UNIVERSITY OF CALIFORNIA
RIVERSIDE

Managing Mobile Applications in Resource Constrained Settings

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Tuan Anh Dao

March 2017

Acknowledgments

I am grateful to my advisor, Prof. Srikanth Krishnamurthy, for his guidance, understanding and encouragement. I would not be able to finish my Ph.D. program without him.

I would also like to thank Prof. Harsha Madhyastha, and Prof. Amit K. Roy-Chowdhury, who have given me many valuable suggestions to finish my research projects. Furthermore, I am thankful to the contributions of all co-authors of my published conference and journal papers, whose contents are included in this dissertation. Specifically, chapter 2 and 3 were published in ACM CoNEXT 2014 and IEEE ICDCS 2015, respectively. The content of chapter 4 is under submission to IPSN 2017.

I am grateful to the members of my dissertation committee, who have given me many constructive and directive advices to finish this dissertation.

Finally, I want to express my deepest gratitude to my wife and my parents for their sacrifices, understanding and support. They have always been the source of my strength to finish this prolonged Ph.D. journey.

To my wife and my parents for all the support.

ABSTRACT OF THE DISSERTATION

Managing Mobile Applications in Resource Constrained Settings

by

Tuan Anh Dao

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, March 2017
Dr. Srikanth V. Krishnamurthy, Chairperson

Even though wireless and mobile devices have evolved with richer capabilities, resources in such devices are still limited, and thus in many cases, are insufficient to accomplish the tasks entrusted in them. The goal of my work is to improve the efficiencies of applications on wireless and mobile devices, focusing on resource constrained settings.

First, we develop TIDE, a user-centric framework that helps to identify high energy consuming applications on users' smartphones. TIDE identifies energy hungry applications by looking at the correlation between applications' activities and high drainage periods on the smartphones. Experiments on Android smartphones show that TIDE is able to identify correctly $\approx 90\%$ of the energy hungry applications, while imposing reasonably low energy overheads.

Subsequently, we develop a framework to identify redundant images uploaded from multiple wireless devices in bandwidth constrained networks, e.g., the destructed networks at natural disaster scenes. Our framework intelligently combines state-of-the-art vision techniques to identify redundant images uploaded to a server. Suppressing the transfer of redundant contents significantly lowers network load, so that the delay in transferring unique and important contents in such critical scenarios is reduced up to $\approx 44\%$.

We then design ACTION, a framework for accurate and timely object (e.g., human) detection in bandwidth constrained settings. In ACTION, the objects of interest are effectively detected at individual camera sensors. Metadata of detected objects is then aggregated at a designated fusion node to improve the detection accuracy. Most accurate information of each detected object is then chosen to upload to a central controller, while adhering to the bandwidth constraints. We show that ACTION helps reduce up to three folds the amount of transferred data, while still delivering important information to the central node.

Finally, we design EECS, a framework for adaptive detection algorithm selection in multi-camera settings. In EECS, only a subset of camera sensors is chosen to detect objects; further, the most energy efficient algorithm is assigned to each camera to reduce energy consumption, while still ensuring a desired accuracy. We show that EECS can be tuned to achieve the right trade-offs between energy efficiency and desired accuracy.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Wireless and mobile devices are widely used in many activities in our modern lives. Even though wireless devices have evolved with richer capabilities, resources in such devices are still limited, and thus in many cases, are insufficient to accomplish the tasks entrusted in them. For example, it is not very uncommon for a user to find out her smartphone running out of battery during a prolonged working day. As another example, in a surveillance system where wireless camera sensors connect to a central controller via a bandwidth limited network, the volume of data generated by the sensors might be significantly higher than the network capacity. In such cases, data cannot be timely and fully transferred back to the controller to provide situation-aware information about the scene.

Motivated by such concerns, our works have been focusing on improving the efficiency of mobile/wireless applications, especially in the resource (e.g., energy and bandwidth) constrained settings.

In our first project, our primary goal is to help normal smartphone users to understand the energy efficiency of the applications they have been using on the phones, and then identify the energy hungry applications. There are thousands of mobile applications currently on the market; however, we believe that normal users are not fully aware of the energy efficiency of the applications they have installed. For example, if the phone is draining battery faster than usual, among the running applications, it is not always possible for the user to tell which applications are the real culprit for such high drainage. For that reason, we develop TIDE, a framework to efficiently detect energy hungry applications on smartphones. As our target is the normal smartphone users, our framework does not require the user to root her device, which might void the phone warranty, or to make any modifications to the phone operating system (OS). Our user-centric framework is able to classify an application as energy hungry, or energy efficient, based on the actual usage pattern of the user on her particular phone, without using a crowd sourcing approach [1]. We conduct a thorough evaluation of our framework by a user-study which involves ≈ 20 Android smartphone users at UCR (IRB HS-13-076). The result shows that, TIDE is able to correctly identify 225/238 high energy consuming applications, while imposing only 0.5% of overhead on the average consumption of the phones battery per hour. Our implementation focuses mainly on Android smartphones, due to the open nature of this platform; however, the principles of the approach can be applied to other platforms as well.

In our second project, we pay attention to reduce the degree of redundancy in images/videos uploaded by multiple wireless devices in a bandwidth limited scenario, e.g., in a natural disaster rescue mission. Studies have shown that when a disaster happens, people produce and upload a great amount of media contents (images/videos) in order to report about the situation, or to assist the rescue team. In addition, autonomous robots could also be deployed to take pictures at the scene and upload those pictures to a central controller. Unfortunately, in such scenarios, available bandwidth is limited because the network infrastructure is usually partially destroyed by the disaster [2, 3]. The constraints in network connectivity, together with the high volume of uploaded data, would hinder communication, or even worse, cause network outages in such life threatening scenarios. Prior studies showed that the pictures and videos taken and uploaded in those scenarios eventually contain a high degree of redundancy, e.g., 50% of the images uploaded during the San Diego wildfire are nearly identical or similar [4], and thus, are redundant. Our primary goal is to develop a lightweight, yet powerful, framework that can detect if an about to be uploaded image has a similar version on the server. If that is the case, the server should notify the devices to suppress such uploads, or defer until the network is less congested. Doing so helps save precious bandwidth for unique content, which could be more useful in assisting the rescue mission. In our approach, only compact meta-data from the images is used to detect image similarity with a high accuracy rate. In the final phase of our approach, user feedback is also leverage to boost the true positive rate.

We implement and evaluate our approach on a 20-node Android smartphone testbed in various conditions with the Kentucky [5], and an US cities image data set that we put together. We find that our multi-stage approach for uploading images correctly identifies the presence of similar images on the server with ≈70% accuracy, while ensuring a low false positive rate of 1%. More importantly, our framework's suppression of uploads of similar content enables the network to tolerate 60% higher load (for target delay requirements), as compared to a setting without our framework.

In our third project, we focus mainly on the object detection problem, especially the human detection problem, which has significant meaning in rescue and tactical missions. Natural disasters usually have a high associated human cost; for example, the recent Nepal earthquake killed more than 8,000 people, injured more than 14,000 and over 300 people are still missing [6]. Today, advanced technologies can help in significantly enhancing search and rescue missions; sensors, often with camera capabilities can be deployed in the field, to provide situational awareness back to a central controller. However, in such scenarios, the bandwidth may be limited, and transferring video content from all of the cameras may not only be wasteful, but may delay the transfer of key information with regards to human victims. Furthermore, transferring all video may cause an inherent information overload on a human who mans the central controller. Thus, we seek to design a framework, that facilitates the transfer of proper situational awareness information from a camera network to a central controller, when subject to significant bandwidth constraints.

ACTION, our framework for timely and accurate human detection, consists of three main components: (i) Camera sensors where the actual object (e.g., human) detection happens, (ii) a fusion node to boost the accuracy and choose the best relevant information for uploading, and finally (iii) a central controller node where detection information is used to assist the rescue mission. At the sensor cameras, we leverage state-of-the-art computer vision techniques to detect the presence of objects (e.g., humans) with low overheads and high accuracy. Meta data information of detected objects are sent to a designated fusion node to improve accuracy. At the fusion node, detection information of the same humans, captured from different views/cameras is aggregated by using spatial and color features. For each detected object, the image frame with the highest confidence values (detection probabilities) are chosen to transfer to the controller node, adhering to the given bandwidth constraints.

We implement ACTION on Android devices preloaded with a dataset that consists of video sequences captured from 4 different cameras [7]. Our evaluations show that by considering views from multiple cameras, ACTION detects $\approx 20\%$ more humans than using the video from a single camera. Further, with multiple cameras, it achieves a very high accuracy rate of $\approx 90\%$ (with a single camera it can be at most $\approx 72\ \%$). In terms of resource usage, ACTION can reduce the bandwidth usage threefold, compared to uploading all the detected frames directly to the central controller. In addition, with ACTION, even though information from 4 cameras is used, the amount of transferred data is only ≈ 1.4 times higher than the amount of data transferred when one camera is used, while providing a significant higher detection accuracy. The energy overhead with ACTION is 102 J at the camera node and 39 J for the fusion node to process a video sequence of 3.2 minutes; this low consumption allows a camera node to last for about 19 hours (assuming a smartphone battery).

In our latest work, we design and implement EECS, a framework for energy efficient object detection in camera sensor networks. EECS supports the co-ordination across a set of camera sensors to achieve a desired object detection accuracy but while achieving significant energy savings. Specifically, the framework ensures that cameras do not all unnecessarily use highly optimal but energy heavy video processing algorithms for object detection. In essence, it facilitates the adaptive choice a subset of cameras, and causes some of the chosen cameras to use sub-optimal detection algorithms to conserve energy while still achieving the pre-defined desired accuracy. Our evaluations on 3 different datasets show that, EECS helps save more than 40% of the energy consumed compared to a case where all cameras use the optimal algorithm for detection and transfer key images relating to detected objects; however, it still achieves $\approx 86\%$ the accuracy achieved when the best algorithms are used at all of the camera nodes. EECS can be tuned to achieve the right trade-offs between energy efficiency and desired accuracy.

The rest of the dissertation is organized as follows. Chapter 2 describes TIDE, our framework to detect high energy applications on smartphones. Chapter 3 represents our work on building a lightweight yet effective framework for image similarity detection. Chapter 4 describes ACTION, our framework for timely and accurate object detection in bandwidth constraint settings. Chapter 5 presents our latest work, the EECS framework for energy efficient object detection. Finally, chapter 6 provides the summary and conclusions of my work.

Chapter 2

TIDE: A User-Centric Tool for Identifying Energy Hungry Applications on Smartphones

2.1 Introduction

While smartphones are evolving with richer capabilities and more powerful hardware, their batteries are not keeping up. Coupled with the explosion in the number of applications[1] for smartphones, this trend has left users distressed about how long their phone's battery lasts even after a full recharge. A report in 2012 [8] says that "Despite activities such as web browsing, watching videos, and using downloadable apps have become

[1]We use the terms app and application interchangeably.

(sic) an everyday part of smartphone use, their impact on battery performance is largely excluded from the data published by manufacturers."

Need for a user-centric app profiling tool: While there exist tools that try to quantify the energy consumption of smartphone apps, they are not user-centric. The target for these tools are software developers who want to check for power inefficiencies in their products before release. These tools either require the instrumentation of the smartphone with specialized external equipment (e.g., a power meter), or require modifications to the smartphone's operating system (OS). A typical user cannot perform either. In addition, these tools need to be run continuously to track an app's operations, and hence consume significant energy themselves.

Instead, it is desirable to have a tool that is capable of reporting which apps on a user's phone dominate battery consumption. This tool should not simply focus on detecting apps that have energy bugs [9] or ignore user-specific factors that influence battery drainage (e.g., as in [1]); for each user, it should identify apps that consume a disproportionate amount of energy *on that user's phone.* When run on a particular user's phone, one could envision this tool as roughly categorizing every app as energy-hungry, energy-thrifty, or energy-moderate, based on how the app is used by the user and the environment in which it is used. Once energy hungry apps are identified, a user can reduce her use of or cease to use such apps when needed.

Challenges: Unfortunately, developing such a user-centric tool to detect high energy apps on smartphones is a hard problem. Since normal users will be reluctant to install modifications to the smartphone OS (this voids the phone's warranty), the identification of energy-hungry apps must be based on information exported by the OS to the application layer. This information is however insufficient for directly measuring the precise amount of resources, and hence energy, consumed by any specific app. First, the OS only reports aggregate resource usage metrics to the application layer. Second, at the application layer, one can only measure the durations between instances when the residual battery life decreases by 1%. During any one such interval, there are typically several apps running simultaneously on the phone.

On the other hand, offline calibration of an app's energy consumption is insufficient, since the determination as to whether a specific app is energy hungry critically depends on how and in what setting the app is used. First, battery drainage is affected by a variety of factors, including the features of the device, the processing invoked by each app, and network conditions. Thus, the power consumed by the same app can significantly vary across different settings. In addition, different users may interact with an app in different ways (e.g., the energy consumed by a video sharing app can differ based on whether the user views videos of high or low quality). Therefore, an app that is energy hungry on one user's phone may not be so on another's.

Due to all of the above factors, it is a significant challenge to tease out the apps that are the real culprits with respect to energy drainage on a particular user's phone.

Our contributions: In this paper, we first undertake an extensive measurement study on a testbed of 22 Android phones. Our study demonstrates how differing network conditions, device features, and usage patterns influence the energy consumed by apps. Our study also highlights the challenges that need to be addressed in building a user-centric tool as described above. These challenges include the need to (a) sample the information exported by the OS in an effective way, and (b) filter noisy data due to the typical co-existence of multiple active apps on a smartphone. Finally, as our main contribution, we design, implement, and evaluate TIDE, a user-centric tool that can be readily installed and used by real users for identifying the energy hungry apps specific to their usage profiles. TIDE is itself implemented as a smartphone app, which continually performs lightweight monitoring of a user's usage of apps and the resources that these apps consume. This information is then fed to a classifier which efficiently categorizes apps as high, moderate, or low consumers of the phone's battery. In our evaluation of TIDE, based on a detailed emulation of traces of usage patterns from 17 volunteer users, we find that it correctly estimates the level of energy consumption for 225 out of 238 apps. Furthermore, TIDE delivers this level of accuracy while imposing only 0.5% of overhead on the average consumption of the phone's battery per hour.

2.2 Related work

Android provides a battery manager tool [10] which estimates the percentage of battery consumed by each app. It considers the resource consumption of an app with respect to the number of CPU ticks, the number of bytes transferred over the network, the time for which the display was active, etc. It uses a model-based estimate of how much energy is consumed due to the use of a *unit* of each specific resource (e.g., per CPU tick, per TCP byte transferred) and multiplies this value by the number of units of that resource used by an app. The tool however does not account for several user-specific factors that influence energy consumption per-unit resource, e.g., link quality influences the energy consumed per byte transferred on the network. In Section 2.3, we show via measurements that these factors can have a significant impact on an app's energy consumption.

Prior efforts on estimating application-specific energy/power consumption can be broadly classified into three major classes.

User-centric tools: Current tools that try to characterize the power consumed by apps either use offline tests and/or fail to account for one or more factors that affect the battery drainage due to an app. PowerTutor [11] estimates an app's power consumption due to its interactions with different hardware components (e.g., LCD, GPS, WiFi, and 3G interfaces) based on a regression model. Unlike TIDE, a) PowerTutor itself consumes high power since it queries the OS at a high sampling rate, b) it depends on per-app resource consumption information, which is not readily available in newer versions of Android, and c) it requires offline calibration for every device type.

Carat [1] uses crowdsourcing to estimate the energy impact of an app; it compares battery drainage statistics with and without the app. This approach however fails to account for both user-specific app usage and user-specific network conditions, which can affect battery behavior, as we show later. Further, unlike Carat, TIDE only runs on user's devices and performs all analysis locally on any particular device, i.e., there is no need for either offline calibration or server-side aggregation. Falaki et al. [12] also highlight the impact of user-specific factors on battery consumption; they suggest that 'diversity' across users in terms of their app interactions can influence battery drainage rates. However, they did not focus on the development of a tool such as TIDE for user-specific estimation of app energy consumptions.

Determining energy bugs: Another body of work tries to detect energy bugs in apps. Yoon et al. [13] use Kprobes, a Linux kernel module in Android, to track native system calls for detecting anomalous behaviors. Pathak et al. [9] design a framework that needs access to system calls and applications' native code, in order to detect energy bugs. However, such tools require an external power meter for energy measurements and/or the modification of the underlying OS. eDoctor [14] identifies abnormal drain issues on phones by comparing app behaviors with well known good versions. Their goal is different from ours; we seek to identify apps consuming energy on individual users' phones, regardless of whether the high energy consumption is due to a bug.

Characterizing energy consumption by individual components: Finally, there are efforts that try to assess the power consumed by smartphone components (as opposed to apps). Shye et al. [15] build a model which estimates the breakdown of power consumption in different hardware components, based on a set of apps. However, their estimation does not work for new apps not present in this set. WattsOn [16] is an energy emulator that uses power models developed offline for individual smartphone components. However, to emulate an app's usage pattern on WattsOn, we would need to capture a user's interactions with the apps on her phone, and collecting this information would require rooting the phone; most users are unlikely to permit this. Most smartphones use battery models to provide the user with coarse-grained battery usage statistics; Sesame [17] argues that such models must be generated based on measurements using individual smartphones, rather than offline in a lab setting. Carroll et al. [18] instrument the components of an Android device offline, and measure the power consumed by each while running various benchmarks. Balasubramanian et al. [19] focus specifically on the energy consumed by the network using different technologies. eCalc [20] estimates the energy consumption of the CPU when an app is executed by profiling the app's binary. None of these efforts look into developing a user-centric tool for identifying energy hungry apps.

2.3 Showcasing user-centric app behaviors

In this section, we present an extensive measurement study to demonstrate that user behaviors, network conditions, and even phone features impact the energy consumption of apps. These demonstrate that crowdsourcing (e.g., Carat [1]) cannot accurately account for user-specific app behaviors. We also showcase the limitations of the Android system tool in capturing energy consumption behaviors of apps.

2.3.1 Impact of network conditions

First, we show that the network types and link qualities significantly affect the energy consumed by an app. We experiment with four HTC Touch 4G phones, each of which uses a different network with different qualities. All the phones use the same email account and we write a script to send emails to the logged in accounts. Emails are sent at high (every 30 seconds), moderate (every 5 minutes rate) or low (every 10 minutes) rates. We turn off the display and all background activities to make sure that the network I/O is the only contributor to battery drain. The phones are notified of new emails via push

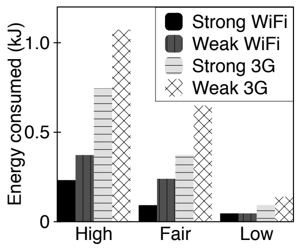


Figure 2.1: Network impact on energy

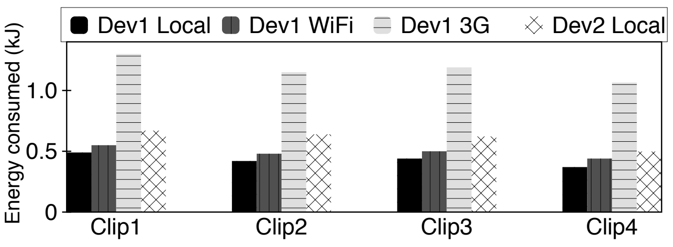


Figure 2.2: YouTube's energy consumption when playing different videos

notification messages. These messages wake up the phones if they are in the sleep state. A pair of phones use 3G connections, while another pair uses WiFi. For the pair of phones on the same network, we put one phone at a location with good signal strength (between -69 and -55 dBm) and the other at a location with poor signal strength (between -103 and -97 dBm). We fully charge the phones before the experiment and measure the energy consumed after 1 hour.

Results: Fig. 2.1 shows the battery drainage with each phone in different network conditions. In poor signal conditions, as one might expect, (i) the amount of energy used to transfer packets is higher [16], and (ii) the amount of corrupted packets is significantly higher [21], which causes many packet retransmissions. Thus, the energy consumption is much higher; for example, with a high volume of data, in 1 hour, the phone with poor 3G signal consumes more than 8% of the battery, while the phone with good 3G signal consumes only around 5%.

Thus, these experiments show that the energy consumption of an app not only depends on the *amount of network traffic* that it sends and receives, but also on the *type* and *quality* of the network connection that the user experiences. We repeated the experiments in this section with different pairs of phone models and different network providers, and we still observed qualitatively similar results. We do not report the other results here due to space limitations.

2.3.2 Impact of user behaviors and phone features

Beyond variance in network conditions, different users can potentially use the same application quite differently, which can in turn affect that app's energy consumption.

An example with YouTube: To demonstrate the impact of user-specific workloads on energy consumption, we perform experiments with YouTube. We play different videos on a smartphone (*Dev 1*). Videos 1 and 2 are full screen; however, video 1 is of high quality (480p) whereas video 2 is of default (360p) quality. Videos 3 and 4 cover $3/4^{\text{th}}$ of the screen when playing; again, the former is of high (480p) quality and the latter is of normal (360p) quality. We play these videos on *Dev1* when the video files are (a) stored locally on the smartphone's memory card, (b) downloaded over WiFi, or (c) downloaded over 3G. Finally, we repeat case (a) with a different smartphone (*Dev2*). *Dev1* is a Samsung Galaxy SII and *Dev2* is a HTC MyTouch 4G phone.

Results: The results of our experiments are shown in Fig. 2.2. On one hand, with *Dev1*, we observe that streaming over 3G always consumes the most energy; streaming over WiFi consumes slightly more energy than when playing local files. This reaffirms our previous finding that, depending on the network coverage (3G versus WiFi) enjoyed by the user, the energy consumption of an app can differ.

On the other hand, we also observe significant differences in the energy consumed when playing different videos (all playing on the same device); between the two videos,

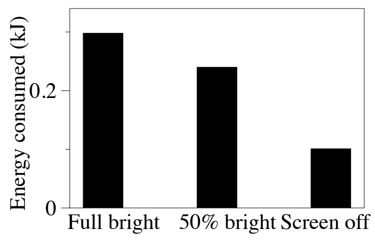


Figure 2.3: MusicFolder-Player's energy consumption

```
<device name="Android">
<item name="screen.full">211.6</item>
  <item name="WiFi.on">1.38</item>
  <item name="WiFi.active">62.09</item>
  <item name="WiFi.scan">52.1</item>
  <item name="radio.active">185.6</item>
(... file content is shortened for clarity...)
```

Table 2.1: power_profile.xml

we see a difference of as much as 20% in terms of the time taken to deplete the battery by 1%. Thus, depending on the video itself (rate of motion, black and white versus color, etc.), its resolution (high quality versus low quality), and the display size, the YouTube app's energy consumption may vary. As the choice of video, resolution, etcdepend on user preferences and choices, the user's behavior strongly influences the energy consumption of this application.

Finally, we also observe differences in the energy consumptions across devices when playing the same video file (from local memory). In fact, the difference is as high as 49%; this is primarily due to the differences in the hardware on the two phones. *Dev1* uses a Super AMOLED Plus display, which does not require a backlight and is thus, more energy-thrifty as compared to the LCD display on *Dev2*.

Other examples: While the above example was with respect to YouTube, other apps also exhibit such *multi-modal* energy consumption patterns based on their usage.

MusicFolderPlayer: The MusicFolderPlayer app allows a user to either keep the screen on or off when playing music. Depending on which option a user chooses to use, the energy consumed by this app can vary. Fig. 2.3 shows the energy consumed by this app in

5 minutes in three different modes. As one might expect, if the screen is on, this app is a high energy app; else, it behaves as a low energy app.

Angry Birds: We next consider a game app and observe varied energy consumption depending on the expertise of the user playing the game. Specifically, we have two users play the Angry Birds game for 10 minutes each. One user, who is well-versed with the game, plays the game constantly and moves to higher levels of play. The other novice user progresses through the game at a slower pace as he takes time figuring out how to play at each level. On a Galaxy SII phone, we observe that the novice user's usage of the game consumes 0.72 kJ of energy as compared to the 0.91 kJ consumed by the expert user. This amounts to a difference of 26.39 % ($\approx$ 4.8 % in terms of the battery percentage consumed) *per hour* of play.

The Android system tool does not account for user-centric factors: As discussed in Section 2.2, the Android system tool attributes energy consumption to an app based on its usage of specific resources. For each app, the tool records the number of units of each hardware component used by the app. This number is multiplied with the average energy consumption of the corresponding component to estimate the energy consumed by the app due to the use of that component. The sum of these values across all components is the energy consumed by the app. In an Android device, the average power consumption values of the various components (in mAh) are stored in the `power_profile.xml` file provided by the manufacturer; a shortened version of the file is shown in Table 2.1. Note that the contents of the file are fixed and not updated (the energy information is not re-calibrated) when the environment changes. We see that the average energy used by the WiFi interface in one time unit is shown on line 4. Similarly, line 6 shows the average power used by the cellular interface. It is evident that the network link quality is not accounted for by the Android tool.

Further, from the source code of the tool [10], one can see that while computing the energy consumption due to an app's network activities, the tool does not differentiate between the app's use of WiFi and cellular networks. If the total amount of data sent and received by all apps over the cellular and WiFi interfaces are *mobileData* and $wifiData$, respectively, then the Android OS computes the average power consumed per byte as $(3GEnergyPerByte * 3GData + wifiEnergyPerByte * wifiData)/(3GData + wifiData)$, where $3GEnergyPerByte$ and $wifiEnergyPerByte$ are obtained from the power model (Table 2.1). For each app, the OS then computes the energy consumed due to network activities by simply multiplying the average energy per byte computed above with the total amount of data transferred by the app over all interfaces.

Since network conditions are not taken into account, the tool may not always yield accurate outputs. To validate this hypothesis, we conduct an experiment wherein three different applications read the same file in the memory card and send the content to our server. The apps are run on the same device and use exactly the same source code but send data in different network settings. We turn off the WiFi connection on the device and run App1, thus causing it to send data over the 3G network. Subsequently, with WiFi turned on, we run App2 at a location near an access point such that the device enjoys good signal strength. Finally, App3 is run at a location with weak WiFi signal strength. The Android system tool shows App1, App2 and App3 consume 2%, 3% and 3% of the phone's battery, respectively. These numbers are far from what we get from direct measurement with a power meter; the measurements show that the three apps consume 6%, 1% and 2.5% of the battery, respectively.

These experiments show that results from the Android System tool do not capture changes in the energy due to specifics of the usage environment (the actual conditions) in which the user applications are executed; in other words, the tool is not user-centric.

Solutions such as Carat [1] cannot be easily extended to account for user-centric behaviors: By its very nature, crowdsourcing (the basis for Carat [1]) ignores user-specific characteristics of apps. We downloaded and tested Carat on our own Android phones for a week. Carat classified two of our appsGoogle Maps and Skypeas energy hogs. However, we had only used Google Maps for a very short time during the study and it barely consumed any energy. Further, we used Skype with audio only and over WiFi, because of which it consumed little energy; Carat classified it as a energy hog since most users used it with video. Other users of Carat have experienced similar issues [22]. One can think of extending Carat to check if an app is an energy hog on a particular users phone by comparing energy consumption on that phone across periods when the app was active/inactive. We did examine this approach with a rudimentary implementation but found that it mis-classified low energy apps as high energy ones. This was primarily because such apps often executed simultaneously with other high energy apps, and it was difficult to isolate their behaviors in terms of energy consumption. Further, the approach did not account for multi-modal behaviors of apps (described later in section 2.4.3). We address these challenges in TIDE.

Summary: Our experiments show that the energy consumption of an app depends on several factors: (i) network conditions experienced by the user, (ii) her usage patterns, and (iii) her device's characteristics. This highlights the need for user-centric classification of apps, i.e., it must account for the user's typical profile in terms of the above factors.

2.4 Challenges in designing TIDE

Having motivated the need for a user-centric tool for identifying high energy apps, we now highlight the challenges in building such a tool on the Android platform. Based on our preliminary studies, we believe that iOS has similar limitations and poses similar challenges.

2.4.1 Lack of OS support

Developing TIDE would be easy if smartphone OSes monitored all the activities or resource usage of every app and exported this information to all other apps. However, as one would expect, smartphone OSes either do not record the necessary details for energy efficiency or hide this information because of security concerns. As a result, smartphone OSes complicate the development of TIDE in several ways.

Lack of precise energy usage information: In prior work, researchers have either instrumented smartphones with devices such as the Monsoon meter [23], or plugged special sense resistors into hardware components on the phone to measure the energy consumed [18]. Such setups were then used to either measure the energy consumption of a single app in isolation or to build power models of individual hardware components. In contrast, for our goal of developing the TIDE app, smartphone OSes do not provide such precise measurements of energy consumption. The only energy-related information exported by the OS is the battery level, which is reported with a 1% granularity.

Thus, TIDE's estimation of energy consumption by apps has to be based on its observation of when the phone's battery level changes, i.e., drops by 1%. Hereafter, we refer to each time period in which the battery drains by 1% as simply an *interval.* In Section 2.5.2, we elaborate on how this information can be captured on the Android platform.

Lack of app-specific resource usage information: A potential approach to side-step the limitation of the lack of precise energy information is as follows. For each type of phone, one can construct an accurate power model for every hardware component (e.g., LCD display, network interfaces, and CPU) in every environment (e.g., LCD power consumption as a function of brightness and 3G power consumption as a function of signal strength). Discounting the fact that gathering such a power model will be cumbersome, TIDE can then estimate the energy consumption of any particular app by 1) monitoring the environment in which the phone is used and the app's usage of each of the phone's components, 2) for every component, multiplying the app's usage of that component with the power coefficient value of the component, and 3) summing up this value across all components.

Unfortunately, such an approach would be hard to implement on today's smartphone OSes since, for many of the phone's hardware components (e.g., display, GPS), the OS only provides aggregate resource usage for the whole phone and not for each individual app. For example, to track LCD usage, Android permits an app to register for the events corresponding to the screen being turned on or off. While this would enable TIDE to determine the time for which the phone's LCD was on, it cannot determine how much of this usage can be attributed to each app on the phone. Thus, when many apps are running simultaneously, though the OS lets an app query for the list of all other apps active on the phone, it would be difficult for TIDE to partition the aggregate resource consumption across these apps.

While the OS does track and export per-app usage of some resources, there are complications involved even in their use. For example, Android maintains two files—`/proc/uid_stat/[uid]/tcp_snd` and `/proc/uid_stat/[uid]/tcp_rcv`—which list the amount of TCP traffic sent and received over the network (both 3G and WiFi) by an app; here *uid* is the unique identifier of the app on the device. However, this feature is optional and is disabled in some phone models (e.g., Galaxy Nexus and Sony Ericsson Xperia X10 Mini Pro); thus, on such phones, a user will have to root the phone and install a new kernel for the OS to be able to track TCP traffic. Moreover, power consumption of the network interfaces also depends on packet arrival rates, which determine the energy drainage during transmission tail periods [24].

The only resource whose usage TIDE can track on a per-app basis is the CPU. On Android, every running app has a unique process ID (pid) and its CPU usage is provided in the file `/proc/[pid]/stat`. The CPU usage time is measured in 'system ticks'. In Android, the number of ticks per second is usually set to 100 [25].

Overhead of querying information: One way to cope with the availability of only aggregate resource usage information would be to have TIDE query the OS frequently

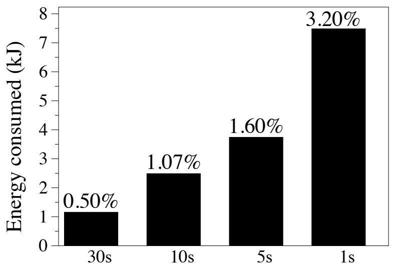


Figure 2.4: TIDE's energy consumption vs. sampling rates

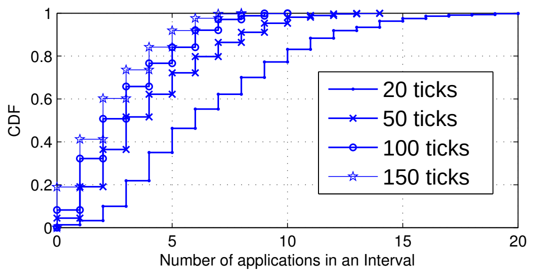


Figure 2.5: Number of active apps

(e.g., every second). TIDE can then attribute all the resource consumption in the last second to the app that was actively used in that period. On the Android OS, TIDE can discover the app currently being used by querying the OS for the foreground app. However, frequently querying the OS for both the foreground app and the usage of all resources can itself consume high energy. Fig. 2.4 shows the power consumed over an hour when querying Android on a Galaxy Nexus phone at different rates; we perform this measurement on a phone where only our querying application was active and all other apps were disabled. If we query every second, TIDE would itself consume 3.2% of the battery in an hour on the tested phone. Consuming over 3% of the phone's battery every hour would make TIDE prohibitive for use. On the other hand, if we query every 30 seconds, the querying application only consumes 0.5% of the battery in an hour; however, this leads to the challenges discussed next.

2.4.2 Challenges in associating energy consumption to specific apps

It is difficult to tease out app-specific energy consumption from the inherently noisy data that the OS provides when queried less frequently (e.g., once every 30 seconds). To show this, we not only perform select experiments on our smartphones, but also rely on measurements from the smartphones of real users. Specifically, we distributed an Android app to 17 volunteer users with IRB approval (details later in Section 2.6.1).

Co-existence of multiple active applications: A major obstacle in attributing the energy consumed to a specific (say target) app is that there are many co-existing active apps when the target app is running; in our measurements, almost all intervals contain multiple concurrently active apps. There are several reasons for this. First, there are background processes (including system processes) that continuously run on a phone. Second, users often switch between multiple apps; for example, a user may switch between checking email, posting on Facebook, and listening to music within a short time. Finally, to reduce load times for recently used apps, Android keeps an app in memory even after use; it kills the app only when the phone's memory has to be devoted for other apps. Thus, many recently used apps are included in the list of active apps reported by the Android OS.

To determine the apps in the active list that actually contribute to energy consumption, we need to estimate their activity levels. One way to estimate an app's activity level is based on the app's CPU usage (the OS can be queried for this information); note that an app consumes a non-trivial number of CPU ticks even when it sends/receives data over the network. Simply eliminating all apps that have consumed zero CPU ticks in the interval is insufficient because some apps may use a little CPU only to periodically poll for updates; these apps are unlikely to contribute much to battery drainage in that interval. Hence, we need to use a threshold to filter out apps that were largely dormant. However, determining a good threshold for CPU ticks is challenging; this threshold will depend on the smartphone architecture and on an application's implementation.

In Fig. 2.5, we plot the CDF of the number of simultaneous apps (from the dataset for one user from our study) with different thresholds for CPU ticks. We see that if a low threshold is used, we cannot filter out apps that run for short periods. For example, with a threshold of 20 ticks, 60% of the intervals have more than 5 simultaneously active apps. However, if the threshold is too high, a majority of apps are filtered out, some of which may be energy hungry. Note that this profile (how many simultaneous apps are active in an interval) is user-specific.

Work delegation between apps: Another major hurdle in attributing energy consumption to specific apps is work delegation, which is possible on Android devices. Specifically, the functions of one app are delegated to another app. One example of an app that receives many such delegated functions is the Mediaserver app. Every media app delegates data retrieval operations to Mediaserver; once Mediaserver has received data over the network, the data is exported to the appropriate app. For example, when a user is viewing a video with the YouTube app, the video streaming is delegated to the Mediaserver app. A naive energy monitoring tool would hold Mediaserver responsible for the energy consumed due to network transfers. Based on this information, since Mediaserver is a system application that cannot be completely disabled, the user may continue to use YouTube as normal and drain her phone's battery. To be accurate, TIDE must identify YouTube as the main culprit for energy drainage in this case.

2.4.3 Multi-modality of apps

Finally, the determination of energy hungry apps is complicated by the various modes in which a single app can function. There are several apps that consume high energy only when they use a high amount of a specific resource(s). As we show later in Section 2.6.3.2, YouTube and Pandora are two examples of multi-modal apps. YouTube's classification as an energy hungry app depends on the network quality, whereas the Pandora app consumes high energy only while the display is on. Therefore, TIDE must have the capability to classify apps under different usage scenarios.

2.5 TIDE: Architecture and Implementation

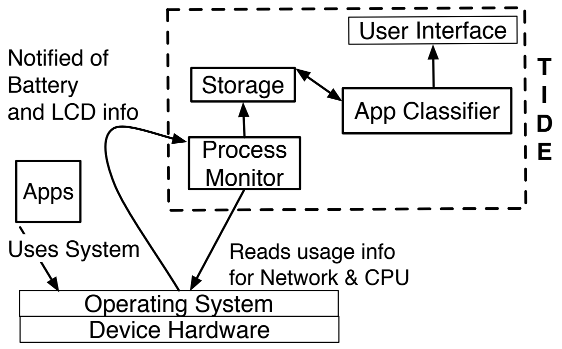


Figure 2.6: TIDE architecture

We next describe the architecture of TIDE and provide the details of our implementation. Since TIDE seeks to capture user-centric attributes, it runs on every user's own smartphone and identifies energy hungry apps based on the user's profile. Specifically, it inspects the correlation of apps' occurrences and high energy/resource usage periods on the phone. TIDE seeks to identify the energy-hungry apps by long term profiling; thus, the more the user invokes an app, the higher the accuracy of TIDE's classification of the app.

We wish to point out here that when we classify apps with TIDE, we focus on the energy consumption due to the CPU, the network interfaces, and the display. However, the framework that we use in TIDE is extensible to account for other resources. For example, one resource whose use is known to lead to high energy consumption is the GPS. Similar to techniques that we describe in this section, TIDE can identify an app's energy consumption due to use of the GPS by correlating periods when the GPS is turned on with intervals in which the app either has significant CPU activity or is in the foreground.

2.5.1 System architecture

Fig. 2.6 depicts the architecture of TIDE; it consists of two main components: *Process Monitor* and *App Classifier*.

2.5.1.1 Process Monitor

TIDE's first component profiles app behaviors on the user's phone. Recall that the smartphone OS does not provide fine-grained information with regards to energy consumption; the only information that the OS exports are the durations between instances when the battery level drops by 1% (intervals). The Process Monitor runs in the background and keeps track of these intervals. At the end of each interval, it queries the OS for the resource usage information in that interval. Specifically, it obtains information relating to (i) the duration for which the screen was on during the interval, and (ii) the aggregate network usage in that interval (in bytes). Within each interval, the Process Monitor also queries the OS periodically (once every τ seconds) for a list of the running apps and the CPU usage of each app in the preceding τ seconds. The information collected is stored in the phone's SD card and is later processed by the App Classifier.

Adaptive sampling: In TIDE's querying of the OS once every τ seconds for a list of active apps, there is an inherent trade-off in choosing a value for τ. On one hand, the larger the value of τ, the more coarse grained the information obtained from the OS. As a result, the query returns co-existing apps more often than not. Further, it cannot accurately map resource usage to apps; this makes it especially difficult to capture multi-modal behaviors. On the other hand, Process Monitor can query the OS more often (e.g., $\tau = 1$ second), but this increases the energy overhead imposed by TIDE[2].

To address this trade-off in TIDE, we use an adaptive sampling approach. Specifically, Process Monitor queries the OS more often when the battery drainage is heavy (i.e., when it observes short intervals) and less often when battery drainage is minimal (long intervals). The basis for this is that, in order to identify energy hungry apps, fine grained information is required only during those periods when the rate of energy consumption is high. In more detail, after a high-drainage interval is seen, the Process Monitor switches to fine-grained sampling, and τ is set to 1 second. Typically, during high usage periods, short intervals appear in bursts (we observe this in our experiments) and thus, the next interval is also likely to be a short one. On the other hand, after k long (short-drainage) intervals, the Process Monitor returns to coarse-grained sampling; in our implementation (described later), we find that $k = 1$ works well and we set τ to 30 seconds for coarse-grained sampling. We evaluate the overhead and efficiency with adaptive sampling in Section 2.6.

2.5.1.2 App Classifier

The output of the Process Monitor contains the set of co-existing apps detected with each query, as well as the resource usage (screen/network) during an interval. The

[2]Note that the number of co-existing apps with $\tau = 1$ sec is drastically lower than when $\tau = 30$ secs, but apps may still co-exist.

App Classifier takes this as an input and tries to identify the high-energy apps from this noisy data. It performs this classification in three phases (summarized in Algorithm 1).

Phase 1: Using interval lengths to categorize apps: First, we observe that longer intervals correspond to slower battery drainage and shorter intervals correspond to faster battery drainage. Therefore, a long interval serves as evidence that *all* of the apps observed in that interval have low energy consumption during the interval. If *any* of the apps used in a particular interval consumed high energy, then that interval would be short. On the other hand, if a single app was active in a short interval, then that app was definitely the cause for the fast battery drainage in that interval.

Based on these observations, our first phase of app classification works as follows. For any app X, we consider all the intervals in which this app is seen to be active (details in section 2.5.2). Among these intervals, if the fraction of intervals that are short and have no other concurrent app with X is greater than a threshold f_H, then we mark X as an energy hungry application. Similarly, among the intervals in which an app Y occurs, if the fraction of long intervals is greater than a second threshold f_L, then we consider Y to be a battery-thrifty application.

However, the above procedure by itself is insufficient to classify all apps. This is because, as discussed earlier in Section 2.4, many intervals include several concurrently active apps. Hence, if a short interval includes many active apps, we cannot attribute the high energy consumption in that interval to any one app with certainty.

Phase 2: A greedy algorithm to handle co-existing apps: To account for multiple active apps in short intervals, we use a greedy algorithm in the second phase of the

App Classifier's execution. In a nutshell, the larger the fraction of short intervals among the intervals in which an app is active, we can have greater confidence in declaring the app as energy hungry. The algorithm identifies energy hungry apps in the decreasing order of associated confidence. Once a particular app is marked as energy hungry, we *greedily* attribute all the energy consumption on the phone to this app in all the short intervals in which the app is active.

In more detail, let us define the confidence value for an application X being energy hungry, *conf(X)*, to be the probability that an interval which contains X is also a high battery drainage interval. The App Classifier deems an application X as energy hungry if *conf(X)* is more than a threshold (say γ). Once app X is marked as energy hungry, the classifier discards all high battery drainage intervals that contain the app from future consideration; this essentially attributes the high battery consumption in these intervals to app X. The classifier thereafter repeats the procedure of identifying the app with the next highest confidence value ($\geq \gamma$) based on the intervals that have not yet been discarded. We repeat this process until no apps with a confidence value $\geq \gamma$ remain.

In the above algorithm, one can envision cases where a high energy app Y gets filtered out simply because it also appears with another high energy app X. However, first we argue that these cases are rare in practice (as also seen in our experiments). When a high energy app is being executed, the phone drains energy very quickly (in less than 2 minutes in our setting). In such a short interval, the likelihood that the user uses and switches between several high energy apps (such as games, video streaming apps, etc.) is really low; such apps usually require user involvement. This decreases the likelihood that such cases happen to begin with.

Second, TIDE fails to identify Y from being a high energy app only if Y is not frequently used by the user. In such cases, Y may not be executed in isolation by the user in the near future; if the user uses app Y frequently, in the long run (say, 1 week), there will be intervals in which Y does not co-occur with other high energy apps (e.g., X) and will thus be correctly classified. We show this later in section 2.6.

Finally, one might expect the user to stop the usage of app X because of TIDEs classification. This then precludes the simultaneous execution of X and Y and thus, the high energy usage of Y will be discovered by TIDE much more quickly and efficiently.

To improve the effectiveness of TIDE in such cases, viz., when app X and app Y are *almost* always executed together, they can be considered as a tuple $\{X, Y\}$ that causes high drainage on the phone. We defer such optimizations to future work.

Phase 3: Dealing with multi-modal apps: Multi-modal apps that exhibit different energy consumption rates in different execution modes may however have a low confidence value, since intervals containing an app X combine data from all of X's modes. To handle such cases, in App Classifier's final phase, we also define the confidence value for a tuple of application X and resource R, *conf(X,R)*, to be the probability that an interval which contains X and has high utilization of R is a high battery drainage interval. Using *conf(X,R)*, TIDE is able to detect apps that are energy hungry only in execution modes where a specific resource (e.g., network, screen) is intensively used. This information will allow a user to decide how to (or rather how not to) use certain apps, e.g., the user may decide against uploading videos to Facebook if TIDE determines that Facebook's high energy consumption is correlated with heavy network usage.

In TIDE, the environmental factors and user behaviors are fully captured when classifying apps. Specifically, it detects high energy apps by capturing the correlation between app activities and the energy drainage rate on the phone. The drainage rate implicitly accounts for how the user interacts with the apps, as well as how much and in what conditions resources are consumed. If the same "amount of" resource is consumed in "favorable" conditions (e.g., good network, low quality video), the drainage rate would be lower, and vice versa. Thus, even though we only provide coarse grained classification information, the results are fully user-centric and accurately capture energy consumption of the apps on the specific user's phone.

2.5.2 Implementation details

Next, we describe our Java-based implementation of TIDE for Android phones[3]

Process Monitor: TIDE captures a phone's battery usage by monitoring what are called "Intent" messages on the Android platform. The Android OS broadcasts notifications about important system events to apps (with the right permissions) through Intents. TIDE registers for the ACTION_BATTERY_CHANGED event, and by means of the associated Intent message that it receives, determines when the residual battery level drops by a percent. TIDE also registers for the ACTION_POWER_CONNECTED and ACTION_POWER_DISCONNECTED events; with these, it is notified when the phone is

[3]We are working towards releasing TIDE on the Google Play store; a preliminary version can be found at `http://bit.ly/1lnp51f`.

Algorithm 1 TIDE's algorithm for app classification

```
1: //Phase 1
2: for all app $x$ do
3:     $s$ := Fraction of intervals containing only $x$ that are short
4:     $l$ := Fraction of intervals with $x$ that are long
5:     if $s \geq f_H$ then
6:         Mark $x$ as $HIGH$
7:     else if $l \geq f_L$ then
8:         Mark $x$ as $LOW$
9:     end if
10: end for
11: //Phase 2
12: $\forall$ unclassified app $x$, calculate $conf(x)$
13: while $\exists$ unclassified app $x$ with $conf(x) \geq \gamma$ do
14:     Find app $x$ that has the highest confidence
15:     Mark $x$ as $HIGH$
16:     Remove all short intervals that contain app $x$
17:     Recalculate confidence values of unclassified apps
18: end while
19: //Phase 3
20: Multi-mode candidates = apps classified in phase 1 $\cup$ all unclassified apps
21: for all multi-mode candidate app $x$ do
22:     Calculate $conf(x, r)$ for app $x$ and resource $r$
23: end for
24: while $\exists$ tuple $(x, r)$ with $conf(x, r) \geq \gamma$ do
25:     Find tuple $(x, r)$ that has the highest confidence
26:     Mark app $x$ as $HIGH$ when it intensively uses resource $r$
27:     Remove short intervals with app $x$ and high utilization of $r$
28:     Recalculate confidence values of remaining tuples
29: end while
30: Mark all unclassified apps as $MODERATE$
```

plugged in or unplugged from the power outlet. Lastly, TIDE registers to be notified of the ACTION_SCREEN_ON and ACTION_SCREEN_OFF events; it can thus determine when the display (screen) is turned on or off.

At the end of each interval, TIDE reads the system files in the folder /sys/class/net for aggregate network usage information; for example, `/sys/class/net/wlan0/statistics /tx_bytes` and `/sys/class/net/wlan0/statistics/rx_bytes` provide information about the number of bytes sent or received through the WiFi interface. In each interval, TIDE periodically queries the Android OS for a list of running apps, and for each app, it reads the system file `/proc/[pid]/stat` (where *pid* is the process identifier of the app) for the number of CPU ticks consumed by the app. All the aforementioned files reflect the resources consumed from the time that the phone was last booted.

In addition, Process Monitor identifies work delegations by tracking the corresponding Intent messages that are invoked. However, these Intent messages can be transparent to the application layer if the delegation is to a system process (e.g., Mediaserver). We believe that such cases are extremely small in number, and hence, identify such cases manually. Among all of the apps we considered, we found that the Mediaserver app was the system-level app to which work was often delegated.

App Classifier: The App Classifier first filters out inactive apps or apps that do not significantly contribute to the energy consumption in each considered interval. It primarily considers an app to be active if it consumes more CPU ticks than a predefined threshold. In some outlier cases, an app may use the LCD but not the CPU; to account for such cases, TIDE also looks at whether an app is a foreground app in high energy intervals.

If so, the app's energy consumption due to the display can be directly computed, and thus, TIDE can determine if it is energy hungry in this mode.

Choosing a CPU threshold: We classify an app as active only if it uses more than a threshold number of CPU ticks; even if the app uses other resources (e.g. to render graphics on the screen, to stream data, etc.), it requires a significant number of CPU cycles. To establish the right threshold, we installed many popular apps from the Android market on a Galaxy SII phone and monitored their CPU usage with a real user's usage pattern (recall our study from section 2.6.1). With this, we determined when the apps were actually being executed and when they were idle in memory. We considered two types of apps: one set which have high CPU usage (e.g., Skype, Angry Birds), and another set with low CPU usage (e.g., MusicFolderPlayer, Advanced Task Manager). We found that a threshold of 150 CPU ticks when $\tau = 30$, works well to ensure that we do not filter active periods of low CPU usage apps but do filter dormant periods of high CPU usage apps. For $\tau = 1$, a threshold of 5 CPU ticks accurately assigns the resource usage to an active app. We repeated the experiment with three other users' traces and obtained almost identical results. This leads us to believe that these thresholds on the Galaxy SII phone are appropriate for use in TIDE to identify active apps. When TIDE is used on other phone models, we apply a linear scaling between the CPU frequency of the new model and the reference model (Galaxy SII) to determine the CPU ticks threshold for the new model. We find that this approximation works well in practice. We also observe that minor variations in the CPU ticks threshold do not affect TIDE's accuracy.

Detecting app LCD usage: Detecting active apps by just using a CPU threshold however is not enough, because an app can keep the screen on without using the CPU. Hence, we consider active apps in an interval to be the ones which either consume CPU or run in the foreground. By using adaptive sampling, in high energy intervals, we sample for the foreground app every second and thereby capture the LCD usage of apps. In other intervals, TIDE can only capture the foreground app once every τ=30 secs; thus, we can miss out on the apps that use the display at other times in between. However, this is not of consequence since, regardless of whether or not the app uses the display, it consumes low energy in such long intervals.

Once the active apps are determined as above, the App Classifier executes the classification algorithm described in Section 2.5.1.2. Here, we need to choose appropriate thresholds for 1) the *long* and *short* intervals in which an app has to appear, in order to be classified as a low or high consumer of energy (referred to as f_L and f_H in Section 2.5.1.2), and 2) the *conf(X)* or *conf(X,R)* values associated with any app X. We experiment with different values for these thresholds with different user workloads and on different types of phones. To keep the false positive rate low, we find that $f_L = f_H = \frac{1}{4}$ and $\gamma = 0.66$ works well. With lower thresholds, false positive rates are high; higher thresholds do not significantly reduce the false positive rate further, without also increasing the false negative rate.

Accounting for work delegation: Finally, whenever an app X (e.g., YouTube) appears in the same interval as another app Y (e.g., Mediaserver) to which X delegates work, we simply attribute all of Y's resource usage in that interval to X. If two apps that delegate work to Y simultaneously appear in an interval, we attribute each app with half of Y's resource usage. A similar approach can be applied to cases with more than two apps. However, in our user traces, we never observed any interval wherein more than two different apps delegated work to the same app within an interval.

Defining high and low drainage intervals: TIDE enables a user to choose the thresholds that define HIGH and LOW drainage intervals based on the user's preferences and expectations. However, for evaluating TIDE's performance, we define intervals in which 1% of the battery is drained in less than 2 minutes as HIGH and intervals in which 1% of battery is drained in more than 6 minutes as LOW. This is based on running known high energy (e.g., Skype) and low energy apps (e.g., MusicFolderPlayer) on our phones and noting how long they take to consume 1% of the battery; for example, Skype takes 1.8 minutes whereas MusicFolderPlayer takes around 6.5 to 9 minutes.

When is resource usage high? When multi-modalities of apps are considered, we need to construct tuples of the form {X, R} to represent the presence of an app X in a high battery drainage interval in which resource R is also heavily utilized. Thus, a question that needs to be answered is: *"when should the usage of resource R be considered high?"* To answer this question, we perform measurements using known resource hungry applications with respect to each resource. Specifically, for network usage, we measure the traffic generated by YouTube while watching 20 random video clips of HD quality, and by Skype during a video conference. We choose these specific apps as they are known to result in high network usage. We measure the volume of traffic while the apps are executed on 4 different devices and in different network conditions. In all our measurements, the apps generate $\geq$ 5.5 MB of traffic per minute, and hence, we set this to be the threshold for high network usage. Similarly, we consider 5 different 3D games (known to be CPU intensive) to set the benchmark for high CPU activity. We find that all of these games consumed more than 1000 CPU ticks per minute. Thus, we set this to be the threshold for high CPU activity. Like with the CPU ticks threshold we use to identify active apps in an interval, here too we linearly scale this threshold for high CPU usage based on the CPU frequency of the phone. As discussed earlier, with adaptive sampling we can capture LCD usage of apps in high energy intervals.

2.6 Evaluation

Next, we present a detailed evaluation of TIDE based on experiments conducted on a testbed of Android phones. Our experiments are driven by traces gathered from the phones of several users. We use a Monsoon power meter for all energy measurements on our testbed.

2.6.1 Collection of real user workloads

To capture user-centric behaviors, we collect data from 17 volunteer users. Our study has been IRB approved by our institution. Since a phone has to be rooted in order to gather the data that we need (note that using TIDE itself does not need the rooting of phones), we handed out rooted smartphones to our volunteers after swapping the phones' SIM cards with the SIM cards from the users' own phones; this obviates the need for volunteers to root their own phones. To ensure consistency, we matched the model of the phone handed out to a user to the user's own phone. The volunteers used our phones for their daily use for a week. The collected user traces are used to generate realistic workloads on our Android testbed for establishing the ground truth (as discussed later in Section 2.6.2). Furthermore, we run TIDE on these phones to get its output assessments.

2.6.1.1 Capturing user interactions

On every phone handed out to our volunteers, we installed a background process that captures all of the user's interactions with her phone. Capturing these interactions in a manner that allows for accurate replay is however a significant challenge. For example, a user's interaction with a web page may be hard to replay since the web page's content may vary over time. Moreover, some apps (e.g., Facebook) may require the user to be logged in, which we cannot emulate during trace replay. To capture interactions in a manner that enables high fidelity trace replay, we adapt the technique proposed by Gomez et al. [26] to capture user input events with low overhead. To do so, we poll the smartphone's system files for events generated by the user's interactions.

Apart from storing user input events, we also need to associate these events to apps. Unfortunately, system files that log user input events do not provide information about the app with which the user is interacting. Therefore, for every interaction, we also capture the foreground app on the phone by querying the ActivityManager class. Since the number of user input events is large (e.g., a simple swipe event on the phone can generate more than 10 records in the `/dev/input/event2` file), in order to minimize overhead, we query the OS for the foreground app only on "key released" records; these records are generated when the user releases her fingers from the screen or from a button. Note that, in order to gather the above information, root privilege on the phone is necessary. Hence, collection of such information is possible only for our purpose of gathering user traces and not as part of TIDE's operation.

We store all of this information in a file so that we can later replay on our testbed all of a user's interactions with every app used by the user. By emulating different network conditions, we can build the ground truth information with regards to the "user-centric" energy consumed by every app.

2.6.1.2 Capturing user-centric resource usage patterns

For privacy reasons, many users were wary of their interactions being captured; in fact only two of our volunteers allowed us to log these interactions. Thus, we seek a different way to estimate the app-specific energy consumption on such users' phones. For this, we capture the resource usage on the phone when an app is running and mimic these utilizations on the same phone to represent the app's execution.

To determine the CPU usage of an app, we read the file `/proc/[pid]/stat` (pid is the process ID of the app). To capture network traffic, we run `tcpdump` on the phone to captures all packets going through all network interfaces. Periodically, we run a modified version of `netstat` (provided by the Busybox tool set [27]) to record all the ports used by each app. We then correlate `tcpdump`'s output with the app to port mapping in order to map every packet to the corresponding app. To measure the time for which an app uses the screen, we access the system logcat information on the phone to estimate how long an app stays in the foreground. Again, note that these methods for capturing app-specific usage of the network/display is possible only with root privileges, and hence, such information is not available to TIDE.

2.6.2 Building the ground truth

To evaluate the accuracy of app classification with TIDE, we first need ground truth information. Specifically, for every app used by a particular user, we need to determine whether or not the app is indeed energy hungry from that user's perspective. Generating this ground truth is non-trivial in itself. In real user workloads, apps do not run in isolation. Furthermore, user-centric factors such as the signal strength of the 3G network experienced at different times are not known. Therefore, to generate the ground truth, for every app used by one of our users, we run the app in isolation as per that user's usage pattern of that app (other apps are turned off), and emulate different network conditions.

Similar to the drainage intervals, we assign one of three labels—HIGH, MODERATE, or LOW—to each app depending on how long it takes the app to consume 1% of the battery. While thresholds for determining these labels can be defined by user preferences in practice, we consider what we believe are reasonable thresholds in this study. For the reasons discussed in section 2.5.2, when replaying apps on a specific phone, we label any app that consumes 1% of the battery in < 2 minutes as HIGH; if this consumption takes $>$ 6 minutes, the app is labeled LOW. We consider apps which consume 1% of the battery in a duration that is in between 2 minutes and 6 minutes as MODERATE. In what follows, for simplicity, we combine both MODERATE and LOW apps and label them as MODERATE, since from a user's perspective it is not vital to distinguish between them.

2.6.2.1 Replaying user traces

Replaying user interactions: As discussed, only two volunteer users let us collect their fine-grained interactions with their phones. We replay these interactions with each app in isolation to quantify the *real* energy consumed by that app.

Replaying app behaviors based on resource usage: For all volunteer users in our study, we replay the resource usage of each app in isolation, to estimate its energy consumption. Currently, we do not consider replaying multiple applications simultaneously, even though there might be mutual influences between them in terms of power consumption. This is because (i) if multiple apps consume a specific amount of energy together, it is not easy to break down the energy consumption due to individual apps, as each app might consume different resources to different extents, and (ii) when there are multiple apps requesting resource access, it is extremely challenging to replay the resource usage exactly in a dependent manner without modifying the Android OS itself. Thus, we defer this to our future work.

For replaying the network usage of an app, we run a server which generates the same network traffic as identified by `tcpdump` in the user trace. We emulate varying network conditions to generate the ground truth in different scenarios. As network activities also consume CPU, we record the number of CPU ticks associated with these activities. When replaying CPU usage of an app, we subtract this number of CPU ticks to preclude network activities.

For replaying display usage, we keep the screen on for the same amount of time and with the same brightness level as from the user-trace. One problem with capturing

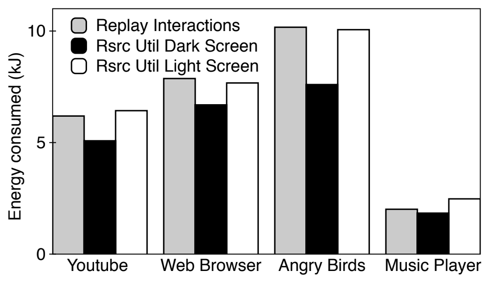


Figure 2.7: Building ground truth on power consumption of an app

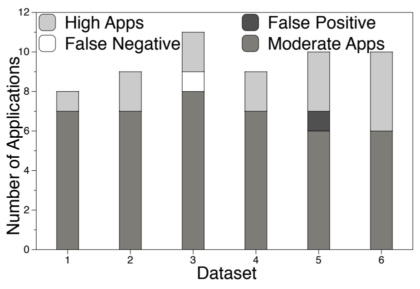


Figure 2.8: Accuracy of TIDE with user interaction based ground truth

the display's usage is that, though we periodically query for the screen's brightness level when an application is running, we do not know the exact content on the screen at specific times. Therefore, in our experiments, we use a static background while replaying an app (the brightness is as per the user's behavior). We try two extreme settings: (i) a dark and (ii) a relatively white background. Note that this limitation with respect to accounting for the impact of the displayed content on energy consumption is inherent in most of the energy models derived based on resource usage (e.g., [11, 17, 13]).

2.6.2.2 Can replaying resource usage patterns capture app energy consumption?

Capturing fine-grained user interactions provides high fidelity in the user-centric classification of apps. However, since we have this detailed information only for two users, we assess how trace replay based on resource usage patterns compares to that based on user interactions. For the two users for whom we could capture their interactions with their phones, we estimate the energy consumed by each app (i) first, by replaying user interactions, and (ii) again, separately, by replaying the associated resource usage from our traces. For clarity, we only show the results for 4 apps in Fig. 2.7; we see similar results with the other apps. We observe that simply using the resource usage provides an estimate of energy consumption that is almost equal to that in the case where we capture user interactions. The use of a relatively white background provides the best estimate; the dark background underestimates the power consumption to some extent. This is to be expected since most apps have bright colored or relatively less dark backgrounds.

2.6.3 Evaluating TIDE

We next evaluate TIDE's accuracy in classifying apps, and we thereafter assess its overhead.

2.6.3.1 App classification accuracy

We determine the accuracy of TIDE's App Classifier first based on ground truth obtained by replaying fine-grained user interactions, and second, based on resource usage information. Note that, on each of our volunteers' phones, TIDE was concurrently running while we were capturing logs that we later used for trace replay (in order to determine the ground truth for energy consumption of every app).

Accuracy as compared to ground truth based on user interactions: The two volunteers, for whom we could capture input events, were seen to use apps under different network conditions; both of these users used Galaxy SII phones. We separated the collected data into 3 sets for each user based on their interactions and network usage; each set contained information spanning at least six hours. Fig. 2.8 shows TIDE's accuracy on these datasets, in comparison with the ground truth. Each bar shows the total number of active apps in the respective dataset. The top and bottom parts of each bar show the number of high energy apps and the number of low/moderate energy apps, which TIDE's App Classifier was able to correctly classify. The middle parts of each bar depict false positive results, wherein LOW or MODERATE apps are mis-labeled as HIGH, and false negative results, where HIGH apps are mis-labeled as LOW or MODERATE.

False positives typically occur when a low energy app co-exists in many of its intervals with other high-energy apps. This can happen for apps that are not frequently used by the user. For example, the one false positive in Fig. 2.8 corresponds to the case where one of the users was using a music player app for 10 minutes while simultaneously surfing the web. In this case, we associate the music player with a high confidence value due to the web browser's high energy consumption. If TIDE monitors this user's phone over a longer period, there are likely to be intervals where the user uses the music player in isolation or only with other LOW apps. TIDE can then be expected to classify the app correctly.

Similarly, false negatives occur when a high-energy app X coexists only with other high-energy apps; when we discard intervals attributing them to these other high-energy apps (with higher confidence values), app X gets filtered out. TIDE then labels such an app as MODERATE. As users use applications for extended periods and increased numbers of times, the coexistence pattern of other apps will vary. As a consequence, the false positive and negative rates can be expected to be lowered over time. We show experimentally that this is the case in Section 2.6.3.3.

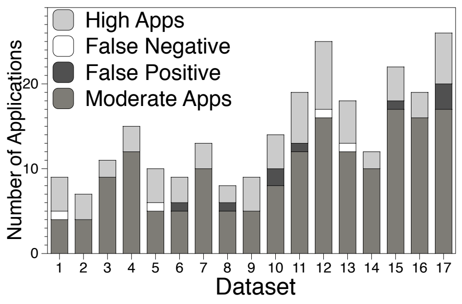


Figure 2.9: Accuracy of TIDE with resource usage based ground truth

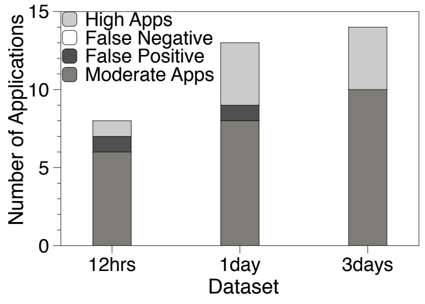


Figure 2.10: Accuracy of TIDE with different amounts of data

Accuracy with respect to ground truth based on resource usage: Next, we examine the larger dataset from our 17 volunteer users, which includes resource usage information based on their daily smartphone use for a week. Fig. 2.9 shows TIDE's accuracy in those 17 datasets, with the results amortized over different network conditions. The representation of the results are in the same form as in the previous case; each bar represents results from a different user's data. In 7 of the datasets, TIDE was able to classify all the apps correctly. In almost every other dataset, we only obtained either one false positive or one false negative.

Dataset 10 and 17 are the only exceptions where we had two and three false positives respectively; however, all the high energy apps were correctly labeled in this user's dataset. In summary, TIDE was able to correctly identify 66 out of 70 HIGH energy apps, and incorrectly classified 9 MODERATE apps as HIGH, from among a total of 168 MODERATE and LOW energy apps.

In the above analysis, we find several cases wherein TIDE correctly identifies the same app as HIGH for one user and LOW/MODERATE for another user. For example, TIDE identifies the YouTube app as HIGH for a user who always uses the 3G network on

		2-minute threshold		3-minute threshold	
Application	Condition	Ground truth	Result	Ground truth	Result
Skype	Strong WiFi	**H**	**H**	**H**	**H**
	Weak WiFi	**H**	**H**	**H**	**H**
	Strong 3G/4G	**H**	**H**	**H**	**H**
	Weak 3G/4G	**H**	**H**	**H**	**H**
Web browser	Strong WiFi	M	M	M	M
	Weak WiFi	M	M	**H**	**H**
	Strong 3G/4G	**H**	**H**	**H**	**H**
	Weak 3G/4G	**H**	**H**	**H**	**H**
Pandora	Strong WiFi	M	M	M	M
	Weak WiFi	M	M	M	M
	Strong 3G/4G	M	M	M	M
	Weak 3G/4G	M	M	**H**	**H**
YouTube	Strong WiFi	M	M	M	M
	Weak WiFi	M	M	M	M
	Strong 3G/4G	**H**	**H**	**H**	**H**
	Weak 3G/4G	**H**	**H**	**H**	**H**
Angry Birds	Strong WiFi	M	M	**H**	**H**
	Weak WiFi	M	M	**H**	**H**
	Strong 3G/4G	**H**	**H**	**H**	**H**
	Weak 3G/4G	**H**	**H**	**H**	**H**
Note: **H** - HIGH ; M - MODERATE					

Table 2.2: An app's energy consumption varies with network conditions

his phone. For another user who typically uses WiFi, TIDE correctly identifies YouTube as a MODERATE app from that user's perspective. Thus, TIDE is able to accurately account for user-centric factors that cause differences in an app's energy consumption across users.

Capturing user-centric app behaviors: Next, we demonstrate TIDE's ability to capture the user-centric attributes of apps. Specifically, here we consider apps that change their behaviors from HIGH to MODERATE or vice versa, depending on network conditions.

We conduct in house experiments with five popular apps—Skype, YouTube, the default Android web browser, Angry Birds, and Pandora—on a Galaxy SII smartphone.

First, we use each app for at least 15 minutes and capture all of the user's interactions. Thereafter, we replay all those apps jointly under 4 different network conditions: strong WiFi, weak WiFi, strong 3G/4G, and weak 3G/4G. The reported signal strength from the phone was between -105 and -97 dBm under weak signal conditions, and between -69 and -55 dBm under good signal conditions.

Table 2.2 shows the ground truth information and the results with TIDE. The ground truth labels are built by replaying the input events under the appropriate network conditions. Note that here we also experiment with two different thresholds to label an app as HIGH; an app is labeled HIGH if it consumes 1% of the battery (i) in less than 2 minutes in one case, and (ii) in less than 3 minutes in another case. The results demonstrate the low sensitivity of TIDE to the threshold.

In our experiments, Skype is always labeled HIGH, regardless of network conditions. Other apps, such as YouTube and the web browser, change their energy consumption profiles under different conditions. TIDE is able to capture these behaviors. In this experiment, we account for work delegation, and assign the resource usage by Mediaserver to YouTube (or Pandora) when they co-exist in the same interval. Without this, YouTube will always be labeled LOW.

2.6.3.2 Capturing multi-modal apps

We next conduct an experiment to evaluate TIDE's ability to classify multi-modal apps. Here, we first play Pandora for 1 hour using the 3G network while keeping the screen off. Subsequently, we set the screen at the highest brightness level and continue playing Pandora for the next 30 minutes. After this, we use YouTube for an hour using WiFi

(Pandora is now off). Finally, we continue with YouTube but switch to 3G for the last 30 minutes. We keep the screen at the highest brightness level while using YouTube. During the entire experiment, we also have other apps (auxiliary apps) that run simultaneously with Pandora and YouTube. With Pandora, we run an app that executes in the foreground and simply turns on the display while Pandora runs in the background; here our goal is to see if Pandora is correctly identified as a low energy app. With YouTube, we run an app that receives updates from a Twitter account; our goal is to see if TIDE can accurately capture YouTube's high energy when the network usage is high. The auxiliary apps are turned on and off at random. When turned on they remain on for a uniformly chosen random period between 3 and 5 minutes; when turned off, they remain in that state for a uniformly chosen period between 7 and 10 minutes. Both of these auxiliary apps continue to run for 2 hours after the Pandora and YouTube apps are terminated. We find that TIDE accurately classifies all of the apps above. Specifically, it finds that: (i) Pandora consumes high energy only when the screen is turned on, (ii) YouTube consumes high energy only if 3G is used, and (iii) both our auxiliary apps consume low energy.

In more detail, the confidence value of Pandora in general, without considering its different usage patterns, is quite low (20% out of 16 intervals). Thus, TIDE classifies Pandora as a MODERATE application. However, when TIDE considers Pandora only in intervals in which the LCD is intensively used, the confidence value of the tuple (Pandora, LCD) is high (80%) and TIDE identifies Pandora as an energy hungry application. As for YouTube, the confidence value in general is low (33% out of 24 intervals). However, considered only when the 3G network is used, its confidence value is 100%; TIDE thus identifies YouTube as a high energy app under high 3G utilization.

2.6.3.3 Accuracy versus dataset size

TIDE monitors user-specific factors (network, screen, CPU ticks) to create a profile of which apps consume high energy and how often, and what resource usage accompanies them. As a result, the longer the observation period, the better TIDE's accuracy. Fig. 2.10 shows the impact of the number of observed intervals on the accuracy of TIDE with one of our datasets (results with other sets are similar). With the data collected for 12 hours, there was only one high energy app invoked by the user, and TIDE produced one false positive result. This is primarily because of the limited volume of data used to build the profile. With the data collected for a day, the user used more high energy apps and TIDE was able to detect all 4 of them. The earlier, wrongly classified app is now correctly labeled as MODERATE; however, a new (previously unseen) app is mis-labeled as HIGH. With the data collected for 3 days, no more new high energy apps were detected. Importantly, the mis-labeled app is now correctly labeled as MODERATE. To ensure that the periods are long, but are not influenced by stale behaviors, we set the monitoring period to one week by default. However, the user can choose the period over which TIDE should use data to classify apps (e.g., 1 day, 3 days, or a month).

App/ Phone Component	**Android Tool**	**Power Tutor**	**TIDE**	**Ground truth***
Skype	6%	2.2 KJ	90%-**H**	4.8 KJ - 14% - **H**
Youtube (40 mins: WiFi+3G)	5%	3.2 KJ	47%-M	6.2 KJ - 17.5% - M
Youtube (the last 20 mins: 3G)	N/A	N/A	91%-**H**	3.9 KJ - 10.9% - **H**
Netflix	2%	1.9 KJ	20%-M	3.0 KJ - 8.5% - M
Pandora (3G)	1%	1.8 KJ	30%-M	3.4 KJ - 9.6% - M
AngryBird	3%	0.9 KJ	50%-M	3.3 KJ - 9.3% - M
Hill Climb	1%	1.6 KJ	73%-**H**	3.6 KJ - 10.1% - **H**
System	10%	4.9 KJ		
MediaServer	5%	1.3 KJ		
Screen	30%			
*Ground truth (x KJ - y% - M/**H**): the app consumes x kilo-Joules $\approx$ y% of the battery capacity, and is classified as a Medium or **H**igh energy app				
Note: Apps use WiFi unless stated otherwise				

Table 2.3: Comparing TIDE and other approaches

2.6.3.4 TIDE versus other popular approaches

Next, we compare the efficiency of TIDE in identifying high energy apps with that of the Android System Tool and the popular PowerTutor [11] tool; the latter has more than 500,000 downloads.

Experimental setup: We run 6 popular Android applications separately on a Galaxy S4 phone. Each application, apart from Youtube, is executed for 20 minutes; for each application, either a WiFi connection or a 3G connection is used to transfer data during the entire time the app is executed. Youtube is the only exception, wherein we use a WiFi connection for the first 20 minutes and a 3G connection for another 20 minutes to emulate a multi-modal app. Subsequently, we capture and compare the results from TIDE and the other tools, as shown in Table 2.3. With respect to the Android Tool and PowerTutor, we show the total amounts of energy reported to be consumed by each app by the tools. Specifically, for each app, (i) the Android Tool reports the percentage of energy consumed by the app with respect to the total energy consumption (by all the apps) on the phone, computed since the last time the battery was fully charged. By knowing how much energy the phone consumes in total, we convert these values into percentages of battery capacity and present them in Table 2.3, (ii) PowerTutor reports the total energy consumption (in kilo Joules) of the app. For TIDE, we show the confidence values with respect to app classification and include the classification labels (as **H** or M). We also show the ground truth information captured by replaying these 6 apps with an external power meter (the real power consumed). The ground truth information is represented in terms of the energy consumption (in kilo Joules), corresponding battery percentage and the correct classification label for each app.

The Android System tool: The tool only reports total energy consumption but does not capture the *average* consumption of the apps, which is more important in identifying energy hungry apps. An app should be identified as a high energy one only if it consumes a disproportionate amount of energy relative to its runtime, and not because it is continuously used for a long period of time. Further, the tool does not capture work delegation between media apps and the MediaServer process for media retrieval; thus, MediaServer is shown to consume a high amount of energy, but eventually, the media apps should be considered to be the main culprits for the drainage. The System process, which takes care of network data transfers at the kernel level for all other apps (and thus, has a high CPU load), is another case for work delegation and identified as a high consuming app.

More importantly, the tool does not breakdown the energy consumed by the screen to individual apps; in most cases, the screen consumes the highest amount of energy (30% of the battery). Thus, the energy consumption of all the apps shown by the tool is far from the ground truth results. For example, as the energy consumed by the screen and media retrieval is not contributed to the app, the tool shows that Pandora consumes about 1% of the battery. In reality, it consumes about 10% instead.

Finally, the tool does not capture multi-modal apps. Specifically, the tool does not differentiate Youtube when it uses (i) a WiFi and (ii) a 3G connections; thus, it only provides the energy consumption information of Youtube for the entire time the app is executed. Consequently, the tool does not identify Youtube as a high energy app when it uses the 3G connection for the last 20 minutes.

PowerTutor: PowerTutor is able to capture the total and the average energy consumption of apps by recording their runtimes. However, the accuracy of the PowerTutor is highly device dependent, since the tool estimates the energy consumption of an app by multiplying the amount of resource utilization with the corresponding average energy consumption for each of the resources. The average energy consumption information is calibrated for only a limited number of phone models; thus, when used on an unsupported phone, the results from PowerTutor might be significantly different from the ground truth information. For example, the tool reports that Youtube only consumes 3.2 KJ, whereas it actually consumes 6.2 KJ (when measured with the power meter). Further, the tool is not able to deal with work delegation or multi-modal apps (similar to the Android System tool).

TIDE: With TIDE, our main goal is to classify an app as HIGH or MODERATE/LOW, the tool relies on the rate of battery drainage reported by the phone and thus, does not require calibration for each specific phone model, as with PowerTutor. Further, with TIDE, when an app is the main culprit for high energy drainage, the correlation between its occurrences and short intervals is high. Therefore, the app is correctly identify as a high energy one instead of system processes which might be interacting with the app. Finally, TIDE is the only approach that is able to detect Youtube as a high consuming app, when the app uses 3G for downloading data. As shown in Table 2.3, TIDE is able to correctly classifying all the 6 apps.

2.6.3.5 Overheads

We examine TIDE's overhead along three dimensions: 1) energy consumed due to TIDE's periodic querying of the OS, 2) the execution time of TIDE's greedy algorithm, and 3) the storage space consumed by TIDE's logs.

Energy overhead: TIDE runs in the background and queries the OS periodically. We earlier showed in Fig. 2.4 that with a sampling rate of 30 seconds, TIDE consumes about 0.5% of the battery per hour. The power consumed by the App Classifier is negligible (especially if the processing is done when the phone is being charged). Even otherwise, to process a data file with 700 intervals, the execution of the App Classifier consumes roughly 192 Joules ($\approx 0.78\%$ of the battery capacity on a Galaxy SII phone).

Overhead with adaptive sampling: To quantify the energy costs with adaptive sampling, we perform the following experiment. We use the gathered data from one of our volunteers with a Galaxy Nexus phone; this was the phone on which we previously measured the energy consumed due to the monitoring process (see Section 2.4) with different sampling intervals. In this dataset, for each day, we pick the period from 9 AM to 5 PM (this is the time when the user uses her phone the most). We consider energy-heavy intervals to be of duration 2 minutes or less; in such periods, we assume that we query the OS every second. For other intervals (considered low energy periods), we only sample once every 30 seconds. We measure the energy consumed with three different sampling schemes: (a) sampling periodically every second, (b) sampling periodically every 30 seconds, and (c) adaptive sampling as above. The mean values of the energy consumed by the three schemes (based on a 5 day user activity) are 3.20%, 0.50%, and 0.76% of the phone's battery per hour, respectively. It is apparent that while adaptive sampling does increase TIDE's energy overhead, the increase is not exorbitant and thus, the approach is viable.

One can claim adaptive sampling makes the phone consume more energy when the battery drain is already high, which will possibly affect user experience. To show otherwise, we do an experiment to measure additional overhead caused by adaptive sampling during high usage intervals. Specifically, we measure the energy overhead with adaptive sampling during a video conference using Skype. In those intervals, the phone consumes 1% of the battery on average in 108 seconds without having any sampling. With adaptive sampling enabled, the phone consumes 1% in 101 seconds. In other words, the penalty is $\approx 7\%$. This indicates that adaptive sampling is unlikely to significantly degrade user experience during high activity periods.

Processing time of the greedy algorithm: Fig. 2.11 shows the execution times of the App Classifier with data collected over different numbers of intervals. We see that,

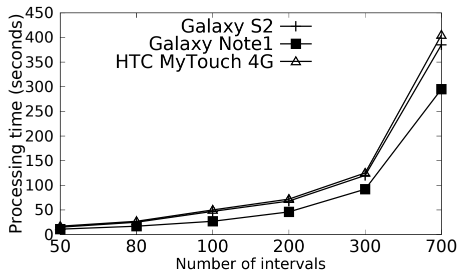


Figure 2.11: App Classifier's processing time on different phones

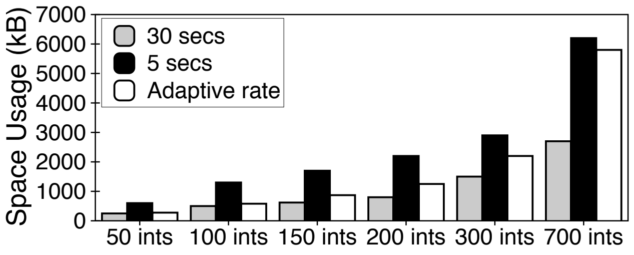


Figure 2.12: Space used to store Process Monitor's logs

even if the data in the input file spans 700 intervals ($\approx$ a week of data), the processing time is $\leq$ 7 minutes. This processing can be done offline when the user is not using the phone (e.g., when it is plugged into a power outlet for charging at night).

Storage space: Fig. 2.12 shows the average storage space used to store the input data collected by the Process Monitor, for different sampling rates. We see that, even when the collected input data spans 700 intervals, TIDE uses less than 6.5MB. Note that old data is purged as new data is accumulated, and hence, TIDE's storage overhead does not continuously grow over time.

2.7 Discussion

In this section, we discuss issues that may need further attention as TIDE is revised and improved in the future.

Availability of resource usage information: TIDE directly reads system files exported by the OS in order to capture resource usage information of apps. It is possible that smartphone OSes may limit access to these system files in the future, for security purposes. However, since many popular user-space tools (e.g., System Monitor, Task Manager) depend on access to this data, it is our belief that future releases of the Android OS will still permit user-level apps to access the system files used by TIDE.

Determining usage thresholds: As evident from prior discussions, TIDE uses a few thresholds for classification purposes; these thresholds are determined by measurements from the usage traces of several users. Note that, our goal is not to determine exactly how much energy each application consumes, but to classify apps into coarse-grained categories (specifically, HIGH and MODERATE energy apps); thus, our thresholds are not very sensitive to user behaviors or phone models. The chosen thresholds work well for all the users and devices in our collected dataset. While an alternative approach based on machine learning could be used to learn the appropriate values for these thresholds, the training process required by such an approach can potentially consume high energy. This requires more careful consideration in the future. In contrast, our simple approach offers high classification accuracy while being energy thrifty.

Low activity background apps: We do not focus on short-lived apps that are executed in very short periods (e.g., a few seconds). In each interval, these apps are typically not the main culprits for energy drainage. However, periodically executing a short-lived task (e.g., the network keepalive activity in many apps) may potentially consume high energy over a long duration. We examine the popular apps that synchronize data periodically (e.g., Facebook and Twitter) and find that these apps are taken into account by TIDE. Since the default sync intervals for such apps are typically set at around 30 minutes, network keepalive activity does not consume high energy ($< 2.5\%$ of the battery per day [28]) and TIDE is able to infer this. Further, if a short-lived app synchronizes its state more frequently, it consumes high CPU or network, and is treated as other normal apps by TIDE. In such cases, the app could be classified as a high energy app.

Consistent coexistent apps: TIDE's identification of energy-hungry apps depends on correlations between an app's occurrences and periods of high energy/resource usage. Thus, if two or more apps are *always* used simultaneously, TIDE cannot identify which of the two apps is energy hungry. However, over long usage periods (e.g., a week), we observe that this situation rarely occurs. As soon as the user invokes the apps separately, the real culprit will be associated with a higher confidence value and will thus be correctly classified.

2.8 Conclusions

In this chapter, we argue that there is a need for a user-centric tool to identify energy hungry apps on a users smartphone. We design and implement such a tool, TIDE. The key challenges addressed in TIDE are (a) it provides a lightweight way to determine active apps based on adaptive sampling and (b) it uses a novel greedy algorithm to filter out the real energy hungry apps from multiple simultaneously running apps on the users phone. It also effectively captures multi-modal energy behaviors. We show via both in house experiments and user- trace driven emulations that TIDE classifies apps as energy hungry (or not) with very high accuracy and low overhead.

Chapter 3

Managing Redundant Content in Wireless Constrained Settings

3.1 Introduction

A recent report estimates that there were around 350 million photos uploaded to Facebook and more than 50 million photos uploaded to Instagram on a daily basis in 2013 [29]. While new technologies attempt to increase wireless capacity (e.g., MIMO), users still find wireless networks to be a significant bottleneck in crowded settings, e.g., at football games [30]. Moreover, the demand for wireless capacity is likely to be exacerbated when unforeseen events such as natural disasters occur. In such scenarios, the network further gets overwhelmed due to a combination of the physical destruction of the underlying infrastructure (which severely impacts both network capacity and coverage) [31][3] and users generating more content than usual [32]. For example, there was a sudden increase in the number of images related to the hurricane Sandy that were uploaded to Flickr during the hour when the hurricane made landfall in New Jersey [33]. Even disaster rescue teams may upload images/videos to a control center, to allow the center to appropriately distribute resources/help. The higher traffic demands combined with the strapped wireless infrastructure can significantly hinder information delivery in such scenarios [4].

The large volume of images/videos that users attempt to transfer during such events is likely to have significant redundancies in information (photos of the same event taken by different users). For example, Weinsberg *et al.* [4] study the images taken by people in the San Diego fire disaster in 2007 and the Haiti earthquake disaster in 2010. They found that 53% of the images in the San Diego set and 22% of the images in the Haiti set were similar to each other (and in essence contained redundant content). Suppressing transfers of such redundant content can ease the load on the network, and allow unique[1] information (possibly critical) to be transferred with low latency. Subsequently, the redundant content can be lazily uploaded when the network conditions are more benign. Content suppression and lazy uploading can also benefit users in more generic settings (e.g., a flash crowd scenario during a sporting event); users can save on their cellular data plans, as well as the energy on their smartphones. These are likely to be taxed when the available bandwidth is low.

Arguably, the biggest challenge in suppressing the transfer of redundant content is to determine whether or not content generated by disparate clients are similar (e.g., photos of the same event, captured almost at the same time). This is inherently hard since the service to which clients are uploading images (e.g., Flickr or a server at a disaster control

[1]In the context of this chapter, unique information refers to content in dissimilar images, which contain objects or surroundings that are not covered in other images.

center) must make this determination *before* a client uploads the image content. Even if any one among a set of similar photos has been previously uploaded, the service has to determine if a *second* photo [2] that is being considered for upload is similar to the one that has already been transferred; if the transfer of the first photo is underway, the process is even harder.

The computer vision community has studied the problem of identifying similar images largely in the setting where the two images being compared are at the same client/server. However, requiring clients to upload images before the service can check for similarity with previously uploaded images nullifies the utility of our framework. Therefore, to suppress the uploads of redundant image content, we leverage the *metadata* used by the computer vision techniques that detect image similarity. Specifically, we have the client first extract metadata from the image it wishes to upload, and upload this metadata to the service. The client then uploads the image content only if the service is unable find any similar images using the uploaded metadata.

This approach however presents a fundamental trade-off. On the one hand, the more fine-grained the metadata extracted by the client from its image, the better the service's ability to correctly identify whether similar images have been previously uploaded. It is important to ensure not only a low false positive rate so that clients do not miss uploading critical images but also a high true positive rate to reduce as much of the redundant content as possible. On the other hand, it is vital that the metadata extraction at the client, the metadata exchange between the client and the service, and the metadata-based lookup by the service all be lightweight. If not, high processing overheads at the client-side or server-

[2]We use the terms image and photo interchangeably.

side, or large delays incurred in transferring the metadata over the network, can render moot our goal of reducing image upload times by suppressing the uploads of redundant content; the same time can instead be spent to simply upload the image.

To address this trade-off between minimizing the metadata-related overhead and maximizing the accuracy of suppressing redundant content, we break up the photo uploading process into multiple phases. Our goal here is that, when a client is attempting to upload a photo, if no similar image was previously uploaded to the service, we seek to determine this with the least amount of metadata exchange, so that the upload of the image's content can begin at the earliest. For this, the first phase involves the exchange of a very small amount of coarse level metadata between the client trying to upload the photo and the service, which allows us to determine if a more careful comparison is even necessary; if there are no images that match the candidate image to be uploaded even at this level, the client can simply proceed with the upload.

If matches are found in the first phase, we employ a series of vision algorithms and exchange fine-grained metadata to increase the fidelity of the comparison. We first seek to minimize the amount of metadata/processing needed (we combine [34] and [35]). However, this does not adequately ensure that false positives are rare. Hence, we slightly increase the overhead by adding additional metadata using the approach in [36]. While this allows us to bring down the false positive rate, we are still unable to reach a reasonably high true positive rate. Hence, we incorporate a third phase which involves human feedback based on thumbnails (again, we increase the metadata by a small amount) returned from the service; this drives up the true positive rate without introducing additional false positives. In combination, these three phases are able to significantly reduce the amount of redundancy in transferred content and thereby the congestion, while ensuring very low false positive rates and low processing overheads at the client and the server.

Our contributions: In this paper, we propose a framework for identifying and suppressing the transfer of redundant image content in bandwidth constrained wireless networks. A key component of our framework is the aforementioned three-phase approach for metadata exchange between the generators of content (smartphones) and the service which receives the images. The framework allows a client to estimate if the service is already in possession of content that is similar to that in an image being considered for upload. If the estimation suggests that this is the case, the image upload is suppressed and deferred for a lazy transfer at a later time when conditions are more benign; else the transfer proceeds.

We implement and evaluate our approach on a 20-node Android smartphone testbed in various conditions with the Kentucky [5], and an US cities image data set that we put together. We find that our multi-stage approach for uploading images correctly identifies the presence of similar images on the service with ≈70% accuracy, while ensuring a low false positive rate of 1%. More importantly, our framework's suppression of uploads of similar content enables the network to tolerate 60% higher load (for target delay requirements), as compared to a setting without our framework. We obtain similar results even at scale, when using ns-3 based simulations. Finally, we also show that the overheads imposed by our framework in terms of bandwidth and energy are very small, therefore making it viable for use.

3.2 Related Work

Improving network performance and reliability during disasters or flash-crowd events: There has been research on the impact of flash-crowd events [37, 38] and natural disasters [31, 3] on network performance and connectivity. Proposed solutions allow the network to adapt and survive in such scenarios [39, 40]. From among these, the work that is closest to ours is CARE [4], which is a framework for image redundancy elimination to improve content delivery in challenged, capacity-limited networks. While the premise is similar, our work differs in terms of how image similarity is detected. In CARE, it is assumed that central infrastructure is unavailable and thus, content is transferred in a peer-to-peer fashion. Similarity detection takes place locally at a chosen node, where the images to be compared are first made available. This node transfers unique images when a DTN (delay tolerant network) relay with infrastructure connectivity is available. In our system, we assume that users have access to central infrastructure; only metadata that is extracted from the images is used for similarity detection. Our goal is to preemptively suppress the uploading of similar images on the bandwidth constrained wireless network. We acknowledge that the authors of CARE were the first to suggest the use of similarity detection in images to reduce content; we believe that our proposed work is complementary to CARE, both in terms of the setting considered and the actual approach itself.

Data deduplication in network services: Orthogonal to our work, data deduplication has been used to reduce storage capacity [41] and bandwidth [42, 43] requirements in systems which involve storing and moving large amounts of data. However, these efforts do not consider content semantics as we do here.

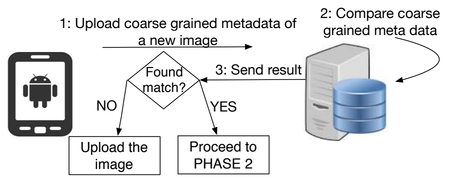
Image similarity detection: Our work leverages state-of-the-art approaches in computer vision for image feature extraction and object matching. Over the last decade, many algorithms have been proposed for robust extraction of global [35, 44] and local key-point [34, 45] features. The bag-of-words (BoW) approach, which had been originally used in text document classification, was applied in computer vision for image classification and matching by building a visual codebook from image local key-points [46]. The min-hash technique was proposed by Chum *et al.* [35, 47] to effectively estimate similarity between images represented in the BoW format. Recent work has been focusing on geometry verification to improve similarity detection accuracy [36, 48]. In section 3.3, we describe in detail how we effectively combine these techniques to create a lightweight, yet accurate image similarity detection system.

3.3 Efficient, lightweight detection of redundancies in images

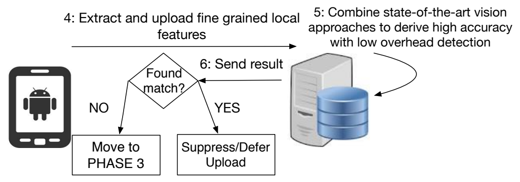
Our goal is to determine if there are similarities between images that are to be uploaded by a plurality of spatially disparate uncoordinated clients. We seek to do so with a very low overhead while still sustaining a high accuracy for detecting similar images. We envision that these images are to be transferred over a wireless network to a central server. The server has access to all the images that were previously uploaded to it.

3.3.1 Our framework in brief

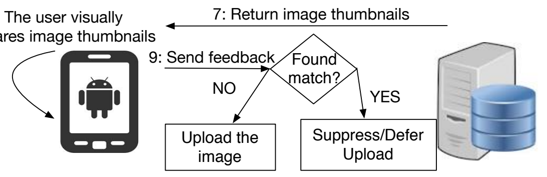
Figure 3.1 presents an overview of our framework, which can be adopted by any service to which users uploads photos, such as Flickr or Facebook, or even a server at a


1: Upload coarse grained metadata of a new image
2: Compare coarse grained meta data
3: Send result
Found match?
NO
YES
Upload the image
Proceed to PHASE 2

(a) Phase 1: Coarse grained metadata matching


4: Extract and upload fine grained local features
5: Combine state-of-the-art vision approaches to derive high accuracy with low overhead detection
6: Send result
Found match?
NO
YES
Move to PHASE 3
Suppress/Defer Upload

(b) Phase 2: Fine grained feature matching


8: The user visually compares image thumbnails
7: Return image thumbnails
9: Send feedback
Found match?
NO
YES
Upload the image
Suppress/Defer Upload

(c) Phase 3: Thumbnail feedback

Figure 3.1: Our framework for determining and suppressing images that contain similar (redundant) information

disaster response control center. When a new image is considered for upload to the server, a small amount of metadata is first extracted from the image and transmitted to the server. The server compares this metadata with that from images that were previously uploaded, and determines if a similar photo is already available. If this determination yields a positive outcome, the photo upload is suppressed for the time being; else the device seeking to upload the image proceeds to do so.

This seemingly simple high-level approach has three phases, with the aim of reducing the overhead associated with identifying similar images. The first two phases form a hierarchical, automated approach for image similarity detection. First, when a client seeks to upload an image, it extracts certain coarse-grained global features and sends these to the server. If the server finds that there is (are) a previously uploaded image(s) with similar features, it invokes the second phase. In this phase, the client intelligently combines state of the art vision algorithms to extract fine-grained local features from the image. A compact representation of these features is then sent to the server. The server performs a further comparison of these features with those in its pre-existing set of images. If there is a further match, it is deemed that similar content exists, and the upload of the candidate image is suppressed.

For all images that pass the first check but fail the second check, the server sends back thumbnails of a small set of the closest matching images in its pre-existing set to the client. In fact, in the scenarios of interest, a small set of pre-existing images may turn out be the closest matching ones to multiple images (being uploaded by disparate users) that are being considered for transfer; in such cases, the server can simply broadcast these thumbnails. If a client device is in the possession of a human user (e.g., a smartphone), the user can look at the thumbnail and then make a final decision on whether or not to continue with the image upload.

Table 3.1 provides a summary of the techniques used in our framework. In the subsequent subsections, we elaborate on how these techniques are combined to efficiently detect image similarity.

Scope of our work: While our approach is applicable to different forms of rich content, we limit ourselves to image/photo uploads/transfers in this work. Extension of the work to video is possible [49] but will be considered in the future. Further, our focus is the identification/suppression of redundant content in this work. In scenarios such as disasters, it is conceivable that some images are more important than others (e.g., a human

Technique	Usage	Goal	Section
OCS color histogram	In Phase 1: Compare Euclidean distance between color histograms to determine server has candidate similar images	Lightweight, but coarse-grained similarity detection	3.3.2
ORB local key-points	In Phase 2: Capture distinctive patches on an image; these can be matched to find similar images	Facilitate highly accurate similarity detection; uses image local features	3.3.3.1
BoW representation	In Phase 2: Compute the Bag of Visual Words (BoW) representation of an image by mapping its key-points into pre-computed clusters	Provide the inputs for computing the min-hash values	3.3.3.2
Image min-hash values	In Phase 2: Convert a BoW representation into a fixed number of hash values	Reduce communication and processing overhead	3.3.3.3
Geometry visual phrases	In Phase 2: Add geometry information to reduce false visual word matches	Reduce false positive rate	3.3.3.4
Thumbnail feedback	In Phase 3: Feedback image thumbnails	Use user input to increase true positive rates	3.3.4

Table 3.1: Summary of techniques combined to form our framework

in need of rescue versus a damaged uninhabited vehicle). Thus, one could conceivably target prioritizing image uploads based on content; however, we defer studies of such possibilities to the future.

In this work, we assume that if there is no prior image that is similar to the one that is considered for transfer, the image is transferred; else it is suppressed. We do not take into account things like the quality of the image (e.g., resolution), or the coverage (e.g., close up versus wide angle) as criteria for the above determination. Accounting for these factors is a harder challenge; the service will need to delay image transfers, compare all metadata from images that are being considered for uploaded and explicitly pull images from a chosen client based on some criteria (e.g., HD quality image with a close up of a house). Furthermore, the vision algorithms that we use here will not provide such assessments.

Our work primarily targets public services that require the transfer of images (e.g. photos transferred during a disaster to facilitate rescue operations). Our approach can be potentially leveraged in flash crowd scenarios where bandwidth is scarce; for example, an image sharing service can use our approach to provide mobile users an *option* to temporarily point to a similar version of an image (that they seek to upload), which is already available on the server side. A seamless upload and replacement with the user's own image could be done lazily when the network is under less duress; from the user's perspective, such an approach would save both on the data usage (if WiFi was used instead of 4G later) and energy costs that could be heavy due to retransmissions when the bandwidth is poor.

Finally, we do not leverage device features (e.g., GPS location, geotags, camera orientation) to assess if two images could be similar; these features could be useful in reducing the search space at the server side (e.g., it can compare images that are taken by cameras in close proximity only). Leveraging such features is orthogonal to, and can be used in conjunction with our framework.

How do you determine if content in two images is similar?: Whether or not the content in one image is similar to that in another is a subjective matter; different human users may perceive things differently and with respect to different images as well. Moreover, a general user may choose to upload his image regardless of whether or not someone else has uploaded a similar image. We assume that (i) savings in terms of data usage and energy will incentivize users to suppress their image transfers, especially when the network is congested and, (ii) in scenarios such as disaster recovery, smartphones could be used by the relief crew, who will want to suppress redundant content to reduce congestion and thus, aid relief operations.

In this work, we seek to ensure that if it is highly likely that a typical human does not perceive that two images are similar, they are classified as dissimilar. In other words, our framework must minimize false positives when classifying images as similar. Keeping this primary goal of a very low false positive rate, we seek to eliminate redundancies via such similarity detection to the extent possible, using state-of-the-art computer vision algorithms. We use known data sets (discussed later) to get objective evaluations of our framework; these evaluations show that our framework is extremely effective in decreasing network congestion.

3.3.2 Phase 1: Use of a coarse-grained global feature

Global features capture the entire content in an image. Examples include the color pattern or the scene pattern in the image. A global feature is represented by a single feature vector. As color is an important image attribute, a histogram of the color distribution in an image is widely used as a global feature for determining if two images are similar.

To construct such a histogram, we use the opponent color space (OCS) [35] to determine image similarity. We use the OCS color space, since it is not very sensitive to illumination (brightness) variations, unlike the RGB space. In brief, there are three components in the OCS space: an intensity component and two opponent colors. The components in the RGB color space can be used to compute the OCS color components using the following equation.

$$\begin{aligned} I &= (R+G+B)/3 \\ O_1 &= (R+G-2B)/4+0.5 \\ O_2 &= (R-2G+B)/4+0.5 \end{aligned} \tag{3.1}$$

The intensity component is quantized into 64 bins, while the other two components are quantized into 32 bins. The histogram vector is normalized so as to represent each component with 1 byte; thus, 128 bytes are used overall to represent the histogram. Once these 128 bytes are sent to the server, the server compares the bin values with those of the images that it has in its data set (previously uploaded). If the Euclidean distance of the histogram of any image on the server side and the histogram of the image about to be uploaded is less than a threshold τ_1, the system enters the second phase for similarity detection; otherwise, the client uploads the new image.

3.3.3 Phase 2: Using fine-grained local features

In the first phase, only the global distribution of colors and intensity were examined. If the server finds matching histograms, in the second phase, finer grained local features are extracted from the image and uploaded as metadata for further comparisons. Contrary to global features, local features are extracted from small patches in the image. When combined together, such local features (called key-points) represent the characteristics of the entire image. Fine-grained local features can be used to detect image similarity with high accuracy. Our approach for using local features consists of the following steps.

3.3.3.1 Extraction of key-points from an image

As mentioned above, the local features that we compare in order to assess the similarity of images are *key-points*. Key-points are small patches of an image that differ significantly from the surrounding areas (in the image). In computer vision, SIFT (Scale Invariant Feature Transform) is the most widely used algorithm for determining the key-points in images [45]. However, SIFT typically imposes a very high processing complexity and is thus, not a viable solution for resource (battery) limited devices like smart-phones. In our experiments, extracting key-points of a high resolution scenery image (approximately 2 MB of data) with SIFT requires about 30 seconds or even more.

Hence, we choose ORB [34] as our algorithm to extract image key-points, instead of SIFT. Experiments from other research groups have shown that ORB is about two orders of magnitude faster than SIFT while offering comparable results in many situations [50][51]. Each ORB key-point is described by 256 binary digits, whereas with SIFT, each key-point is described by a 128-dimensional vector. The number of key-points depends on the image size, the image resolution and the number of objects in the image. Normally, the amount of data associated with the key-points in an image is far greater than the size of the image itself! Therefore, directly comparing and matching key-points of images is not an option for our framework; this would violate our goal of exchanging a very limited amount of metadata for determining the similarity across disparate images.

3.3.3.2 Bag of Words (BoW) representation

Instead of directly working with image key-points, we use the bag-of-words (BoW) approach [46] to build what is called a "visual codebook." Any image can be represented as a bag of *visual words*, which is much more compact than simply representing the image via key-points. We describe below how the visual words are determined. For now, we point out that a visual word in the ORB representation is simply described by a 256 bit-vector; each element of the vector is called a dimension. A comparison of the visual words representing two images could be used to determine if the two images are similar. Representing an image by a bag of visual words is performed as follows.

Determining the visual words: First, the ORB key-points of a large set of representative images are extracted. These key-points are all grouped into a pre-defined number of clusters using any good clustering algorithm. In our approach, we use a modified version of the k-means clustering algorithm to partition and group binary vectors [52]. With this algorithm, the input key-points are mapped onto k different clusters based on the Euclidean distance between the key points and the cluster centroids. However, the Euclidean distance is not suitable for binary data such as the ORB key-point descriptors. Therefore, in our framework, we use the Hamming distance instead of the Euclidean distance. The Hamming distance between two binary vectors is simply the number of bits that are different in the two vectors.

The centroid of each cluster is randomly chosen first but is iteratively refined, as key points are added to the cluster; details are available in [53]. With the binary vector representation, in order to determine the centroid of a cluster, we count the number of zeroes and ones in each of the 256 dimensions, for all the data points (key points) that are associated with the cluster. If the number of zeroes is greater than the number of ones, the value of the corresponding dimension for the centroid is a zero, else it is a one. If there is a tie between the number of zeroes and ones, the value of the that dimension for the centroid is randomly assigned as either a “0” or a “1”. Each such cluster centroid is then considered to be a visual word in the aforementioned codebook.

When an image is considered for transfer, each ORB key-point in the image is mapped on to the closest cluster centroid in terms of the Hamming distance. With such a mapping, each image is now represented by a histogram of visual words; the number of key-points mapped on to a cluster reflects the *value* of the corresponding visual word in the histogram.

We wish to point out here that the codebook can be pre-loaded onto the clients when our software framework is installed (under benign conditions of connectivity); thus, there is no need to exchange it each time a comparison is to be done across images. In the BoW approach, the number of clusters is generally chosen between tens of thousands to hundreds of thousands.

3.3.3.3 Reducing detection overhead using min-hashes

Transferring the histogram of visual words constructed as above will incur significant overhead since it would require at least $n * k$ bytes if each component in the histogram can be represented by n bytes and we have k such components. For example, even with n = 2 and k = 50000, this corresponds to 100 KB. To adhere to our goal of having very little overhead of exchanging metadata, we use the min-hash approach proposed by Chum *et al.* [35, 47] in conjunction with the BoW representation.

To define the min-hash function of an image in the BoW representation we do the following. First, if the value associated with a visual word in the histogram is greater than 0, the word is simply considered to be included in a set that is associated with the image. Simply accounting for whether or not a visual word is present in an image (as above) is a weaker representation of the BoW vector, since the number of occurrences of a word is not taken into account. Now that each image is simply represented by a set of words, the similarity of two images with sets of visual words I_1 and I_2 respectively, is defined by the following equation.

$$sim(I_1, I_2) = \frac{|I_1 \cap I_2|}{|I_1 \cup I_2|} \tag{3.2}$$

The min-hash function approximates the similarity in Equation 3.2 between two images as follows. Let h be a hash function that maps all members (visual words) of set I (or I') into distinct integer numbers (called labels). The min-hash value of an image I is the minimum from all the hash values associated with the visual words in that image. Formally, for each visual word X, a unique hash value $h(X)$ is assigned. The min-hash value of image I is $H(I) = min\{h(X), X \in I\}$. The client uses M different assignments of labels to the visual words. Specifically, let us say there are N visual words; each is assigned a unique label value $\in \{1, N\}$ at random. This is referred as one assignment or permutation. The client can perform M (different) such permutations and compute the min-hash value in each case. The number of identical min-hash values between two images can be used to assess the similarity level between the two.

We use an example from [35] to demonstrate how the min-hash approach works. Consider a vocabulary of six visual words A, B, C, D, E and F and three different images. Each image contains three of these visual words, specifically, $I_1 = \{$A,B,C$\}$, $I_2 = \{$B,C,D$\}$ and $I_3 = \{$A,E,F$\}$. For each image, 4 min-hash functions are generated by using different permutations as shown in Table 3.2. For example, the min-hash of I_1, corresponding to the first permutation (row 1) is the value associated with C and is thus equal to 2. As I_1 and I_2 have 3 identical min-hash values out of 4, their similarity is $\frac{3}{4}$=75%; for the same reason, the similarity between I_1 and I_3 is $\frac{1}{4}$=25%.

Proof sketch: The skeleton of a simple proof for why min-hash approach yields the similarity between two images is as follows. Let $\pi(S)$ be a random permutation on a set S, and let X be the element which has the minimum hash value in $\pi(I_1 \cup I_2)$. Because π is a random permutation, the probability that X is *any* element in the set $I_1 \cup I_2$ is equal for all elements. If $X \in (I_1 \cap I_2)$, then obviously, $H(I_1) = H(I_2) = h(X)$. Otherwise, without loss of generality, assume $X \in I_1 \setminus I_2$; then, $H(I_1) < H(I_2)$. To summarize, two images have the same min-hash value if and only if the element X, which has the minimum hash value, is included in both of them. It is easy to see that, as a consequence, the probability that two images I_1 and I_2 have the same min hash value is equal to their similarity, i.e., $sim(I_1, I_2)$, as defined in Equation 3.2.

3.3.3.4 Improving detection accuracy by using geometry visual phrases (GVPs)

In the bag of words representation and the inferred min-hash information, the geometry information of each key-point (for example, the location of a key-point in the

	A	B	C	D	E	F	I_1=ABC	I_2=BCD	I_3=AEF
permutations	3	6	2	5	4	1	2	2	1
	1	2	6	3	5	4	1	2	1
	3	2	1	6	4	5	1	1	3
	4	3	5	6	1	2	3	3	1
	Label assignments						Min-hash values		

Table 3.2: An example of min-hash functions: Four permutations of label assignments are shown.

image) is lost. To reduce the likelihood of false matches because of the above, we use geometry visual phrases (GVP) in combination with the min-hash values. This reduces the false positive rates significantly.

We describe in brief how GVPs are computed and used; more details are in [36]. Towards determining the GVPs between two images, each image is divided into a fixed number of bins (same for both images regardless of the size of the image). In each image, each key-point corresponding to a visual word is then mapped into this offset space and is represented by the co-ordinates $\{x, y\}$, of the bin index to which it belongs; this is referred to as the geometry information. Each pair of equal min-hash functions identify a visual word that occurs in both images. For each of these visual words, the differences in geometry information of the key-points (denoted by Δx and Δy) are computed. If these "difference" values for say L visual words are the same, this implies that these L key-points are likely to be mapped onto the same (corresponding) objects in the two images and thus, are said to form a *co-occurring* GVP of length L. To illustrate, let us consider the example in Figure 3.2, which shows two different images of a house (found online) that was damaged due to hurricane Sandy. The Δx and Δy values for the key points A, B and C are all $\{0, 0\}$. Thus, these three points together could potentially map onto something common in the two

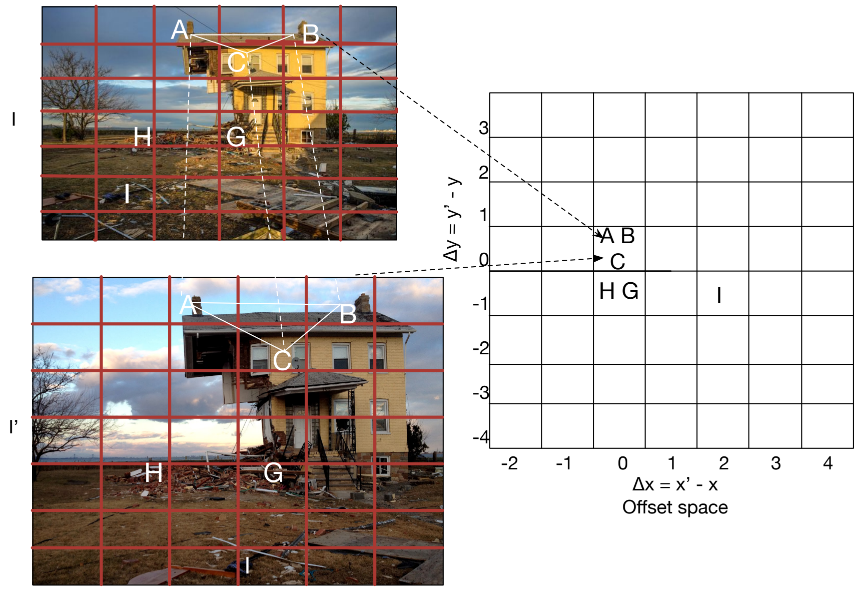


Figure 3.2: An example of visual phrases between two images

images and this forms a GVP of length 3. Similarly, for the two key points G and H, the Δx and Δy values are 0 and -1 respectively; thus, these two key points could potentially map on to identical constructs in the two images. This is a GVP of length 2.

Given two images I and I', the similarity score based on visual phrases of length L is defined in equation 3.3.

$$sim^L(I, I') = \sum_s \frac{\binom{M_s}{L}}{\binom{M}{L}}, \tag{3.3}$$

In equation 3.3, M_s is the number of key-points in bin s of the offset space, and thus $\binom{M_s}{L}$ is the number of GVPs of length L in that bin. For example, in bin {0,0}, there are 3 key-points viz., A, B and C; if $L = 2$, the number of length-2 GVPs in the bin is 3, corresponding to AB, AC and BC. Thus, the numerator on the RHS of equation 3.3 is simply the total number of GVPs of length L between two images. The denominator, $\binom{M}{L}$, corresponds to the maximum possible number of GVPs of length L that can be formed between the two images, given that M min-hash functions are used. Hence, the similarity score between two images is the number of GVPs of length L that are common between them, normalized by the maximum possible number of GVPs length L that can be created by using M min-hash functions.

If the similarity scores between the user's image and a candidate image is greater than a threshold (say τ_2), the images are deemed similar.

3.3.4 Phase 3: Thumbnail feedback

If at the end of phase 2, if the server finds no matches for the image considered for upload, it invokes an optional phase 3, seeking user input for finally making a decision on whether or not to have the image uploaded.

Upon failing to find a match in phase 2, the server rank orders the images in the candidate set based on the similarity scores with respect to the image being considered for upload. For each of the top k images in this ordered list, a small thumbnail is sent back to the client device; photo sharing services typically generate a thumbnail for every image at the time it is uploaded [54]. The user of the client device can visually compare her image with the received thumbnails and assess whether or not similar images are already available at the server; based on this, she can decide whether or not to transfer the image.

3.3.5 Handling parallel transfers of similar content

Thus far, we implicitly assumed that an image being considered for upload is compared with images that were already uploaded previously and stored in the server database. However, it is quite possible that in our scenarios of interest, multiple user devices attempt to upload similar images almost at the same time (close to when the event is occurring). Due to the shared access to the wireless medium, these attempts could be proceeding in parallel. If bandwidth is limited, it becomes important to reduce the load especially in such settings; for example, multiple such critical events (people needing to be rescued) could be ongoing at the same time and it is desirable to have (unique) information associated with all such events. The challenge here is to essentially compare such parallel uploads and determine if such attempts are towards transferring similar content.

When a client sends an OCS histogram of a new image, the server inserts an entry (with this information) for the image, into a queue. When there are other parallel uploads, the server not only compares the histogram of a candidate image with that of the images in its database, but also with the histograms of entries in this queue. If there are similar histograms (in either the database or in the above queue), the client is instructed to upload the local features as before. When the server receives the local features of an image, it associates them with the proper entry in the queue. It then compares these local features with the images from the database that were classified to be likely candidates with similar content *as well as* the local features of entries in the queue that are already available (with matching histograms). If there is a match with either, the image transfer is suppressed and the corresponding entry is deleted from the queue; else, the client is instructed to upload the image. After an image is completely received, the corresponding entry is deleted from the queue and the image is added to the server database. We point out that we only use the first two phases in determining if images that are considered for upload almost simultaneously, are similar. When this determination is taking place, only metadata of the images is available at the server side (the thumbnails of such images are not yet available). Since we desire that the decisions on whether or not to upload be quick, we avoid waiting for the complete information towards generating thumbnails and providing subsequent user feedback; this would cause delays and affect user experience. However, recall that the thumbnail feedback in phase 3 is mainly to help increase the true positive rate; thus, for the images considered for upload almost simultaneously, our system still achieves a low false positive rate.

3.4 System Implementation

In this section, we describe the prototype implementation of our framework. Our prototype consists of a central server which accesses a database where a set of previously uploaded images are stored. A number of mobile client devices generate new images and attempt to upload them to the server. We use the Kentucky image data set (described later in Section 3.5) to learn the appropriate values for the parameters in our implementation.

3.4.1 Image server

The server stores the images that it receives in a central database. For each image, it extracts and stores the image's 128-byte OCS histogram and the image's min-hash values as described in Section 3.3. At the server side, we choose to construct 512 permutations (recall Section 3.3.3) and determine the corresponding hash values for each image; each hash value is stored using 2 bytes. As one might expect, the larger the number of permutations, the higher the accuracy in similarity determination. However, Zhang *et al.* [36] showed that when more than 512 hash functions are used, the gain in accuracy is at a point of diminishing returns due to an increase in the imposed processing overhead.

For each hash value, the server also stores the geometry information of the visual word that corresponds to that min-hash function (the visual word which is assigned the minimum value by the hash function). Specifically, for each visual word, we store the x,y indices of the bin to which the key-point associated with that visual word belongs, as described in Section 3.3.3. For each image, we use a 10x10 bin-space as in [36]; thus, the geometry information of a min-hash value consists of 1 byte, including 4 bits for the horizontal bin index and 4 bits for the vertical bin index. Therefore, the total byte count to capture the local features of an image is 1536; this includes 1024 bytes for the min-hash values and 512 bytes for the geometry information. In total, we impose only 1664 bytes overhead for each image, together for both the global and local features. This is less than 1% of the size of a normal quality image taken with a modern smartphone. Our server application is implemented in C++ and uses the OpenCV library [55] to extract global and local features of the images.

Server operations: When a client application is about to upload an image, it sends the OCS histogram of the image first. The server searches its database to find images with similar histograms (the component values are within a threshold τ_1 of the incoming image's histogram) based on Euclidean distance. If such similar histograms are found, the corresponding images are considered as candidates for containing the content in the image to be uploaded; then, the client is asked to transfer min-hash values and the geometry information of its image. If no similar histograms are found, the client is instructed to upload its image.

In the next phase, when the local features of the image are received, the server calculates the geometry similarity score of that image with respect to each of the images in the candidate set according to Equation 3.3. Here, the scores are based on GVPs of length 2; it has been shown that this provides a good enough detection accuracy when compared to using GVPs of other lengths [36]. If any of these similarity scores is $\geq \tau_2$, the server deems that the content is similar and notifies the client application to suppress the image upload.

Otherwise, the server chooses those images that have the highest similarity scores and sends back thumbnails of these, to the client (a delayed multicast is possible to reach a plurality of clients, but we don't implement this). The human user of the client device can then check to see if the images are similar, and only choose to upload the image if she feels that they are not. We assume that users are objective and suppress an upload if a thumbnail is indeed of a similar image (we discuss our data sets and their objective usability for determining the similarity of images in Section 3.5).

Fast histogram matching in Phase 1: For each OCS histogram that is received (from the clients), the server needs to find the set of images in its database with similar histograms. The database could potentially contain a large set of images, and brute-force checks are thus not viable. To achieve an efficient search, we utilize the fast "k nearest neighbor" search (***knn***) to find a good approximate set of the candidate images. We use the FLANN library [56] which is freely available for knn search. The histograms of all the images on the server side are grouped into hierarchical clusters. Specifically, based on the histograms, all the images are first grouped into N clusters; each such cluster in turn is recursively partitioned into N sub-clusters and so on, up to a maximum number of iterations. Here, we choose a default value of N=32, as suggested by the FLANN library.

Choosing the key parameter for the **knn** *search:* One important parameter when using ***knn*** search is the value of k, the maximum number of nearest neighbors the library should return. The higher the value of k, the library explores a larger number of branches in the cluster-tree and is thus able to find a larger set of similar images. However, the search time also increases.

To determine a good value for k, we conduct an experiment using the Kentucky image set with 10200 images. This set contains groups of images that are similar; each group contains four images (ground truth). We build a test set that consists of 500 images, such that no image is similar to any other image in the set. For each image, we execute FLANN with different values for k, and record the processing times and the number of similar images found on the server. Figure 3.3 shows the search time (for query processing) for different values of k, and Figure 3.4 shows the accuracy of the search results in terms of the percentage of similar images that are found for a candidate image (as compared to the ground truth). These graphs show that, if we set k=500, we are able find over 90% of similar images with an expected processing delay of only about 33ms. Thus, we choose this to be the default value in our server implementation.

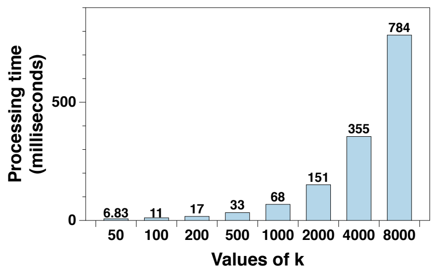


Figure 3.3: Search time for different values of k with $\boldsymbol{knn}$

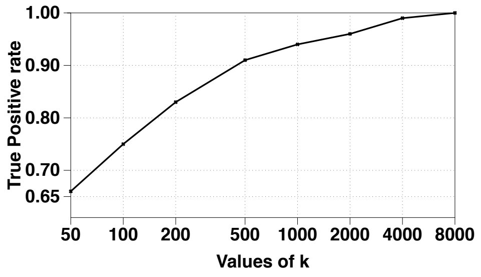


Figure 3.4: Percentage of images similar to an incoming image found

3.4.2 Client application

The client app is implemented on Android smart-phones using Java and native C++ code (JNI). Upon capturing an image, our client app is invoked for attempting an upload of the image to the server. The Java code is only for the graphical interface; the image processing code is written in C++ and is linked with the OpenCV library for feature extraction.

Client operations: First, the client application extracts the OCS histogram of the image and sends it to the server. It then awaits a server notification with regards to whether or not there are images with similar histograms in the server's database. If such candidate images exist, the client app extracts the local features, i.e., the min-hash values and their corresponding geometry information, from the image. To calculate the min-hash values, first the ORB key-points of the images are extracted. Next, the client application converts the image into the BoW presentation by mapping the key-points into visual words based on a vocabulary file. The vocabulary file contains the book of visual words (codebook) that is pre-built and preloaded from a set of training images (these are also available to the server). We choose to use a codebook of 20000 clusters. With this, the processing time at the client for each image is approximately 1.2 seconds. With a larger codebook, (e.g., with 50000 clusters) the processing time is around 3.5 seconds. Thus, choosing this larger codebook will degrade the performance of our framework (increased latencies). Furthermore, with 20000 visual words, we only need to use 2 bytes to represent each cluster and this limits the metadata overhead; larger numbers of clusters will increase these overhead costs.

Next, the client reads 512 different hash values (permutations) from a pre-built data file; for each permutation, a unique label is assigned to a hash value and thus, each visual word. It identifies the label of the visual word with the min-hash value for each permutation, and computes the geometry information for the visual word associated with that value. It sends both the min-hash values and the geometry information back to the server (local features).

Pre-installed data on client side: We pre-install a vocabulary file for the BoW processing and a permutation file for determining the min-hash labels for images on the client side. Thus, this information does not need to be exchanged for each image. With 20000 clusters, the size of the vocabulary file is only 640 KB; each cluster centroid of an ORB key-point is just 256 bits (32 bytes). The permutation file contains 512 permutations; each permutation in turn contains 20000 assignments which map each cluster on to a unique 2-byte label value. To reiterate, each permutation essentially reorders the identification labels to be assigned to the clusters. Thus, the size of the permutation file is $\approx$ 20MB. These files are also available to the server (which essentially provides the software at install to each client). The only time that these files need to be rebuilt is if there are large changes to the image database maintained by the server.

An alternative approach using SIFT key-points [45]: In addition to the ORB-based approach, we implement an alternative approach which leverages the SIFT key-points, as SIFT is the most widely used key-point extraction technique in image similarity detection [47][36]. We also use the OpenCV library (with the default parameters described by the author of SIFT in [45]) to extract key-points of the images. Each SIFT key-point is represented by a 128-dimensional vector, we use the k-mean technique to cluster the key-points and build a vocabulary (codebook) including of 20000 visual words. The SIFT key-points in each image are then mapped into the nearest cluster center by using the Euclidean distance, instead of the Hamming distance as in the ORB-based approach. After the vocabulary is built, the remaining steps in this approach, including building the BoW representation, computing min-hash and geometry distances are exactly the same as described in section 3.3, when ORB key-points are used. Be noted that the biggest concern when SIFT is used is its high processing overhead, as shown in [34][51]. Specifically, the average times to process one image in our dataset (described in section 3.5.1) on a Google Nexus 4 phone are 0.8 seconds and 13 seconds when ORB and SIFT are used, respectively. The impact of using SIFT instead of ORB on detection accuracy and transfer delay is discussed later in section 3.5.

3.5 Evaluation

In this section, we describe the evaluations of our framework. We perform experiments on a testbed of Android phones, as well as simulations using ns3 to showcase the performance as well as the benefits of our approach.

3.5.1 Training and test image sets

We begin with describing our image data sets and how we use them to evaluate the accuracy with which our framework can identify similar images. We use the Kentucky image set, which has been widely used in computer vision, primarily because of the availability of ground-truth information. We also use an image set of US cities that we collected on the Internet as described below.

The Kentucky image set [5]: The image set consists of 10200 images forming 2550 groups. In each group, there are 4 images of the same object taken from different angles; such images match our requirement/definition of similar images.

The city image set: We use the Bing image search service to find one image each for 5000 popular US cities. As these pictures are taken from different cities, there are no pairs of similar images in the entire image set.

We change the format and increase the file size of all the images to ≈ 700 KB using ImageMagick [57] in order to ensure that the evaluations are consistent across the different datasets that we consider. The tool converts the image to a different format (e.g., JPEG to BMP) in order to change the file size; thus the image is practically unchanged but the file size is now different. This size reflects the average size of normal-quality images taken by smartphones today[3]. Further, robust image key-points (such as SIFT or ORB, which we used in our approach) are scale-invariant; in other words, the key-points are stable even if the images are resized to some extent.

Building the training and test data sets: We evaluate the accuracy of our framework by partitioning our image data sets into a training set and a test set. We pre-upload images in the training set to the server and evaluate the accuracy with which our framework is able to identify images in the test set (when we try to upload these images) as having corresponding similar images on the server.

We randomly pick 2000 images from the US cities set and 2000 images from the Kentucky set to create a test set of 4000 images. Though images from the Kentucky data set are chosen randomly, we ensure that no more than 2 images are taken from the same group (the aforementioned 4 images of the same object). The remaining images from the Kentucky data set and the US cities data set are used as our training set; this set is used to build the codebook for the BoW representation and stored at the server.

To eliminate biases with a specific test set, we construct 5 different test sets (by randomly choosing images from the Kentucky and US Cities data sets) and the corresponding training sets. By default, the results reported are the average from our experiments with these 5 different sets. Note that by adopting the process above, we essentially ensure that for each image in our test set that is taken from the Kentucky set, there are at least 2 similar versions of the image in the server database; this can be used as ground truth while estimating our true positive rates in identifying similar/redundant content. For each image

[3]With modern smartphones, the average file size of high quality images can be between 2 and 2.5 MB [58].

taken from the US cities set, there is no similar version in the server database; thus, this set is useful in estimating the false positive rates with our framework.

Remarks: We do recognize that the Kentucky and the US cities data sets contain largely dissimilar images. We tried to perform experiments with a large set of images from a disaster scenario (Hurricane Sandy) but had difficulty in establishing the ground truth for the purposes of quantifying true and false positive rates. While we believe that our framework works well in such cases (based on some limited experiments where we tried to upload about a 100 photos and manually checked for similarity), we need a set of volunteer users to categorize whether the images are similar or not; for a large set of images, this was difficult to do. Using the Kentucky and US cities data sets allowed us to evaluate the accuracy of our framework without human involvement in an objective way.

When emulating human feedback based on thumbnails, we again rely on the objectivity possible with the above data sets. If two images are indeed similar (based on the ground truth), we assume that the user will correctly classify it to be the case; if the images aren't, we assume that the user will correctly decide to upload her image.

A training set of images (and a corresponding codebook) for a specific disaster location, can be built by using images of the same location before the disaster *and* images of the same kind of disaster (for example, an earthquake or a wildfire) that had previously occurred at other locations. A codebook built based on the two image sets is likely to contain key points that are similar to the key points in the images captured at the disaster scene. This determination is based on results from prior research on disaster images; specifically, Yang *et al.* [59] found that images of the same kind of disaster have many similar local features.

3.5.2 Experimental setup

Our experimental system consists of a server with an associated database; all the images in the training set and their global and local features are stored in the database. We have 20 Android-based smartphones as our client devices; the test images are divided equally among these phones. The smartphones connect to an access point (AP) on a WiFi network; the server is also connected directly to the AP via a 100 Mbps Ethernet cable. To emulate bandwidth constrained settings, we set the network bitrate to 6 Mbps. We experiment with different workloads (upload rates) from our smartphones. Note that we vary the network bandwidth in our simulations in Section 3.5.8; we also consider uploads using the cellular infrastructure in those studies.

Remark: Unfortunately, we were unable to showcase the performance of our framework via real experiments on cellular networks. Specifically, we do not have a sufficiently large set of phones to create enough load that strained the network bandwidth. Further, we were unable to determine the bit rate on the LTE links and could not accurately characterize the network load; thus, it was difficult to objectively quantify the benefits from our framework. However, in the scenarios of interest (e.g., disasters), we expect that there will be sufficiently large user activity that will strain the capacity of the network [4, 60].

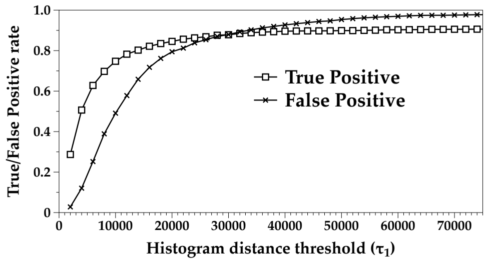


Figure 3.5: Accuracy with different histogram thresholds

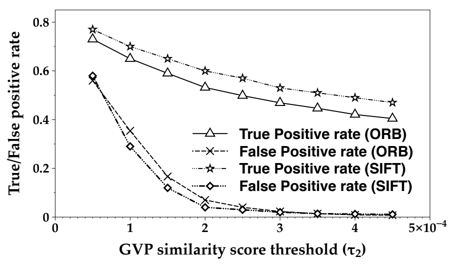


Figure 3.6: Accuracy with different GVP similarity scores

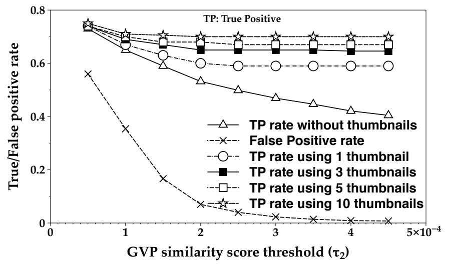


Figure 3.7: Accuracy with different numbers of image thumbnails

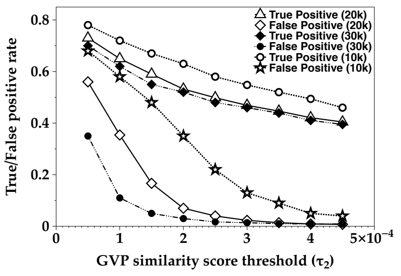


Figure 3.8: Accuracy with different number of words in the codebook

3.5.3 Accuracy of detecting similar content

Detection accuracy with global features only. First, we examine the accuracy with which Phase 1 of our framework determines if or not the server is in possession of a similar image as compared to one being considered for upload. Recall that image similarity is determined here only by comparing the global OCS histogram associated with two images. Figure 3.5 shows the true positive rates (correctly detecting similar content) and the false positive rates (wrongly classifying images as containing similar content) with different histogram distance thresholds. If we set a very low threshold (meaning that the Euclidean distance between the histogram of the image to be uploaded and a candidate image in the server database should be very small), we will end up not identifying any similar images; here, the true positive rate will be very low. To increase the true positive rate, we will need to increase the threshold so that we have a bigger likelihood of identifying candidate images in the server database, but this will have the undesired effect of increasing the false positive rate, since some wrong images in the database will also be classified as candidates for similarity checks. Based on Figure 3.5, we choose a threshold of τ_1=14000 towards achieving $\approx$ 80% true positive rate; however, this results in a 63% false positive rate, which we seek to drastically decrease with Phase 2.

An alternate distance measure: We did experiments with the Earth Mover distance (EMD) and we observed a very slight difference compared to using Euclidean distance. When using EMD, we are able to achieve a true positive rate of 80% but we encounter a false positive rate of $\approx$ 60% (compared to 80% and 63% respectively when using the Euclidean distance). The reason was that while color features are good at identifying similar images, they are not distinctive enough to identify dissimilar images. Since the complexity of computing EMD distance is also significantly higher ($O(N^3 logN)$ [61]), we use the Euclidean distance. In our experiments using the OpenCV library, we found that computing the EMD distance is 500 to 1000 times slower than computing the Euclidean distance with a 128-bin histogram.

Improving detection accuracy with local features. In Phase 2 of our approach, the assessment of image similarity is refined by calculating the similarity scores based on GVPs (see Equation 3.3). Figure 3.6 depicts the true and false positive rates after this phase, when different similarity score thresholds are used. We show results for two different approaches, (i) one which uses ORB key-points and (ii) the other using SIFT key-points. For both approaches, we observe that with a very low threshold, the false positive rate is very high (images are wrongly classified as similar) but then drops drastically as we increase the threshold. However, increasing the threshold decreases the true positive rate as well, since similar images are discarded for "not being good enough". To avoid missing critical image uploads, a very low false positive rate ($\approx 1\%$) is desirable. If we set a threshold to achieve this, the true positive rate is $\approx 47\%$ and $\approx 51\%$ with the ORB-based and SIFT-based approaches, respectively; this implies that approximately half of the images which have redundant content are detected and are subsequently suppressed at the end of this phase. These results also show that using SIFT only offers a slightly better detection accuracy while imposing a significant higher processing overhead (more than 15 times higher compared to using ORB, as described in section 3.4.2). Because of its high processing time, the SIFT-based approach has a noticeably undesirable impact on image transfer delays, which we show later in section 3.5.4.1. Thus, using ORB is a more viable option for resource-constrained wireless devices. Hereafter, we only focus on and show the evaluation results for our efficient ORB-based approach.

Feeding back image thumbnails to further increase the true positive rate. To further improve the detection of similar images, in Phase 3, the server sends back thumbnails to the user for visual inspection (Section 3.3.4). In our experiments, the size of an image's thumbnail is ≈ 11 KB; we believe that this a reasonably small volume of data needed for improving accuracy.[4] Figure 3.7 shows the increase in accuracy when different numbers of thumbnails are fed back to the user. With 3 to 5 image thumbnails, we find that the true positive rate increases to about 68% (from 47% with the ORB-based approach at the end of Phase 2). Beyond that, we find that we hit a point of diminishing returns; for

[4]The overheads due thumbnails are discussed later.

example, with 10 thumbnails, the true positive rate increases to just over 70%. Note here that the false positive rates do not change after Phase 3 (the human accurately determines if the content is similar or not).

Detection accuracy with parallel uploads. Next, we consider the case where multiple client devices are attempting uploads of similar content in parallel (almost simultaneously). Specifically, we conduct an experiment where three smartphones attempt parallel uploads of an ordered set of 500 images to the server. The database on the server side does not contain any images that are similar to the test images. We manually ensure that the images at the same position in the ordered sets at the three clients are similar; for example, the first image on the first client is similar to the first image on the second and the third client and so on.

First, we conduct the experiment without performing any similarity detection; as a result, all the 1500 images from the clients are uploaded to the server. Next, we implement similarity detection using the process described in Section 3.3.5. Here, we observe that only 585 images are uploaded to the server. Specifically, 500 images with unique content and 85 images with redundant content are uploaded; this corresponds to a 17% contribution from redundant information.

Impact of system parameters on accuracy. We vary each of the system parameters (e.g., the number of words in the codebook, number of histogram bins), and report the values that work best with our data set. Due to constraints on page count, we only show the impact of changing the number of visual words used in building the vocabulary (described in section 3.3.3.2). In Figure 3.8, we show the true positive and false positive rates with our system, when 10000, 20000, and 30000 words are used in the codebook. When 10000 words are used, the system gets a high true positive rate but suffers from a relatively high false positive rate. Since the number of words is relatively lower, similar key-points have a higher likelihood of being grouped in the same clusters; however, in some cases, dissimilar key-points end up in the same clusters as well. When 20000 or 30000 words are used, both the true positive and false positive rates drop. The figure shows that, when choosing a high GVP threshold to keep to false positive rate of $\approx 1\%$, using 20000 or 30000 words allow the system to achieve higher true positive rates. There are no further significant improvements when the number of words is increased from 20000 to 30000; thus, we choose to use 20000 words to build the vocabulary. Note that a similar process is used to learn the best values for the other parameters. We discuss how to choose the parameters in dynamic settings in section 3.6.

3.5.4 Impact of redundant content suppression on network performance

In this section, we seek to understand the impact of redundant content reduction on network performance. Specifically, we seek to quantify the impact on (i) the delay experienced during image uploads (where we can expect a decrease) and (ii) the total sustainable load (where we can expect an increase). We also quantify the overheads due to our approach.

For the experiments in this section, each smartphone sends a test set of 50 images, back to back to the server. The test set consists of 25 images from the Kentucky data set with similar versions on the server, and 25 images from the US cities set. The total size of the test set is ≈ 32 Megabytes. Subsequently, we vary the proportion of the redundant content in the test set and quantify the impact on network performance. First, we show the results in an ideal case wherein all redundant content is correctly detected and suppressed; in other words, we assume a 100% detection accuracy unlike what we expect in practice. With our framework, the proportional improvements are reduced by a factor equal to the complement of the true positive rate; we show this later in Section 3.5.7.

3.5.4.1 Delays under different network loads

The normalized network load (also referred to as simply network load) is defined as $\frac{n\lambda}{\mu}$, where n is the number of devices in the network, and λ is the load generated per device, and μ is the rate achievable on the transfer link. To vary load, we first fix λ, but vary the number of clients that are attempting image transfers per unit time (n). Each client transfers/suppresses one image completely, before attempting the next transfer. Specifically, we assume that a new image is generated every t seconds. We set $t = 6$ seconds (we have other results but do not report them as they are similar), which implies that a client device (given the 32 MB content volume consisting of 50 images) generates an average load of $\lambda = 0.85$ Mbps. With the wireless bandwidth of 6 Mbps (μ), this corresponds to *each* client generating a *normalized* load of 14% (0.85/6), on average.

Figure 3.9 demonstrates the reduction in the delay experienced under different normalized network loads, due to similarity detection/redundancy elimination. The delay experienced by an image is defined as the duration between when the image is generated and when it reaches the server; the delay is computed for only those images that are transferred to the server. We show the average upload delay values in three scenarios: (i) No similarity

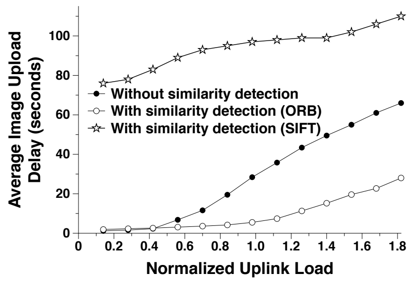


Figure 3.9: Impact on network load and image transfer delay (Ideal case with 100% detection accuracy)

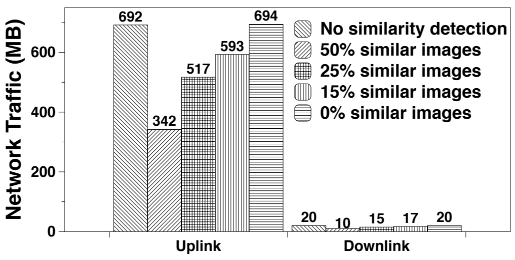


Figure 3.10: Varying the proportion of similar images at the server (Ideal case with 100% detection accuracy)

detection is employed (ii) Similar images are detected by using ORB key-points (iii) Similar images are detected by using SIFT key-points. Under light network load (e.g., below 40%), sending the images directly without any similarity detection/redundancy suppression is faster! This is because in these regimes, there is no congestion and it is possible to transfer images without much delay; the process of similarity detection adds processing/metadata exchange delays but does not contribute to a reduction in congestion.

However, when the load $> 50\%$, the network transitions into a congested state; this is the regime where redundant content reduction will benefit performance. The figure demonstrates that similarity detection (using ORB key-points) and redundancy suppression allow us to tolerate up to a 100% increase in load. Similarly, at high loads (e.g., at loads $>$ 1.0), more than a 100% reduction in the experienced delay is possible.

On the other hand, the SIFT-based approach imposes high computational overheads and increases the upload delay with all network loads. The time taken to process images becomes the bottleneck and dominates the upload delay (instead of network transfer time). In all cases, the upload delay is much higher than sending the images directly! These

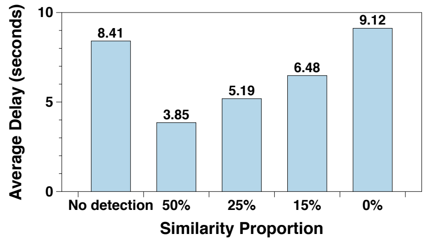


Figure 3.11: Upload delay with different proportions of similar images (Ideal case with 100% detection accuracy.)

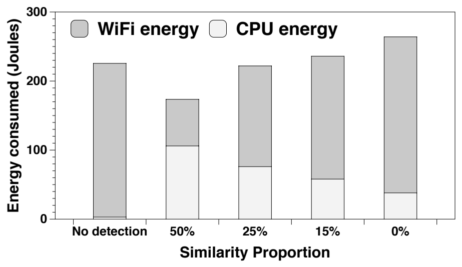


Figure 3.12: Impact on Energy

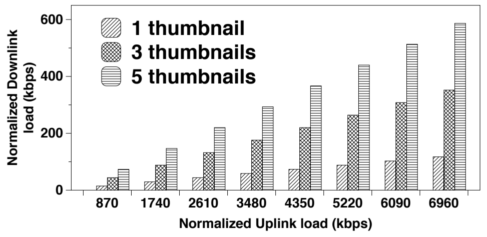


Figure 3.13: Overheads from thumbnails

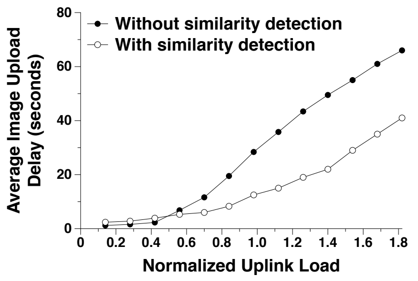


Figure 3.14: Upload delay with our framework (True Positive rate ≈ 70%)

results show that using ORB key-points, as in our approach, is an essential part for ensuring the effectiveness of our framework.

Note that these results are based on about 50% of the images to be uploaded having similar counterparts at the server.

3.5.4.2 Varying the proportion of similar images available at the server

Next, we vary the fraction of uploaded images that have similar counterparts at the server. In these experiments, we use `tcpdump` to capture network traffic transferred over the wireless network. We show the results when the normalized load is 0.6; the behavioral results are similar at other loads. The results are shown in Figure 3.10.

First, the figure depicts a case where the server does not contain any image that is similar to any image being considered for upload. In this case, there is an overhead associated with each image due to the metadata, but this overhead serves no purpose (images are ultimately uploaded). In this extreme case, we find that the performance with our framework is only slightly worse than in the case without it; the upload data volume increases from 692 MB to 694 MB. Second, as one might expect, as the likelihood of the server finding a similar image increases (the proportion of similar images present is increased), the performance with our framework improves in terms of a drastic reduction in network load. The figure shows that when the redundancy in content is about 50%, the decrease in network traffic (because of redundancy elimination) is in fact slightly higher than 50%. The main reason for this artifact is that the reduction in network load also reduces the overheads due to retransmissions of corrupted packets that are typically incurred, if an image is in fact uploaded. These experiments also inherently account for the uplink overheads with our framework; the results suggest that these overheads are extremely low (because the gains are as expected in an ideal setting with no overhead). Finally, the metadata overhead consumed in the reverse direction (from the server to the smartphones) is also depicted; this corresponds to a very small fraction of the upload content volume ($\approx$ 3%).

We also examine the delays incurred in transferring images, while varying the proportion of similar images available to the server. The results are shown in Figure 3.11. Again, if no similar images are present at the server for any of the images being considered for upload, there is a very slight increase in delay (from 8.41 seconds to 9.12 seconds) due to the metadata exchange and processing. This demonstrates the extremely low overheads with our approach. As the proportion of similar images increase, drastic reductions (54% when this proportion is 50%) in image transfer delays are realized with our framework as depicted in the figure.

3.5.5 Impact on energy consumption

Our next set of experiments capture the impact of our framework on the energy consumption on the client devices. Specifically, we pay particular attention to the (i) energy consumed due to processing, towards extracting local and global features, and (ii) the energy consumed by the network interfaces due to content/metadata transfers. We compare the energy consumed on smartphones with and without our framework.

We use the PowerTutor tool [11] to capture the energy usage on our smartphones. For clarity, we only show the results with a Sony Ericsson Xperia Arc phone in our test bed. However, we observe similar results on different phones from different vendors as well. Figure 3.12 shows total energy consumed and the energy breakdown when a set of 50 images is uploaded from the phone. It is observed that our approach only induces a very small energy overhead on client devices when the similarity detection fails in all cases (server does not have any similar images to the ones considered for upload). As the volume of the transferred data is reduced, the energy consumed by the WiFi connection is also reduced. However, the energy consumed due to processing increases (due to the computation of global/local features of the images). The highest difference in energy consumption is between when no similarity detection is deployed and when there are no images with similar versions on the server side; this is about 40 Joules. On today's modern smartphones, the battery capacity is around 1800mAh with a voltage of 3.7 Volts; the above energy overhead only corresponds to about 0.2% of the battery capacity. Given the large number of image transfers considered from each phone here, the overhead is likely to be even lower in practice and thus, will not adversely affect user experience.

3.5.6 Overhead due to thumbnails

As our final experiment on our Android testbed, we seek to quantify the overheads incurred in sending different numbers of thumbnails from the server to the smartphone clients. In this experiment, 50% of the images that are considered for uploads have similar versions on the server side. Figure 3.13 shows the normalized (upload) load and the corresponding overhead in terms of download load if 1, 3 and 5 thumbnails are generated for those images for which the server comes up with a negative result at the end of Phase 2, in our system. It is observed that when the upload rate is increases, the overhead of generating thumbnails also increases. However, even when the generated upload load is higher than the capacity of our link (6Mbps), the overhead of generating 5 thumbnails is still less than 10% of the link capacity. This demonstrates that Phase 3 of our framework is lightweight and is a viable option in practice.

3.5.7 Improvement in network performance with our framework in practice

Thus far, we have shown the improvements on network performance in an idealized setting where we assume that similarity detection can be performed with 100% accuracy. Next we show the impact on network performance with our framework, using our test set

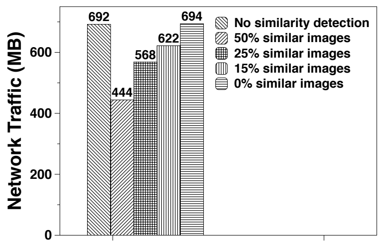


Figure 3.15: Uplink Network traffic with our framework (True Positive rate ≈ 70%)

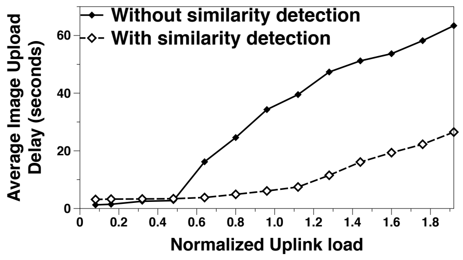


Figure 3.16: Delay vs load with WiFi (ns3)

of images. Here, the true positive rate is approximately 70% (as described in Section 3.5.1). Figure 3.14 shows the reduction in upload delay if the proportion of redundant content is 50%. Unlike in an ideal case, our framework is able to eliminate ≈70% of the redundant content (given the true positive rate achieved). The figure shows that when the load is high, the upload delay without similarity detection is about ≈44% higher (even though only ≈35% of the data considered for upload gets suppressed). The reason for this higher than expected delay reduction is the same as that in the ideal case. The retransmission overheads due to corrupted packets decrease (to significant extents in cases where the link quality is poor) as compared to a case without similarity detection (when the images actually get uploaded); this in turn further reduces the aggregate network load and thus decreases delay. The elimination in redundant content also allows the network to sustain a higher load. For a target expected delay of 30 seconds, the sustainable load increases by about 60% as seen in the figure.

Figure 3.15 shows the uplink network traffic when different amounts of redundant content are present. For the same reason as above (fewer retransmissions), the reduction in the total (uplink) traffic is typically higher than the proportion of redundant content that is suppressed except in the case where there are no similar images at all (0% of similar images). In this extreme case, all images are uploaded, and there are slight overhead penalties due to our framework.

3.5.8 Evaluations via simulations

Finally, we examine the impact of our framework on network performance using ns-3 based simulations, which allow us to experiment with different network set-ups and scales.

Simulation set-up: We evaluate the network performance with both WiFi and LTE: (i) In the WiFi set-up, all the mobile devices connect to the same wireless access point (AP) through an 802.11 link[5], which in turn connects to a server node through a dedicated link of 100 Mbps. All the clients are initially distributed evenly in a square area of 50x50 meters; the AP is positioned at the center of the area. We use a random walk mobility model for the clients; each client moves at a random speed in a random direction inside the area. The WiFi channel is characterized by a distance based loss propagation model. The channel bit rate is kept constant (but varied in different experiments) to allow us to compute the normalized load. (ii) In our LTE set-up, all the smartphones connect to the same base station (regarded to as an enb node in ns-3) through a LTE network; the base station is connected to a LTE gateway, which in turns connects to the server node via

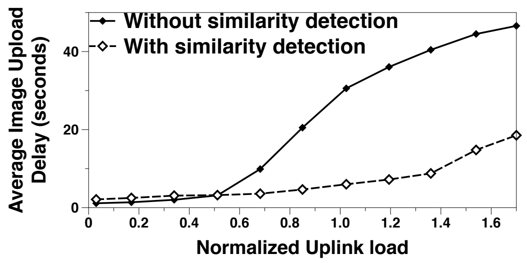


Figure 3.17: Delay vs load with LTE (ns3)

[5]We experiment with different 802.11 standards and observe similar results.

a dedicated link of 100 Mbps. In this set-up, all the LTE clients are initially distributed evenly in a square area of 100x100 meters; the base station is also positioned at the center of this area. We again use the random walk mobility model. The LTE channel is characterized by the default Friis path loss model. Again, the channel rate is kept at a constant value (but varied across experiments) in order to be able to characterize the normalized load.

Impact on image delay time: We examine the impact of our framework with different loads and with different numbers of mobile devices when 50% of the content to be uploaded is redundant, and is correctly eliminated. In all our experiments, under the same load, we observe consistent and very similar results even if the number of devices and data rates are changed. For simplicity, we only show one sample result from our WiFi experiments and one result from our LTE experiments. Figure 3.16 shows the results with a 54 Mbps WiFi network and 200 smartphones. Figure 3.17 shows the results with a 15 Mbps LTE network and 50 smartphones. In both experiments, a new image is generated every 10 seconds. These results match those obtained from the real experiments with our 20-phone Android testbed (shown in Figure 3.9); more than a 100 % improvement in the average network delay is achieved at high loads (> 0.8).

3.6 Discussion

Leveraging location and time metadata: Images taken from different places might exhibit high levels of similarity. For example, images of street corners which have similar high rise buildings, cars, and street signs may be classified to be similar. In such cases, one can leverage embedded GPS and timestamp information to reduce the false positive rate. In this work, we assume that the images are taken at approximately the same times by humans or devices, in the same geographical region; thus, image content is the primary material used for comparison in our approach. Extending this to exploit other sensors will be considered in future work.

Images are uploaded to different servers: Our approach requires the meta-data associated with the images to be available to a server that performs similarity detection. In image sharing services (e.g., Flickr), we assume that requests from the same geographical region will be served by the same servers. We also assume that the databases of servers in nearby proximity will be synchronized within a short period. Such synchronization is needed to ensure consistency across the databases and we expect content providers to implement algorithms for ensuring that this will be the case.

Scenarios of relevance: We target image-sharing services in cases such as disasters wherein the network is congested. We expect the users to be altruistic and provide access to the images they upload to search and rescue personnel or others. Since users may not all be able to upload heavy media content (videos/images) due to losses in infrastructure, we expect that they will approve of suppressing their uploads if needed. The user can upload her images when the network is less congested. This altruistic assumption has also been made in CARE [4]. Services such as CNN's iReport where users share stories about an event are apt use cases for our system.

Choosing system parameters: It is extremely challenging to come up with a set of parameters that will work well with different (variable) input data; thus, we conduct experiments and choose the best parameters for our dataset. An alternative approach would be to have multiple training data sets (subsets of the training set) and learn the best parameters for each such subset. When input data changes, the system can compare and identify the subset that is most similar to the new input data. Subsequently, the system can apply the parameters that work well with that subset to the new input data. We defer such possibilities to future work.

3.7 Conclusions

In this paper, we propose a framework for detecting similar content in a distributed manner, and suppressing the transfer of such content in bandwidth constrained wireless networks. Such constraints are likely to be imposed on networks during events such as disasters. With our framework, we seek to enable the timely delivery of every unique piece (possibly critical in some cases) of information. In building our framework we tackle several challenges, primary among which is the lightweight decentralized detection of redundancies in image content. We leverage, but intelligently combine, a plurality of state-of-the-art vision algorithms in tackling this challenge. We perform both experiments on a 20-node Android smartphone testbed and ns-3 simulations to demonstrate the effectiveness of our approach in decreasing network congestion, and thereby ensuring the timely delivery of unique content.

Chapter 4

ACTION: Accurate and Timely Situation Awareness Retrieval from Bandwidth Constrained Wireless Cameras

4.1 Introduction

Natural disasters usually have a high associated human cost; for example, the recent Nepal earthquake resulted in the death of more than 8,000 people, injury to more than 14,000, and over 300 people are still missing [6]. Today, advanced technologies can help in significantly enhancing search and rescue missions; sensors, often with camera capabilities can be deployed in the field, to provide situation awareness back to a central operations center or controller. Specifically, this is information with regards to particular objects of interest (e.g., distressed or injured humans, or animals) that would be critical in aiding search and rescue. In other situations (e.g., Boston marathon bombing), being able to quickly detect unattended objects such as luggage or backpacks using such cameras could help prevent tragic disasters from happening.

Unfortunately, in many such scenarios, in the aftermath of disasters the bandwidth is likely to be limited[1]. Blindly sending the video feeds from camera nodes in the field is likely to be unfeasible because of bandwidth limitations. As information collected from multiple cameras with overlapping views tends to contain content with a high level of redundancy, transferring all raw video content from all of the cameras is also likely to be wasteful. Doing so may also delay the transfer of key information with regards to some of the objects of interest. Finally, having to look at large volumes of video may cause an inherent information overload on humans who man the central controller.

In this work, "we seek to extract accurate situation awareness information from the camera feeds from a set of wireless cameras, and deliver it in a timely way to an operations center that handles search and rescue, when presented with bandwidth constraints." Before we describe the challenges in addressing this overarching objective and our contributions, we first formally define our view of how the situation awareness information is gathered and sent to the operations center. We envision that multiple autonomous camera-equipped sensors with possibly overlapping views, are deployed in the field and are used towards searching for particular objects of interest. The camera nodes possess processing capabilities and can

[1]Natural disasters come with at least some destruction of physical network infrastructure [2] and this impacts communications [62].

locally extract situational information about the objects of interest from captured video feeds. They can then report information extracted from the videos (e.g., frames or parts of a frame) to a central controller via a wireless network with limited available bandwidth. The controller sends a queries which seek for example, to determine whether there was an object of interest (e.g., a human, a vehicle, or a backpack) present at a specified location at a specified time.

Challenges: In order to achieve our overarching goal, we need to tackle some key challenges. First, we need to identify those videos that contain the "same" object of interest (at a given location and at a given time) autonomously; in essence an object needs to be re-identified across the cameras. This is critical in elimination of redundant content (only if the camera views are capturing the same object can they be considered redundant). Depending on the location of the object relative to the camera, today's vision algorithms might not be able to categorically determine if there is a real object in view. They can only provide an assessment of the accuracy of their detection. Thus, a challenge we need to address is "how to effectively aggregate information sent from multiple cameras to improve the quality of object detection?" Finally, the transfer of all of the raw data with respect to all the detected objects may still be beyond the network capacity. Thus, how can we identify redundant content, and choose only the most relevant sub-set of this content and transfer this information back to the central node?

Our framework in brief: Towards addressing the above challenges and providing accurate and timely situational awareness with regards to objects of interest in the field, we design and implement a framework that we call ACTION. ACTION has the following component modules: (i) each individual camera has a module that aids lightweight object detection from the video collected; here we leverage state of the art computer vision algorithms (ii) a novel module that facilitates coordination across multiple cameras to aggregate information towards (a) re-identification of an object across multiple cameras and (b) using the joint camera views of the object to improve the quality of detection, and finally, (iii) a module that, based on the previous step, selects a sub-set of the video feeds for transfer to the central controller, that provide the highest accuracy given a bandwidth constraint. In what follows, we briefly describe the functions of each of the three modules. To keep the narrative clean, we focus on human detection; with minor modifications, ACTION is applicable for the detection of other types of objects (e.g., animals or luggage).

ACTION in action: To facilitate effective object detection at each camera node locally, we leverage state-of-the-art detection algorithms from the computer vision community. In brief, a sliding window (covering a block of pixels) is used on key frames to detect whether or not an object of interest is present in that block. Unfortunately, even these algorithms may suffer from false positives or false negatives due to occluded views where the objects are partially covered by other objects. Thus, as discussed below ACTION combines the information from multiple camera views to significantly enhance accuracy.

Object reidentification across cameras: The first challenge ACTION resolves is to determine if what is classified as an object of interest by one camera is also perceived to be the same object by another camera (or cameras) with an overlapping view. This is referred to as the re-identification problem (objects are re-identified across cameras). ACTION uses a novel method that maps the 2D view to a 3D location. This 3D location as perceived by the different cameras, is used jointly with the color features of the object to perform the re-identification.

Maximizing accuracy using multiple camera views given bandwidth constraints: Given the bandwidth constraints, ACTION does not exchange raw videos among the cameras. Instead, all nodes with overlapping views extract metadata with respect to detected objects, and send this metadata to a common node (called the fusion node). The fusion node jointly examines the metadata and first resolves the aforementioned re-identification problem. Next, the ACTION software at the fusion node identifies the combination of camera views of a particular object that yield highest associated detection accuracy (lower false positive and false negative rates) while adhering to a timeliness constraint (that is determined by the bandwidth available for transferring the information). It essentially models the network as a knapsack, and the gain associated with each frame is dictated by the probability that an object is correctly detected in that frame. It then uses a greedy approach to select and send those frames that either satisfy the detection criterion, or fill the knapsack (sends all the frames that the bandwidth permits). It relays this information back to the cameras which transfer the actual frames.

Evaluations: We consider human detection (i.e., humans are the objects of interest) and evaluate ACTION's performance. Specifically, we emulate our scenario by implementing ACTION on Android devices preloaded with a dataset that consists of video sequences captured from 4 different cameras [7]. Our evaluations show that by considering views from multiple cameras, ACTION can detect $\approx 20\%$ more humans than when using the video from a single camera. Further, it achieves a very high accuracy rate of $\approx 90\%$, if an object is detected by at least 2 cameras (with a single camera it can be at most ≈ 72 %). In terms of resource usage, ACTION can reduce the bandwidth usage threefold, compared to uploading all the detected frames directly to the central controller. In addition, with ACTION, even though information from 4 cameras is used, the amount of transferred data is only ≈ 1.4 times higher than the amount of data transferred when one camera is used, while providing a significant higher detection accuracy.

As an auxiliary result we show that the energy overhead with ACTION is 102 J at the camera node and 39 J for the fusion node to process a video sequence of 3.2 minutes; this low consumption allows a camera node to last for about 19 hours (assuming a smartphone like battery).

4.2 Related work

We divide relevant related work into 4 main categories:

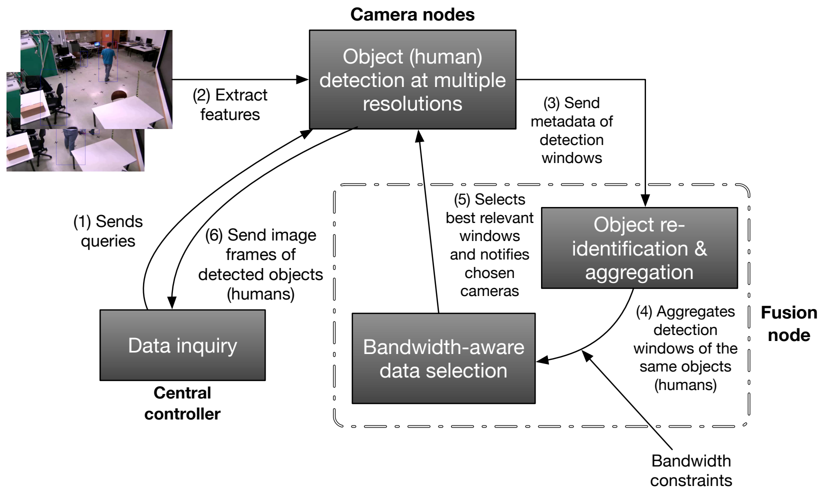
Object detection: We leverage state-of-the-art human detection algorithms from the computer vision community. These algorithms can be easily applied to detect other types of objects (e.g., vehicle, animals) with appropriate datasets. For human detection, different features [63][64][65][66] and machine learning techniques for building the detection model [67][68][69][70] have been proposed. We leverage the technique described in [68] to effectively detect humans in videos at individual camera nodes, in ACTION. Details on how we do so will be provided in Section 4.3.2.

Object association across camera views: Object association refers to the identification of the same objects (e.g., humans) captured from overlapping camera views.

In [71], people are assumed to stand on the ground; human positions are then computed by projecting the detected positions onto the ground-plane. Positions that are within a distance threshold on this plane, are considered as being associated with the same human. Implicitly they assume that the ground co-ordinate is known. However, we do not assume that this will be the case in ACTION; humans can stand on objects or on uneven ground. We provide a novel association mechanism (we call it re-identification) in such cases. In [72] and [73], color features of detected regions are leveraged (compared across cameras) to identify images of the same human. In ACTION, we jointly consider such color information with the 3D position information (obtained using our approach), relating to each human detected from different camera views, to achieve a high association or re-identification accuracy.

Detecting and tracking objects in multi-camera networks: In [74], multiple cameras collaborate to detect human heads; for each detected head position in an image, the associated 3D position of the head is estimated. Nearby head locations are identified and combined by comparing their Euclidean distances. Other efforts on object tracking [75][7][76] build complicated 2D to 3D mapping models which require high computational overheads. Since we seek to ensure low energy consumption, these cannot be applied in our framework. Further, data communications between the nodes is not a concern in these designs; in ACTION we seek to limit the amount of data transferred to a central controller.

Redundancy reduction in video transfers over networks: Recent related work considers the reduction of redundancy of content transferred over networks [4, 77]. In these efforts, metadata (or the entire video/image content) is exchanged between nodes and the controller to detect and suppress redundant content. However, they do not eliminate



Camera nodes
Object (human) detection at multiple resolutions
(2) Extract features
(3) Send metadata of detection windows
(1) Sends queries
(6) Send image frames of detected objects (humans)
(5) Selects best relevant windows and notifies chosen cameras
Object re-identification & aggregation
Fusion node
Data inquiry
Central controller
Bandwidth-aware data selection
(4) Aggregates detection windows of the same objects (humans)
Bandwidth constraints

Figure 4.1: Architecture of ACTION

unnecessary content (e.g., views that do not contain relevant objects). They are also unconcerned about the accuracy of detection of such objects (as is the case with ACTION); in fact, some redundant content may be transferred in ACTION to improve the accuracy of object detection.

Zhang *et al.* [78] build a multi-camera surveillance system. In their system, if an object is detected by multiple cameras, only *a single* view containing the object is uploaded; *all* other views that contain the same object are treated as redundant and suppressed. They only consider sufficiently high resolution input video feeds; and thus, all objects are considered correctly detected. In ACTION, information from overlapped cameras is "aggregated" to improve the detection accuracy and views are chosen depending on the bandwidth constraints.

4.3 The ACTION framework

In this section, we describe our framework, ACTION, for extracting and reporting situational-awareness information in bandwidth constrained multi-camera networks. To aid the narrative, we use human detection as a specific use case. In some parts of ACTION, therefore, we leverage state-of-the-art techniques in human detection from the computer vision community to identify the presence of humans from the video feeds. However, the same or very similar techniques can be used to detect other types of objects (e.g., vehicles, suitcases, animals)[2]. We begin with an overview and then, describe each of ACTION's modules in detail. Figure 4.1 depicts the architecture of ACTION.

4.3.1 Overview

The ACTION software is housed on three main components: camera nodes, a fusion node and a central controller. The camera nodes are deployed in the field. They may have overlapping views, i.e., a location on the field could be covered by multiple camera nodes. We assume that the camera nodes communicate with the central controller using either a cellular (i.e., 4G/LTE) connection, or a WiFi connection. The communications between individual camera nodes, and one of these chosen to be a fusion node is via a wireless channel (using 802.11 ad hoc connection temporarily set up at the scene).

We assume that a user who mans the central controller sends queries to the camera nodes. The queries seek the images of humans (the objects of interest in our use case) who appeared in the field of view within a specified period of time. Each camera node extracts

[2]This is because these techniques are based on machine learning and are trained using appropriate datasets collected in such settings; if an appropriate dataset with regard to the different types of objects are used to train the system, the algorithms can be used in these other cases.

features from the video sequences that it has captured towards detecting the presence of humans. When a human is found, metadata associated with the position of the human (we call this the detection window), is computed and sent to a fusion node. The content in the metadata includes information such as the timestamp, the probability (or confidence) that the detected object is a human, and location of the human in the snapshot image, etc.; more details are provided later in the section.

Upon receiving all the metadata from the camera nodes sharing overlapping views, the fusion node first performs re-identification of the human object across the cameras (to determine which camera views correspond to the same human). We simply assume that the fusion node is one of the camera nodes which share overlapping views (this set can be determined by transmission of beacons). We arbitrarily select one of these nodes for collecting and aggregating the metadata; more intelligent algorithms can be used to choose the fusion node [79, 80], but that is not the focus of this work.

After re-identification, the fusion node runs a greedy algorithm to identify the "most relevant" detection windows from the plurality of camera views. In brief, we formulate the problem as a variant of the classical Knapsack problem [81] and design an efficient greed algorithm to solve it. The available bandwidth is equally shared between among all detected humans. Thus, for each human, the bandwidth allocated dictates the size of the knapsack and the accuracy requirements determines which video frames are inserted into the knapsack (chosen for transfer). The fusion node notifies all the cameras about the windows that are selected for transfer by the greedy algorithm. Upon receiving this notification, the chosen camera nodes send the relevant frames directly to the central controller. We wish to point out that the fusion node only receives the lightweight metadata information; we avoid transferring the actual video data between nodes.

4.3.2 Object detection at individual cameras

Upon receiving a query from the central controller, the camera nodes process their locally stored media source (videos/ images) to locate images of humans to respond to the query. If the media source is a video sequence, in order to reduce the processing overhead, only the key frames or video frames at pre-specified intervals are processed for human detection. This interval typically should be chosen based on the dynamics of the setting.

We do not design new computer vision algorithms for object detection in ACTION. Rather, we leverage a state-of-the-art human detection technique proposed by Dollar et al. [68]. With this technique, a detection window of size 128x64 pixels is slid across the input image to check for the presence of humans (at different positions within the image). A detection model based on such a fixed size detection window, is only effective in detecting humans whose sizes are similar to that window size. However, humans that appear in a frame could vary in size depending on the distance between the camera and each such human. To overcome this problem, each input frame is scaled to different sizes (so that humans in the view are also scaled up and down by different factors) to try to match the size of the detection window. The sliding detection window is then applied to all sizes of the input image. A detection hit on a “resized” version of the image will be mapped onto the corresponding position in the original sized frame.

When the sliding window is moved to a new position, the pixel values inside the window are computed and used as features for the human detection process. Specifically, the image is transformed into 10 different representations (called image channels). These include 3 color channels in the LUV color space, 1 gradient magnitude channel and 6 gradient orientation channels (for more details refer to [64]). Each pixel in each channel is used as a feature associated with the detection window; therefore there is a total of 128x64x10 features (corresponding to the window size chosen). To reduce this high number of features, the Adaboost algorithm [82] is used (discussed below).

First, a training set which includes "positive" image windows (those with humans) and negative image windows (non-human images) is built offline. All the samples in this training set are resized to 128x64 pixels and have the same number of features. Subsequently, the Adaboost algorithm is applied to learn the classification rule as briefly captured in Alg. 2. The number of learning steps T is pre-defined. In each step t, a *weak classifier* h is learnt. Thus, at the end of T steps, we have T weak classifiers. The final *strong classifier* H is constructed as a linear combination of these T weak classifiers.

In its simplest form, a weak classifier h_j is a classification rule (for a detection window x) defined based on a single feature f_j. It is defined based on whether or not f_j (a positive value) is higher or lower than an associated (prespecified) threshold τ_j and a polarity ($\theta_j = \pm 1$, chosen to reflect the correct inequality), as:

$$h_j(x) = \begin{cases} 1 & f_j(x).\theta_j \leq \tau_j.\theta_j \\ -1 & otherwise \end{cases} \tag{4.1}$$

where, $h_i(x) = 1$ indicates that x is a positive window and a negative window otherwise. The polarity θ_i is used to determine the correct inequality i.e., whether the value of $f_i \leq \tau_i$ indicates a positive window, or otherwise.

Algorithm 2 Adaboost learning process

```
Inputs:
  • Training samples $\{x_1, y_1\}, ..., \{x_n, y_n\}$ where $y_i = \{-1, 1\}$ indicates a negative and a
    positive sample, respectively
  • T: number of learning steps
  • $N$, $N'$: number of positive and negative training samples, respectively
Initialize: For each item $x_i$, set weight $w_{0,i}=\frac{1}{2N}$, $\frac{1}{2N'}$ for positive and negative samples
respectively
 1: for $t = 1$ to $T$ do
 2:     Normalize weights: For each item $x_i$, set $w_{t,i} = \frac{w_{t,i}}{\sum_i w_{t,i}}$
 3:     for each feature $f_j$ do
 4:         Train a weak classifier $h_j$ as in Eq (4.1)
 5:         Compute the error rate $\xi_j = \sum_i w_{t,i} I(h_j(x_i) \neq y_i)$, where I is the identity function
 6:     end for
 7:     Choose the weak classifier which has the lowest error rate $\xi_t$
 8:     Set $\beta_t = \frac{\xi_t}{1-\xi_t}$, and $\alpha_t = -log\beta_t$
 9:     for each learning sample $x_i$ do
10:         Update its weight: $w_{t+1,i} = w_{t,i}\beta_t^{1-e_i}$; where $e_i = 0$ if sample $x_i$ is classified
            correctly, otherwise $e_i = 1$
11:     end for
12: end for
Output: The final strong classifier $H(x) = \sum_{t=1}^{T} \alpha_t h_t(x)$
```

Let N' and N be the number of negative (no human) and positive (with human) learning samples, respectively. Initially, all the positive samples x, are assigned the same weight, $\frac{1}{2N}$. The weights of the negative samples, x' are set to $\frac{1}{2N'}$. At each step t, the feature which produces the lowest error rate ξ_t (discussed below) is chosen as the weak classifier.

For each feature, the error rate is the sum of the weights of all the samples x for which $h_i(x) \neq y(x)$, where $y(x) = \pm 1$ is the label of training sample x. At each learning step, the key idea of Adaboost is to decrease the weights of those samples that are correctly classified. Specifically, the weight of sample x_i is is kept unchanged if it is incorrectly classified. If x_i is correctly classified, its weight is reduced (adjusted *down*) by a factor of β_t that is a function of the error (line 8 in Alg. 2). Thus, the weights of incorrectly classified samples are higher than those samples that are correctly classified.

The strong classifier H will be distributed to the camera nodes to detect humans in their captured videos/images.

4.3.3 Putting things together: Jointly considering overlapping views

When an individual camera believes that it has detected a human, it extracts metadata pertaining to the detection window in which the human is detected. It then sends this metadata to a fusion node. The fusion node is simply one of the camera nodes which share the overlapping views as described earlier. The metadata associated with each detection window includes (i) A timestamp which indicates when the human appears in the field of view, (ii) The location of the detected human in a 2D image coordinate system (as discussed later this location is converted to a location in the real 3D world using a novel approach as discussed later), (iii) The probability that the human is actually captured in that window P_i; we refer to this as the detection probability. P_i is computed from the value of the strong classifier H (described in section 4.3.2), (iv) a compact color feature of the detection window, and finally, (v) the size of the detection window.

At the fusion node, the goal is to combine the information from the detection windows obtained from the different cameras that are associated with the same human. In order to do so, the fusion node converts the 2D locations from the detection windows in the image coordinate system to a location in the 3D real world coordinate system. These locations with respect to the plurality of cameras are then compared towards re-identifying the human in different camera views. If the locations of two windows are within a threshold T_p, we consider that the two windows are most probably associated with the same human.

In many cases, there could be several people standing close to each other; in order to verify and reduce the false matches in such situations, we also compare the color features from the detected windows. If the distance associated with their color features are also within a threshold (T_c), the fusion node concludes that the two windows depict the same human.

In the following, we provide details on how ACTION computes the 3D location in the real world and uses the color features in association.

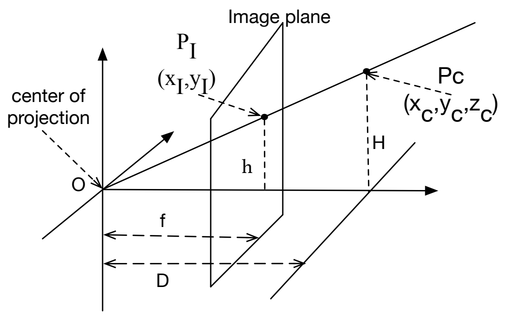


Figure 4.2: Camera intrinsic information

Estimation of the 3D location of the detected human: We use the camera calibration information of an individual camera to *estimate* a 3D location $P_w(x_w, y_w, z_w)$ of a human in the real world coordinate system (with a pre-defined origin agreed upon by all cameras) from its 2D location $P_I(x_I, y_I)$ in the image coordinate system. The camera calibration information consists of the intrinsic information **K**, the rotation information **R** and the translation information **T** of the camera [83]. **R** provides information with regards to the angle the camera is tilted with respect to the three axes in the real world coordinate system. **T** provides information about the location of the camera itself in the real world coordinate system. **R** and **T** form the extrinsic information of the camera. Note here that **R** and **T** are matrices.

In the default setting, the three axes in the real world and in the camera coordinate system are the same. When the camera makes a rotation of α about the z-axis, then a rotation of β about the y-axis, and finally a rotation of γ about the x-axis, it can be shown that $\mathbf{R}_{(\alpha,\beta,\gamma)}$ is given by (more detail in [84]):

$$\begin{bmatrix} cos\alpha cos\beta & cos\alpha sin\beta sin\gamma - sin\alpha cos\gamma & cos\alpha sin\beta cos\gamma + sin\alpha sin\gamma \\ sin\alpha cos\beta & sin\alpha sin\beta sin\gamma + cos\alpha cos\gamma & sin\alpha sin\beta cos\gamma - cos\alpha sin\gamma \\ -sin\beta & cos\beta sin\gamma & cos\beta cos\gamma \end{bmatrix}$$

If the camera is also translated by distances of (T_X,T_Y,T_Z) from the origin along the three axes (with respect to the rotated camera coordinate system), it is easy to see that **T** is $[-T_X - T_Y - T_Z]^T$ (the point $\{T_X$,T_Y,$T_Z\}$ now becomes the new origin of the camera coordinate system).

The **R** and the **T** matrices are used to convert the 3D location $P_w(x_w, y_w, z_w)$ of a human in the real world coordinate system to a 3D location $P_C(x_C, y_C, z_C)$ (of the same human) in the camera coordinate system where, the camera itself is the origin and the axes are tilted or rotated in conjunction with the camera. This is captured in Eq 4.2:

$$P_C = \mathbf{R} * P_w + \mathbf{T} \tag{4.2}$$

where, "$*$" represents the product operation (of the matrices).

The intrinsic matrix **K**, which contains information about the camera "projection point" and the focal length, helps convert the 3D location P_C into a 2D location P_I in the image plane, as shown in Fig. 4.2. In essence, it is simply a projection of the 3D object onto the 2D plane in the camera's view.

The task of converting the position of the detected human in the image plane to a position in the real world coordinate system consists of the following steps: (i) convert the 2D location P_I in the image plane to a 3D location P_C in the camera coordinate system, and (ii) convert the 3D location P_C to the 3D location P_w in the real world coordinate system. It is a challenge to accurately compute P_C because of the lack of the third dimensional information when converting a point from a 2D world to one in a 3D world. As evident from Fig. 4.2, multiple 3D objects can project to the same 2D object in the image coordinate system (e.g., any point along the line connecting P_I and P_C is projected onto the same point P_I in the image plane). Note here that the distance D between the detected human and the camera is not known.

In ACTION, we estimate the value of P_C by considering a plurality of human or object heights and determining whether or not two camera views converge in their 3D location estimates for *any* of these heights.. In more detail, let h be the height of the detected human in the image coordinate system (the height of the detection window). Let H be the height of the human in real world. D is computed as:

$$\frac{D}{f} = \frac{H}{h} \tag{4.3}$$

where f is the distance between the center of projection (see Fig.4.2) of the camera to the image plane (this is the camera's focal length and is provided in the camera's intrinsic information $\mathbf{K}$). Once D is known, P_C can be computed from P_I easily.

We assume a set of possible values of the height of the detected human, $H = \{H_1, H_2, ...H_n\}$ (e.g., from 3ft to 6ft, etc.)[3], and come up with the corresponding values of D_j using Eq 4.3 (for each $H_j, j \in \{1, n\}$). The values of P_I (the 2D location) and each D_j (distance to the camera) are then used to compute the possible values of $P_C = \{P_{C1}, P_{C2}, ..., P_{Cn}\}$, as depicted in Fig. 4.2. Finally, the corresponding values of P_w are computed for each P_{Cj}, using Eq 4.2.

Each camera node sends its intrinsic information (including the focal length), location and orientation computed based on the reference world coordinate system (which are used to calculate **R** and **T** [83]) to the fusion node[4]. The camera nodes update the fusion node if the positions or orientations of the cameras change. With respect to each detection window, its size and the coordinates P_I of the center point of the window are are also included as metadata. This allows the fusion node to compute P_w, with respect to the window for each considered $H_j, j \in \{1, n\}$. Subsequently, the fusion node groups the "nearby" windows (if the Euclidean distance between the P_w values, for any j, is less than a pre-defined threshold T_p) from multiple cameras, together into candidate sets. Note that this requires a pairwise comparison of the P_ws from each camera for all values of j.

Using color and texture features: Unfortunately, humans in the field can stand close to the others and this can lead to false matches, i.e., wrong inferences can be made with respect to re-identification, if only the Euclidean distances were considered (as computed in the above discussion). Therefore, ACTION also compares the color and

[3]If ACTION is used to detect other types of objects, appropriate heights of the objects (e.g., 4ft to 6ft for sedan cars) should be used.

[4]Camera calibration information for distributed fixed camera networks can be effectively obtained using prior approaches on computer vision (e.g.,[85, 86]).

texture features in the detected windows of the candidate sets to reduce the number of false matches.

We use the “Mean Color” feature proposed by Hirzer et al. [87], which is extracted from each detection window in the above step. An input image is divided into small 8x16 patches and the patches are arranged such that there is a 50% overlap between the neighboring patches in both dimensions (x and y). Each patch is converted into the HSV and LAB color spaces (see [87]). The sum of all values of the pixels in the HSV representation, and the sum of all values of the pixels in the LAB representation of each patch are computed; these two values are concatenated and used as the *color feature* of the patch. In addition, the 256-byte LBP histogram [88] of the patch is computed and used as its *texture feature.*

The color feature and texture features of a patch are finally concatenated to form its *local feature.* The local features from all the patches are then again concatenated into a single *global feature* of the whole detection window.

The mean color feature of a 64x128 image is a 55,000-dimensional vector and therefore, here, Euclidean distance is not an effective measure for comparing the feature vectors between detection windows. First, we use principal component analysis (PCA) to reduce the dimensionality of the color features. Subsequently, we use the Mahalanobis distance (proposed in [89]) to compare the features. The Mahalanobis distance between two vectors x_i and x_j is defined as:

$$d(x_i, x_j) = \sqrt{(x_i - x_j)^T \mathbf{A}(x_i - x_j)} \tag{4.4}$$

where $(x_i - x_j)^T$ is a transposed matrix, and $\mathbf{A}$ is called the Mahalanobis matrix and is learnt from a training set. Our training set consists of two subsets. The first subset S, contains

Algorithm 3 Greedy algorithm for choosing detection windows for a detected human

```
Inputs:
  $\tau$: Transfer time for this human
1: Remove detection windows $W_i$ whose $\frac{size(W_i)}{B_i} > \tau$
2: Sort remaining detection windows (in descending order) by "profit densities" $PD_i = \frac{|\log F_i| B_i}{size(W_i)}$
3: Compute k=min{j∈1,...,n}: $\sum_{i=1}^{j} \frac{size(W_i)}{B_i} > \tau$
4: Compute $V_1^{k-1} = \sum_{i=1}^{k-1} |\log F_i|$ and $V_k = |\log F_k|$
5: if $V_1^{k-1} > V_k$ then
6:     Choose $\{W_1, ..., W_{k-1}\}$
7: else
8:     Choose $W_k$
9: end if
```

images of the same humans (with the same timestamp) collected from different cameras; the second subset D contains images of different humans (with the same timestamp) collected from different cameras. Matrix $\mathbf{A}$ is trained to minimize the distances between the elements in S while maximizing the distances between the elements in D. The problem is formulated as a constrained optimization problem, and an iterative framework based on a binary search is used to find an optimal matrix $\mathbf{A}$ (see [89]). If the Mahalanobis distance between the color features of two detection windows (which have been already matched up with respect to position) is less than or equal T_c, we assume that the humans in the two windows are the same.

4.3.4 Transferring the most relevant information given bandwidth constraints

We assume that the bandwidth from each camera node to the central controller is known (Tools such as iPerf can be used for short durations for determining this [90]). We assume that the bandwidth information is also conveyed to the fusion node as part of the metadata. For each detected human, we seek to select and transfer the most relevant camera views sufficient to achieve a desired detection probability P (e.g., $P = 0.9$), within a pre-specified delay. However, achieving the detection probability and this delay simultaneously may be impossible depending on the available bandwidth. Thus, we modify our objective to maximizing the detection probability (of humans), subject to an available bandwidth constraint, which is formalized in Eq 4.5.

$$\begin{aligned} &maximize_{P_i \in \{0,1\}} \quad P = 1 - \prod_i (1 - P_i) = 1 - \prod_i F_i \\ &\text{subject to} \quad \sum_i \frac{size(W_i)}{B_i} \leq \tau. \end{aligned} \tag{4.5}$$

In Eq (4.5), P_i, $F_i = 1 - P_i$, and $size(W_i)$ are the detection probability, false detection probability and size of a detection window W_i (in bytes) from each detected human, respectively; the index i varies over all views of that human. τ is the delay requirement specified in seconds and B_i bytes/second is the bandwidth from camera i to the central controller (note that the channel conditions and contention would dictate this bandwidth). The value of P is computed based on detection probabilities associated with the windows considered for transfer. Specifically, if multiple detection windows W_i agree on a human presence at a given place, at the query specified time, the probability of a false detection at that location (when all cameras yield incorrect results) is $F = \prod_i (1 - P_i) = \prod_i F_i$; thus, $P = (1 - F)$.

Thus, those windows which maximize P, are chosen for transfer, while adhering to the bandwidth allocated.

Algorithm 4 Algorithm for sending best relevant detection information to the central controller

```
Inputs:
  $N$: Number of humans present in the field
  $B_j$: Available bandwidth of each camera $j$
  $P$: Desired detection probability for each human
  $\tau$: Delay requirement //for all detected humans

Initialize: Remaining humans, $n = N$

    for each human $H_i$, $i = 1$ to $N$ do
2:      Available transmission time $\tau_i = \frac{\tau}{n}$
        False detection rate $F_{Hi} = 1$
4:      Select detection windows using the Greedy algorithm (Al. 3) with corresponding $\tau_i$
        for each selected windows $W_{ij}$ do
6:          Send $W_{ij}$ //view $j$ for human $i$
            $\tau_i = \tau_i - \frac{size(W_{ij})}{B_j}$
8:          $F_{Hi} = F_{Hi}(1 - P_{ij})$ //update false detection rate
            if $F_{Hi} \leq (1 - P)$ then
10:             break; move to next human;
            end if
12:     end for
        $\tau = \tau - (\frac{\tau}{n} - \tau_i)$ //share leftover time for other humans
14:     $n = n - 1$ //number of humans need to send data
    end for
```

The objective in Eq (4.5) is equivalent to the minimization of $\prod_i F_i$, $F_i \in (0,1]$ which in turn, is equivalent to the minimization of $\sum_i \log F_i$, for all $F_i \in (0,1]$, with the same bandwidth constraint as before. Since $\log F_i \leq 0$ for all possible values of F_i (F_i

values are less than 1 since they represent probabilities), the minimization of $\sum_i \log F_i$ is equivalent to the maximization of $\sum_i |\log F_i|$ ($\log F_i$ is negative and monotonic). Thus, the optimization problem in Eq (4.5) is equivalent to the following problem in Eq 4.6.

$$\begin{aligned} &maximize \;\; V = \sum_i |\log F_i| \\ &\text{subject to} \qquad \sum_i \frac{size(W_i)}{B_i} \le \tau \end{aligned} \tag{4.6}$$

The above problem can be mapped onto a Knapsack problem [81]. The maximum tolerable delay (specified) for transferring the information pertaining to each user, to the central controller, corresponds to the knapsack size. The goal of the fusion node then, is to choose the views that maximize the sum of the utilities ($|\log F_i|$) of the objects (camera views) that are placed into the knapsack. Unfortunately, the problem as defined above is known to be NP hard [81]. Therefore, ACTION uses a well known greedy algorithm (detailed in [91]) to fill the knapsack. As the name suggests, the algorithm greedily chooses the most relevant windows from the multiple cameras for being sent back to the central controller in response to its query, while adhering to the delay constraint given the bandwidth.

With the greedy algorithm, the detection windows are sorted (in descending order) at the fusion node, in terms of their "profit densities" which are defined as $PD_i = \frac{|\log F_i| B_i}{size(W_i)}$. Here, note that in the general case, the the detection windows from the different camera nodes vary in terms of the number of bytes. The knapsack is filled from this sorted list; the view with the highest profit density is inserted first and so on. Let W_k be the first window (as the list is traversed) that causes a violation to the bandwidth constraint; here, k=min{j∈1,...,n} such that $\sum_{i=1}^{j} \frac{size(W_i)}{B_i} > \tau$. The greedy algorithm then chooses either the set of windows $\{W_i, ..., W_{k-1}\}$ or the single window W_k, depending on whether the value $V_1^{k-1} = \sum_{i=1}^{k-1} |\log F_i|$ is higher or lower than $V_k = |\log F_k|$, respectively. It is shown in [91] that the profit V_G obtained using the greedy algorithm ($V_G = max\{V_1^{k-1} = \sum_{i=1}^{k-1} |\log F_i|, V_k = |\log F_k|\}$) is guaranteed to be $\geq \frac{1}{2}V_{OPT}$, where V_{OPT} is value of the Knapsack when filled optimally [91]. The pseudocode for the greedy algorithm is shown in Algorithm 3.

Operations in Practice: The pseudocode of our bandwidth-aware data selection process at the fusion node is presented in Algorithm 4. Specifically, we divide the available time equally among all of the humans that are detected[5]. For example, if a 1 second period is available to transfer the information and there are 3 humans detected, we allocate 0.33 seconds to transfer the detection windows associated with each human. For each human H_i, we apply the greedy algorithm (Algorithm 3) with available transfer time τ_i of that human to select best relevant windows to fill the knapsack. The relevant detection windows are transferred in order (we assume that the fusion node tells each camera node when and what frames to transmit) until one of two conditions is met (i) the duration for sending data with respect to that human expires, or (ii) the required detection accuracy with respect to this human has been met.

We modify the Knapsack problem to ensure that we do not waste bandwidth if the detection probability is higher than a sufficient (desired) value. In other words, if a desired P is achieved (i.e., the detection accuracy is met) but there is leftover bandwidth (time) after the associated, “sufficient” set of detection windows relating to the human are

[5]In ACTION, a record with respect to a human is put into a queue when all the information related to that human is available at the fusion node. Information relating to the human whose associated record has the earliest time stamp is considered first for transfer. Other policies are possible (prioritization of records as per other criteria) but we do not consider this.

transferred, the residual time (bandwidth) is equally shared for the transfer of windows with respect to the other humans, as shown on line 13 of the algorithm.

4.4 Implementation

In this section, we describe the implementation of ACTION. Our implementation consists of a fusion application, a detection application and a server application. The fusion application runs on a pre-specified Android phone and receives metadata relating to the detection windows, from multiple instances of the detection application that are in turn running on Android smartphones. The detection applications interact with the fusion application, receive fusion decisions from the fusion node and finally upload chosen detection windows to a server application which represents the central controller.

The detection application: We implement our detection application on Asus ZenFone II smartphones with Android 5.0 OS. These are used as the camera nodes in our prototype. The application was written in both Java and native C++ (JNI). We partly use the source code provided by the authors of [68], to convert the input image into different channels and to extract its features for human detection (as described in 4.3.2). We use the OpenCV library for all other image processing tasks.

Note here that we use smartphones, as they are popularly used for video capture today [92], to *emulate* camera nodes with local processing capabilities. In a real scenario, one could envision static, programmable cameras (e.g., Pixy [93]) that are mounted on ceilings or walls are used. Such cameras could capture higher resolution videos; however, we believe that today's smartphones already offer very high resolutions and devices that are similar architecturally can be used for this purpose. However, we acknowledge that with other platforms, the results could differ from those that we present in Section 4.5. For example, other platforms may have more powerful batteries.

An input image is scaled into 23 different sizes for human detection. In order to reduce the computational overheads, we only compute image features at 3 specific base scales; image features at other sizes are estimated based on the features determined at the base scales, as described in [94].

Camera node operations: Upon receiving a query from the server application, our detection application reads the input video sequence and checks for humans in every 10^{th} frame (once every 0.5 seconds). In practice, this parameter should be chosen based on the area of the monitored field, number of deployed cameras and the moving speed of tracked humans. The value 0.5s chosen in our implementation is based on the availability of the ground truth information in our data set [7], which is described later in section 4.5. When a human is found, the application extracts the metadata associated with the detection window as described in section 4.3.3 and uploads this information to the fusion node.

The output of the human detection algorithm is a detection score of the window. The higher the score, the higher the probability that the window contains an image of a human. We convert the detection score to a corresponding detection probability using the training data as follows. The detection scores are quantized into 20 bins. For each bin value x, the probability is given by the ratio of the number of *correct* detection windows (based on the ground truth information) to all the detection windows that have a detection score of at least x.

To extract the color features from the detection windows, we first resize all the windows into 64x128 pixels. This ensures that the color features of all detection windows will have the same length. The extracted mean color feature of each window is a 55,000 dimensional vector. To reduce the communication overhead, we apply PCA to reduce this size to 40 dimensions, as in [87], before uploading the information to the fusion node. Further, we compress the detection window (using the jpeg format) and use the compressed version for transfer.

The fusion application: The fusion application is written in Java and can be executed on any Android smartphone. Currently, we statically assign one of the camera nodes to act as the fusion node; however, in practice, nearby camera nodes can be grouped into clusters, and for each cluster, the fusion node can be chosen based on which of the nodes has highest computational power or residual energy. In Section 4.5, we conduct experiments to show that the fusion application is lightweight and does not consume much energy overhead; thus, ACTION is not sensitive to the choice of the fusion node.

Fusion node operations: The fusion node will receive metadata with regards to detection windows from multiple surrounding cameras. For each detection window, the fusion considers a set of of 17 possible heights of common humans (from 120 cm to 200 cm) and converts the 2D location in the image to a set of 17 possible 3D locations of the detected human. Locations and color features of the detection windows are used to detect and associate windows that capture the same human. Specifically, we use the threshold $T_p = 1.2m$ for location, and $T_c = 18,000$ for color (Mahalanobis) distance to group detection windows. We show the effectiveness of using these values in Section 4.5.

When the metadata with regards to a new detection window arrives the fusion node checks to see if it can be correlated with the metadata from that of another camera node. If it cannot find a correlation with any previously received metadata, the fusion node considers the window to be the first window of a newly detected human and creates a new group for it. When metadata associated with other windows corresponding to the same human arrive, they will be grouped together. Locations of individual windows in a group are averaged to compute the location of the centroid position, which is used to represent the whole group. For each group, there are multiple positions P_w of the centroid corresponding to the considered heights, H_j; in order to be considered to belong to a group, the newly received metadata must reflect a position that is within the threshold T_p with respect to one of the n, P_w values (i.e., corresponding to each of the heights H_j). Further, the distance between the color feature of the window and that of at least one of the windows in the group must be within T_c.

In cases when one detection window can be grouped into different groups, the fusion nodes includes the new window in the group which has the minimum Mahalanobis color distance to the window, since "color distance" is more distinctive than "position distance" if people are close to each other (these are the primary reasons for errors in reidentification based on position alone).

After partitioning the detection windows into groups, the fusion node chooses the most relevant windows from each group based on the greedy algorithm described in section 4.3.4, and notifies the camera nodes if they have the chosen windows. Subsequently, the camera nodes transfer image data directly to the central controller.

The server application: The server application runs on a Linux server; its only duty is to send queries with specific time-stamps, and receive content (parts of frames with detected humans) from the camera nodes after being instructed to do so by the fusion node.

4.5 Evaluations

In this section, we evaluate both the accuracy as well as the efficiency (in terms of resource consumption) of ACTION. We begin with a description of the datasets we use and how we determine the ground truth. Later, we provide our results.

4.5.1 Training and test datasets

4.5.1.1 Data set

We use the "Multi-camera multi-object tracking" data set, made available by the Computer Graphics and Vision group at Graz University of Technology [7]. The dataset contains 6 indoor video scenarios. Each scenario has 4 different synchronized video sequences, captured by 4 different cameras. In each video, there are about 4 to 6 people walking around in the same room. At different times in the videos, the people can be separated spatially or could be closely clustered together. We use the first video sequence as the training set to calibrate ACTION, and the other sequences as the test data.

The human detection model is built using the Inria pedestrian dataset [63]. This contains thousands of images of humans in different poses. Thus, the detection model is not calibrated by the small set of humans that appear in our first data set, and can be used to detect humans in general cases.

4.5.1.2 Obtaining ground-truth information

The data set that we consider is annotated with information that provides ground truth at a coarse-grained level. Specifically, once every 10 frames, the 3-D, real world coordinates of the "foot" of each person in the frame, is provided. As discussed earlier, we can convert these co-ordinates to the 2-D coordinates in the image coordinate system of each camera (using Eq (4.2)). The data set also provides the identifier of the human the foot belongs to (human ID).

Each detection window is a rectangular area within the frame as discussed earlier. We check if the aforementioned 2D coordinates (of the "foot") fall within each detection window. Each detection window which contains such coordinates, is associated with a human ID. In cases where there is more than one human in the detection window, we may have false positives. To eliminate these, we mark those windows and manually check the ground truth information (to determine which human is in the detection window).

4.5.2 Improving detection accuracy with overlapped camera views

4.5.2.1 Setting thresholds for accurate detection

Below, we first describe how we determine the thresholds T_p (the Euclidean distance between the plausible 3D locations of a detected human, from the perspective of multiple cameras) and T_c (the Mahalanobis distance between the color features that correspond to the same human, from different cameras).

From the training video sequence, we create two different sets: (a) set S contains pairs of detection windows that show the same human, captured at the same time by different cameras, and (b) set D contains pairs of windows that show different humans captured by different cameras. For each detection window, we compute the possible positions of the human (based on a set of considered heights) as described in section 4.3.3, and compute the minimum Euclidean distance (with respect to all considered heights) between each pair of windows in set S and in set D. We show the CDF of the distances in Fig. 4.3. Based on the results, we set threshold $T_p = 1.2m$; from the figure we see that this results in the detection of more than 80% of the windows of the same human with about $\approx 30\%$ false matches. An increase in this threshold would result in a higher false positive rate; a decrease would reduce the number of correct detections (true positives). This seems like a reasonable compromise. Next, we compute the Mahalanobis distance between the color features of each pair in set S, and in set D. We show the CDF of this distance in Fig. 4.4. Based on the results, we set threshold $T_c = 18,000$. With this threshold, we are able to detect more than 90% of windows with the same human, with an expense of ≈25% false matches (false positives).

We point out that it is important to achieve a high true positive rate while keeping the false positive rate low. The former would reduce bandwidth usage as fewer windows of the same human need to be transferred; however, incorrect matching of different humans might cause the missing of the transfer of data associated with a particular person. *When both the position and color are combined, we are able to achieve a true positive rate of $\approx 91\%$ and a false positive rate (when windows containing different humans are incorrectly grouped) of ≈ 9 %.* ACTION classifies a group of detection windows as "correctly matched," based on a majority rule. If at least half of the detection windows correspond to the same human

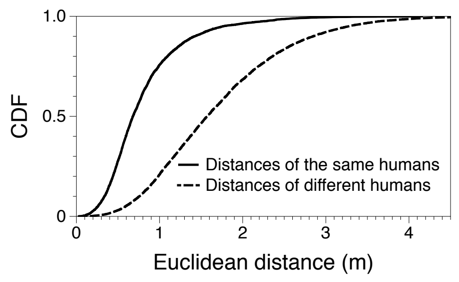


Figure 4.3: Distances between windows of the same and different humans

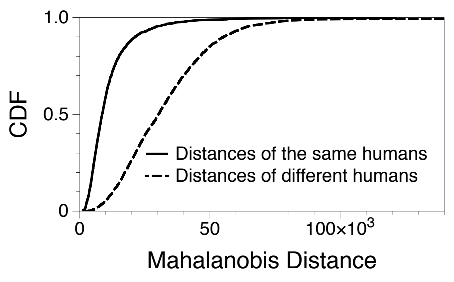


Figure 4.4: Distances between color features of the same and different humans

(based on the ground truth information), the matching is considered correct; otherwise it is considered incorrect.

4.5.2.2 Does the use of more cameras yield better performance?

Next, we perform experiments to determine the benefits of using a plurality of cameras in ACTION. One of the metrics we use is what we call the *recall* value. This value is the number of times that a human is correctly identified from among all the times she appears in the captured videos (transferred to the central controller). It is expressed as a percentage. If multiple cameras are used, at least one of the cameras needs to correctly detect the human. We measure how the recall value changes, when different numbers of cameras are used for human detection.

When only one camera is used, all its detection windows that reflect the presence of a human are transferred. If multiple cameras are used, we posit a requirement of 0.9 on the detection probability P. In other words, the fusion node requires the transfer of detection windows until this requirement is met or the time constraint imposed (by the

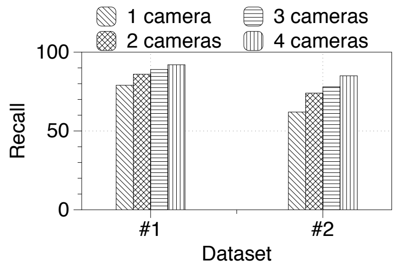


Figure 4.5: Average recall values with different numbers of cameras

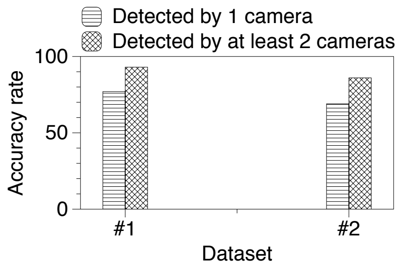


Figure 4.6: Accuracy rate when combining data from more than one camera

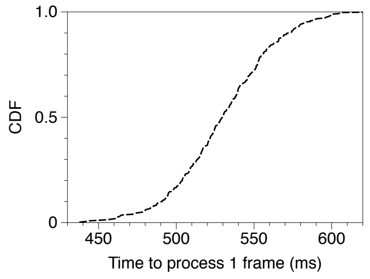


Figure 4.7: Image processing time

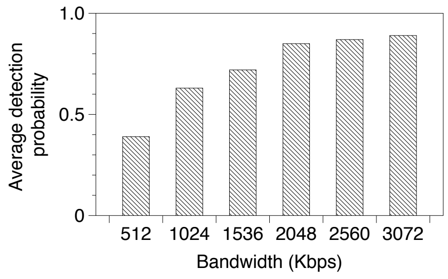


Figure 4.8: Average detection probability under different bandwidth constraints

bandwidth requirement) does not allow any additional transfers. In these experiments, that time constraint is set to 0.5 seconds.

In Fig. 4.5, we show the results from both of our data sets. As one might expect, the use of a higher number of cameras results in a higher recall value. However, the improvements depend on the extent to which humans are occluded from camera views. In the first data set, humans are separated with little occlusion. Thus, the increase is only modest as seen in the figure. However, in the second dataset, the people are close to each other and typically there is higher occlusion. Here, the use of a plurality of cameras with ACTION results in a significant performance improvement. Specifically, with only one camera, the recall value is ≈ 64 %. It increases to more than 85 % with four cameras.

We next evaluate ACTION in terms of the *precision* of detection (aka the accuracy rate). Specifically, from among all the detection windows reported to detect humans, the accuracy rate represents the fraction that are correct reports. In the case of multiple cameras, we require that at least two of them correctly identify the (same) human (majority rule is applied as discussed earlier). We again observe that the use of multiple cameras significantly improves the accuracy rate. With dataset 1, the improvement is about 15 % while with dataset 2, the improvement is about 20 %.

4.5.2.3 Impact of bandwidth constraints

In our next experiment, we illustrate how the greedy algorithm for transferring relevant detection windows in ACTION, performs as we vary the available bandwidth. The individual nodes process the video sequences and upload metadata to the fusion node once every 10 frames. The fusion node determines the set of detection windows to be uploaded (as described in Section 4.3.4). The corresponding cameras are required to transfer the selected information in 0.5 seconds. For ease of disposition, we assume that the system is homogeneous i.e., the bandwidth between the controller and each camera node is identical (we set this in our implementation). However, the results can easily carry over to heterogeneous settings. We again posit a requirement of 0.9 on the detection probability.

The results, presented in Fig. 4.8, show that under strict bandwidth constraints only a small fraction of the detection windows associated with each human is transferred. Thus the detection probability requirement is not met; the achieved detection probabilities are really low. However, as more bandwidth is available, a higher number of detection windows associated with each human can be transferred. The detection probability increases gradually. However, there is a "saturation point," (available bandwidth = 2048 kbps) after which there is sufficient bandwidth to transmit enough windows for achieving the required detection probability P; beyond this point there is a negligible improvement in P (if at all), even with a bandwidth increase.

4.5.3 Resource usage

In this section, we seek to evaluate ACTION in terms of quantifying the bandwidth savings that it provides and its processing and energy overheads.

Bandwidth usage: Fig. 4.9 shows the bandwidth usage in three different scenarios: (i) when images of all detection windows from a single camera, are transferred directly to the command node, (ii) when all 4 cameras transfer images of all detection windows directly to the command node, and (iii) ACTION is used and a target detection probability of $\approx 90\%$ is required with regards to each detected human. In the former two scenarios, ACTION is not used. We show the results for the case where the input video is processed every (i) 0.5s and (ii) 2s, as in the previous experiment. Without ACTION, the volume of data transferred by the 4 cameras is around 3 times as compared to when ACTION is used. With ACTION, typically, only data from the best 1 or 2 cameras need to be transferred; thus a significant amount of unnecessary data transfers is avoided. Note here that if more than 4 (overlapping) cameras are available, one could conceivably achieve even a higher reduction in bandwidth usage. Further, note that with ACTION (since only the most relevant information is transferred), the total amount of transferred data (from 4 cameras) is only ≈ 1.4 times the data transferred by 1 camera.

Processing times on individual nodes: Fig 4.7 shows the distribution of the time taken to process 1 frame towards detecting a human on our Android smartphone. The resolution of our test data is 1024x768 pixels. With this setup, the maximum processing

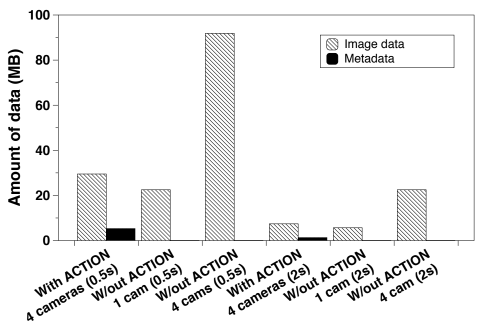


Figure 4.9: Bandwidth usage

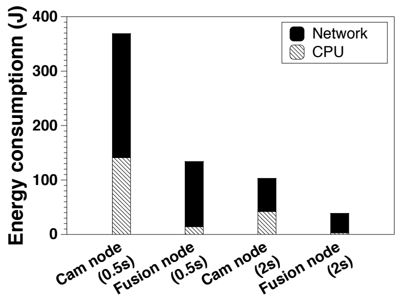


Figure 4.10: Energy overheads

time is $600ms$; in other words the local algorithm for human detection on our smartphone-based individual camera nodes in ACTION, achieves a processing rate of $\approx 1.7 frames/sec$. This in turn suggests that a platform such as a smartphone is sufficiently powerful to process the video in near real-time. Since video frames can be processed well in advance of a query, we believe that the system is sufficiently responsive and deployable in real contexts.

Energy overhead: Fig. 4.10 depicts the energy consumed due to human detection on individual phones and due to the fusion operation. We consider a 3.2 minute input video sequence. We consider two possibilities: (a) every 10th frame *or* (b) every 40th frame, is considered for human detection. The detection windows are sent to the fusion node once every 0.5 seconds in the former case and every 2 seconds in the latter. We use the popular PowerTutor[6] tool [11] to estimate power usage due to the following components: (i) the CPU usage to extract image features locally on the camera nodes, (ii) the CPU consumed to combine the metadata and choose detection windows for transfer, at the fusion node, and (iii) the WiFi network interface towards transferring and receiving the data. Note that we only report the energy consumed by the operations of our ACTION framework; the energy

[6]We modify the source code of PowerTutor to allow it to capture the network usage information on the ZenFone II we use.

consumed by unrelated processes and components on the phones (e.g., screen usage, phone standby energy, system background processes, etc.) is not taken into account.

The result shows that, if the video sequence is processed every 0.5s, the detection application consumes 369J. The fusion operations consume 133J. On our smartphones, which have a battery capacity of 3000mAh (and 4.5v), these two values correspond to 0.76% and 0.27% of the battery respectively. If the operations are continuous then, the energy of the camera node lasts for about 7 hours.

If the processing interval increases, the energy consumption decreases as expected. The human detection and fusion operations consume 102J and 39J, respectively, if the video is processed every 2 seconds. Those values correspond to 0.2% and 0.08% of the battery capacity, respectively. At this rate, the energy of the camera node lasts for 26 hours. Note that if the camera nodes have batteries that are better than today's smartphone batteries, they can last for much longer.

4.6 Conclusions

In this paper, we consider the problem of retrieving situation awareness information from a multi-camera network in scenarios such as natural disasters where the bandwidth is limited due to compromised infrastructure. In such scenarios, the cameras cannot all transfer their content to a central controller handling search and rescue operations. Thus, we seek to only transfer those camera feeds that can provide highly accurate input to the controller while ensuring the timeliness of the transferred content. Towards this, we design and implement a framework, ACTION. ACTION (i) uses state of the art computer vision algorithms to detect objects of interest (e.g., humans or animals), (ii) uses novel approaches to jointly consider views from multiple cameras and determine the views that yield the best accuracy with respect to an object of interest, and (iii) only transfers the best views to a controller while adhering to bandwidth constraints. We implement ACTION on a smartphone based testbed and show that it achieves a high accuracy of ≈ 90 % in terms of detecting humans who are considered as objects of interest, while reducing the bandwidth consumption threefold.

Chapter 5

Energy Efficient Object Detection in Camera Sensor Networks

5.1 Introduction

Timely and accurate detection of objects of interest (e.g., humans) is critical in many scenarios of interest. For example, in rescue and recovery missions following natural disasters, one might want to detect humans or animals in distress. Homeland security might be interested in automatically and proactively, tracking unattended baggage in airports or bus stations. Today, camera sensors equipped with computation capabilities can be deployed in the field to provide situation awareness information in such scenarios. In fact, battery operated, low power embedded camera devices (e.g, the CMUcam series [95]) that can be used for such purposes are already emerging and on the market. Using common programming languages, these devices can be programmed to do multiple types of on-board processing tasks (e.g., object and face detection).

A network of such camera sensors can significantly improve the accuracy of object detection. With such a network, objects that might be obstructed or hidden from specific angles of view, can still be potentially detected. However, simply sending video feeds from all such camera sensors to a central controller that is responsible for operations (e.g., search and rescue) might not only be wasteful, but could result in unnecessary energy expenditures and hurt the longevity of the network. In addition, each camera could be trained to use highly optimal, domain specific, algorithms to process the captured video. However, the higher the fidelity of a processing algorithm, the higher the cost in terms of processing energy. Thus, when a plurality of cameras detect the same object, it might be unnecessary for all of the cameras to use the optimal (possibly the most energy expensive) algorithm. Some of the cameras could use sub-optimal processing to save energy while ensuring that the detection accuracy does not take a big hit.

In this paper, our goal is to design a framework EECS, that can facilitate co-ordination among the cameras in such a network to realize significant energy savings compared to cases where there is no such co-ordination, and yet, achieve a high detection accuracy. The framework determines (a) which cameras are suitable for capturing an object of interest, (b) what domain specific algorithm to use for processing the captured video and (c) which cameras views are to be transferred to the central controller that is responsible for operations.

Challenges: In order to achieve our overarching goal, we need to tackle a set of key challenges. First, since the scenarios are likely to be unknown a priori, the optimal or the most accurate video processing or detection algorithm for each camera sensor is not known. The problem is harder if we need to rank order the processing algorithms in terms of the accuracy they yield and the energy expenditures they incur for the scenario. This essentially requires the assessment of similarities between the video captured of an unknown scenario and a set of pre-installed training videos corresponding to a set of (pre-determined) scenarios; such an online comparison is very challenging for complicated and high dimensional signals like video feeds. Second, we require EECS to be able to identify a subset of camera sensors whose detection yields are together sufficient to achieve a desired accuracy. This ensures that EECS does not unnecessarily invoke all camera sensors (and thus, helps reduce energy consumption). Third, we need to determine which camera nodes should utilize sub-optimal (energy efficient) detection algorithms, instead of using the most accurate (possibly more expensive) algorithm for processing the videos while still adhering to the accuracy requirements.

EECS in brief: EECS is designed to address the above challenges. To solve the first challenge, it leverages state-of-the-art video comparison algorithms to identify the most effective detection algorithm for each individual camera. In brief, each camera captures a short video feed and compares the feed with pre-loaded training videos to find the closest match; the algorithm that works best for the matching training video is then chosen. When the environment changes, the process is repeated. The process which is referred to as "domain adaptation has been used for efficient video comparisons [96]. Specifically, principle component analysis (PCA) is applied on the captured and training videos to remove the unimportant features, and to reduce the signal dimensionality; the PCA-processed signals are then projected onto a subspace called the Grassmannian manifold for comparison. The manifold is created in a manner that ensures that a small distance between two projected points in the manifold also indicates a high level of similarity between two associated video feeds.

The above approach however, only allows each individual camera to dynamically choose the most accurate algorithm to process the captured video feeds. EECS ranks the camera nodes based on individual accuracies and applies a novel greedy algorithm to choose a subset of cameras that jointly can achieve a predefined desired accuracy (thus addressing the second challenge). This requires EECS to be able to identify and aggregate detection information of the same objects from different views/cameras, and then assesses the detection accuracy based on the aggregated information. Subsequently towards addressing the third challenge, for each chosen camera, EECS determines whether each camera can use a less energy expensive algorithm that satisfies the desired detection accuracy requirement and if yes, chooses the less expensive algorithm for processing.

Novelty: To the best of our knowledge, EECS is the first to support co-ordination across a set of battery operated camera nodes towards reducing energy consumption while ensuring high accuracy of object detection. While the design of domain adaption for individual cameras has been studied in the computer vision community, coordination across cameras to determine the algorithms that different cameras should use to achieve a certain detection accuracy has not been considered before. Finally, energy was not a consideration in determining the choice of this set of algorithms.

Evaluations: We implement EECS on a testbed of Android phones, which have pre-installed video feeds captured from overlapping cameras. We implement three different video processing algorithms on each of the cameras. The algorithms are adaptively chosen by EECS depending on the environment and requirements. Our evaluations show that EECS achieves both higher precision (more accurate detection) and recall (more objects/humans are detected) than using the same algorithm to process all data sets. In addition, EECS's resource-aware algorithm selection approach helps to reduce up to 40% of the total energy consumption while still achieving $\approx$ 86% of the highest accuracy (achieved when the optimal algorithms are used at all individual cameras).

5.2 Related work

Object detection algorithms: While our work is applicable to object detection in general, we focus mainly on the humans as the objects of interest. Different features (color, gradient, texture) and machine learning techniques (SVM, boosting) have been used in object detection [63, 68, 97, 98, 64]. However, each algorithm only works well in specific scenarios and conditions. In this work, we propose a framework for adaptively choosing an appropriate algorithm depending on the environment/condition.

Domain adaptation: Domain adaptation is used for learning classification rules for a target (e.g., indoor) dataset from pre-trained source dataset [96, 99]. In EECS, domain adaptation is used to find the correspondences between features of the two datasets. This correspondence is used to assess video similarity, and identify the most appropriate detection algorithm for an unknown incoming video feed.

Adaptive algorithm selection: Algorithm selection has been studied in several recent works for other problems. In [100], a model to predict the performance of different image segmentation algorithms is developed. In [101], pixels in an image are segmented into different regions, and different detection algorithms are applied for the different regions. In these works, the values of the selected features are used to determine which algorithms are used. In our approach, the similarities between features from different video feeds are used to select the algorithm (not the values of the features directly). The work in [102] is the closest to our work; the authors consider using different algorithms to detect humans in different video feeds. However, they only consider choosing the most efficient algorithm to process captured video feeds for a *single* camera. EECS, on the other hand, focuses on multi-camera settings in which if one camera already chooses the best (yet energy hungry) algorithms, other cameras might consider choosing sub-optimal algorithms instead to reduce the total energy consumption.

Object detection using multiple cameras: Works such as [78, 75, 76] focus on detecting objects of interest using a network of camera sensors. However, they only employ a specific video processing algorithm to detect objects. On the other hand, EECS allows changing the detection algorithm adaptively as the environment changes to improve detection accuracy while ensuring energy efficiency.

Efficient video processing on mobile devices: Li *et al.* [103] propose data manipulation techniques to reduce memory access related energy consumption on mobile devices. Lee *et al.* [104] implement and study the energy consumption of different video encryption schemes on mobile devices. The authors in [105, 106, 107] study and improve the energy efficiency of video streaming applications for mobile devices. Improving the efficiency of object detection or other video processing algorithms is not our focus; we instead design a framework to choose the most energy efficient algorithm from a set of available algorithms while ensuring given accuracy requirements.

5.3 Video comparison using domain adaptation

When a video feed is captured, each camera needs to determine which video processing algorithm is most effective (in terms of accurately detecting objects of interest) in the feed. In order to do so, in EECS, the new feed is compared against a training set of videos that are pre-loaded onto the cameras. To determine matches between the new feed and the videos in the training set, an efficient video comparison technique is essential. The process is referred to as domain adaptation i.e., determining which algorithm is best suited for the domain under consideration (pertaining to the captured scene in the new feed).

Domain adaptation for single cameras has been studied in the computer vision community [96, 99] and can be used to determine the similarity between videos. In many cases, directly comparing video features in their original domains does not yield good results; for example, images taken in different conditions (indoor/outdoor, illumination, variations in size of an object from different views, etc.) can be quite different, but actually should be processed using the same algorithm in order to achieve the highest accuracy, as shown

Symbol	**Meaning**
α, β	Sizes of feature space and PCA subspace
t_i	Features of the training video item T_i
v_j	Features of the incoming video item V_j
k_1, k_2	Number of frames used to represent T_i and V_j
x_i, z_j	Basis of the PCA-projected subspaces of t_i and v_j
$\tilde{x_i}$	Orthogonal complement to x_i, namely, $\tilde{x_i}^T x_i = 0$
$\theta(y)$	Geodesic flow function
U, V, Σ_1, Σ_2	Matrices used to compute $\theta(y)$ and W_{ij}
W_{ij}	Geodesic kernel, used to compute distance between x_i and z_j

Table 5.1: List of symbols used in computing video similarity

in [102]. In such cases, the similarity between two video feeds[1] is much more noticeable if the features in the feeds are projected on to a common subspace. We use image key-points, and the histogram of gradient (HOG) as features of the image frames in a video feed for comparison; the chosen features (to be used in EECS) will be described in detail later, in section 5.5.

The key idea in domain adaptation, is to project the two video feeds (a training video and an unknown video feed) onto a common subspace, in which similar patterns between the videos can be better identified. Here, we choose to project the training and the incoming videos (also referred to as data items) onto a Grassmannian manifold, as in [96]. The geodesic flow curve is the *shortest path* that links two projected items on the manifold, and represents a measure of similarity of the data distributions on the manifold. If two items have similar distributions on the Grassmannian manifold, the same video processing or detection algorithm should be applied on the two video feeds [102].

[1]We use the terms video feed and video item interchangeably.

In more detail, we formulate the problem of assessing the similarity between two videos as follows [2]. Let the features to be compared in the training and captured videos, T_i and V_i be $t_i \in \mathbb{R}^{k_1 \times \alpha}, v_j \in \mathbb{R}^{k_2 \times \alpha}$, respectively. Here, k_1, k_2 are the number of key frames (images) chosen in T_i and V_j, respectively, to represent the entire video feeds to reduce the computational overhead. In addition, α is the dimension of the feature vector of each chosen key frame (image). In other words, each video feed is represented as a set of images, where each image is then represented as a feature vector in the $\mathbb{R}^{\alpha}$ space.

Using principal component analysis (PCA), we project all the images in T_i onto a $\mathbb{R}^{\beta}$ subspace in which the variances of data are maximized; typically $\beta < \alpha$. The basis of such a subspace consists of β orthogonal basis vectors of size α (α-dimensional vectors). Let x_i be the basis of the subspace, then $x_i \in \mathbb{R}^{\alpha \times \beta}$. Let z_j be the basis of the subspace obtained when applying PCA on v_j; similarly, $z_j \in \mathbb{R}^{\alpha \times \beta}$.

Let $Gr(\beta, \mathbb{R}^{\alpha})$ be a special space that contains all the subspaces of size β in $\mathbb{R}^{\alpha}$; this space is called a Grassmannian manifold of $\mathbb{R}^{\alpha}$. Then, both the subspaces represented by x_i and v_j lie on $Gr(\beta, \mathbb{R}^{\alpha})$. Let $\tilde{x_i} \in \mathbb{R}^{\alpha \times (\alpha - \beta)}$ be the orthogonal complement to x_i, namely $\tilde{x_i}^T x_i = 0$, where x^T denotes the transpose of matrix x.

Given the Grassmanian manifold, the geodesic flow connecting x_i and z_j on the manifold, is defined as [96]:

$$\int_0^1 (\theta(y) t_i)^T (\theta(y) v_j) dy = t_i^T W_{ij} v_j. \tag{5.1}$$

[2]The list of symbols used in this section is summarized in Table 5.1.

The left hand side of (5.1) provides the definition of the geodesic flow, whose value can be computed using the right side of the equation. $\theta(y)$ is the geodesic flow function, parameterized by a continuous variable $y \in [0, 1]$ [96].

The geodesic flow can be computed by computing the kernel function W_{ij} between the two feature vectors t_i and v_j. W_{ij} is defined as:

$$W_{ij} = [x_i U \quad \tilde{x}_i V] \begin{bmatrix} \Lambda_1 & \Lambda_2 \\ \Lambda_2 & \Lambda_3 \end{bmatrix} \begin{bmatrix} U^T x_i^T \\ V^T \tilde{x}_i{}^T \end{bmatrix}, \tag{5.2}$$

where U, V are the left singular matrices when applying singular value decomposition (SVD) to $x_i^T z_j$ and $\tilde{x}_i^T z_j$, respectively. Further, let Σ_1 and Σ_2 be the diagonal matrices of such SVDs; the values of Λ_1, Λ_2, and Λ_3 matrices are computed from both Σ_1 and Σ_2 [96]. The kernel function W_{ij} provides an effective way to compute the inner product of high dimensional vectors t_i and v_j, which is widely used to compute the similarity between the vectors [108].

The kernel distance between the two video feeds T_i and V_j (i.e., between features of two sets of images t_i and v_j) on the manifold is computed based on the geodesic flow connecting them as [109]:

$$K(T_i, V_j) = t_i^T W_{ij} t_i + v_j^T W_{ij} v_j - 2 t_i^T W_{ij} v_j, \tag{5.3}$$

where $K(T_i, V_j)$ is a $k_1 \times k_2$ matrix, representing the individual distances (on the manifold) from each image in T_i to each image in V_j.

We define the total distance between the two video feeds on the manifold to be the mean of all the kernel distances between the individual images from the two feeds:

$$M_d(T_i, V_j) = \frac{1}{k_1 k_2} \sum_{m_1} \sum_{m_2} K_{(m_1, m_2)}(T_i, V_j), \tag{5.4}$$

where m_1, m_2 are integers in $\{1, \cdots, k_1\}, \{1, \cdots, k_2\}$, respectively, and $K_{(m_1,m_2)}(T_i, V_j)$ is the element (m_1, m_2) of the matrix $K(T_i, V_j)$.

Finally, we define the similarity of the two videos T_i, V_i as:

$$Sim(T_i, V_j) = e^{-M_d(T_i,V_j)}. \tag{5.5}$$

Notice that $Sim(T_i, V_j) \in [0, 1]$, for $M_d(T_i, V_j) \geq 0$. A higher distance corresponds to a lower similarity value; further, the similarity approaches 0 exponentially fast beyond some certain threshold (e.g., when $M_d(T_i, V_j) \geq 4$). In such cases, the video feeds are considered dissimilar.

5.4 EECS system design

In this section, we describe the design of our camera co-ordination framework EECS, consisting of two main components: camera sensors and a central controller (we also use the terms cameras and controller). The goal of EECS is to allow the central controller to collect visual features (discussed later in section 5.5) from the camera sensors and use these to determine the most effective detection (video processing) algorithm (in terms of accuracy and energy consumption) for each camera, and what combination of views yields the desired object detection accuracy, while ensuring that the energy drain at the camera sensors adheres to a set energy budget. In EECS, video analytics and algorithm selection happen at the controller to avoid storing information about training video feeds and executing processing-expensive, domain adaptation at each battery-operated camera sensor. Here, as typically the case, we assume that the controller does not have energy constraints and can easily perform these required operations.

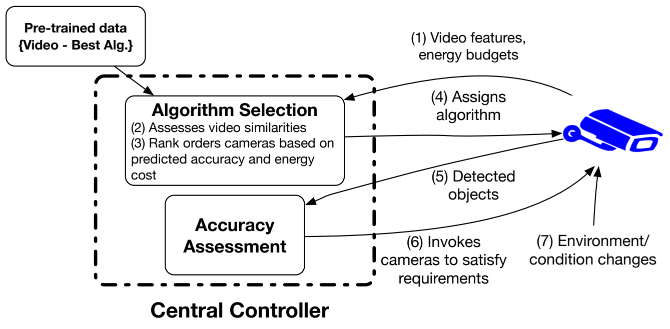

Pre-trained data
{Video - Best Alg.}
Algorithm Selection
(2) Assesses video similarities
(3) Rank orders cameras based on predicted accuracy and energy cost
Accuracy Assessment
Central Controller
(1) Video features, energy budgets
(4) Assigns algorithm
(5) Detected objects
(6) Invokes cameras to satisfy requirements
(7) Environment/ condition changes

Figure 5.1: EECS system for adaptively choosing detection algorithms in a camera network.

Each camera node individually executes a detection (video processing) algorithm to detect the presence of objects (e.g. humans) in the scene. The detection accuracy of a certain algorithm depends on how well it matches the environment conditions, which are dictated by attributes such as brightness and indoor versus outdoor, etc. At the same time, different algorithms consume different amounts of energy. The controller collects information relating to the detected objects (details discussed later) as well as residual energy information from the camera sensors. Based on the assessment of the achieved global detection accuracy and the energy budgets and expenditures at each camera, the controller node adaptively invokes different sets of cameras and/or different detection algorithms to meet both the accuracy and energy requirements. Fig. 5.1 depicts the functional view of system with its different components. In the following, we describe the details of the EECS framework.

Let $\mathcal{T} = \{T_1, T_2, \cdots, T_N\}$ be the set of training videos at the controller. Let $\mathcal{A} = \{A_1, A_2, \cdots, A_H\}$ be the set of available detection algorithms pre-installed at each individual camera. Each camera sensor $S_j \in \mathcal{S}$ of M cameras has an energy budget B_j, which is a function of the required operation time as well as other processing parameters (such as number of frames processed per second). In addition, each camera has a communication cost C_j, which depends on the link quality from the camera to the central controller and is independent of the detection algorithm assigned to the camera[3]. Let $\mathcal{V}$ be the set of captured video feeds, where $V_j \in \mathcal{V}$ is the video feed captured at camera S_j. The goal of EECS is to identify a subset of the camera sensors $\mathcal{S}' \subseteq \mathcal{S}$ and the corresponding video processing algorithm $A'_j \in \mathcal{A}$ to be used at *each* camera $S_j \in \mathcal{S}'$ to satisfy a predefined global accuracy D, while adhering to the energy constraints $c(A'_j) + C_j \leq B_j$. Here, $c(A'_j)$ represents the computation cost for algorithm A'_j.

5.4.1 Offline training

A key task of the controller is to rank order the video processing algorithms based on their accuracies and identify the most accurate algorithm for the captured video feed V_j with regards to each individual camera sensor S_j. EECS performs a video comparison of the incoming video with the training videos using the domain adaptation technique, described in section 5.3.

First, the controller node applies each available detection algorithm to process each training item, and measures the detection accuracy achieved and the computational cost (a total of $H \times N$ combinations). Specifically, for each training item, the controller measures the precision and recall values for each algorithm. The precision value is the number of correctly identified objects from among the detected objects, while the recall is the number of detected objects from among the objects actually in the scene. The *f_score*

[3]Specifically, C_j depends on the resolution of the captured video, and the available bandwidth between the camera sensor and the central controller. This can be estimated using tools such as iPerf [90], by transferring some sampled frames and recording the consumed energy.

value [110], which is usually used to assess the accuracy of a detection algorithm, is then computed as $f\_score = 2 \times \frac{recall \times precision}{recall + precision}$. For each training item, a ranked list of algorithms is then constructed based on the f_score. For each training video $T_i \in \mathcal{T}$, the most accurate detection algorithm, labeled $A_i^* \in \mathcal{A}$, is identified.

5.4.2 Resource-aware algorithm selection

In this subsection we describe how the information gathered at the controller node from the camera sensors is used to adaptively choose the detection algorithm for each camera sensor.

5.4.2.1 Uploading video features

When the camera sensors start up, or when they detect surrounding environment changes, each sensor S_j extracts and uploads features (such as image key points and histogram of gradient; details discussed in section 5.5) of the captured video feed V_j to the controller. Further, each camera notifies the controller node about its energy budget B_j, as well as the energy cost to process an image frame using each of the available algorithms. A camera sensor estimates the processing energy costs of the algorithms by applying each algorithm to a few sampled frames and recording the consumed energy, using tools such as PowerTutor [11].

5.4.2.2 Rank ordering the detection algorithms

Once the controller node receives video features V_j from camera sensor S_j, it determines the video similarities between the input and the items in its training set, and identifies the closest training item $T_i^* \in \mathcal{T}$ that is most similar to V_j using Equation (5.5). Because EECS uses state-of-the art video comparison techniques, a high similarity between T_i^* and V_j will indicate that the two videos should be processed by the same algorithm [102]. Thus, the A_i^* that is associated with T_i^*, will be the most accurate algorithm that can be used to process V_j. Further, the process ensures that the rank ordering of the detection algorithms will also be similar (if not identical) between the videos.

5.4.2.3 Choosing a subset of cameras

Periodically, for a short *accuracy assessment* duration (e.g., 100 frames), the controller coordinates across all the cameras, so that each uses its most accurate detection algorithm, whose processing cost, together with the communication cost, are not higher than the energy budget. Then, metadata[4] about detected objects is returned to the controller. This information is then used to estimate the best possible global accuracy that can be achieved (described later). The cameras are then rank ordered based on their individual accuracies in the list $\mathcal{S}_o = \{S_1, S_2, ..., S_m\}$.

Next, the controller node sequentially invokes the cameras in the list $\mathcal{S}_o$, one by one in order, until the desired global accuracy is satisfied. This set of chosen cameras is denoted by $\mathcal{S}' \subset \mathcal{S}$. This approach ensures that EECS does not invoke all the camera sensors unnecessarily, but only invokes a sufficient set of cameras to satisfy the detection accuracy requirements while conserving energy.

[4]The metadata consists of features extracted from the video (e.g., color features) and will be defined formally later in this section.

5.4.2.4 Choosing detection algorithms

In the previous step, a set of cameras that sufficiently satisfies the required accuracy using the associated "best" detection algorithms $A_j^*, \forall j \in \{1, 2, \cdots, M\}$ are chosen. Here, we further seek to reduce the energy usage at the camera sensors while still satisfying the detection accuracy requirements.

First, we browse the list $\mathcal{S}'$ in reverse order to consider the cameras with the lower accuracies first, towards reducing energy expenses. Then, for each camera, we check whether a different lower energy algorithm can be used, while still retaining the required global accuracy D. This reduction in accuracy is a direct consequence of the algorithm chosen at that camera. In other words, at each step the accuracy with regards to the other cameras is not affected. Note that, in order to reduce the number of alternatives that need to be explored, EECS only pays attention to algorithms that have higher $\frac{f\_score}{\text{energy cost}}$ values compared to the most accurate algorithm. Metadata relating to each object detected by such algorithms is uploaded to the controller, which in turn determines the algorithm that consumes the least energy, and yet ensures that the required accuracy is achieved. It selects this algorithm for the corresponding camera and feeds back this information to the camera sensor. If such an algorithm is not found, then this process stops. Otherwise, the process continues with the next camera in $\mathcal{S}'$.

The set of cameras $\mathcal{S}'$ are used, along with the selected detection algorithms, until the next *re-calibration interval* (e.g., 500 frames), when a new accuracy assessment process starts, and a new set of camera sensors and detection algorithms might be chosen.

5.4.3 Global detection accuracy

In the above discussion, we implicitly assumed that the central controller can assess the global detection accuracy, given the accuracy assessments from the different camera sensors. The process is explained in detail as follows.

In order to assess the global accuracy, the controller needs first to identify the same object (e.g., human) captured from different cameras/views, and estimate the achieved accuracy pertaining to that object. Correctly aggregating the same detected areas (representing objects) from multiple views allow EECS to correctly identify the total number of objects that has been detected in the scene, since those areas will be counted as a single object. Otherwise, the same detected human might be counted multiple times, which leads to an incorrect assessment of the global detection accuracy. However, this re-identification of the same object from multiple camera views is a challenging in itself; this is because images of the same object (e.g., human) can be quite different if viewed from different angles.

Aggregating metadata from multiple cameras: For each detected area, the sensors extract and upload the metadata of that area (representing a potential object), which is used for object re-identification. Specifically, the metadata includes: (i) the location of the area in the image, (ii) color features of the area, and finally (iii) a confidence measure that the detected area is an actual object of interest.

Next, the metadata is extracted and leveraged as follows. For each detected area, a detection algorithm provides the location (rectangular bounding box) of the area, and a score reflecting how confident the algorithm is, with regards to the area representing an object of interest (for example, see the detection algorithm in [63]). This score can be converted into a detection probability via an offline training process.

Next, we discuss how EECS identifies and verifies detected areas from different views as the same area. First, a set of landmark points on the ground are chosen in the real world coordinate system. The locations of these landmarks are then identified in the captured images of each individual cameras. Using the correlation between the locations of the landmarks in the images captured by two cameras facilitates the building of a mapping function (called a homography) between the “ground planes” of the two cameras (e.g., by using RANSAC [111], which produce very accurate results).

Once such a homography is constructed, for each detected area in an image, the software at the central controller extracts the center of the bottom edge in the frame (which is supposed to be on the ground), and then projects that center point onto the ground plane of other camera views to identify detected areas of the same object in the other camera views.

Further, the camera sensors also extract and upload color features (as a part of the metadata) of the detected area to help the controller reduce the false matches due to imperfect homography matching. Specifically, in EECS, we extract the Mean Color feature [87] of a detected area, which is a 55,000-dimensional signal, and then use PCA to reduce the feature to 40 dimensions (as in [87]). Then, the Mahalanobis distance [89] is used to compute the distance between the color features of two pre-matched (by homography mapping) detected areas in different cameras; if the distance is within a certain threshold, we consider the two detected regions to correspond to the same object.

Assessing global detection accuracy: First, we denote the area on the image for each detected object i, using the algorithm running on camera S_j, by R_{ij}. For each R_{ij}, a detection probability P_{ij} is learned, as discussed above, representing the detection precision associated with that area. Thus, P_{ij} indicates the probability that the area R_{ij} is actually an object of interest.

Once the metadata corresponding to the different objects is collected by the controller, the following quantities can be computed to assess detection accuracy: (i) the number of objects on the field jointly detected by the cameras (after re-identification), and (ii) the combined detection probability of each detected object, computed as:

$$P_i = 1 - \prod_j (1 - P_{ij}), \tag{5.6}$$

where $(1 - P_{ij})$ is thus the false positive probability of object i on camera j. The aggregated *true positive* detection probability P is computed as the probability that *all* cameras S_i yield false positive detections.

The controller periodically triggers each camera S_j to use the best algorithm A_j^* to compute the "baseline" (i.e., best possible) global detection accuracy. This reflects the number of detected objects, and the average detection probability. If the current detection accuracy is below such baseline by more than a threshold D, more cameras and/or more expensive algorithms might be invoked.

Summary: Fig. 5.2 summarizes the operations and interactions between video sensors and the central controller. The individual camera sensors capture video feeds, extract specific features and send these features along with their energy residue information, to the central controller. The central controller does the video analytics to select a sub-set

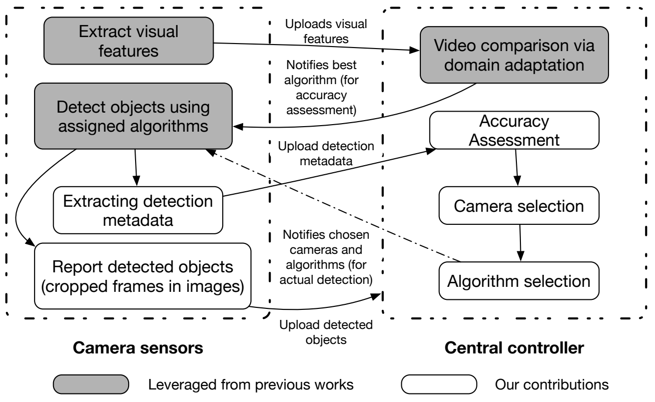

Extract visual features
Uploads visual features
Video comparison via domain adaptation
Notifies best algorithm (for accuracy assessment)
Detect objects using assigned algorithms
Accuracy Assessment
Upload detection metadata
Extracting detection metadata
Camera selection
Notifies chosen cameras and algorithms (for actual detection)
Report detected objects (cropped frames in images)
Algorithm selection
Upload detected objects
Camera sensors
Central controller
Leveraged from previous works
Our contributions

Figure 5.2: Interactions between the sensors and central controller

of cameras, and the video processing algorithms that must be used at these cameras, to achieve a good trade-off between detection accuracy and energy expenses.

5.5 Implementation

In this section, we describe the detailed implementation of EECS; as discussed, it consists of two main components viz., camera sensors and a central controller that gathers the outputs from these cameras.

Note that, in implementing EECS, we focus on "humans" as objects; this has significant importance in rescue, tactical and homeland security missions. The techniques can however, be used to detect other types of objects; only the detection algorithms used at the camera nodes need to be replaced.

5.5.1 The camera sensors

We implement the camera nodes using Asus Zen II Android smartphones. Each node is pre-installed with 4 different human detection algorithms: HOG[5] [63], ACF [68], C4 [98] and LSVM [97]. We use OpenCV to implement HOG, LSVM, and ACF (based on the source code provided by the authors). For C4, we use the source code (in C++) provided by its authors.

First, each captured video feed is represented by 100 image frames. For each frame, we extract the HOG (histogram of gradient) features [63] and SURF (speeded-up robust features) key-points[6] [112] of the image using OpenCV. The HOG features are represented by a 3780-dimension feature vector. For the SURF key-points, we use the bag of words (BoW) approach [46]. Specifically, each SURF key-point is represented by a 64-dimensional vector, called the key-point descriptor. Once a training set is chosen (more details about data sets are provided later in section 5.6), key-point descriptors of the training images are extracted, and then partitioned into predefined k clusters using the k-means clustering algorithm. Each such cluster centroid is called a *visual word* in the vocabulary.

In EECS, a vocabulary of 400 words is built from images of 12 training video feeds. Subsequently, for any given image, the key points of the image are extracted, and each key point is then mapped to the nearest cluster centroid (visual word). The BoW representation of an image, regardless of the image size and number of key-points, is thus a 400-bin histogram, in which the value of each bin is the number of key-points mapped on

[5]We use the term HOG for both the feature (histogram of gradient) and the algorithm that leverages the feature.

[6]Key-points are small patches of an image that differ significantly from the surrounding areas (in the image).

to the associated visual word. Thus, each image frame in a incoming video feed is actually presented by a fixed 4180-dimension feature vector (combining the HOG feature and the BoW representation). The camera nodes then upload the features of the set of chosen image frames of the captured video feed to the central controller for video comparisons and subsequent, processing (detection) algorithm selection.

Once a detection algorithm is assigned by the central controller, the camera sensors detect the presence of objects of interest (i.e., humans), and upload the information relating to the detected areas (the locations and the color features of the objects in the frame) to the controller for accuracy assessment and re-calibration. We also utilize OpenCV to extract features from the detected areas on the smartphones (used as camera nodes).

5.5.2 The central controller

The central controller is implemented on a Linux server. It contains a pre-installed training set consisting of different video items (details in section 5.6). The accuracy (*precision*, *recall* and *f_score* values) of each algorithm on each item in the training set, is computed. The controller then ranks the algorithms based on the *f_score* values.

Once it receives the uploaded video features, the controller estimates the similarity of the received video with the training items; the most similar training item to the captured video feed at each camera is determined. The source code provided by the authors of [96] is used to compute video distances on the Grassmannian manifold. The controller node then follows the framework described in section 5.4.2 to select a subset of cameras and the associated detection algorithms to achieve a desired accuracy.

Verifying the desired detection accuracy is met: In reality, ground truth information may not be available, and involving a human in the loop to manually verify the detection is assumed to be not possible (or can be expensive or if the human has to continuously provide feedback on the results). Thus, an important question is how to verify if a required detection accuracy is satisfied?

In our system, we use the highest possible accuracy, that is, when the most accurate algorithm is used at each individual camera, as the baseline for comparison. Specifically, let N^* and P^* are the number of detected objects and the average detection probability (of all detected objects), respectively, when the *best* algorithm is used at *all* the cameras. The desired accuracy is then defined proportionally to the values of N^* and P^*. Specifically, let D_n and D_p be the desired number of detected objects and the desired mean value for the detection probability, respectively. Here, the values of P^* and D_p are computed as the average probability across all the detected objects, where the probability for each object is computed by Equation 5.6. We then require that $D_n \geq (\gamma_n \times N^*)$, and $D_p \geq (\gamma_p \times P^*)$; the values of γ_n and γ_p can be changed to influence the desired accuracy. Periodically, the detection accuracy is reassessed to determine if more cameras or more accurate algorithms need to be used.

5.6 Evaluations

In this section, we evaluate both the accuracy and energy efficiency of EECS.

Training and test datasets: We use the following publicly available datasets in our evaluation; each of the datasets consists of video feeds captured from 4 overlapping cameras:

- Dataset #1 is the "lab sequences" dataset, provided by EPFL [113]. This dataset consists of indoor video feeds in an empty room setting. In each video there are 6 people walking in the room. The resolution of the video set is 360x288.
- Dataset #2 is the "chap" dataset, provided by Graz University [7]. This dataset also consists of indoor video feeds in a lab setting, in which there are 4-6 people walking in the room. There are furniture items in the lab which might cause false positives in terms of detection; thus this dataset has lower precision than the other datasets. The resolution of this dataset is 1024x768. Thus the energy cost to process this dataset is expected to be higher than in the other datasets.
- Dataset #3 is the "terrace sequences" dataset, also provided by EPFL [113]. This dataset contains outdoor video feeds, with 8 people walking on an empty terrace of a building. The resolution is 360x288.

Since each set contains video feeds captured simultaneously from 4 overlapping cameras, there are 12 videos feeds in total. Each of those video feeds is approximately 3000 frames long. We use the first 1000 frames in each video feed as the training video. The remaining 2000 frames from each video feed are used as test data.

Ground truth information: All the datasets we use contain ground truth information about human locations in the scene. In particular, the 3D locations (in the real-world coordinates) of where the humans stood in the scene are marked. Further, for each video feed in the dataset, the homography used to transform coordinates on the ground plane in the image into real-world coordinates is provided[7]. Using such homography, those 3D locations are converted to 2D locations in each video frame. This is then compared with the results of the detection algorithm to estimate accuracy in our evaluation. For datasets #1 and #3, ground truth information is available every 25 frames, whereas the ground truth is available every 10 frames for dataset #2.

Re-identifying detected objects across cameras: Using the provided homography information described above, EECS can re-identify and aggregate the same objects detected by different cameras. Color features of the matched objects are then verified to reduce false positives, as described in section 5.6.1. By using these two techniques, EECS is able to re-identify objects with a high precision (more than 90%) in all the data sets.

Computing energy costs and budget: For each sensor, we apply each algorithm to each of the videos to learn the processing cost of the algorithm; the process is repeated over 1000 frames to get the average value. We transfer 1000 frames (we compress the frame using the "jpeg" format before transferring) using WiFi in good conditions to estimate the communication cost. In EECS, sensors only transfer cropped image frames containing the detected objects, to the controller. The amount of transferred data thus varies across frames (since the number of detected objects and the sizes of the different objects, also vary). Thus, to estimate the communication cost, we assume the whole frame is transferred, and monitor the consumed energy. Doing so ensures the actual communication cost will never be higher than our estimated value.

[7]Such a homography is built by marking the landmarks in the field and in the images, as described in 5.4.2.

Finally, the energy budget is computed by first defining a expected operation time (e.g., 6 hours) and an expected frame rate (e.g., image frames are processed every 2 seconds). Given these, the number of frames need to be processed during the operation time (e.g., how often batteries need to be recharged or replaced) can be computed. Subsequently, the residual energy capacity is divided by the number of frames to compute the energy budget for each frame. In the evaluations that are presented later, we actually use different budget values to evaluate how EECS adaptively chooses different algorithms under different given budget constraints.

5.6.1 Estimating the detection accuracy

Each camera sensor is pre-installed with 4 different detection algorithms. Video feeds available are split into training segments and test segments. We apply all the algorithms to each of the training video segments to characterize the accuracy of each algorithm on each training segment. Each object detected is assigned a *detection score* by the algorithm, reflecting a measure of confidence in detection. We discard areas on a frame with very low detection score below a cut-off dectection score threshold d_t. Different cut-off thresholds correspond to different values of f_score. For example, a higher threshold will disregard detection areas with lower scores to reduce the false positive rate, but on the other hand, might also cause correctly detected areas with lower scores to be ignored and thus reduce the recall (true positive rate). Thus, we choose a threshold d_t which maximizes the f_{score} value. In other words, for each combination of an algorithm and a training video segment, we record the highest possible accuracy the algorithm can achieve on that segment.

Alg.	Threshold	Recall	Precision	F-score	Energy cost/frame(J)	Processing time/frame (s)
HOG	0.5	0.48	1.0	**0.66**	1.08	1.5
ACF	2	0.34	0.95	0.505	0.07	0.1
C4	0	0.46	1	0.63	4.92	2.4
LSVM	-1.2	0.89	0.9	0.89	3.31	6.2

Table 5.2: Accuracy of different algorithms on dataset #1, camera #1, frame 0→1000, used as a *training* video item.

Alg.	Threshold	Recall	Precision	F-score	Energy cost/frame(J)	Processing time/frame (s)
HOG	0.6	0.8	0.42	0.55	9.86	3.4
ACF	20	0.83	0.89	**0.86**	0.315	0.4
C4	0.5	0.70	0.70	0.70	5.56	6.8
LSVM	-0.2	0.84	0.83	0.84	25.06	32.2

Table 5.3: Accuracy of different algorithms on data set #2, camera #1, frame 0→1000, used as a *training* video item.

The video feeds are split into two segments. The first segment, 1000 frames, is used as training items. Tables 5.2 and 5.3 show the efficiencies of the detection algorithms on the training items extracted from video feeds captured from camera #1, in dataset #1 and in dataset #2, respectively. We also apply the algorithms on the test video segments using the same threshold values learned from the training segments. Table 5.4 shows the accuracy of the detection algorithms on the feed captured from camera #1 in dataset #1, from frame 1001 to 2950. The energy costs shown in these tables include the processing costs of the corresponding algorithm as well as the algorithm-independent communication cost to transfer the images of detected objects to the central controller, as described earlier in this section.

In Tables 5.2 and 5.4, we use two different segments from the same video which is captured by camera #1 on dataset #1. The tables show that the LSVM algorithm, even though has a very high *f_score* value, also has very high energy cost and processing time;

Alg.	Threshold	Recall	Precision	F-score	Energy cost/frame(J)	Processing time/frame (s)
HOG	0.5	0.6	0.99	**0.74**	1.07	1.8
ACF	2	0.52	0.91	0.66	0.07	0.1
C4	0	0.534	0.974	0.69	4.82	2.3
LSVM	-1.2	0.975	0.892	0.93	3.2	6.4

Table 5.4: Accuracy of different algorithms on dataset #1, camera #1, frame 1001 $\rightarrow$ 2950, used as a *test* item.

thus, it can be expensive for use on a battery driven mobile platform. For that reason, we will not consider LSVM in the remainder of this section. Thus, for video feeds on dataset #1, camera #1, HOG will be considered the most accurate detection algorithm.

5.6.2 Evaluating video similarity using domain adaptation

Extracting features from an entire video feed is expensive. Here, we only extract the features (HOG, and BoW) of a 100 consecutive frames. To reduce the bias, the frames are randomly selected from each video feed and the process is repeated 5 times. We then report the average value of the similarity.

Table 5.5 shows the similarities between the training and test video items, computed as described in (5.5). In the table, $T_{x.y}$ (or $V_{x.y}$) indicates the training (or test) video item is from dataset $\#x$ and captured by camera $\#y$. It is shown that, using the manifold distance, in all the cases, we are able to match a test item to the training item corresponding to the same dataset and captured by the same camera. This observation is also confirmed by the results in tables 5.2 and 5.4. These tables show that video items belonging to same camera and same dataset have the same most accurate detection algorithm as well as the same "order" of the algorithms in terms of accuracy.

Test set/ Train set	$V_{1.1}$	$V_{1.2}$	$V_{1.3}$	$V_{1.4}$	$V_{2.1}$	$V_{2.2}$	$V_{2.3}$	$V_{2.4}$	$V_{3.1}$	$V_{3.2}$	$V_{3.3}$	$V_{3.4}$
$T_{1.1}$	**0.78**	0.56	0.53	0.56	0.47	0.49	0.48	0.45	0.46	0.48	0.49	0.44
$T_{1.2}$	0.55	**0.77**	0.54	0.60	0.46	0.48	0.48	0.49	0.48	0.49	0.46	0.40
$T_{1.3}$	0.54	0.54	**0.76**	0.53	0.45	0.47	0.50	0.45	0.48	0.49	0.49	0.41
$T_{1.4}$	0.56	0.61	0.54	**0.76**	0.47	0.50	0.51	0.48	0.48	0.53	0.48	0.40
$T_{2.1}$	0.39	0.39	0.38	0.38	**0.79**	0.45	0.48	0.43	0.37	0.37	0.39	0.34
$T_{2.2}$	0.44	0.47	0.45	0.47	0.48	**0.75**	0.51	0.45	0.43	0.43	0.46	0.39
$T_{2.3}$	0.41	0.45	0.43	0.46	0.49	0.51	**0.81**	0.47	0.45	0.41	0.41	0.37
$T_{2.4}$	0.38	0.43	0.39	0.41	0.45	0.44	0.47	**0.76**	0.41	0.36	0.39	0.37
$T_{3.1}$	0.50	0.54	0.51	0.53	0.43	0.45	0.48	0.49	**0.69**	0.48	0.46	0.45
$T_{3.2}$	0.44	0.47	0.44	0.49	0.41	0.41	0.45	0.39	0.40	**0.69**	0.43	0.38
$T_{3.3}$	0.48	0.48	0.49	0.49	0.47	0.47	0.44	0.43	0.41	0.46	**0.74**	0.49
$T_{3.4}$	0.45	0.43	0.45	0.43	0.42	0.41	0.43	0.42	0.42	0.40	0.51	**0.75**
$T_{x.y}$, $V_{x.y}$ denote training and test video feed captured by camera #x in dataset #y, respectively												

Table 5.5: Video similarities computed using the manifold distance

5.6.3 Benefit of adaptively choosing the detection algorithms

Next, we show the benefits of adaptively choosing different algorithms to process different video feeds.

Fig. 5.3 shows the highest detection accuracy when different algorithms are used to process the video feeds captured by camera #1 in both dataset #1 and dataset #2.

Assuming that the environment changes (from dataset #1 to #2), if the same algorithm is still used, the highest *f_score* the system can achieve by using one detection algorithm is 0.70 (using HOG algorithm) for both datasets. However, if the system adaptively uses the best algorithm for each dataset (specifically, HOG for dataset #1 and ACF for dataset #2), the highest *f_score* achieved is 0.81.

More importantly, adaptively choosing the most accurate algorithm helps increase the recall and precision values *simultaneously*. Specifically, if HOG algorithm is used, the

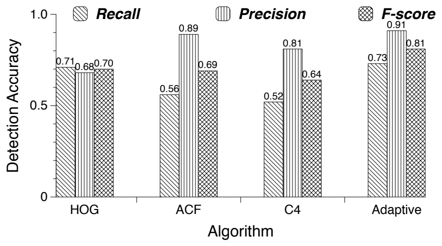


Figure 5.3: Achieved accuracy with different detection algorithms, dataset #1 and #2

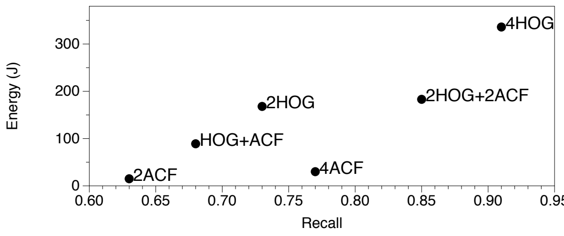


Figure 5.4: Trade-off between accuracy and energy cost to process dataset #1

achieved recall is 0.71, which is close to 0.73 of the adaptive approach. However, the precision is much lower, 0.68, compared to 0.91 achieved when the adaptive approach is used (corresponding to a higher false positive rate compared to the adaptive approach). Similarly, when ACF is used, the precision is good, however, the recall is significantly lower than the adaptive approach. In other words, the false negative rate is high. Thus,

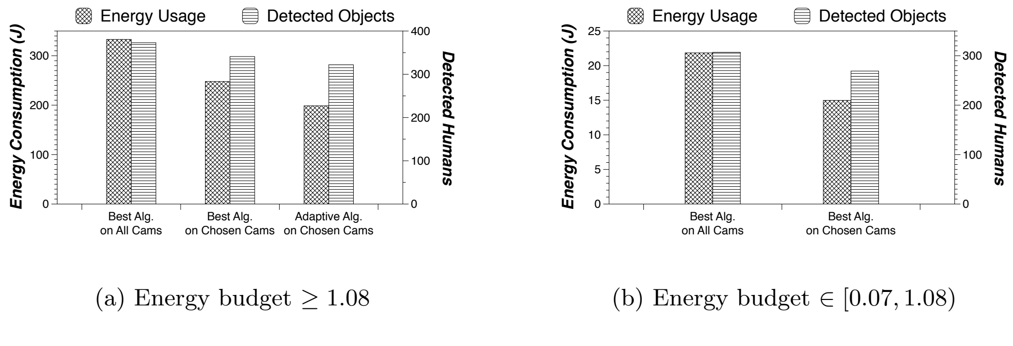


Figure 5.5: Detected humans vs. energy consumption for dataset #1, with different energy budgets

using an adaptive approach helps reduce both the false negative and false positive rates simultaneously.

5.6.4 Should the highest accuracy algorithm always be used?

Based on video comparison, EECS can identify the most accurate detection algorithm for an incoming video feed. However, should the most accurate algorithms *always* be used to process the video feeds?

Fig. 5.4 shows the trade-off between the achieved accuracy (in terms of the number of correctly detected humans) and energy costs when processing the 4 video feeds in dataset #1. We show the values when 2 cameras are used: (i) 2HOG: both cameras used HOG (the most accurate, yet expensive, algorithm), (ii) 2ACF: both cameras used ACF (sub-optimal, yet energy efficient, algorithm), (iii) HOG+ACF: one camera used HOG and the other used ACF; and when 4 cameras are used: (iv) 4HOG: all 4 cameras used HOG, (v) 4ACF: all 4 cameras used ACF, and finally, (vi) 2HOG+2ACF: two camera used HOG while the other two cameras used ACF.

The x-axis shows the recall achieved (the number of humans detected among the humans appearing in the scene)[8], while the y-axis shows the energy consumption for each case.

It is shown that, depending on the desired accuracy, a sub-optimal solution can be used to achieve a significantly lower energy consumption, but with a relatively small accuracy hit. For example, the 2HOG+2ACF option only consumes $\approx 54\%$ of the energy

[8]For dataset #1, the precision is ≥ 0.95 for all the algorithms (Table 5.2). Thus we only pay attention to the recall values in this case.

consumed by the 4HOG option, while in the former case, 85% of objects actually appeared in the scene were detected, compared to 92% in the latter case. the difference in the achieved accuracies was only $\approx 7\%$.

5.6.5 Adaptive choice of algorithms in EECS

Next, we evaluate EECS for adaptively choosing detection algorithms based on the energy budgets and desired detection accuracy. For simplicity, we only show the results for dataset #1 and dataset #2. Similar results are observed in the other dataset.

In this experiment, we choose $\gamma_n = 0.85$ and $\gamma_p = 0.8$, indicating EECS is allowed to reduce the number of cameras, or assign sub-optimal detection algorithms to the cameras, as long as: (i) the number of detected objects is at least 85% of N^*, and (ii) the average detection probability is at least 80% of P^*, where N^* and P^* are the number of detected objects, and the average detection probability of all the detected objects, respectively, when the best algorithms are used by all cameras.

In addition, EECS uses other parameters to control the re-calibration process. In particular, the accuracy assessment period and the re-calibration interval (see section 5.4.2) are set to 100 and 500 frames, respectively. In other words, EECS uses the detection metadata from 100 frames to assess the detection accuracy and decide the set of cameras and associated detection algorithms. This decision is then used for 500 frames before the accuracy is reassessed again. Note that, for datasets #1 and #2, the ground truth is available every 25 frames. To evaluate EECS, we only process frames that have ground truth information. Thus, EECS actually uses the information from 4 frames to assess the accuracy and select the cameras and detection algorithms. Such selection is then used to process the next 20 frames before re-calibration. In practice, EECS computes the highest possible accuracy (by using the most accurate detection algorithms) using detection metadata. This accuracy measure is then used to calibrate the desired accuracy in place of ground truth information.

Fig. 5.5 shows the energy consumption and the number of correctly detected humans when (i) the best algorithm is used in each of the 4 cameras, (ii) EECS only chooses a smaller set of cameras that is sufficient to achieve the desired accuracy, and finally (iii) EECS chooses sub-optimal algorithms on selected cameras while adhering to the required accuracy.

In Fig. 5.5a, the energy budget is relatively high and so that the camera sensors can choose HOG (the most accurate algorithm) to detect humans. When all cameras use the best algorithm, the whole system consumes ≈ 333 Joules and correctly detects 373 humans in total. However, EECS only needs to use 3 cameras to achieve similar accuracy. If all the *selected* cameras still use the most accurate algorithm (HOG), the energy consumption is reduced to $\approx$ 248 Joules ($\approx$ 75% of the highest consumption), while the number of detected humans is 341 ($\approx$ 91% of the highest accuracy). Further, EECS assigns sub-optimal algorithm (ACF) to some of the cameras to further reduce the energy consumption to $\approx$ 131 Joules ($\approx$ 59% of the highest consumption energy cost). This energy conservation is achieved while still detecting 322 humans ($\approx$ 86% of the highest accuracy).

In Fig. 5.5b, the available energy budget is less than the energy costs incurred with HOG. Thus, the camera sensors can now only use the sub-optimal algorithm ACF to detect objects. When all the cameras are used, there are 307 correctly detected humans

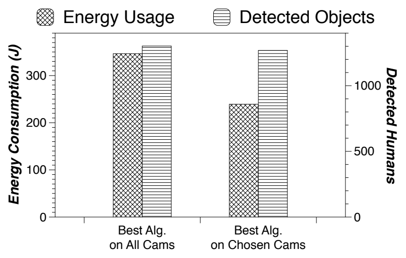


Figure 5.6: Detected humans vs. energy consumption for dataset #2

and the energy consumption is ≈ 22 Joules. EECS, however, uses fewer cameras (2 or 3 cameras) to achieve similar accuracy. Specifically, the framework can detect 269 (≈ 88% of the highest accuracy) humans with an energy consumption of only 15 Joules (≈ 68% of the highest consumption). Since ACF is already the most energy efficient algorithm, EECS cannot further reduce the energy consumption in this case.

In Fig. 5.6, we show the results for dataset #2. For this dataset, ACF is both the most accurate and most energy efficient algorithm. Thus, the results for this case are similar to those in Fig. 5.5b. Even though EECS is not able to reduce energy consumption by using more efficient algorithms, it only uses up to 3 cameras (only 2 cameras are used in some rounds) to achieve similar accuracy compared to when all 4 cameras are used. Specifically, EECS is able to correctly detect 1269 humans (≈ 97% of the highest accuracy), while only consuming 239 Joules (≈ 70% of the highest consumption).

5.7 Discussion

EECS can trade-off the global detection accuracy to some extent (while ensuring a pre-defined accuracy requirement) for energy conservation by assigning sub-optimal algorithms to some of the camera sensors, for certain periods in time. This reduction in detection accuracy could result in a number of undetected objects. However, EECS can be tuned to resist such misses. As it is, objects (e.g., human) that are not detected in some frames are likely to be detected at other frames (e.g., when the objects move to different locations). In addition, EECS can target energy conservation only in some rounds; thus, a lower detection accuracy is only experienced in such rounds. EECS would then periodically enforce higher accuracy requirements in other rounds to catch objects that were possibly missed earlier; we have done some preliminary studies that suggest that this only results in slightly increased energy costs. Specifically, if HOG and ACF are the ideal algorithms (as in our data set #1), and HOG yields a higher accuracy with a higher expense, then HOG can be used intermittently to increase accuracy in those corresponding rounds.

5.8 Conclusions

In this chapter, we present a framework, EECS, for supporting the co-ordination across a set of camera sensors to achieve a desired object detection accuracy but while achieving significant energy savings. Specifically, the framework ensures that cameras do not all unnecessarily use highly optimal but energy heavy video processing algorithms for object detection. In essence, it facilitates the adaptive choice a sub-set of cameras, and causes some of the chosen cameras to use sub-optimal detection algorithms to conserve energy while still achieving the pre-defined desired accuracy. Our evaluation shows that, EECS helps save more than 40% of the energy consumed compared to a case where all cameras use the optimal algorithm for detection and transfer key images relating to detected objects; however, it still achieves $\approx 86\%$ the accuracy achieved when the best algorithms are used at all of the camera nodes. EECS can be tuned to achieve the right trade-offs between energy efficiency and desired accuracy.

Chapter 6

Conclusions

My works have been focusing on improving the efficiencies of mobile applications, especially in the resource (e.g., energy and bandwidth) constrained settings. First, we develop TIDE, a framework to detect high energy consuming applications on smartphones. TIDE is totally user-centric, the high energy applications are identified based on the actual usage patterns on the user's phone. Further, TIDE does not require the users to root the devices or make any modifications to the mobile OSes. Our experiments show that TIDE accurately identifies 225/238 high energy applications, while imposing only 0.5% of overhead on the average consumption of the phone's battery per hour.

Subsequently, we pay attention to improving the efficiencies of mobile applications in large-scale scenarios. We propose and implement a framework to detect similarities in images uploaded my multiple wireless devices in bandwidth-limited settings, such as in rescue missions at natural disaster scenes, or at flash-crowd events. Our framework intelligently combines state-of-the-art techniques in computer vision, together with soliciting user feedback to achieve very high accuracy and low overheads. Experiments show that our framework helps to reduce $\approx$ 40% network delay in transferring unique and important contents.

Next, we present ACTION, our framework for accurate and timely object detection in bandwidth constraint settings. ACTION effectively collects detection information from multiple overlapping camera, aggregates the detection information at a fusion node to improve the detection accuracy. Finally, for each detected object, the most relevant (most accurate) detection information is chosen, based on given bandwidth constraints, to upload to a central controller to assist humans in rescue missions. Our evaluations show that ACTION helps to reduce up to three folds the network load from a testbed with 4 Android smartphones.

Finally, in chapter 5, we present EECS, an adaptive detection algorithm selection framework for multi-camera settings. EECS chooses only a subset of camera sensors for object detection; further, the framework allows each camera to use the most energy efficient algorithm to conserve energy while still ensuring a global desired detection accuracy. Our evaluations show that, EECS helps save more than 40% of the energy consumed compared to a case where all cameras use the optimal algorithm for detection and transfer key images relating to detected objects; however, it still achieves $\approx$ 86% the accuracy achieved when the best algorithms are used at all of the camera nodes. EECS can be tuned to achieve the right trade-offs between energy efficiency and desired accuracy.

Bibliography

[1] A. J. Oliner, A. P. Iyer, I. Stoica, E. Lagerspetz, and S. Tarkoma, "Carat: Collaborative energy diagnosis for mobile devices," in *SenSys*, 2013.

[2] A. M. Townsend and M. L. Moss, "Telecommunications infrastructure in disasters: Preparing cities for crisis communications," Center for Catastrophe Preparedness and Response, NYC, 2005.

[3] C. James, , A. Popescu, and T. Underwood, "Impact of Hurricane Katrina on internet infrastructure," in *Renesys*, 2005.

[4] U. Weinsberg, Q. Li, N. Taft, A. Balachandran, V. Sekar, G. Iannaccone, and S. Seshan, "CARE: Context aware redundancy elimination in challenged networks," in *ACM HotNets*, 2012.

[5] D. Nistér and H. Stewénius, "Scalable recognition with a vocabulary tree," in *CVPR*, 2006.

[6] "Nepal earthquake death toll reaches 8,635, over 300 missing," Press Trust of India, 2015.

[7] H. Possegger, S. Sternig, T. Mauthner, P. M. Roth, and H. Bischof, "Robust real-time tracking of multiple objects by volumetric mass densities," in *CVPR*, 2013.

[8] "Battery life complaints causing operator headaches." `http://bit.ly/PI5Fnj`.

[9] A. Pathak, Y. C. Hu, and M. Zhang, "Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with Eprof," in *EuroSys*, 2012.

[10] "Android battery tool source code." `http://bit.ly/UPV77m`.

[11] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *CODES/ISSS*, 2010.

[12] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan, and D. Estrin, "Diversity in smartphone usage," in *MobiSys*, 2010.

[13] C. Yoon, D. Kim, W. Jung, C. Kang, and H. Cha, “AppScope: Application energy metering framework for Android smartphones using kernel activity monitoring,” in *USENIX ATC*, 2012.

[14] X. Ma, P. Huang, X. Jin, P. Wang, S. Park, D. Shen, Y. Zhou, L. K. Saul, and G. M. Voelker, “eDoctor: Automatically diagnosing abnormal battery drain issues on smartphones,” in *NSDI*, 2013.

[15] A. Shye, B. Scholbrock, and G. Memik, “Into the wild: Studying real user activity patterns to guide power optimizations for mobile architectures,” in *MICRO*, 2009.

[16] R. Mittal, A. Kansal, and R. Chandra, “Empowering developers to estimate app energy consumption,” in *Mobicom*, 2012.

[17] M. Dong and L. Zhong, “Self-constructive high-rate system energy modeling for battery-powered mobile systems,” in *MobiSys*, 2011.

[18] A. Carroll and G. Heiser, “An analysis of power consumption in a smartphone,” in *USENIX ATC*, 2010.

[19] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani, “Energy consumption in mobile phones: A measurement study and implications for network applications,” in *IMC*, 2009.

[20] S. Hao, D. Li, W. Halfond, and R. Govindan, “Estimating Android applications’ CPU energy usage via bytecode profiling,” in *GREENS*, 2012.

[21] A. Nikravesh, D. R. Choffnes, E. Katz-Bassett, Z. M. Mao, and M. Welsh, “Mobile network performance from user devices: A longitudinal, multidimensional analysis,” in *PAM*, 2014.

[22] “Carat - Android version.” `http://bit.ly/1h5EYGS`.

[23] “Monsoon power monitor.” `http://bit.ly/p6qNfY`.

[24] F. Qian, Z. Wang, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck, “Characterizing radio resource allocation for 3G networks,” in *IMC*, 2010.

[25] “Android source code.” `http://bit.ly/SQNHP7`.

[26] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, “RERAN: timing- and touch-sensitive record and replay for Android,” in *ICSE*, 2013.

[27] “Busybox tool set.” `http://www.busybox.net/`.

[28] “Building push applications for android.” `http://bit.ly/1gli69A`.

[29] “More photos shared daily on Snapchat than Facebook, Instagram combined.” `http://bit.ly/1sfog1M`, 2013.

[30] “Ars Technica news article.” `http://goo.gl/KIjO1e`.

[31] T. Anthony and M. Moss, “Preparing cities for crisis communication,” in *CCPR*, 2005.

[32] A. Hughes, L. Palen, J. sutton, S. Liu, and S. Veweg, “Site-seeing in disaster: An examination of on-line social convergence,” in *ISCRAM*, 2008.

[33] T. Preis, H. Moat, S. Bishop, P. Treleavan, and E. Stanley, “Quantifying the digital traces of Hurricane Sandy on Flickr,” in *SREP*, 2013.

[34] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, “ORB: An efficient alternative to SIFT or SURF,” in *ICCV*, 2011.

[35] O. Chum, J. Philbin, M. Isard, and A. Zisserman, “Scalable near identical image and shot detection,” in *CIVR*, 2007.

[36] Y. Zhang, Z. Jia, and T. Chen, “Image retrieval with geometry-preserving visual phrases,” in *CVPR*, 2011.

[37] F. Liu, B. Li, L. Zhong, B. Li, H. Jin, and X. Liao, “Flash crowd in P2P live streaming systems: Fundamental characteristics and design implications,” in *IEEE Trans. Parallel Distrib. Syst.*, 2012.

[38] A. Koehl and H. Wang, “Surviving a search engine overload,” in *WWW*, 2012.

[39] Q. Tran, K. Nguyen, K. E, and Y. S, “Tree-based disaster recovery multihop access network,” in *APCC*, 2013.

[40] S. M. George, W. Zhou, H. Chenji, M. Won, Y. O. Lee, A. Pazarloglou, R. Stoleru, and T. A, “Topics in situation management distressnet: A wireless ad hoc and sensor network architecture for situation management in disaster response,” in *IEEE Communications Magazine*, 2010.

[41] Y. Zhang, Y. Wu, and G. Yang, “Droplet: A distributed solution of data deduplication.,” in *GRID*, 2012.

[42] P. Riteau, C. Morin, and T. Priol, “Shrinker: Improving live migration of virtual clusters over WANs with distributed data deduplication and content-based addressing,” in *Euro-Par*, 2011.

[43] X. Zhang, Z. Huo, J. Ma, and D. Meng, “Exploiting data deduplication to accelerate live virtual machine migration,” in *CLUSTER*, 2010.

[44] M. Douze, H. Jégou, H. Sandhawalia, L. Amsaleg, and C. Schmid, “Evaluation of GIST descriptors for web-scale image search,” in *CIVR*, 2009.

[45] D. G. Lowe, “Distinctive image features from scale-invariant keypoints,” *Int. J. Comput. Vision*, 2004.

[46] J. Sivic and A. Zisserman, “Video Google: A text retrieval approach to object matching in videos,” in *ICCV*, 2003.

[47] O. Chum, J. Philbin, and A. Zisserman, "Near duplicate image detection: min-hash and tf-idf weighting," in *BMVC*, 2008.

[48] Z. Wengang, L. Yijuan, L. Houqiang, S. Yibing, and T. Qi, "Spatial coding for large scale partial-duplicate web image search," in *MM*, 2010.

[49] A. Sarkar, P. Ghosh, E. Moxley, and B. S. Manjunath, "Video fingerprinting: features for duplicate and similar video detection and query-based video retrieval," in *SPIE*, 2008.

[50] O. Miksik and K. Mikolajczyk, "Evaluation of local detectors and descriptors for fast feature matching," in *ICPR*, 2012.

[51] J. Heinly, E. Dunn, and J. Frahm, "Comparative evaluation of binary features," in *ECCV*, 2012.

[52] C. Grana, D. Borghesani, M. Manfredi, and R. Cucchiara, "A fast approach for integrating ORB descriptors in the Bag of Words model," in *IS&T/SPIE*, 2013.

[53] J. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*, 1967.

[54] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel, "Finding a needle in Haystack: Facebook's photo storage," in *OSDI*, 2010.

[55] "OpenCV library." `http://www.opencv.org`.

[56] M. Muja and D. G. Lowe, "Fast approximate nearest neighbors with automatic algorithm configuration," in *VISSAPP*, 2009.

[57] "Image Magick." `http://www.imagemagick.org`.

[58] "Apple iPhone 5s reviews." `http://bit.ly/TXPtqr`, 2013.

[59] Y. Yang, H.-Y. Ha, F. Fleites, S.-C. Chen, and S. Luis, "Hierarchical disaster image classification for situation report enhancement," in *IRI*, 2011.

[60] K. Fall, G. Iannaccone, J. Kannan, F. Silveira, and N. Taft, "A disruption-tolerant architecture for secure and efficient disaster response communications," in *ISCRAM*, 2010.

[61] S. Shirdhonkar and D. W. Jacobs, "Approximate earth movers distance in linear time," in *CVPR*, 2008.

[62] "The federal response to hurricane katrina: Lessons learned," Office of the Assistant to the President for Homeland Security and Couter Terrorism, 2006.

[63] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *CVPR*, 2005.

[64] P. Dollar, Z. Tu, P. Perona, and S. Belongie, "Integral channel features," in *BMVC*, 2009.

[65] N. Dalal and B. Triggs, "Human detection based on a probabilistic assembly of robust part detectors," in *ECCV*, 2004.

[66] W. R. Schwartz, A. Kembhavi, D. Harwood, and L. S. Davis, "Human detection using partial least squares analysis," in *ICCV*, 2009.

[67] Q. Zhu, S. Avidan, M.-C. Yeh, and K.-T. Cheng, "Fast human detection using a cascade of histograms of oriented gradients," in *CVPR*, 2006.

[68] P. Dollar, R. Appel, S. Belongie, , and P. Perona, "Fast feature pyramids for object detection," in *PAMI*, 2014.

[69] R. Benenson, M. Mathias, R. Timofte, and L. V. Gool, "Pedestrian detection at 100 frames per second," in *CVPR*, 2012.

[70] S. Zhang, R. Benenson, and B. Schiele, "Filtered channel features for pedestrian detection," in *CVPR*, 2015.

[71] X. Dai and S. Payandeh, "Geometry-based object association and consistent labeling in multi-camera surveillance," in *IEEE CAS*, 2013.

[72] K. Nummiaro, E. Koller-Meier, T. Svoboda, D. Roth, and L. V. Gool, "Color-based object tracking in multi-camera environments," in *DAGM*, 2003.

[73] M. Tan and S. Ranganath, "Multi-camera people tracking using bayesian networks," in *ICICS*, 2003.

[74] H. Iwaki, G. Srivastava, A. Kosaka, J. Park, and A. Kak, "A novel evidence accumulation framework for robust multi-camera person detection," in *ICDCS*, 2008.

[75] C. Zeng and H. Ma, "Human detection using multi-camera and 3D scene knowledge," in *ICIP*, 2011.

[76] H. Possegger, T. Mauthner, P. M. Roth, and H. Bischof, "Occlusion geodesics for online multi-object tracking," in *CVPR*, 2014.

[77] T. Dao, A. K. Roy-Chowdhury, H. V. Madhyastha, S. V. Krishnamurthy, and T. La Porta, "Managing redundant content in bandwidth constrained wireless networks," in *CoNEXT*, 2014.

[78] T. Zhang, A. Chowdhery, P. V. Bahl, K. Jamieson, and S. Banerjee, "The design and implementation of a wireless video surveillance system," in *ACM MobiCom*, 2015.

[79] W. R. Heinzelman, A. Chandrakasan, and H. Balakrishnan, "Energy-efficient communication protocol for wireless microsensor networks," in *HICSS*, 2000.

[80] Y. Wang, Q. Wang, Z. Jin, and N. Saxena, "Improved cluster heads selection method in wireless sensor networks," in *IEEE GreenCom*, 2010.

[81] H. Kellerer, U. Pferschy, and D. Pisinger, “Knapsack problems,” 2010.

[82] P. Viola and M. Jones, “Robust real-time object detection,” in *International Journal of Computer Vision*, 2001.

[83] R. Y. TSai, “A versatile camera calibration technique for high-accuracy 3D machine vision metrology using off-the-shelf tv cameras and lenses,” in *IEEE Journal of Robotics and Automation*, 1987.

[84] S. LaValle, “Planning algorithms,” ch. 3, Cambridge University Press, 2006.

[85] E. Elhamifar and R. Vidal, “Distributed calibration of camera sensor networks,” in *ICDSC*, 2009.

[86] Z. Zhang, T. Tan, K. Huang, and Y. Wang, “Practical camera calibration from moving objects for traffic scene surveillance,” *IEEE TCSVT*, 2013.

[87] M. Hirzer, P. M. Roth, M. Koestinger, and H. Bischof, “Relaxed pairwise learned metric for person re-identification,” in *ECCV*, 2012.

[88] A. Porebski, N. Vandenbroucke, and D. Hamad, “LBP histogram selection for supervised color texture classification,” in *ICIP*, 2013.

[89] S. Xiang, F. Nie, and C. Zhang, “Learning a Mahalanobis distance metric for data clustering and classification,” in *PR*, 2008.

[90] “iPerf - network bandwidth measurement tool.” `http://iperf.fr/`.

[91] C. P. Gomes and R. Williams, “Approximation algorithms,” in *Search Methodologies*, ch. 18, Springer, 2005.

[92] P. Boutin, “New apps to post videos with ease,” 2011.

[93] “CMU Pixy camera.” `http://bit.ly/1UIln1O`.

[94] P. Dollar, S. Belongie, and P. Perona, “The fastest pedestrian detector in the west,” in *BMVC*, 2010.

[95] “CMUcam: Open source programmable embedded color vision sensors.” `http://www.cmucam.org`.

[96] B. Gong, Y. Shi, F. Sha, and K. Grauman, “Geodesic flow kernel for unsupervised domain adaptation,” in *CVPR*, 2012.

[97] P. F. Felzenszwalb, R. B. Girshick, D. McAllester, and D. Ramanan, “Object detection with discriminatively trained part based models,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2010.

[98] J. Wu, C. Geyer, and J. M. Rehg, “Real-time human detection using contour cues,” in *ICRA*, 2011.

[99] R. Gopalan, R. Li, and R. Chellappa, "Domain adaptation for object recognition: An unsupervised approach," in *ICCV*, 2011.

[100] X. Yong, D. Feng, Z. Rongchun, and M. Petrou, "Learning-based algorithm selection for image segmentation," *Pattern Recogn. Lett.*, vol. 26, no. 8, 2005.

[101] O. Mac Aodha, G. J. Brostow, and M. Pollefeys, "Segmenting video into classes of algorithm-suitability," in *CVPR*, 2010.

[102] S. Zhang, Q. Zhu, and A. K. Roy-Chowdhury, "Adaptive algorithm selection, with applications in pedestrian detection," in *ICIP*, 2016.

[103] Y. Li and T. Zhang, "Reducing dram image data access energy consumption in video processing," in *IEEE Transactions on Multimedia*, 2012.

[104] *An Experimental Study on Energy Consumption of Video Encryption for Mobile Handheld Devices*, 2005.

[105] S. Mohapatra, R. Cornea, N. Dutt, A. Nicolau, and N. Venkatasubramanian, "Integrated power management for video streaming to mobile handheld devices," in *Proceedings of the eleventh ACM international conference on Multimedia*, 2003.

[106] M. Tamai, T. Sun, K. Yasumoto, N. Shibata, and M. Ito, "Energy-aware video streaming with qos control for portable computing devices," in *Proceedings of the 14th international workshop on Network and operating systems support for digital audio and video*, 2004.

[107] L. Zhou, R. Q. Hu, Y. Qian, and H.-H. Chen, "Energy-spectrum efficiency tradeoff for video streaming over mobile ad hoc networks," *IEEE Journal on Selected Areas in Communications*, vol. 31, no. 5, pp. 981–991, 2013.

[108] B. Schiilkopf, "The kernel trick for distances," *Advances in neural information processing systems*, vol. 13, pp. 301–307, 2001.

[109] J. M. Phillips and S. Venkatasubramanian, "A gentle introduction to the kernel distance," *arXiv preprint arXiv:1103.1625*, 2011.

[110] "Precision and recall." `https://en.wikipedia.org/wiki/Precision_and_recall`.

[111] E. Vincent and R. Laganiére, "Detecting planar homographies in an image pair," in *Proceedings of the 2nd International Symposium on Image and Signal Processing and Analysis*, 2001.

[112] H. Bay, T. Tuytelaars, and L. Van Gool, "Surf: Speeded up robust features," in *ECCV*, 2006.

[113] J. Berclaz, F. Fleuret, E. Turetken, and P. Fua, "Multiple Object Tracking using K-Shortest Paths Optimization," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2011.

[114] N. Vallina-Rodriguez, P. Hui, J. Crowcroft, and A. Rice, "Exhausting battery statistics: understanding the energy demands on mobile handsets," in *MobiHeld*, 2010.

[115] M. Chuah, W. Luo, and X. Zhang, "Impacts of inactivity timer values on umts system capacity," in *WCNC*, 2002.

[116] "Configuration of fast dormancy in release 8." 3GPP discussion and decision notes RP-090960, 2009.

[117] A. M. Corley, "Making smartphones power-smarter." IEEE Spectrum, 2010.

[118] "Enable GPS programatically like Tasker." `http://bit.ly/1kuNLCT`.

[119] "We need better battery life and more storage space before 1080p displays arrive." `http://bit.ly/S5yUEa`.

[120] "Samsung commits to increasing smartphone battery life in 2012, hopes for all-day use." `http://bit.ly/TuhumD`.

[121] "3gpp specification - technical specification, section 8.5." `https://goo.gl/7rXjcw`.

[122] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck, "A close examination of performance and power characteristics of 4G LTE networks," in *MobiSys*, 2012.

[123] S. Deng and H. Balakrishnan, "Traffic-Aware Techniques to Reduce 3G/LTE Wireless Energy Consumption," in *CoNEXT*, 2012.

[124] A. Pathak, Y. C. Hu, and M. Zhang, "Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices," in *HotNets*, 2011.

[125] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff, "What is keeping my phone awake? characterizing and detecting no-sleep energy bugs in smartphone apps," in *MobiSys*, 2012.

[126] J. Han, M. Kamber, and J. Pei, *Data Mining - Concepts and Techniques, 3rd ed.* Morgan Kaufmann, 2012.

[127] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck, "A close examination of performance and power characteristics of 4G LTE networks," in *MobiSys*, 2012.

[128] F. Qian, Z. Wang, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck, "Profiling resource usage for mobile applications: A cross-layer approach," in *MobiSys*, 2011.

[129] C. Lee, J. Yeh, and J. Chen, "Impact of inactivity timer on energy consumption in wcdma and cdma2000," in *WTS*, 2004.

[130] F. Qian, Z. Wang, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck, "Top: Tail optimization protocol for cellular radio resource allocation," in *ICNP*, 2010.

[131] P. K. Athivarapu, R. Bhagwan, S. Guha, V. Navda, R. Ramjee, D. Arora, V. N. Padmanabhan, and G. Varghese, "Radiojockey: mining program execution to optimize cellular radio usage," in *Mobicom*, 2012.

[132] F. Qian, Z. Wang, Y. Gao, J. Huang, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck, "Periodic transfers in mobile applications: network-wide origin, impact, and optimization," in *WWW*, 2012.

[133] J. Huang, F. Qian, Z. M. Mao, S. Sen, and O. Spatscheck, "Screen-off traffic characterization and optimization in 3g/4g networks," in *IMC*, 2012.

[134] T. Dao, I. Singh, H. V. Madhyastha, S. V. Krishnamurthy, G. Cao, and P. Mohapatra, "TIDE a user-centric tool for identifying energy hungry applications on smartphones," in *ICDCS*, 2015.

[135] W.LeFebvre, "CNN.com: Facing a world crisis," *Invited talk at LISA'01.*

[136] O. Chum, M. Perdoch, and J. Matas, "Geometric min-Hashing: Finding a (thick) needle in a Haystack," in *CIVR*, 2009.

[137] D. Nister and H. Stewenius, "Scalable recognition with a vocabulary tree," in *CVPR*, 2006.

[138] J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman, "Lost in quantization: Improving particular object retrieval in large scale image databases," in *CVPR*, 2008.

[139] S. Zhang, Q. Huang, G. Hua, S. Jiang, W. Gao, and Q. Tian, "Building contextual visual vocabulary for large-scale image applications," in *MM*, 2010.

[140] H. Jegou, M. Douze, C. Schmid, and P. Perez, "Aggregating local descriptors into a compact image representation," in *CVPR*, 2010.

[141] X. Zhang, Z. Li, L. Zhang, W. Ma, and H.-Y. Shum, "Efficient indexing for large scale visual search," in *ICCV*, 2009.

[142] M. Perdoch, O. Chum, and J. Matas, "Efficient representation of local geometry for large scale object retrieval," in *CVPR*, 2009.

[143] J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman, "Object retrieval with large vocabularies and fast spatial matching," in *CVPR*, 2007.

[144] Z. Wu, Q. Ke, M. Isard, and J. Sun, "Bundling features for large scale partial-duplicate web image search," in *CVPR*, 2009.

[145] D. Xu, T. J. Cham, S. Yan, L. Duan, and S.-F. Chang, "Near duplicate identification with spatially aligned pyramid matching," in *CVPR*, 2008.

[146] W. Zhou and D. yi Wang, "A dynamic-resource-allocation based flash crowd mitigation algorithm for Video-on-Demand network," in *ICCSIT*, 2010.

[147] C.-H. Chi, S. Xu, F. Li, and K.-Y. Lam, “Selection policy of rescue servers based on workload characterization of flash crowd,” in *SKG*, 2010.

[148] H. Wu, K. Xu, M. Zhou, A. Wong, J. Li, and Z. Li, “Multiple-tree topology construction scheme for P2P live streaming systems under flash crowds,” in *WCNC*, 2013.

[149] J. Brodkin, “Why your smart device can’t get WiFi in the home team’s stadium.” `http://arstechnica.com/features/2012/08/why-your-smart-device-cant-get-wifi-in-the-home-teams-stadium/`, 2012.

[150] K. Srinivasan, T. Bisson, G. Goodson, and K. Voruganti, “iDedup: Latency-aware, inline data deduplication for primary storage,” in *FAST*, 2012.

[151] “Mobile apps overtake pc internet usage in U.S..” `http://money.cnn.com/2014/02/28/technology/mobile/mobile-apps-internet/`, 2014.

[152] C. F. Barnes, H. Fritz, and J. Yoo, “Hurricane disaster assessments with image-driven data mining in high-resolution satellite imagery,” in *IEEE Transactions on Geoscience and Remote Sensing*, 2007.

[153] R. Federica and I. Mikio, “Learning from megadisasters: Lessons from the great east japan earthquake,” in *Emergency Communication*, World Bank Publications, 2015.

[154] S. Khan and M. Shah, “Consistent labeling of tracked objects in multiple cameras with overlapping fields of view,” in *IEEE TPAMI*, 2003.

[155] Q. Zhou and J. Aggarwa, “Object tracking in an outdoor environment using fusion of features and cameras,” in *IEEE TPAMI*, 2006.

[156] S. Calderara, R. Cucchiara, and A. Prati, “Bayesian-competitive consistent labeling for people surveillance,” in *IEEE TPAMI*, 2008.

[157] L. Bazzani, M. Cristani, and V. Murino, “Symmetry-driven accumulation of local features for human characterization and re-identification,” in *Computer Vision and Image Understanding 2008*, 2012.

[158] N. Martinel, A. Das, C. Micheloni, and A. K. Roy-Chowdhury, “Re-identification in the function space of feature warps,” in *PAMI*, 2015.