 4S, the number of HTTP requests is reduced to 105 and 67, respectively.

The above results show that the multiple HTTP range requests used by iOS are not due to Vuclip, as the same phenomenon has been observed in other popular streaming services as well. Besides YouTube, we have also tested against two other popular sites Dailymotion and Veoh, we have found similar patterns.

In addition, we also find in Table 3 that the received traffic amount on these iOS devices is significantly larger than the actual file size. Recall that we have observed different amounts of traffic delivered to Android and iOS devices in the server-side log. We are interested in whether such extra traffic received on iOS devices is related to the content requesting approach, i.e., the multiple HTTP range requests.

Inspecting the packet-level workload we have captured for these experiments, we find that while AppleCoreMedia always starts with an HTTP range request instead of a standard HTTP request, it constantly terminates the HTTP connection spontaneously before the full response to that range request is received. Subsequently, it will issue another HTTP range request. Having carefully studied the workload, our conjecture is that such behaviors are closely related to the available memory space in a mobile device. Our packet level traces across all these experiments consistently show that AppleCoreMedia always resets (via TCP-RST) the active connection used for the HTTP request. The most likely reason is due to the lack of the memory space for the playout buffer. With a small amount of available memory, AppleCoreMedia has to frequently abort the current connection because the playout buffer is going to overflow.

Besides highly frequent connection aborts (which also necessitates multiple HTTP range requests after aborts), we also find that AppleCoreMedia always re-downloads the beginning part of the video after it has received the entire video file. Recall that with Pseudo Streaming, the entire file is usually received before the user finishes the playback. However, as shown by the last column in Table 3, a significant amount of traffic has been transmitted afterwards for re-downloading the beginning part of the video again. Such re-downloading is also found in our experiments with Vuclip, Dailymotion, and Veoh. Intuitively, this seems to prepare for the potential re-play activities of the user. With the beginning part in the buffer, the user would experience low start-up delay. However, due to the insufficient memory supply on the mobile devices, the beginning part might have been evicted from the buffer after its first-time playback in order to make room for the to-be-played content. Such re-downloading behavior, likely due to insufficient memory size as well, apparently contributes to the redundant traffic we have observed in Figure 3.

For the same reason, for iOS devices with a larger memory size (such as iPhone 3GS and iPhone 4S), the re-downloading traffic amount is much smaller as shown in Table 3. This indicates that with more available memory, AppleCoreMedia can get more buffer space, and put a larger portion of the video file in its buffer.

We further examine the impact of the memory size by instructing our testing devices to access different video files with an increasing file size. We use three different YouTube videos. Videos are of different durations but are encoded with the same data rate. Table 4 shows the results we have obtained. These results are the average results over multiple experiments. This table shows that devices with different physical memory sizes have different traffic efficiency. If we compare the results in a same row, we can see that when the video file size becomes larger, the amount of redundant traffic would also increase. For example, from Table 4 we can see that the redundant traffic for iPhone 4S is increased from 9% when accessing Video1 to more than 15% when accessing Video2 and Video3.

Table 4: Transferred Traffic vs. File Size (Bytes)

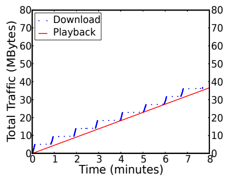
	Video1	Video2	Video3
Duration (sec)	360	480	657
File Size	29,503,221	38,517,389	53,405,910
iPod Touch	42,379,164	57,176,659	90,445,044
iPhone 3G	42,322,498	74,442,375	86,933,886
iPhone 3GS	37,702,143	47,460,396	72,388,936
iPhone 4S	32,248,384	44,538,836	61,731,408

3.2 Android and Stagefright

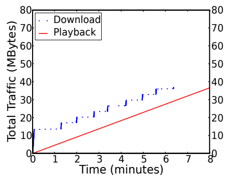
The study of iOS and AppleCoreMedia shows that the memory available to the playout buffer of the mediaplayer is dynamically changing and it plays a critical role in the entire streaming session. In this subsection, we examine if a different type of buffer management method has been used in Android as Android devices have shown different behaviors in accessing streaming media.

Starting from Android 2.3 Gingerbread, a new mediaplayer framework called Stagefright is used in Android. Similar to AppleCoreMedia, Stagefright also supports Pseudo Streaming by using HTTP for requesting video data. On Android devices, a mobile user can access video streaming services from either the mobile browser or applications installed, similar to that on iOS devices. Stagefright is called when a video request needs to be handled. From the server's side, it can be identified with User-Agent of `stagefright/1.x (Linux;Android x.x.x)`. As we shall show later, Stagefright results in a completely different traffic pattern from that of AppleCoreMedia.

To examine how Stagefright works on Android devices, we use our testing devices to access the same 480-second YouTube video (36.7 MBytes) via their native browsers. Again, for each experiment, we let the Android devices watch the entire video for 8 minutes without any manual activities. Figure 5 shows the accumulative traffic pattern of our 2 different Android devices, Nexus One and Kindle Fire, with the corresponding playback speed. We find that downloading is explicitly and periodically paused during the 8-minute playback. With multiple experiments conducted, we find that although the data burst length is different across Nexus One and Kindle Fire, such pausing and resuming behaviors can be consistently observed.



(a) Nexus One



(b) Kindle Fire

Fig. 5: Traffic Pattern of Android Devices Accessing a YouTube Video

Table 5: Android Devices Accessing a 36.7 MB YouTube Video

Name	# of HTTP Connections	Received Traffic (Bytes)	Re-downloaded (Bytes)
Nexus One	1	38,517,389	0
Kindle Fire	1	38,517,389	0

Further inspection of the corresponding packet level workloads reveals that only one single HTTP request is used to download the video file by both Nexus One and Kindle Fire as shown in Table 5. When the downloading is paused, instead of terminating the current TCP connection as AppleCoreMedia does, Stagefright sets the TCP window size to 0, so that the server would not send any more packets to it. When it wants to resume the downloading, it will send a TCP window update message, and the server will start to deliver the data again. Moreover, we find that the total traffic amount is always equal to the video file size, indicating no re-downloading of the beginning part. This is also different from AppleCoreMedia.

Such different behaviors observed on Stagefright in these experiments and in the server-side log motivate us to explore the underlying reasons. Next, we study the Android source code to better understand how Stagefright works.

```
enum {
 kPageSize        = 65535,
 kDefaultHighWaterThreshold = 20 * 1024 * 1024,
 kDefaultLowWaterThreshold  = 4 * 1024 * 1024,
 kDefaultKeepAliveIntervals = 15000000,
};
```

Fig. 6: Code Snippet From /libstagefright/include/NuCachedSource2.h

In the libstagefright framework, the underlying media playout buffer is handled by `NuCachedSource2.cpp`. Basically, it sets a `HighWaterThreshold`. When the total buffer size reaches this threshold, the downloading would be paused. As the playback progresses, the buffer depletes. When the to-be-played data in the buffer drops below another pre-defined threshold `LowWaterThreshold`, the downloading will be resumed. Figure 6 shows some code snippet from the latest Stagefright source code we extract from the Android base. We can see that buffer space is allocated in terms of 65,536 Bytes (64 KB). When the total buffer size reaches 20 MB, downloading would be paused; when the remaining not-played data is less than 4 MB, Stagefright will resume the downloading. As the downloading is paused, in order to keep the connection with the server, it would temporarily resume to download a PageSize (64 KB) of data every 15 seconds and pause the downloading after that. This buffer management method well explains what we have observed in both the server-side log and the client-side experiments.

Further studying the history of earlier versions in the Android code base, we find that the value of these 4 parameters shown in Figure 6 have changed over time. For example, in the earliest version, the `HighWaterThreshold` was set to 3 MB, and the `LowWaterThreshold` was 512 KB. This indicates as Android devices are getting more physical memory, a larger amount of buffer is allocated to the mediaplayer.

Nevertheless, the `HighWaterThreshold` can be seen as the total buffer size used by Stagefright on Android devices. That is, Stagefright would only use a fixed amount of memory despite different video file sizes, and that only a fixed amount of video data would be kept in the buffer. Compared to iOS, this is a simple and static buffer management method.

In addition, different Android devices may use different values for these parameters in their out-of-factory settings. For example, based on Figure 5, we can estimate that the `HighWaterThreshold` for Nexus One is around 5 MB, while Kindle Fire uses a larger value of about 13 MB. By analyzing the debugging log from these Android devices, we are also able to get the accurate value of `LowWaterThreshold`, which is 768 KB for Nexus One and 10 MB for Kindle Fire, respectively.

3.3 Comparisons

Through client-side experiments, we confirm that Android devices often use a single HTTP connection to download the video file unless there is manual interruption of current playback. On the contrary, iOS devices always use multiple HTTP range requests to download the video file. Buffer management wise, by analyzing the source code of Android mediaplayer, we find that Stagefright always uses a fixed/preset amount of memory for the playout buffer, while AppleCoreMedia of iOS devices always adjust the playout buffer dynamically at runtime.

We believe such different buffer management policies have caused iOS and Android devices to exhibit different behaviors when they are used to access streaming videos. Stagefright would always and only store a fixed amount (set by `HighWaterThreshold`) of video data, and may download at most this amount of video data ahead of the playback. If the user stops watching the video in the middle, at most `HighWaterThreshold` amount of data may be wasted. But in normal streaming sessions with few user manual inter-activities, Stagefright on Android devices always downloads the exact amount of data as the video size, while AppleCoreMedia on iOS devices always tries to keep as much video data as possible in the buffer for user's experience, including re-downloading the beginning part. This results in a significant amount of redundant traffic delivered to iOS devices.

4 Related Work

With the increasing video accesses from mobile devices, a lot of research has been conducted to examine Internet mobile streaming, from the client's perspective [2] [10], the video server's perspective [11], and the ISP's perspective [12] [13]. For example, in our prior work, we conduct extensive measurements from the client's perspective about the energy-efficiency of various streaming protocols used by mobile devices today [2]. Li et al. present a detailed analysis of user behaviors and access patterns in mobile video streaming from a server's perspective [11].

Researchers have also studied how accesses from mobile devices and desktop computers are served differently by the video service providers. For example, Rao et al. characterize the traffic pattern of YouTube and Netflix on both desktop computers and mobile devices [10] . Finamore et al. [12] compare the playback performance of PC-players and mobile-players accessing YouTube, and examine the potential causes for the inferior performance of mobile-players.

Different from prior work, in this study, we focus on the streaming access performance of two dominant types of mobile systems Android and iOS. We find that the different content requesting patterns and different playout buffer management policies have caused these devices to have sharply different behaviors.

5 Conclusion

Internet mobile streaming has attracted significant attention from both industry and research community, due to the dominant streaming traffic volume in the entire mobile data traffic. In this work, we focus on the Internet mobile streaming delivery to Android and iOS devices, with an aim to investigate their performance when receiving Internet streaming content. With both server-side log analysis and client-side experiment-based investigations, we find that Andriod and iOS mediaplayers are using different content requesting approaches and different buffer management methods when accessing streaming content, which result in a non-trivial amount of redundant traffic received by iOS devices. This would lead to extra battery power consumption on iOS devices and additional monetary cost if cellular networks have been used. Our study not only provides some guidelines for common mobile device users, but also offers some insights for Internet streaming service providers and mobile mediaplayer developers.

Acknowledgements. We appreciate constructive comments from anonymous referees and our shepherd Edmond W. W. Chan. The work is partially supported by NSF under grants CNS-0746649, CNS-1117300, CCF-0915681, CCF-1146578.

References

1. "Mobile/Tablet OS Market Share," `http://marketshare.hitslink.com/operating-system-market-share.aspx?qprid=8&qpcustomd=1`.
2. Y. Liu, L. Guo, F. Li, and S. Chen, "An Empirical Evaluation of Battery Power Consumption for Streaming Data Transmission to Mobile Devices," in *Proc. of ACM Multimedia*, 2011.
3. "YouTube," http://m.youtube.com/.
4. "Dailymotion," http://touch.dailymotion.com/.
5. "Veoh," http://www.veoh.com/iphone/.
6. "Trace," `http://cs.gmu.edu/~sqchen/open-access/pam13-trace.tgz`.
7. "Vuclip-Chinese Cinema," http://www.vuclip.com/.
8. "Wireshark," http://www.wireshark.org.
9. "Fast Start," `http://www.microsoft.com/windows/windowsmedia/howto/articles/optimize_web.aspx#performance_faststreaming`.
10. 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.
11. 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.
12. 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.
13. 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.