Mobile-Friendly HTTP Middleware with Screen Scrolling

Lei Zhang
School of Computing Science
Simon Fraser University
BC, Canada
Email: lza70@cs.sfu.ca

Feng Wang
Department of Computer and Information Science
The University of Mississippi
MS, USA
Email: fwang@cs.olemiss.edu

Jiangchuan Liu
School of Computing Science
Simon Fraser University
BC, Canada
Email: jcliu@cs.sfu.ca

***Abstract*—The pervasive penetration of mobile smart devices has significantly enriched Internet applications and undoubtedly reshaped the way that users access Internet services. D-ifferent from traditional desktop applications, mobile Internet applications require users to input via touch screens and view outputs on the displays with considerably limited size. The significant conflict between the limited-size of touch screens and the richness of online media contents requires the mobile Internet applications to download contents way beyond the user's viewing region (referred as *viewport*).**

In this paper, we present a Mobile-Friendly HTTP middleware (MF-HTTP), which interprets user touch screen inputs and optimize the HTTP downloading of media objects to improve quality of experience (QoE) and cost efficiency. We first demystify screen scrolling in mobile operating systems and precisely break down the viewport moving process. We identify the key influential factors for media object downloading and develop an optimal download scheme. Towards building a practical middleware, we further discuss and address the implementation issues in detail. We implement a MF-HTTP prototype based on Android platforms and evaluate the performance of MF-HTTP by conducting concrete case studies on two representative applications, namely, web browsing and 360-degree video streaming.

1. Introduction

During the past decade, we have witnessed the pervasive penetration of mobile smart devices such as smartphones, tablets and wearable devices, which significantly enrich Internet applications and improve user experience. In the foreseeable future, mobile smart devices are predicted to take up over 50% of global devices/connections and surpass 4/5 of mobile data traffic by 2021 [1]. Different from traditional desktop applications, in which users interact via input/output devices like large-size displays, keyboards, and mouses, mobile Internet applications require users to input through *touch screens* and allow them to view outputs on the displays of considerably limited size. This distinct feature introduced by mobile hardware interfaces brings both challenges and opportunities to mobile Internet applications. On one hand, the service providers of mobile Internet applications should

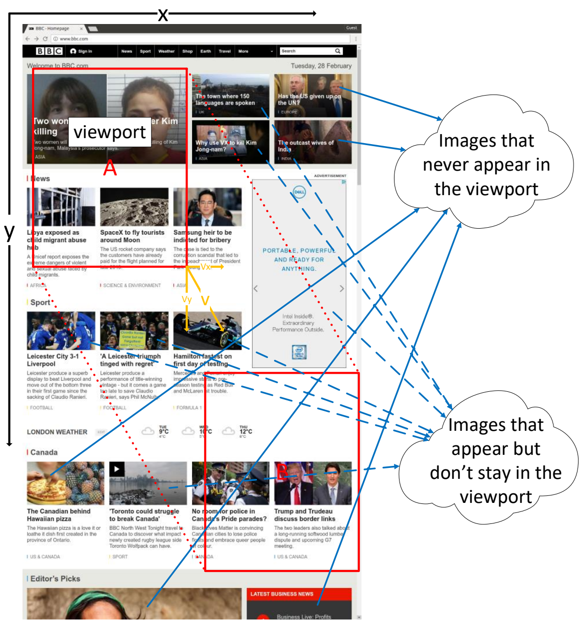

x
y
viewport
A
Vx
Vy
V
Images that never appear in the viewport
Images that appear but don't stay in the viewport

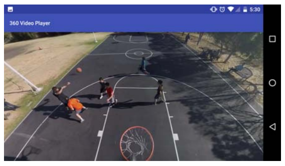
Figure 1: An example of mobile web browsing

provide multiple copies of media contents with different resolutions and even multiple versions of application user interface (UI) layouts to fit various sizes of screens on heterogeneous devices. On the other hand, as media contents are usually organized in certain order/layout in mobile Internet applications, it is possible to predict the viewing region (referred as *viewport* hereafter) given the user inputs and the fixed size of display.

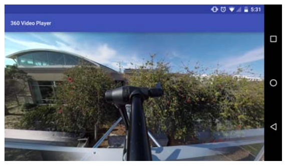
The significant conflict between the limited-size of touch screens and the richness of online media contents requires the mobile Internet applications to download contents way beyond the user viewport. Figs. 1 to 3 show three examples of different applications (i.e., mobile web browsing [2], 360-degree video watching [3], and mobile social networking [4]) that constantly download contents out of the viewports to guarantee good user experience. Understanding how the viewport moves becomes crucial to optimize those applications. Fortunately, as the screen scrolling animation

2575-8411/18/$31.00 ©2018 IEEE
DOI 10.1109/ICDCS.2018.00140

(a) A raw frame

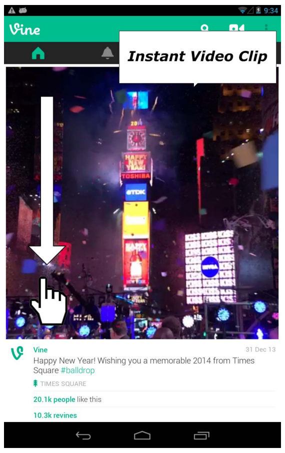


(b) Viewport 1

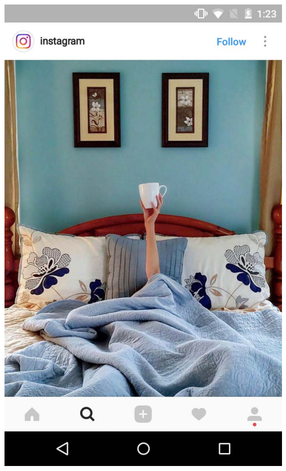


(c) Viewport 2

Figure 2: 360-degree video watching



(a) Vine



(b) Instagram

Figure 3: Mobile social networking

is mostly affected by user touches, once an input gesture is given based on user touches, the following process of viewport movement is predetermined in mobile operating systems. Therefore, by studying the impacts of user touches on screen scrolling, our work targets to improve Quality of Experience (QoE) and cost efficiency for a class of mobile Internet applications that make downloads beyond user viewport.

Taking web browsing as an example, mobile users usually can only view a limited area of the web page. By tracking user touches, it is possible to identify in which direction the viewport moves and where it stops, as well as the area covered during the deceleration. Conventionally, web browsers download all the elements in the web page by default, as users can easily view the whole web page on a desktop display. However, in the mobile scenario, given the entire screen scrolling process, we are able to tell what media contents need to be downloaded. For instance, in Fig. 1, the user browses the web page and scrolls the viewport from position A to position B. The area bounded by the dashed lines is covered during the deceleration of screen scrolling. In this web browsing event, there is no need to download the images that are entirely out of the scrolling covered area, which does not hurt the user experience as they never appear in the viewport.

In this paper, we present the Mobile-Friendly HTTP middleware (MF-HTTP), which acts at the application layer, interprets screen scrolling processes on mobile devices by tracking user touch screen operations, and optimize the downloading of media objects to improve QoE and cost efficiency. We first demystify screen scrolling philosophy in mobile operating system in depth. With the opportunities of collecting and understanding user touch screen operations, we show how to precisely break down the viewport movement, and identify the media objects involved in the process. By examining the key influential factors for media object downloading, we develop an optimal download scheme. Towards building a practical middleware, we further discuss the implementation details for MF-HTTP, based on which we implement a prototype on Android platforms. We conduct two concrete case studies on web browsing and 360-degree video streaming, integrate them with our MF-HTTP middleware implementation, and evaluate the performance through extensive experiments.

2. Background and Related Work

A serial of studies have been conducted to optimize web browsing, an application that is largely affected by user viewport. Prior work [2] suggests that client-only approaches have significant limitations for mobile users: caching [5] web contents does not remove the true bottleneck of web page loading–RTT, and predictive prefetching [6] cannot work well either since most of the pages will only be requested once by a user. A recent measurement study [7] shows that only a few web sites have fully deployed HTTP/2 (the state-of-the-art standard in industry) servers, and few of them have correctly realized the new features in HTTP/2, which implies the desire of research efforts on optimizing web performance. Scheduling network requests is a widely exploited approach to reduce page load time, which is designed base on the dependency between web page elements [8]. Butkiewicz et al. [9] proposed KLOTSKI, a system that prioritizes the contents most relevant to the user preference and with least rendering time. By collecting the traces of user gaze fixation during web browsing, Kelton et al. [10] examined the focus of user attention and reordered the loading of web objects accordingly. To achieve the best performance-energy tradeoff, Ren et al. [11] adopted a machine learning based approach to predict the optimal processor configurations at runtime for heterogeneous mobile platforms.

Video streaming is another killer application influenced by user viewport. Dynamic Adaptive Streaming over HTTP

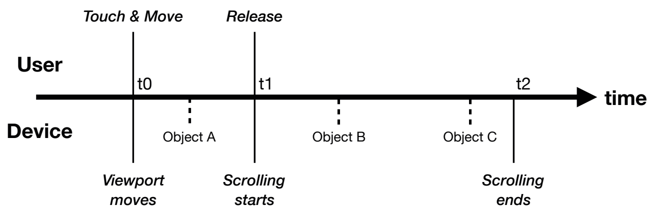


Figure 4: An example of screen scrolling process

(DASH) is widely deployed on the Internet for live and on-demand video streaming services. The rate adaptation scheme is one of the fundamental research issues for DASH. By studying the responsiveness and smoothness trade-off in DASH, Tian et al. [12] showed that client-side buffered video time is a helpful feedback signal to guide rate adaptation. Instead of constantly predicting future capacity, Huang et al. [13] proposed to use simple capacity estimation only in the startup phase and then choose the video rate based on the current buffer occupancy in the steady state. Recently, MPEG DASH standard has included a new Spatial Representation Description (SRD) [14] feature, to support the streaming of spatial sub-parts of a video to display devices, in combination with adaptive multirate streaming that is intrinsically supported by DASH. Following this advance, DASH has been further exploited to stream zoomable and navigable videos [15], virtual reality videos [16], and multiview videos [17]. Recent work have adopted novel technique, e.g., deep learning [18] and emerging computing architecture, e.g., fog/edge computing [19] to improve the rate adaptation for DASH.

Such mobile smart devices as smartphones, phablets, and tablets, undoubtedly reshape the way that users access Internet services, and therefore attract tremendous attention from academia. Existing studies tackle the challenges brought by the intrinsic mobile nature and enhance network protocols to accommodate seamless mobility [20], [21], inefficient retransmission [22], unstable channel quality [23], [24], and unexpected interference [25] in wireless and mobile networks. Yet, very few of them have attempted to improve network protocols for multimedia applications by utilizing rich interfaces and user interactions on mobile smart devices. Rather than optimizing one specific application, our work strives to enhance HTTP for a class of mobile Internet applications that make downloads outside user viewport.

3. MF-HTTP: Architecture and Design

3.1. Middleware Architecture

We first illustrate the opportunities from screen scrolling, the unique user–screen interaction on mobile platforms. Fig. 4 shows an example of screen scrolling process. Along the time axis, the solid line segments indicate the user/device actions, and the dashed line segments indicate when the objects enter the user viewport. From $t0$ to $t1$, the viewport changes as the user moves his/her finger. Once the finger is released at $t1$, the following scrolling process is

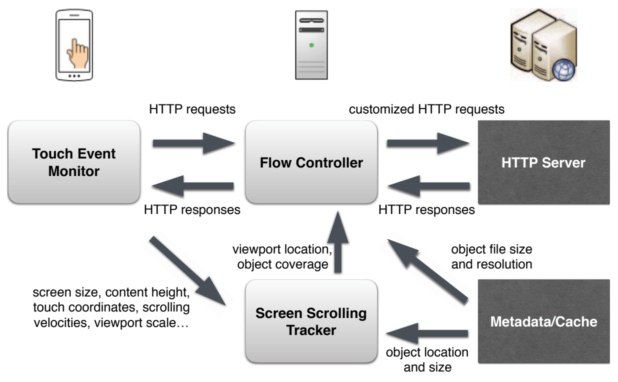


Figure 5: Middleware architecture

predetermined. Thus, at $t1$, we can accurately calculate the viewport's movement and predict object A's exit and object B and C's entrance in the viewport, which will be addressed in detail in later subsections. Given such information, better download arrangements can be made for the media objects in advance.

Our middleware consists of three modules: touch event monitor, screen scrolling tracker and flow controller, each of which will be elaborated in the following subsections. The main work flow is shown in Fig. 5. The touch event monitor attaches on the target mobile app to collect user touch data, which will be sent to the screen scrolling tracker. The middleware server that holds the other two modules can be either a remote content server or a forward or reverse proxy. With the information of user touch and device configuration, the screen scrolling tracker traces and predicts the viewport's movement. Further, given the location and size of the viewport and those of the media objects, the coverages of the media objects in the viewport can be calculated. Finally, with the full knowledge of the screen scrolling process, the flow controller is able to determine the optimal download policy.

3.2. Touch Event Monitor

The touch event monitor is a light-weight module that collects the device specification and configuration (e.g., screen size, pixel density, viewport scale) as well as the user touch data. As the user touch data can only be obtained from the mobile device, this module is designed to locate on the mobile client and provide interfaces for the mobile app developers to feed the user touch data. The collected information and data are sent to the screen scrolling tracker, which only introduces negligible traffic overhead.

In general, there are 3 types of input user gestures: click, drag, and fling, the last two of which can result in screen scrolling animation. Each gesture can be identified by a series of touch events. By detecting and collecting the information about the user's finger touch and release on the screen, the initial scrolling velocity on x axis (denoted as v_x) and that on y axis (denoted as v_y) can be calculated as the displacement divided by the touch time in two axes respectively.

3.3. Screen Scrolling Tracker

3.3.1. Scrolling Animation Philosophy. As we will discuss the practical issues for the implementation in the next section, we first investigate the philosophy of animating the screen scrolling for most mobile operating systems, which is to gradually decelerate the scrolling speed until it reaches zero if there is no other finger touch detected during the deceleration. Taking Android OS as an example, we next show how to calculate the viewport movement and the media object coverage during the scrolling process. Given the user touch data, the initial scrolling speed can be obtained as $v = \sqrt{v_x^2 + v_y^2}$. Android OS uses a threshold for the initial scrolling speed to distinguish between a drag and a fling, whose default value is 50 $pixels/second$ and can be scaled under different configurations based on the actual screen resolution.

For *dragging*, the screen scrolling speed will experience a uniform deceleration, which can be easily interpreted given the deceleration parameter and initial speed. As the deceleration of a dragging event is usually short and has very limited impact on viewport movement, we focus on analyzing the case of *flinging*. If a fling is detected, the deceleration will change with the scrolling speed. Given the initial scrolling speed v, the total fling duration $T(v)$ and the total fling distance $D(v)$ (the viewport displacement caused by the fling) can be calculated by using the following equations:

$$l(v) = log[0.35 \cdot v/(Fric \cdot P_{COEF})], \tag{1}$$

$$T(v) = 1000 \cdot exp[l(v)/(D_{RATE} - 1)], \tag{2}$$

$$D(v) = Fric \cdot P_{COEF} \cdot exp[D_{RATE}/(D_{RATE} - 1) \cdot l(v)], \tag{3}$$

where $D_{RATE} = log(0.78)/log(0.9)$, $Fric$ denotes the friction parameter with the default value as 0.015, and $P_{COEF} = G \cdot 39.37 \cdot ppi \cdot 0.84$. To compute P_{COEF}, G is the gravity of the Earth with a constant value of 9.80665 m/s^2, 39.37 is used for the conversion between meters and inches, and ppi denotes pixel density for the specific mobile device. Note that, as the basis of the following analysis, the above equations are obtained from our analysis of Android OS source code.

3.3.2. Viewport Displacement. Assume that, at time t, which denotes the time elapsed since the scrolling starts, the scrolling speed decreases to v'. From Eq. 2 and Eq. 3, we can have

$$D(v) = Fric \cdot P_{COEF} \cdot (T(v)/1000)^{D_{RATE}}. \tag{4}$$

Given $t = T(v) - T(v')$, the viewport displacement at time t can be calculated as

$$d(t) = D(v) - Fric \cdot P_{COEF} \cdot [(T(v) - t)/1000]^{D_{RATE}}. \tag{5}$$

Upon obtaining $d(t)$, we can further calculate the viewport displacement on x and y axis as $d_x(t) = d(t) \cdot \frac{v_x}{v}$ and $d_y(t) = d(t) \cdot \frac{v_y}{v}$, respectively. Note that, as $d(t)$ can have any direction, which is usually the same as (or opposite to) the direction of the user's finger touch movement, $d_x(t)$ and $d_y(t)$ can be either positive or negative.

3.3.3. Objects Involved in Viewport Movement. As the viewport and the rendered media objects (e.g., objects in a web page) are usually rectangular or bounded by rectangular boxes, let (x_p^0, y_p^0) be the original coordinates of the left-top vertex of a viewport, and w_p and h_p be its width and height. The viewport can be then uniquely defined. Similarly, we define (x_i, y_i), w_i, and h_i as the coordinates of the left-top vertex, the width, and the height of a media object i, respectively.

To identify the media objects covered by a scrolling process, we first determine the area covered by the viewport movement. Given the viewport displacement calculated above, the final location of the viewport's vertices can be obtained. As the viewport can move in any direction in a 2-D plane, the mathematical description of the covered area depends on the specific situation. For simplicity, we study the case of $D_x(v) = D(v) \cdot \frac{v_x}{v} > 0, D_y(v) = D(v) \cdot \frac{v_y}{v} > 0$ (other cases can be studied similarly), in which the covered area is surrounded by the boundary consisting of 6 intersected line segments: (1) $x = x_p^0$; (2) $y = y_p^0$; (3) $x = x_p^0 + w_p + D_x(v)$; (4) $y = y_p^0 + h_p + D_y(v)$; (5) $y = \frac{D_y(v)}{D_x(v)}(x - x_p^0) + y_p^0 + h_p$; (6) $y = \frac{D_y(s)}{D_x(v)}(x - x_p^0 - w_p) + y_p^0$.

To decide whether object i appears in such a bounded area, we check its four vertices to see if it intersects or is located inside. Since the four vertices are correlated, we can further evaluate the case based on the location of one vertex. Specifically, given this boundary, we can then determine object i located in/intersecting the covered area, if (x_i, y_i) meets the following conditions: (1) $x_p^0 - w_i < x_i < x_p^0 + w_p + D_x(v)$; (2) $y_p^0 - h_i < y_i < y_p^0 + h_p + D_y(v)$; (3) $\frac{D_y(v)}{D_x(v)}(x_i - x_p^0 - w_p) + y_p^0 - h_i < y_i < \frac{D_y(v)}{D_x(v)}(x_i + w_i - x_p^0) + y_p^0 + h_p$.

As we are now able to filter the media objects that are involved in a scrolling process, intuitively, the media objects that never appear in the viewport can be omitted for downloading or considered with low priority, which likely causes no difference in user QoE.

3.3.4. Object Coverage in Viewport. For those media objects that appear in the viewport, calculating how much area each of them covers is a straight-forward evaluation of its significance to user's multimedia viewing experience. We next show how to compute the coverage of a media object i in the viewport at a given time t. Based on the analysis in the previous subsections, we have that, at time t, the left-top vertex of object i is moved to $(x_p(t), y_p(t)) = (x_p^0 + d_x(t), y_p^0 + d_y(t))$. Similarly, we consider object i appearing in the viewport at time t, if the two following conditions are satisfied: (1) $x_p(t) - w_i < x_i < x_p(t) + w_p$ and (2) $y_p(t) - h_i < y_i < y_p(t) + h_p$. If object i is identified in the viewport, we can further calculate how much area it covers. Let $s_i(t)$ be the coverage of object i in the viewport at time t, which can be obtained as:

$$s_i(t) = [\min(y_i + h_i, y_p(t) + h_p) - \max(y_i, y_p(t))] \cdot [\min(x_i + w_i, x_p(t) + w_p) - \max(x_i, x_p(t))]. \quad (6)$$

3.4. Flow Controller

The flow controller determines and executes the optimal download policy for the media objects identified in the last step. We next present the formulation of the download optimization problem, which is solved in this module.

Consider n media objects that are involved in a screen scrolling event. A media object can be an image in a web page or a video segment in a DASH stream. To accommodate the heterogeneity of mobile platforms, the service/content providers usually offer multiple versions of media objects, e.g., images/video segments with different qualities. Assume that each object $i \in [1, n]$ have m versions ordered increasingly by resolution. Let t_i be the time when object i first appears in the viewport. Assume that the media objects are indexed based on the order in which they enter the viewport, which implies $t_1 \le t_2 \le ... \le t_n$. Let $B(t)$ be the available bandwidth at time t and $f_{i,j}$ be the file size of object i with resolution r_j ($j \in [1, m]$). We further define the cost function as $c(f_{i,j})$, which denotes the cost of download with the given file size. We use $k_{i,j} \in \{0, 1\}$ to denote the download policy for the given object, where the binary variable $k_{i,j} = 1$ indicates the object i of version j will be downloaded, and $k_{i,j} = 0$ otherwise.

3.4.1. Performance Metrics. We propose two metrics to evaluate the performance gain as well as the download cost for a media object, namely the QoE model and the cost model.

Based on Section 3.3.4, object i covers a faction $\frac{s_i(t)}{S}$ of the viewport at time t, where S is the area of the viewport. In practice, the user QoE is not only influenced by a media object's coverage and resolution, but also depends on how long the object stays in the viewport. Following this intuition, our QoE model consists of two parts. The first part $Q1(i, j)$ weights the object based on its coverage during the screen scrolling, which can be calculated as the normalized integral of $s_i(t)$ in discrete time with resolution r_j:

$$Q1(i, j) = \frac{1}{T(v)} \frac{r_j}{r_m} \sum_{t=1}^{T(v)} \frac{s_i(t)}{S} = \frac{1}{T(v)} \frac{1}{S} \frac{r_j}{r_m} \sum_{t=1}^{T(v)} s_i(t), \quad (7)$$

where $S = w_p \cdot h_p$. The terms in the denominator are used to normalize $Q1(i, j)$ so that its value is between 0 and 1.

The second part $Q2(i)$ is an binary indicator which checks whether the object appears in the final viewport when the screen scrolling stops:

$$Q2(i) = \mathbb{1}_{[s_i(T(v))>0]}, \quad (8)$$

where $\mathbb{1}_{[\cdot]}$ is the indicator function.

The QoE metric of object i with resolution r_j is defined as a weighted sum of the two parts defined above:

$$Q_{i,j} = a \cdot Q1(i, j) + b \cdot Q2(i). \quad (9)$$

To simply our QoE model, we set $a = b = 1/2$, so that $Q_{i,j}$ is between 0 and 1, and the QoE score of the object in the final viewport will never be lower than that of the object out of the viewport.

The performance gain comes with a price. The download cost of a object can be obtained from the cost function $c(f_{i,j})$ given the file size $f_{i,j}$. We calculate the normalized cost for downloading object i with resolution r_j as

$$C_{i,j} = c(f_{i,j})/c_M \quad (10)$$

where c_M is the highest download cost during the scrolling process. As c_M is reached when all the involved media objects are downloaded at the highest resolutions or the bandwidth is completely consumed, it can be calculated as $c_M = c(\min(\sum_{i=1}^{n} f_{i,m}, \sum_{t=1}^{T(v)} B(t)))$. We keep the cost model generic so that it can be easily adapted to different practical scenarios.

3.4.2. Performance Optimization. The goal is to generate the optimal download policy for all the media objects, which maximizes the QoE gain and minimizes the download cost. The objective function can be formulated as

$$\sum_{i=1}^{n} \sum_{j=1}^{m} k_{i,j}(p \cdot Q_{i,j} - q \cdot C_{i,j}) =$$
$$\sum_{i=1}^{n} \sum_{j=1}^{m} k_{i,j}\left(\frac{p}{2} \frac{1}{2T(v)} \frac{1}{S} \frac{r_j}{r_m} \sum_{t=1}^{T(v)} s_i(t) + \frac{p}{2} \mathbb{1}_{[s_i(T(v))>0]} - q \frac{c(f_{i,j})}{c_M}\right), \quad (11)$$

where p and q are the weighting parameters.

For the proposed optimization problem, the following two constraints must be satisfied.

(1) Each object is downloaded once at most:

$$\forall i \in [1, n], \sum_{j=1}^{m} k_{i,j} \le 1. \quad (12)$$

(2) The bandwidth should be enough to download the objects in time:

$$\forall i' \in [1, n], \sum_{i=1}^{i'} \sum_{j=1}^{m} k_{i,j} \cdot f_{i,j} \le \sum_{t=1}^{t_{i'}} B(t). \quad (13)$$

The first constraint ensures that no more than one copy (with a certain resolution) of each object can be downloaded. The second constraint implies that, when any object i' appears in the viewport at time $t_{i'}$, there should be enough bandwidth to download it and all the other selected objects that enter the viewport before it. Given the download policy, the underlying scheduling scheme hinted by Eq. 13 is to schedule the download in the same order that the objects are requested in the application.

We solve the formulated optimization problem by converting it to a variation of the 0-1 Knapsack problem.

Define the value of object i with with resolution r_j as $v(i,j) = p \cdot Q_{i,j} - q \cdot C_{i,j}$, its weight as $w(i,j) = f_{i,j}$, and the maximum weight capacity as $W(t') = \sum_{t=1}^{t'} B(t)$. The key difference is that, in our problem, $W(t')$ (the available bandwidth till a given time t') varies with time. Define $M[i,l]$ as to be the maximum value that can be attained with weight less than or equal to l using first i items. Inspired by the solution of 0-1 Knapsack problem, we solve the formulated problem by dynamic programming, in which the recursive equation is:

$$\begin{aligned} M[i,l] = \max_{j \in [1,m]} \{ & M[i-1, \min(l, W(t_{i-1}))], \\ & M[i-1, \min(l - w(i,j), W(t_{i-1}))] + v(i,j)\}, \end{aligned} \quad (14)$$

and the initial setting is:

$$M[i,l] = 0, \forall i \in [1,n], l \in [0, W(t_n)]. \quad (15)$$

In the modified solution to our problem, $v(i,j)$ and $w(i,j)$ are initialized according to the definitions. The maximum weight capacity is carefully updated as it increases with larger i. The time complexity of the designed algorithm is $O(nm^2 W(t_n))$. Although the algorithm is executed whenever a user touch event is detected, given that any user gesture can only affect a limited number of media objects for a very short time, n, m, and $W(t_n)$ are most likely to have very small values, and thus our algorithm can run efficiently.

4. Middleware Implementation

In this section, we present and discuss the implementation issues for the MF-HTTP middleware.

4.1. Touch Event Monitor

The touch event monitor is implemented on the mobile side. The middleware should introduce least modifications on mobile clients and HTTP servers for multimedia services. As the user touch events need to be collected from mobile devices, integrating the touch event monitor to the client-side software, typically a mobile app, is however inevitable. It thus should be effortless for general mobile app developers to implement and integrate the touch event monitor, which employs simple and standard APIs.

Taking Android platform as an example, the user interface for an Android app is built using a hierarchy of layouts (`ViewGroup` objects) and widgets (`View` objects). The widget that occupies (a part of) the device's screen can listen to and handle user touch events on it. The idea is to find the proper View object class in the application's source code, which can also be provided by developers, and attache this module to the scrollable View objects that display the scrolling effect in response to touch gestures. Next, we override the `onTouchEvent` method of the target View objects to handle key touch screen motion events such as `ACTION_DOWN`, `ACTION_MOVE`, and `ACTION_UP`, based on which the input gesture can be identified as a fling or a drag. We further decouple the scrolling animation from the original mobile application to produce a well-controlled scrolling process, by employing the `Scroller` class to animate scrolling over time using platform-standard scrolling physics (friction, velocity, etc.). The corresponding scrolling offsets for both drag and fling events are calculated and sent to the screen scrolling tracker.

4.2. Screen Scrolling Tracker

The screen scrolling tracker requires certain knowledge about the mobile multimedia services. As such knowledge can be hardly collected from client side, we implement the screen scrolling tracker on the middleware server. This module can thus access the related data on the cache of the middleware server or directly from the multimedia service server with very low cost.

During the consumption of mobile Internet services, the screen scrolling tracker first retrieves the device specification and configuration information from the touch event monitor. Second, the user touch data is constantly transmitted to the module through a TCP socket connection. Based on the analysis in Section 3.3, the viewport locations and object coverages the during the scrolling process can be calculated. Whenever a touch event with a newer timestamp arrives, the simulation of current/unfinished scrolling is aborted.

4.3. Flow Controller

The flow controller is also implemented on the middleware server and runs in a separate thread from the screen scrolling tracker sharing necessary global variables. We adopt the *mitmdump*[1] tool, run MF-HTTP as a man-in-the-middle proxy, and redirect the mobile client's HTTP traffic to the middleware server. We develop a Python script to run with *mitmdump* on the middleware server to identify and handle the HTTP traffic generated by the target mobile multimedia service, and modify the script with the `@concurrent` setting so that MF-HTTP proxy works in a non-blocking mode to process multiple HTTP requests at the same time. The control of media object downloading is realized by modifying, deferring, or blocking the target HTTP headers, requests and responses.

The flow controller executes the optimization logic presented in Section 3.4. It is worth noting that, our optimization model of MF-HTTP can adapt to various user requirements and different practical scenarios, as the cost function and the weights of performance metrics are adjustable. Moreover, as the inputs, the outputs, and the interfaces employed by MF-HTTP are simple and straightforward, users of MF-HTTP can design and implement their own optimization logics.

5. Case Studies

MF-HTTP targets to optimize a class of mobile Internet applications that make downloads outside user viewport.

1. https://mitmproxy.org/

For different applications, the knowledge assumed from the last section can be carefully obtained or bypassed. We next present concrete case studies on two representative applications, web browsing and 360-degree video streaming, and discuss the light and practical adjustments for the MF-HTTP prototype.

5.1. Mobile Web Browsing

The media objects that are critical to mobile web browsing experience are the images in the web page (the videos are often marked by their thumbnails before being selected to play). Therefore, our scrolling-aware HTTP middleware can be adjusted for the download of images.

5.1.1. Implementation Details. We develop a light-weight web browser based on the `WebView`[2] class from Android API with the touch event monitor integrated, whose `onTouchEvent` method is customized as described in the last section. Note that `WebView` share the same rendering engine as Chrome for Android. To collect the information needed by the screen scrolling tracking, we use Chrome's developer tool to emulate the web page layout under different screen sizes. Every time the web page is requested, this reference is built and updated, so that the middleware keeps refreshing the information of web page layouts proactively. The dependencies between web objects can be profiled using tools such as Wprof [26]. As mentioned, we focus on and modify the download of images, among which dependencies rarely exist. We keep the download sequence of styling rules and scripts unchanged to ensure that MF-HTTP does not violate the dependencies of the web page.

5.1.2. Optimization Workflow. As bandwidth is rarely the bottleneck for web browsing [2], we release the bandwidth constraint from the formulated problem in Section 3.4. Rather than modifying the hardcore of the web engine to have fine-grained control over the download of web page objects, MF-HTTP adopts simple but effective approaches. The flow controller is adjusted to execute the following work process. (1) When a web page is requested, as the images' source URLs are already collected, the flow controller maintains a block list of source URLs for the images outside the initial viewport. (2) For each data flow, it checks the header to see if the requested URL is in the block list. If so, it blocks the HTTP request. (3) By receiving the updates of viewport location, viewport displacement, and object coverage from the screen scrolling tracker, the flow controller is able to determine whether an image appears in the viewport in the scrolling process. If the image is never involved in the scrolling, it remains in the block list. For web browsing, the images in the viewport before and after its moving are the most crucial to user QoE. Thus such images are in the current viewport or in the final viewport when the scrolling stops are identified and removed from the block list. For the images that appear but fail to stay in the viewport, the flow controller evaluates their values $p{\cdot}Q_{i,j} - q{\cdot}C_{i,j}$ as in Eq. 11. The images with positive values are allowed to download, while others with negative values are kept in the block list. (4) Whenever a new user touch event is detected, the flow controller receives the updates from the screen scrolling tracker and reacts in the same logic as described above.

5.2. 360-Degree Video Streaming

Different from web browsing, video streaming is bandwidth-sensitive and -intensive, which can also benefit from MF-HTTP. 360-degree videos provide users with panoramic views and create unique viewing experience, which are now popular on major video sharing platforms such as YouTube and Facebook. In our case study, 360-degree videos are consumed as navigable videos from mobile clients with limited viewports. As shown in Fig. 2, the user's viewport is significantly confined by the device's size of display, while the whole raw frame is streamed back with large portions outside the viewport. We next discuss how to enable the key idea of MF-HTTP for 360-degree video streaming.

5.2.1. Implementation Details. The touch event monitor is implemented and attached to an open source 360-degree video player[3]. In this case, we directly pin the touch event monitor to the player's main View object class, which extends the `TextureView` class from Android API, to handle the touch events and output the user gestures and the scrolling offsets.

As the delivery schemes for traditional Internet videos are inefficient for 360-degree videos and provides no flexibility to adapt to the change of user's Region Of Interest (ROI), we use the tile-based streaming approach [27], [28] to adapt the user's viewport. To map the viewport in the spherical view to the tiles of the rectangular raw video frame, we adopt the widely used equirectangular projection [29] as the sphere-to-play mapping scheme, which unwraps a sphere with a radius of r on a 2D rectangular plane with the dimensions of $(2\pi r, \pi r)$.

5.2.2. Optimization Workflow. As user interest for video contents is usually coherent in one viewing session, 360-degree video users produce much more drag events than fling events if there are any. Given that a DASH segment's duration is usually much longer than a scrolling, instead of interpreting viewport movement, the screen scrolling tracker only keeps a close track of the viewport's current location by monitoring the user drag events. The tiles are thus classified into two categories: tiles that appear in the viewport and tiles that have no overlap with the viewport. In the original formulation, media objects with different resolutions are evaluated and selected separately, which can be simplified here by setting $Q_{i,j}$ to be binary, as the tiles that are appear in the viewport should be of the same quality so as to

2. https://developer.android.com/reference/android/webkit/WebView.html

3. https://github.com/fbsamples/360-degree-video-player-for-android

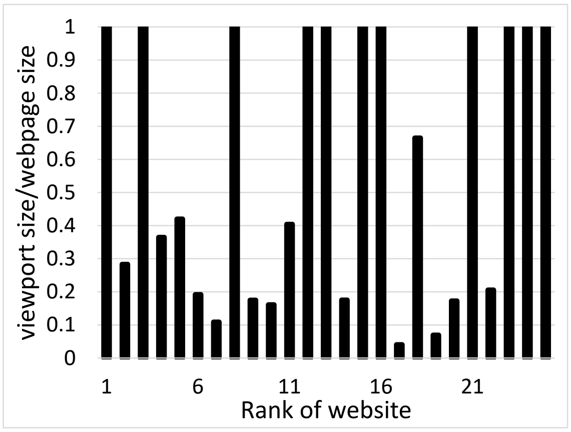


Figure 6: viewport size/webpage size

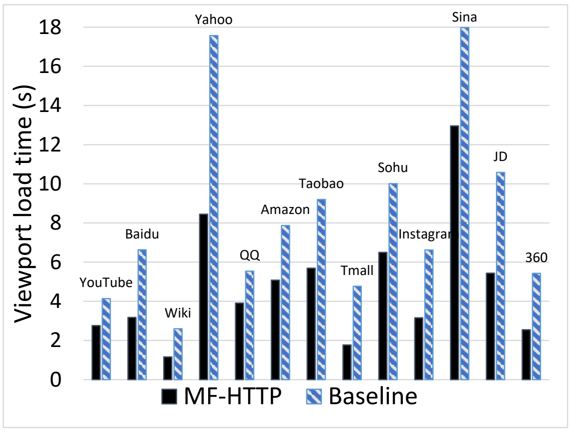


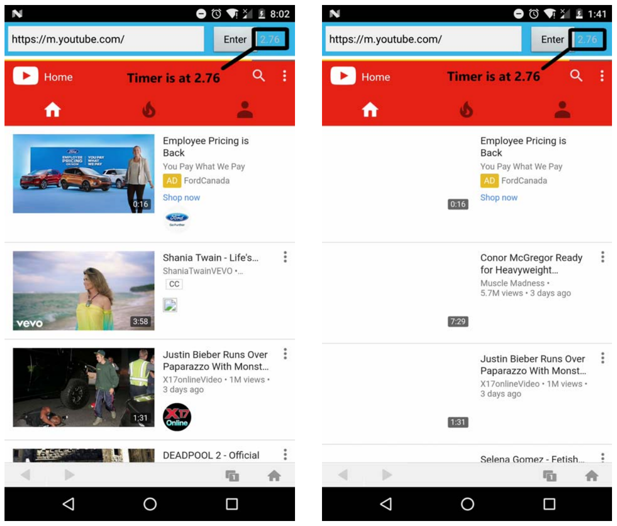
Figure 7: Viewport load time

provide better and consistent QoE for video watching. As the design of more sophisticated algorithms specifically for 360-degree video DASH streaming optimization is out of the scope of this paper, therefore, for illustration purpose, here the flow controller adopts the following principle for tile-based 360-degree video DASH streaming: given the available bandwidth, minimize the quality of the tiles that have no overlap with the viewport and maximize the quality of the tiles that appear in the viewport.

6. Performance Evaluation

6.1. Experiments for Web Browsing

6.1.1. Test Platforms and Settings. In this subsection, we evaluate the performance improvement of our MF-HTTP middleware for the mobile web browsing case study. We use a Nexus 6 phone running Android 7.0 as the mobile client, and a desktop computer with Intel Core i7-3770 CPU @ 3.40GHz $\times$ 8 and 16 GB memory running Ubuntu 14.04 LTS as the MF-HTTP middleware. As the touch interface



(a) MF-HTTP enabled (b) MF-HTTP disabled

Figure 8: Screenshots of two browsing sessions with the same timestamp

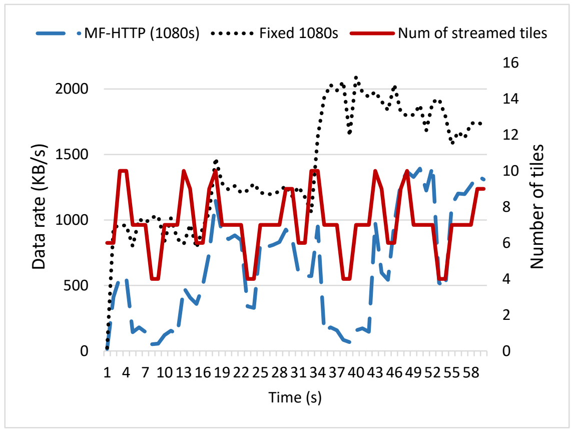


Figure 9: A sample trace of one video watching session

has no dramatic change across different generations of hardware and software platforms, similar experiment results are observed with other phones. The mobile client is connected to MF-HTTP through an IEEE 802.11 WLAN router. Both of the middleware and the router locate in the university campus network, and the network condition is good and stable. We use the browser to access the Alexa's top 25 global websites [30]. Each browsing session consists of default viewport loading followed by a random scrolling touch. We set the weight of cost metric $q = 0$ to maximize the viewing experience. To better trace the loading performance, we add a timer to the browser. We compare the performance of browsing with and without MF-HTTP enabled.

6.1.2. Results. We first check the default viewport size against the webpage size, where Fig. 6 shows the ratio of top 25 websites. In particular, there are 11 websites

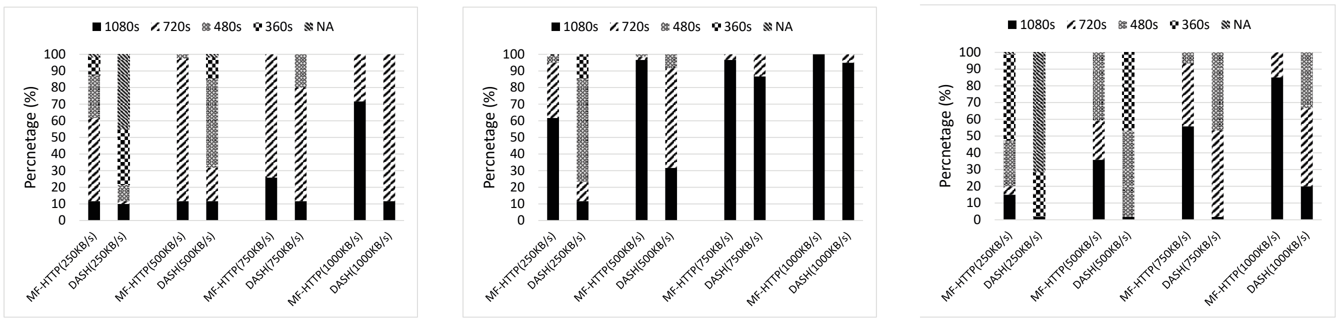


Figure 10: Video quality constitutions with different bandwidth (Video 1 to 3 from left to right)

having full-size viewports and 14 websites having limited-size viewports. The 11 websites with full-size viewports are mainly search engines (e.g., Google global and 4 other regions, Microsoft Live) and login pages (e.g. Facebook, Twitter, and Linkedin), while the other 14 websites stand for more general and various types of websites, from which mobile users can only view a small portion of the whole page (as low as 4.1% in the case of Sohu). It is worth noting that, some websites (e.g., YouTube and Yahoo) have pages of varying length, which will always load new contents when users hit the bottom. In theory, these websites can have unlimited length of contents, and thus the impacts of limited viewports become even more notable.

Rather than using page load time, one of the major performance metrics for web browsing, we use a new metric, *viewport load time*, which is the elapsed duration when the viewport is fully loaded. We record the screen of the test smartphone and replay the video to track the loading process as well as the timer. As shown in Fig. 7, MF-HTTP significantly improves the loading performance for the websites with limited-size viewports as it prioritizes the downloads of the objects in the viewport. In average, MF-HTTP reduces the viewport load time by 44.3%. Fig. 8 further shows two screenshots taken at the same time for two YouTube browsing sessions using different approaches. In this example, MF-HTTP finishes loading the viewport, while the baseline approach still struggles in downloading objects disregarding whether they are in the viewport.

6.2. Simulations for 360-Degree Video Streaming

6.2.1. Data Collection. In this subsection, we evaluate the performance improvement of our MF-HTTP middleware for the 360-degree video streaming case study. We obtain three test videos from YouTube[4] at 4 different resolutions: 1080s, 720s, 480s, and 360-degrees ("s" stands for spherical). We recruit 10 volunteers to watch each video on the Nexus 6 phone and modify the 360-degree video player to record user touches during the video watching. Each video watching session lasts for 1 minute. To support tile-based DASH streaming, we use the *GPAC*[5] toolbox to slice and package the 360-degree videos into into 4×4 tiles. We further do a segmentation on the encoded tile-based videos and generate segments with duration of 1 second as well as the MPD files, which are ready to be DASHed. The viewport movement and the resulting tile and rate selection are generated by MF-HTTP based on the collected traces of user touches.

6.2.2. Results. We first check the bandwidth consumption for MF-HTTP and plot a sample trace of one 1080s video watching session in Fig. 9. Compared to the baseline approach, streaming the whole frame with a fixed resolution without considering the viewport, MF-HTTP significantly reduces the bandwidth consumption. The result also suggests that MF-HTTP does not necessarily share network load peaks with the baseline steaming approach, and its bandwidth consumption is closely affected by the number of tiles that appear in or overlap the viewport, as the valleys of the two curves match in Fig. 9.

We next vary the available bandwidth from 250KB/s to 1000KB/s to examine the streaming quality of MF-HTTP, and compare its performance with a greedy DASH scheme that maximizes bandwidth usage and streams at the highest possible resolution. Fig. 10 shows how much time (in percentage) the test videos are played at different resolutions using two streaming approaches, where "NA" denotes the bandwidth is insufficient for any of the given resolutions. As shown, MF-HTTP constantly outperforms the greedy DASH scheme under all bandwidth conditions for all test videos. MF-HTTP can maintain good video quality when the bandwidth is low, and it quickly responds to the increase of the bandwidth. This result suggests that MF-HTTP can more efficiently utilize the network resource to focus on downloading the high quality video segments in the viewport.

7. Conclusion

In this paper, we presented the Mobile-Friendly HTTP middleware (MF-HTTP). MF-HTTP acts at the application

4. YouTube IDs of the three test videos are: -xNN-bJQ4vI, rG4jSz_2HDY, wXeKxY3F0sE.

5. https://gpac.wp.imt.fr/home/

layer and interprets screen scrolling processes on mobile devices by tracking user touch screen operations. Based on the information from the screen scrolling processes, MF-HTTP further optimizes the downloading of media objects to improve QoE and cost efficiency. To achieve this, we first demystified the detailed screen scrolling philosophy in mobile system and showed how to precisely break down the viewport movement. We then identified the key influential factors for media object downloading, and developed an optimal downloading scheme. We further discussed practical issues towards the implementation of MF-HTTP. Finally, we implemented a prototype based on Android platforms and conducted two concrete case studies, namely, web browsing and 360-degree video streaming, to demonstrate the superior performance of MF-HTTP.

Acknowledgment

This publication was made possible by NPRP grant #[8-519-1-108] from the Qatar National Research Fund (a member of Qatar Foundation). The findings achieved herein are solely the responsibility of the authors. It is also supported by the Natural Sciences and Engineering Research Natural Sciences and Engineering Research Council (NSERC) of Canada.

References

[1] Cisco, "Global mobile data traffic forecast update, 2016-2021 white paper," Cisco Visual Networking Index, Tech. Rep.

[2] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie, "How far can client-only solutions go for mobile browser speed?" in *Proceedings of ACM WWW*, 2012, pp. 31–40.

[3] C. Zhou, Z. Li, and Y. Liu, "A measurement study of oculus 360 degree video streaming," in *Proceedings of ACM MMSys*, 2017, pp. 27–37.

[4] L. Zhang, F. Wang, and J. Liu, "Mobile instant video clip sharing with screen scrolling: Measurement and enhancement," *IEEE Transactions on Multimedia*, 2018.

[5] F. Qian, K. S. Quah, J. Huang, J. Erman, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck, "Web caching on smartphones: ideal vs. reality," in *Proceedings of ACM MobiSys*, 2012, pp. 127–140.

[6] V. N. Padmanabhan and J. C. Mogul, "Using predictive prefetching to improve world wide web latency," *ACM SIGCOMM Computer Communication Review*, vol. 26, no. 3, pp. 22–36, 1996.

[7] M. Jiang, X. Luo, T. Miu, S. Hu, and W. Rao, "Are http/2 servers ready yet?" in *Proceedings of IEEE ICDCS*, 2017, pp. 1661–1671.

[8] R. Netravali, A. Goyal, J. Mickens, and H. Balakrishnan, "Polaris: Faster page loads using fine-grained dependency tracking," in *Proceedings of USENIX NSDI*, 2016.

[9] M. Butkiewicz, D. Wang, Z. Wu, H. V. Madhyastha, and V. Sekar, "Klotski: Reprioritizing web content to improve user experience on mobile devices." in *Proceedings of USENIX NSDI*, 2015, pp. 439–453.

[10] C. Kelton, J. Ryoo, A. Balasubramanian, and S. R. Das, "Improving user perceived page load times using gaze." in *Proceedings of USENIX NSDI*, 2017, pp. 545–559.

[11] J. Ren, L. Gao, H. Wang, and Z. Wang, "Optimise web browsing on heterogeneous mobile platforms: a machine learning based approach," in *Proceedings of IEEE INFOCOM*, 2017.

[12] G. Tian and Y. Liu, "Towards agile and smooth video adaptation in dynamic http streaming," in *Proceedings of ACM CoNEXT*, 2012, pp. 109–120.

[13] T.-Y. Huang, R. Johari, N. McKeown, M. Trunnell, and M. Watson, "A buffer-based approach to rate adaptation: Evidence from a large video streaming service," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 187–198, 2015.

[14] O. A. Niamut, E. Thomas, L. D'Acunto, C. Concolato, F. Denoual, and S. Y. Lim, "Mpeg dash srd: spatial relationship description," in *Proceedings of ACM MMSys*, 2016, p. 5.

[15] L. D'Acunto, J. van den Berg, E. Thomas, and O. Niamut, "Using mpeg dash srd for zoomable and navigable video," in *Proceedings of ACM MMSys*, 2016, p. 34.

[16] M. Hosseini and V. Swaminathan, "Adaptive 360 vr video streaming: Divide and conquer," in *Proceedings of IEEE ISM*, 2016, pp. 107–110.

[17] K. Diab and M. Hefeeda, "Mash: Adaptive streaming of multiview videos over http," in *Proceedings of IEEE INFOCOM*, 2017.

[18] H. Mao, R. Netravali, and M. Alizadeh, "Neural adaptive video streaming with pensieve," in *Proceedings of ACM SIGCOMM*, 2017.

[19] Y. Im, J. Han, J. H. Lee, Y. Kwon, C. Joe-Wong, T. Kwon, and S. Ha, "Flare: Coordinated rate adaptation for http adaptive streaming in cellular networks," in *Proceedings of IEEE ICDCS*, 2017, pp. 298–307.

[20] A. Yadav and A. Venkataramani, "msocket: System support for mobile, multipath, and middlebox-agnostic applications," in *Proceedings of IEEE ICNP*, 2016, pp. 1–10.

[21] L. Zhang, F. Wang, and J. Liu, "Dispersing social content in mobile crowd through opportunistic contacts," in *Proceedings of IEEE ICDCS*, 2017, pp. 2276–2281.

[22] M. O. Khan, L. Qiu, A. Bhartia, and K. C.-J. Lin, "Smart retransmission and rate adaptation in wifi," in *Proceedings of IEEE ICNP*, 2015, pp. 54–65.

[23] A. Aqil, A. O. Atya, S. V. Krishnamurthy, and G. Papageorgiou, "Streaming lower quality video over lte: How much energy can you save?" in *Proceedings of IEEE ICNP*, 2015, pp. 156–167.

[24] L. Zhang, D. Fu, J. Liu, E. C.-H. Ngai, and W. Zhu, "On energy-efficient offloading in mobile cloud for real-time video applications," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 27, no. 1, pp. 170–181, 2017.

[25] C.-J. Liu and L. Xiao, "Rmip: Resource management with interference precancellation in heterogeneous cellular networks," in *Proceedings of IEEE ICNP*, 2016, pp. 1–10.

[26] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall, "Demystifying page load performance with wprof." in *Proceedings of USENIX NSDI*, 2013, pp. 473–485.

[27] N. Quang Minh Khiem, G. Ravindra, A. Carlier, and W. T. Ooi, "Supporting zoomable video streams with dynamic region-of-interest cropping," in *Proceedings of ACM MMSys*, 2010, pp. 259–270.

[28] M. Xiao, C. Zhou, Y. Liu, and S. Chen, "Optile: Toward optimal tiling in 360-degree video streaming," in *Proceedings of ACM MM*, 2017, pp. 708–716.

[29] "Equirectangular projection," https://en.wikipedia.org/wiki/Equirectangular_projection, accessed: 2017.

[30] "The top 500 sites on the web," http://www.alexa.com/topsites, accessed: 2017, July.