Effectively Minimizing Redundant Internet Streaming Traffic to iOS Devices

Yao Liu[1] Fei Li[1] Lei Guo[2] Bo Shen[3] Songqing Chen[1]
[1]George Mason University [2]Ohio State University [3]Vuclip
{yliud, lifei, sqchen}@cs.gmu.edu lguo@cse.ohio-state.edu bshen@vuclip.com

Abstract—**The Internet has witnessed rapidly increasing streaming traffic to various mobile devices. In this paper, we find that for the popular iOS based mobile devices, accessing popular Internet streaming services typically involves about 10% - 70% unnecessary 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 users' mobile devices and leads to possible monetary cost. To alleviate such a situation without changing the server side or the iOS, we design and implement a CStreamer prototype that can transparently work between existing iOS devices and media servers. We also build a CStreamer iOS App to enable end users to access Internet streaming services via CStreamer. Experiments conducted based on this prototype running on Amazon EC2 show that CStreamer can completely eliminate the redundant traffic without degrading user's QoS.**

I. Introduction

Today mobile devices, such as iPhone and iPad, are becoming more and more popular. Accesses from mobile devices are directed to all kinds of Internet streaming services. For example, popular video sharing websites such as YouTube [1], Dailymotion [2], and Veoh [3], all allow mobile users to access their services. As a result, today mobile video traffic dominates the Internet mobile traffic. According to Cisco's report [4], mobile video traffic accounts for 52% of total mobile data traffic in 2011, and is predicted to exceed two thirds by 2016. It has been found that 80% of the mobile video accesses take place on iOS devices [5].

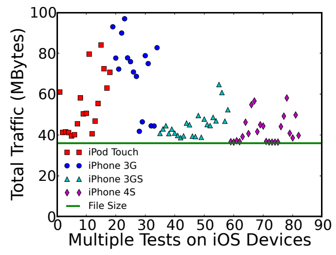
Compared to general web surfing on the Internet, streaming applications often involve bulk data transmission in a continuous fashion, which may deplete the limited battery power supply at a fast pace and incur extra monetary cost for cellular data plan users. Therefore, for mobile devices and mobile 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, we find that for the popular iOS based mobile devices, accessing streaming services typically involves about 10% to 70% unnecessary 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 iOS MediaPlayer 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. 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 potential extra monetary cost).

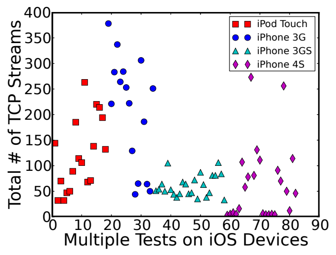
Motivated by our measurement results, we examine the potential causes for such unnecessary traffic in normal mobile streaming accesses. We find these problems are mainly due to the limited memory size of popular iOS devices and the too fast/slow network connections. These findings motivate us to seek 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 downloading with the MediaPlayer's 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.

To evaluate the effectiveness of CStreamer in minimizing the redundant traffic, we have implemented a prototype of CStreamer running on Amazon EC2. To enable transparent user accesses, we have also built a corresponding CStreamer App. 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 perceived streaming experience. In summary, this paper makes the following contributions:

- We find that the current 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).



(a) Traffic Received by iOS Devices (MBytes)



(b) Total # of TCP Streams per Watching Session

Fig. 1: Statistics of iOS devices watching YouTube

- Conducting client-side experiments, we investigate the potential reasons of such redundant traffic. We find it is mainly attributed to the limited memory size of mobile devices, 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.

II. HTTP Range Request and Streaming to iOS

Among the popular mobile devices, iOS based devices are leading the market [6]. According to Freewheel, 80% of wireless video views take place on iOS devices [5]. iOS supports two streaming protocols: Pseudo streaming and HTTP Live Streaming (HLS) [7]. Pseudo streaming today carries more mobile traffic than HLS, as it is often used by video streaming services like YouTube [1] and DailyMotion [2], and YouTube alone contributes 27% of mobile traffic in North America [8].

With Pseudo streaming, the client can download the media content from an HTTP server. The playback can start before the entire file is downloaded. The client uses 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 data 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`). It would first ask 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 would send multiple HTTP requests for the video file, and specifies a range to download in each request.

III. Local Measurement Results

We have conducted experiments with four devices running different versions of iOS: iPod Touch (iOS 3.1.2), iPhone 3G (iOS 4.2.1), iPhone 3GS (iOS 5.0.1), and iPhone 4S (iOS 5.1). These devices are instructed to access the streaming services of YouTube, Dailymotion, and Veoh via MobileSafari. Due to space limit, we only show the results of YouTube. During the experiments, to record all the incoming and outgoing packets, we setup Wireshark to listen on the same channel as the testing device in promiscuous mode and capture all packets received/delivered from our testing devices.

We use our iOS devices to watch a same 480-second YouTube video repeatedly. We capture all the 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. In these experiments, all viewing sessions are normal sessions without early termination, seeking, or replay. Figure 1(a) shows the results. 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 TCP payload (i.e., video data) of HTTP responses here, without taking into account the protocol headers. Our experiments with Dailymotion and Veoh show similar results.

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 device's interests. On one hand, receiving more data would make the wireless network interface card (WNIC) on the mobile device 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 user's 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. Summary of Redundant Traffic Analysis

To find out why such redundant traffic is transmitted, we closely study the captured workloads and further conduct experiments to validate our findings. Our experiments and analysis reveal three causes of the redundant streaming traffic. Due to space limit, we only present our major findings here. Please refer to our technical report [9] for detailed analysis.

With Pseudo streaming, the entire video file may have been downloaded while the video is still playing. However, the iOS MediaPlayer automatically starts to request the file from the beginning again afterwards. Our conjecture for this behavior is that the MediaPlayer downloads the data again in case the user wants to re-watch the video. Due to the limited amount of memory that is made available to the MediaPlayer by the OS, it is sometimes not feasible for the mobile device to fit the entire video in the memory. Therefore, the MediaPlayer has to request the missing part of the video that is not cached in the memory.

Besides re-downloading, we find that the MediaPlayer often aborts the TCP connection before it receives all the requested data, and starts a new connection afterwards. A new HTTP range request would be 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 1(b) shows the total number of TCP connections used to download the entire video file from the YouTube server. Such abnormal aborts cause traffic in flow wasted. Our conjecture is that when MediaPlayer finds the downloading speed is so fast that the downloaded but not-played part of the video has nearly filled up all the available memory, it decides to abort the connection and let the playback buffer consume before it resumes the downloading.

While the fast serving/downloading speed from the server can cause redundant traffic, we find that a slow speed causes problems as well. When the MediaPlayer finds that the downloading speed of the current connection is not fast enough to keep up with the playback progress, it would start new connections to request data in the unit of 64 KBytes continuously. This helps smoothly play the video by fetching desired data directly. However, the original "slow" connection is not terminated, and continues to download.

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 incurs extra battery power consumption and potential 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, given that iOS is a closed system, it is difficult to address the redundant traffic problem by modifying 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 operating system 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 phenomena does not happen when videos are delivered

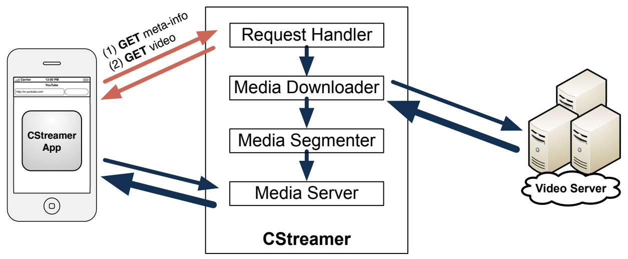


Fig. 2: Overview of CStreamer

with HTTP Live Streaming (HLS). 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 2 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 re-written URL, it downloads the desired video from the video server using a single 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 HLS, 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. Therefore, the request rate is not as aggressive as in Pseudo streaming, HTTP requests are not aborted even the available memory size of the requesting device is small. 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 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 would always wait to receive the full response of the current request, without sending out any additional overlapping requests. 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.

B. *CStreamer Implementation*

Our CStreamer prototype has 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 would request the desired content. 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 would send out a *video request* directing for the CStreamer URL. When the Request Handler receives such a video request, it would call 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 the downloaded content to the Media Segmenter, which segments the video without waiting for the downloading 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. The pipelined execution of the Media Downloader and the Media Segmenter makes CStreamer very fast to prepare the video content.

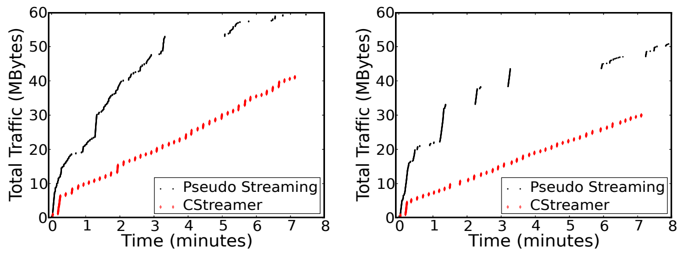
After the requested video has been processed, the Media Downloader and the Media Segmenter can move on to process another version of the same video, either in higher quality or lower quality.

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 would wait and retrieve 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, it 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.

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.



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

Fig. 3: CStreamer Eliminates Redundant Traffic

VI. Performance Evaluation

To evaluate the effectiveness of CStreamer, we implement our prototype CStreamer server and run it on Amazon EC2. 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 have repeated the experiments 10 times consecutively. In our experiments, we focus on whether CStreamer can serve users' 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 would empty the media server's storage.

Figure 3(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. Results with other iOS devices are similar, and we omit them due to space limitation. With Pseudo streaming, over 59 MBytes of traffic was delivered. With CStreamer, the 480-second video is segmented into 48 segments. Each segment is delivered every 10 seconds, except for the first 5 segments, which were requested aggressively by the MediaPlayer. As a result, (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 3GS did 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 32% of traffic is saved compared to using Pseudo streaming.

Similarly, Figure 3(b) shows the traffic patterns of watching a 478-second video on Dailymotion. More than 50 MBytes of traffic was transmitted using Pseudo streaming, while CStreamer did not cause any redundant traffic.

TABLE I: Estimated Start-up Delay (seconds)

Name	Pseudo streaming	CStreamer
YouTube	1.78	1.75
Dailymotion	2.42	2.87

TABLE II: Average WNIC Sleep Time (%)

Name	Pseudo streaming	CStreamer
YouTube	80.9	87.7
Dailymotion	79.8	90.5

We further examine if the start-up delay is increased due to additional processing between the client and the server. We estimate the start-up delay of Pseudo streaming by examining the period between the HTTP request for the video is sent and when the first 10-second 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 I 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 1.75 seconds to download the first 10-second segment. 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 perceived QoS, is not affected by 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, in the YouTube experiment, the WNIC is able to sleep 86.8% (416 seconds) of time during the 480 second playback; 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 II.

VII. Related Work

In recent years, the Internet mobile streaming traffic has increased dramatically. Pseudo streaming (as used by popular video services like YouTube) and HTTP Live Streaming (HLS) (as promoted by Apple) are among the most popular streaming protocols used by mobile devices today. Plenty of previous work had focused on analyzing these two protocols. In our prior work [10], we compared the energy-efficiency of different streaming protocols, including HLS and Pseudo streaming. Finamore et al. collected traffic from several edge locations and studied the potential reasons for the inferior streaming experience of mobile YouTube users [11]. Erman et al. examined mobile video traffic (HLS and Pseudo streaming combined) over cellular networks from the ISP's perspective [12]. Li et al. examined an iOS mobile TV that uses HLS based on server-side logs [13]. In this work, focusing on the dominant iOS devices, we find that streaming to these iOS devices has introduced a significant amount of redundant traffic in the current practice, which 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, CNS-1117300, CCF-0915681, CCF-1146578.

References

[1] "YouTube," http://m.youtube.com/.
[2] "Dailymotion," http://touch.dailymotion.com/.
[3] "Veoh," http://www.veoh.com/iphone/.
[4] "Cisco VNI," http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-520862.pdf.
[5] "FREEWHEEL VIDEO MONETIZATION REPORT," http://www.freewheel.tv/docs/FreeWheelMonetizationReport_Q1_2011.pdf.
[6] "Mobile/Tablet OS Market Share," http://marketshare.hitslink.com/operating-system-market-share.aspx?qprid=8&qpcustomd=1.
[7] "Apple HTTP Live Streaming," http://tools.ietf.org/html/draft-pantos-http-live-streaming.
[8] "Sandvine Global Internet Phenomena Report," http://www.sandvine.com/downloads/documents/Phenomena_1H_2012/Sandvine_Global_Internet_Phenomena_Report_1H_2012.pdf.
[9] "Effectively Minimizing Redundant Internet Streaming Traffic to iOS Devices," http://cs.gmu.edu/~sqchen/open-access/range-tr.pdf.
[10] 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.
[11] 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.
[12] 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.
[13] 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.