CS237 Project Report: Edge Computing in TIPPERS

Janus Varmarken
jvarmark@uci.edu

Nishanth Devarajan
devarajn@uci.edu

Victor Hsiao
vwhsiao@uci.edu

June 2018

1 Introduction and Objective

TIPPERS[1] [6] is a research project that investigates Internet of Things (IoT) based smart spaces, but differentiates itself from similar projects by emphasizing the privacy of its users. Specifically, TIPPERS provides the user with the option to specify policies for what of their data (collected by an extensive set of different types of sensors deployed throughout Donald Bren Hall) may be used when servicing queries about them (e.g., a query about the user's current location). Most of the sensors that partake in the TIPPERS system perform periodic readings, all of which are uploaded to—and (later) served to clients by—the central TIPPERS backend. Given the multitude of sensors and their high sampling rates, the current scheme produces a vast amount of (redundant) data, which in turn results in long service times for data queries and excessive, unnecessary utilization of network bandwidth. Naturally, if TIPPERS is extended to encompass the entire UCI campus as planned for, this problem of data redundancy and excessive network bandwidth utilization becomes even more prevalent.

The primary objective of our project lies in addressing the problem described in the previous paragraph, i.e., our goal is to propose a solution that reduces the large amount of data that TIPPERS is currently accumulating as a result of the high sampling rates of the sensors. Currently, all sensor data is being recorded at the end devices, each with their own sampling rate, and sent directly to the TIPPERS backend. This is done, in most cases, without taking into consideration if the newly sampled data is meaningful or not (i.e., if it is redundant) when viewed in conjunction with past samples. The surveillance camera system is a good example. At the moment, the cameras take a picture roughly every two seconds and send the picture to the TIPPERS backend. Even if a camera has taken a picture of the same empty hallway for the past 100 samples, the new sample (yet another picture of an empty hallway) is still sent to, and stored in, the TIPPERS backend. A simple observation shows that there is potential for substantially reducing the amount of data uploaded to the TIPPERS backend in this scenario: if the camera (or a computational device in its vicinity) can detect that two successive images are logically identical (e.g., two pictures of an empty hallway), there is no need to upload both to the TIPPERS backend, provided that the data queries at the backend are updated to "understand" that a temporal gap between successive recorded sensor samples (in this case images) means that there was no change in the sampled data during that gap.

Motivated by the example above, we propose and implement a software framework that can perform sensor sample filtering at the edge of the network. Since the problem of redundant sensor data is a general one (i.e., it is *not* a problem that is unique to the surveillance camara system), we take special care to ensure that our software framework remains completely oblivious to the type of sensor being sampled, thereby making it easily applicable to *any* sensor type. As a proof of concept, we apply our framework to the example scenario discussed above (the surveillance camera system) and show that it can potentially reduce the amount of data uploaded to the TIPPERS backend over night by a single camera from 1.3 GB to less than 100 KB.

The remainder of this paper is structured as follows. Section 2 provides context by defining the field of study, i.e., edge computing. Section 3 highlights a couple of related works. Section 4 describes the implementation of our software framework for sensor sample filtering at the edge of the network. Section 5 evaluates the correctness and potential benefits of our proof-of-concept implementation for the surveillance camera system. Finally, Section 6 concludes the paper.

2 Definition of the Field of Study

Edge computing—also referred to as *fog computing*[2]—is a computing paradigm in which (*some*) computational services move closer to the extremities (the edge) of the network. Specifically, as noted by Bonomi et

[1] See `http://tippersweb.ics.uci.edu/`.

[2] The two terms will be used interchangeably throughout this document.

al. [2], who coined the term fog computing, fog computing is not intended as a successor of cloud computing. Instead, fog computing *augments* cloud computing in order to enable a new breed of applications[3]—especially those that have strict real-time requirements for some computations, yet rely on the cloud for functionality such as persistent storage and/or big data analytics. For example, neighboring smart traffic lights could communicate directly with one another to create waves of green lights (a decision made in the fog that has real-time requirements), and each individual smart traffic light could continuously upload (aggregate) traffic information to the cloud which could in turn be in charge of calculating large scale analytics such as (hourly) city-wide heat-maps of traffic congestion.

Vaquero and Rodero-Merino [14] argue that confusion about the meaning of a term often arises in the information and communication technologies (ICT) community due to the lack of a clear definition from the onset. They proceed to argue that fog computing is even more likely to cause such confusion as it is not constrained to a single technological area, but rather is the (natural) product of a set of converging technological trends. In continuation of this argument, they deem the definition in [2] insufficient, claiming that it fails to convey the novelty of fog computing as it leaves the reader with the impression that fog computing is merely a (trivial) extension of cloud computing. They ultimately present their own definition, arguing that it is more holistic as it takes into account a set of features identified as key ingredients of the fog (e.g., ubiquity and support for device cooperation). Yi et al. [15] praise this definition for being integrative as it is based on a careful analysis of what technologies enable fog computing, and what challenges fog computing must overcome, but argue that it fails to capture the unique relationship between the fog and the cloud and therefore propose their own definition, which unifies the comprehensive definition in [14] and the relation to the cloud as captured in the definition by Bonomi et al. [2]:

> "Fog computing is a geographically distributed computing architecture with a resource pool [that] consists of one or more ubiquitously connected heterogeneous devices (including edge devices) at the edge of [the] network and not exclusively seamlessly backed by cloud services, [in order] to collaboratively provide elastic computation, storage and communication (and many other new services and tasks) in isolated environments to a large scale of clients in proximity."

We argue that our proposed software framework fits nicely under this definition. The cameras—and potentially other sensors later on (heterogeneity aspect)—upload their data (pictures) to a node in their physical vicinity (edge of the network aspect) which in turn computes if the data should be discarded or uploaded to the central TIPPERS backend server (cloud interplay aspect).

3 Related Work

In this section, we present two related works that motivate the direction our class project moves in, and a direction the TIPPERS project would highly benefit from, looking forward: (1) GigaSight [11], a hybrid cloud architecture that decentralizes cloud computing infrastructure to the edge, significantly reducing network bandwidth utilization; and (2) device-to-device communication (D2D) which is the concept of a predictive framework that rests on the notion that information demand patterns by users are to an extent predictable, and where such predictability is exploited to minimize peak load at the server by precaching desired information at the edge.

3.1 GigaSight

Similar to the implementation goals that we envision for our class project, [11] presents GigaSight—a hybrid cloud architecture that uses a decentralized cloud computing infrastructure. GigaSight employs what the

[3]In fact, the choice of name neatly captures the interplay with the cloud as a fog is simply a cloud that is close to the ground [2].

authors call *cloudlets*. A cloudlet is essentially a small scale data center—a "datacenter in a box"—that resides close to the edge of the network, hence forming an intermediary between mobile devices and the cloud. They specifically deal with video data, as video streaming services such as YouTube places heavy bandwidth demands on the network—for example, YouTube's recommended upload rate of 8.5 Mbps effectively means that a million concurrent uploads would require 8.5 Tbytes per second. Satyanarayanan et al. implement the cloudlet as a VM capable of performing required operations on the live video data with as much as possibly reduced computational complexity. They describe a GigaSight prototype that uses a Python-based implementation of an image categorization and segmentation algorithm by Jamie Shotton et al [12]. Furthermore, they provide an analysis that studies the optimal placement of cloudlets, taking into account the environment, i.e connectivity (3G, 4G, 5G etc), and show the potential benefits of using such an architecture with not only video cameras as sensors (as case studied here), but any content/service type in the IoT that requires a high data rate.

GigaSight also helps with processing the data to gather useful insights. In particular, the authors argue that the richness of content and the possibility of unanticipated value distinguishes video from simpler sensor data, e.g., surveillance videos were crucial for discovering the Boston Marathon bombers in 2013. In addition, GigaSight aids in preserving the privacy of user data (what they call *denaturing*):

> "An important type of 'analytics' supported on cloudlets is automated modification of video streams to preserve privacy. For example, this might involve editing out frames or blurring individual objects within frames. What needs to be removed or altered is highly specific to the owner of a video stream, but no user has time to go through and manually edit video captured on a continuous basis. Denaturing must strike a balance between privacy and value. At one extreme is a blank video: perfect privacy but zero value. At the other extreme is the original video at its capture resolution and frame rate. This has the highest value for potential customers, but it also incurs the highest exposure of privacy. Where to strike the balance is a difficult question that is best answered individually, by each user. This decision will most probably be context-sensitive." [11].

In fact, privacy at the edge is a well researched niche in the field of edge computing with immense benefits [4], [3] and [10]. In fact, for TIPPERS especially - providing many crowd sourcing applications must handle user privacy satisfactorily, the lack of which would signify severe privacy breach implications otherwise [13].

During our meetings with Roberto Yus, a PhD student working on the TIPPERS project, a continued direction along these lines—namely to implement blurring of users' faces based on their privacy preferences—was discussed. And we believe that such continued work would set the basis to enhance the privacy preserving aspects at the edge, for the TIPPERS system.

3.2 D2D

At the moment, one of the ideas for the future of the 5G standard that is currently being developed is the idea of having IoT compatibility and Device-to-Device (D2D) communications. D2D is the concept of 5G devices communicating with nearby devices through a local link to exchange information and data. It is through this D2D concept that Bastug et al. see as a way to achieve what they call *proactive networks* [1]. By combining small base stations with high storage capacities and utilizing D2D communications and the corresponding storage space on the devices, they believe that it is possible to proactively deliver content by analyzing and predicting usage patterns:

> "The predictive framework rests on the notion that information demand patterns of mobile users are, to a certain extent, predictable. Such predictability can be exploited to minimize the peak load of cellular networks by proactively pre-caching desired information to selected users before they actually request it. [...] That is, when the proactive network serves users' requests before their deadlines, the corresponding data is stored in the user device, and when the request is

actually initiated, the information is pulled out directly from the cached memory instead of accessing the wireless network."[1].

With the utilization of small base stations with large storage capacities, Bastug et al. hope to achieve lower latencies by having the edge base stations predict and proactively cache data on users' devices so that there is no need to traverse the network to retrieve the file. Similarly, once the base station receives a user-requested file, it will cache it for future requests on the off-chance the file was not predicted to be accessed by a user.

Bastug et al. continue to add on and say that by leveraging social networks and the information that they provide based on social circles, interests, and general user data, it is possible to predict a user's future data needs. For example, if the analysis of the user's social network shows that they will be attending a certain event, the network can proactively load the event webpage (e.g., for tickets and/or the schedule) onto the user's device and present this cached version if the user does indeed choose to visit the site. In addition, in order to promote D2D and its potential for reducing latency, the authors suggest that the base station can use social graph analysis to calculate a so-called *influencer* for a file and redirect other users to retrieve the file through D2D with the influencer. The influencer is calculated based on a history of users' encounters and file requests. Essentially, the idea is that you are likely to be in proximity of your friends, hence the latency involved in retrieving the file from your friend using D2D should be considerably lower than retrieving the file from the core of the network.

4 Project Implementation

The problem of redundant sensor data is a general one, i.e., it is *not* a problem that is unique to the images captured by the surveillance cameras. For example, temperature sensors may measure the same temperature for many successive samples, and Wi-Fi access points may observe the same set of connected MAC addresses for an extended period of time. As such, sensor sample filtering may be relevant to a multitude of sensors other than just surveillance cameras. To accommodate easy application of our approach in the context of other sensors, we have put significant effort into providing a generic implementation that is completely oblivious to the specific type of sensor data being filtered. This generic framework is described in Section 4.1. Section 4.2 describes how the generic framework is used in the context of the surveillance cameras for discarding redundant images.

4.1 Generic Framework

Implemented in Java, we achieve this generality by specifying an interface, `SampleProvider<S>` (see listing 1), and an abstract class, `AbstractSampleHandler<S>` (see listing 2). In both cases, the type parameter, `S`, is used to designate a class that models a sensor reading. `S` is not bounded, it can be anything ranging from a simple `Double` that models a temperature reading to a complex type that also carries metadata such as a timestamp, the sensor's ID etc. The `SampleProvider` interface (listing 1) provides third-party code with means for specifying that a particular class models a sensor that can be sampled. The external code simply has to provide an implementation of `sample()`, which, as the name suggests, samples the sensor and returns the data read from the sensor. The use of a common interface (i.e., `SampleProvider`) allows `AbstractSampleHandler` (and its subclasses) to be implemented in terms of the interface and thereby remain general-purpose.

`AbstractSampleHandler` (listing 2) is the base class for implementing sensor sample filtering: when filtering is to be implemented for a new sensor type, the third-party code simply subclasses `AbstractSampleHandler` (or one of its utility subclasses[4]). In doing so, the external code is required to implement the two abstract

[4]For example, we provide a convenience subclass, `AbstractPeriodicSampleHandler`, that samples a sensor and processes the sampled data at a fixed, configurable rate (e.g., retrieving and processing a sample every other second).

methods of `AbstractSampleHandler`, i.e., `shouldIncludeSample(S)` and `uploadSample(S)`. The former is invoked by `AbstractSampleHandler` whenever a new sensor sample is retrieved (passing the sample as the argument) and is `AbstractSampleHandler`'s way of delegating the sample inclusion decision to its subclass. This pattern allows the subclass to implement any arbitrary decision logic, yet keeps the base class in the loop by virtue of the method's boolean return value, allowing it to perform further actions based on the subclass' inclusion decision, such as invoking `uploadSample(S)` to trigger the code that uploads the sample to the backend. In order to accommodate easy implementation of filtering decisions based on past sensor samples, `AbstractSampleHandler` maintains, and exposes to its subclasses, a fixed-size cache (`mSampleCache`) of the most recent samples that were uploaded to the backend server.

Listing 1: The `SampleProvider` interface.

```
public interface SampleProvider<S> {
    /**
     * Sample the sensor (retrieve a new reading).
     * @return The new sample (reading) or {@code null} if no data is available or an error occurred.
     */
    S sample();
}
```

Listing 2: The `AbstractSampleHandler` class.

```
public abstract class AbstractSampleHandler<S> {

    protected final SampleProvider<S> mSampleProvider;

    protected final List<S> mSampleCache = new ArrayList<S>() {
        @Override
        public boolean add(S s) {
            boolean added = super.add(s);
            if (size() > AbstractSampleHandler.this.mSampleCacheSize) {
                removeRange(0, size() - AbstractSampleHandler.this.mSampleCacheSize);
            }
            return added;
        }
    };

    protected final int mSampleCacheSize;

    public AbstractSampleHandler(SampleProvider<S> sampleProvider, int sampleCacheSize) { /* Assign member variables... */ }

    public void sampleAndUpload() {
        S sample = mSampleProvider.sample();
        if (sample != null && shouldIncludeSample(sample)) {
            // Sample is valid and should be included. Attempt upload of sample to TIPPERS backend db.
            boolean uploaded = uploadSample(sample);
            if (uploaded) {
                // Only cache sample if sample was persisted at the backend.
                mSampleCache.add(sample);
            }
        }
    }

    abstract protected boolean shouldIncludeSample(S sample);

    abstract protected boolean uploadSample(S sample);
}
```

4.2 Applying the Generic Framework to Surveillance Cameras

We apply the generic framework discussed in Section 4.1 to images captured by the surveillance cameras in Donald Bren Hall by implementing `CameraSampleHandler` as a subclass of `AbstractPeriodicSampleHandler`, specifying a sample rate of one image every two seconds. `CameraSampleHandler` is passed an implementation of `SampleProvider`, called `CameraRestClient`, whose implementation of `sample()` invokes the camera's

REST API to capture and download an image and ultimately returns the local path to the downloaded image.

Our image filtering logic is based on the idea that one of two successive images is redundant if both images essentially reflect the same *scene*. A scene is defined by the *ordered* list of objects (e.g., people and things) present in an image. Our implementation of `shouldIncludeSample` in `CameraSampleHandler` (see listing 3) therefore performs object detection on the newly sampled image and compares the list of identified objects with the list of objects present in the image that was most recently uploaded to the backend (extracted from `mSampleCache` of `AbstractSampleHandler`). The object detection is facilitated by creating a Java wrapper around YOLOv3 [9], a neural network based object detection system. The wrapper executes YOLOv3 in a separate process and parses its output, constructing a list of objects detected by YOLOv3 along with YOLOv3's confidence in its detection of each separate object (output as an integer percentage). This confidence is also considered when establishing if two scenes are identical. In fact, in our current implementation, two images (scenes) are only considered identical if the same set of objects are present—in the same order, and with the same confidence—in both images. The reasoning behind this equality condition is that we want two scenes to be considered *different* if they include the same set of objects, but in different in-image locations, so as to capture moving objects (e.g., a person moving towards the camera from the end of the hallway). The ordering of objects, as discussed above, is only able to partially capture this aspect—for example, it will fail if there is only one object present, or if the the set of detected objects move in synchrony (are all shifted in the same direction). Our assumption is that when an object moves, YOLOv3's confidence in its detection of said object will vary slightly from one image to the next[5], and hence the algorithm will appropriately select both images for inclusion. We acknowledge that relying on the confidence to be *exactly* the same may cause two images with negligible differences to both be selected for inclusion. As such, we suggest that one settles on a level of slack (in terms of how much the confidence in the detection of an object may vary across two scenes) that is appropriate for the specific use case.

Listing 3: The `CameraSampleHandler` class.

```
public class CameraSampleHandler extends AbstractPeriodicSampleHandler<String> {
    // Our Java YOLOv3 wrapper.
    private final DarknetProcess mDarknetProcess;

    /* ...some fields and constructor omitted */

    @Override
    protected boolean shouldIncludeSample(String sample) {
        String previousImg = null;
        synchronized (mSampleCache) {
            // Get the location of the most-recently cached image, if any.
            if (mSampleCache.size() > 0) { previousImg = mSampleCache.get(mSampleCache.size()-1); }
        }
        // Always upload if no previously cached image.
        if (previousImg == null) { return true; }
        try {
            List<DarknetProcess.DetectedObject> oldScene = mDarknetProcess.exec(previousImg);
            List<DarknetProcess.DetectedObject> newScene = mDarknetProcess.exec(sample);
            // Scenes definitely differ if there is a different number of objects in the two.
            if (oldScene.size() != newScene.size()) { return true; }
            int matchedObjects = 0;
            for (int i = 0; i < oldScene.size(); i++) {
                if (oldScene.get(i).equals(newScene.get(i))) { matchedObjects++; }
            }
            boolean identical = matchedObjects == oldScene.size();
            // Only upload this image if there is a discrepancy between the objects of the new and the old scene.
            return !identical;
        } catch (IOException|InterruptedException e) {
            // Always upload on error -- TODO: better strategy?
            return true;
        }
    }

    /* ...image upload code omitted */
}
```

[5]For example, YOLOv3 is likely to have more confidence in its detection of a person if the entire body of that person is in the picture than if only the person's (side-facing) torso is in the picture.

5 Evaluation and Results

We evaluate our system in terms of its correctness, its potential benefits (i.e., its effect on the storage needs of the TIPPERS backend), and its hardware-needs (i.e., the computational power necessary at the edge).

5.1 Correctness and Benefits

In order to verify the correctness of our inclusion logic, we created a sample set of 30 images pulled from a camera observing a hallway at the second floor of DBH. The sample set consists of nine images of an empty hallway followed by one image with a person present, then another nine images of an empty hallway followed by one image with a person present, etc. The images are fed to the inclusion algorithm in the described order. As per the inclusion logic described in Section 4.2, the algorithm should then select six pictures for inclusion, namely the first[6], the 10th, the 11th, the 20th, the 21st, and the 30th, as these are the "border" images, i.e., the images where the scene changes from an empty hallway to a hallway with a person present or vice versa. Our test case did indeed output exactly this set of images that were selected for inclusion.

The size of an image taken by the camera ranges from 91 KB to 117 KB. Assuming that the hallway lies dormant from 11 p.m. to 7 a.m. every day, and that the camera is configured to take a picture every two seconds, our system will hence reduce the number of images uploaded to the TIPPERS backend from $\frac{8\text{h}\times 60\text{m/h}\times 60\text{s/m}}{2\text{s/img}} = 14,400$ to 1. Even for the smallest image size mentioned above, this translates to a decrease in the daily expansion of the database from $14,400\text{img} \times 91\text{KB/img} = 1,310,400\text{KB} = 1.3104\text{GB}$ to just 91 KB.

5.2 Necessary Edge Hardware

We provide a rough estimate of the necessary edge hardware by benchmarking the average time it takes to process an image. In the benchmark, we execute the YOLOv3 portion of our code on a set of 100 sample images captured by the same camera as used in the correctness and benefits evaluation described above. The benchmark was executed on a mid 2014 13-inch MacBook Pro Retina, equipped with a 2.6 Ghz Intel Core i5 dual-core processor and 8 GB of memory. Table 1 summarizes our results. We see that it takes roughly 12.5 seconds to process each image on this somewhat capable hardware. As such, it would require a cluster of $\lceil \frac{12.5\text{s/img}}{2\text{s/img}} \rceil = 7$ of such laptops to keep up with the work resulting from a single camera taking a picture every two seconds[7]. Needless to say, such a setup is infeasible given its monetary cost. We therefore propose that one pursues one of the following edge hardware solutions when deploying the system:

- Build a cluster of Raspberry Pis and off-load processing of each newly captured image in a round-robin fashion. The benchmark would have to be rerun on a Raspberry Pi to arrive at the number of Pis necessary in the cluster (calculated using the same equation as used above) as the clock speed of a Raspberry Pi is significantly lower than that of the MacBook used in our benchmark (1.4 GHz vs. 2.6 GHz, respectively). A (very) pessimistic guess of the number of Raspberry Pis needed to keep up with the image capture rate would be 20, leaving the hardware cost at a reasonable $700 given that the most recent Raspberry Pi retails for approximately $35.
- Deploy a small-scale, inexpensive, many-core server or desktop computer and execute the YOLOv3

[6]The first image is a special case. It is always selected for inclusion as the cache of previous images is empty.

[7]We processed the images sequentially in our benchmark (i.e., we deferred starting a new YOLOv3 process until the previous one completed), assuming that the YOLOv3 process would utilize all available cores for processing a single image. However, judging from the CPU utilization during the benchmark, YOLOv3 seems to be sequential (i.e., it only utilizes a single core), hence the number of required laptops could be cut in half (i.e., become $\lceil \frac{12.5\text{s/img}}{2\text{s/img}\times 2} \rceil = 4$) if one makes sure to always execute two YOLOv3 processes in parallel.

process in parallel on all cores. For example, one can currently acquire a 6-core AMD Ryzen for as little as $190 [5].

- Deploy a small-scale, inexpensive server or desktop computer with a reasonable, yet inexpensive, GPU and execute the YOLOv3 portion of the code on the GPU. As described in [8], YOLOv3 runs much faster on the GPU ("...like 500 times faster..." [7]), but complicates the necessary setup a little extra as one must install CUDA. Nevertheless, this solution seems like the most reasonable as it achieves the best speedup per dollar spent.

Table 1: Image Processing Statistics (ms)

Processed Image Count	Mean Time	Standard Deviation	Variance
100	12,488.53	86.13	7,419.06

6 Conclusion

We described the implementation of a software framework that performs sensor sample filtering at the edge of the network. The framework is completely generic, making it applicable to *any* type of sensor. As a proof-of-concept, we applied the framework to the surveillance cameras that are part of the TIPPERS system. We showed that the framework's ability to detect if successive pictures taken by the cameras are identical (e.g., two photos of the same, empty hallway) can potentially reduce the amount of data uploaded to the TIPPERS backend over night by a single camera from 1.3 GB to less than 100 KB. As TIPPERS moves forward in its development and implementation, there is no doubt that more sensors will be added to the system, and scalability hence becomes an even more pressing concern. As our software framework can easily be applied to arbitrary sensor types, we believe that it may be a handy toolbox for dealing with these scalability issues and that it may help secure the practicality of extending TIPPERS to the entire UCI campus.

References

[1] Ejder Bastug, Mehdi Bennis, and Mérouane Debbah. Living on the edge: The role of proactive caching in 5G wireless networks. *IEEE Communications Magazine*, 52(8):82–89, 2014.

[2] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog Computing and Its Role in the Internet of Things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, pages 13–16, New York, NY, USA, 2012. ACM.

[3] J. Fan, H. Luo, M. S. Hacid, and E. Bertino. A novel approach for privacy-preserving video sharing. In *In Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 609–616, 2005.

[4] P. Gilbert, L. P. Cox, J. Jung, and D. Wetherall. Toward trustworthy mobile sensing. In *In Proceedings of the Eleventh Workshop on Mobile Computing Systems and Applications*, pages 31–36, 2010.

[5] Newegg.com. AMD RYZEN 5 2600 6-Core 3.4 GHz (3.9 GHz Max Boost) Socket AM4 65W YD2600BBAFBOX Desktop Processor. [Online; accessed 2018-06-13].

[6] P. Pappachan, M. Degeling, R. Yus, A. Das, S. Bhagavatula, W. Melicher, P. E. Naeini, S. Zhang, L. Bauer, A. Kobsa, S. Mehrotra, N. Sadeh, and N. Venkatasubramanian. Towards privacy-aware smart buildings: Capturing, communicating, and enforcing privacy policies and preferences. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 193–198, June 2017.

[7] Joseph Redmon. Installing Darknet. `https://pjreddie.com/darknet/install/`. [Online; accessed 2018-06-13].

[8] Joseph Redmon. YOLO: Real-Time Object Detection. `https://pjreddie.com/darknet/yolo/`. [Online; accessed 2018-06-13].

[9] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv*, 2018.

[10] S. Saroiu and A. Wolman. I am a sensor, and i approve this message. In *In Proceedings of the Eleventh Workshop on Mobile Computing Systems and Applications*, pages pp. 37–42, 2010.

[11] M. Satyanarayanan, P. Simoens, Y. Xiao, P. Pillai, Z. Chen, K. Ha, W. Hu, and B. Amos. Edge Analytics in the Internet of Things. *IEEE Pervasive Computing*, 14(2):24–31, Apr 2015.

[12] J. Shotton, M. Johnson, and R. Cipolla. Semantic texton forests for image categorization and segmentation. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, 2008.

[13] P. Simoens, Y. Xiao, P. Pillai, Z. Chen, K. Ha, and M. Satyanarayanan. Scalable crowd-sourcing of video from mobile devices. In *In Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, pages pp. 139–152, 2013.

[14] Luis M. Vaquero and Luis Rodero-Merino. Finding Your Way in the Fog: Towards a Comprehensive Definition of Fog Computing. *SIGCOMM Comput. Commun. Rev.*, 44(5):27–32, October 2014.

[15] S. Yi, Z. Hao, Z. Qin, and Q. Li. Fog Computing: Platform and Applications. In *2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*, pages 73–78, Nov 2015.