More Than The Sum of Its Things: Resource Sharing Across IoTs at The Edge

Aliaa Essameldin *
Carnegie Mellon University
essameldin@cmu.edu

Mohammed Nurul Hoque *
Carnegie Mellon University
mnur@cmu.edu

Khaled A. Harras
Carnegie Mellon University
kharras@cs.cmu.edu

2020 IEEE/ACM Symposium on Edge Computing (SEC) | 978-1-7281-5943-0/20/$31.00 ©2020 IEEE | DOI: 10.1109/SEC50012.2020.00034

***Abstract*—The extreme growth and diversity of IoT applications, along with the heterogeneity of these devices, has led to numerous middleware solutions emerging to address various relevant challenges. These solutions have been shifting from cloud-based approaches to edge-technologies in order to handle issues related to privacy, latency, and bandwidth. Within this context, we identify an opportunity to introduce a novel IoT Middleware system that enables more seamless resource and context sharing at the edge. We propose the Hive, a middleware system that allows heterogeneous off-the-shelf devices in a common edge network to seamlessly and efficiently utilize each other's sensory and computational resources. The Hive architecture enables a new wave of multi-modal sensory applications, leveraging a pool of IoT devices, that would otherwise be unattainable. We accomplish this by decoupling the hardware, processing, and application layers within IoT devices from each other. The Hive abstracts each of these layers into a single resource pool that is shared and cross utilized on-demand within the edge network by any individual device. We implement the Hive, along with a dedicated communication protocol for our system. We evaluate the effectiveness of the Hive by integrating it with two sample IoT applications: an audio-based emotion recognition system, and a video-based facial detection application. We extensively evaluate the impact the Hive has on these new applications after integration, and additionally investigate its impact at scale. Our results show how the hive boosts the overall utilization of resources in an edge IoT network, reduces computational delay in complex applications, and most importantly, enables applications to perform at higher level of effectiveness.**

I. Introduction

Internet of Things is the notion that any "thing" can be connected to the Internet including machines, data management systems, and services, creating high opportunities for intelligent applications. Nowadays, we have allegedly reached 26.66 billion IoT connected devices [1], almost three times the world's current population. This abundance of devices and data has resulted in IoTs becoming an integral tool in numerous fields that include smart homes [2]–[9], factory automation [10], [11], agriculture [12], healthcare [13]–[16], and vision-based systems as well as augmented reality systems [17]–[21].

This IoT growth is empowered by a plethora of Middleware solutions that address common IoT challenges (*Section II*). While these solutions circumvent the latency network bottleneck of utilizing the cloud, they do not fully utilize the operating devices themselves; such solutions do not exploit the system's ability to capitalize on the overall computational and sensory capabilities of its nearby devices. Other prior work suggests Semantic Context Sharing for improved resource utilization, but these solutions only serve a limited and predefined set of applications which is a poor fit for the dynamic and highly diverse nature of today's IoT networks.

We adopt a representative smart-home scenario to demonstrate the gap that we identify in prior work and show how it motivates our proposed work in this paper *(Section III)*. The scenario is one amongst many that highlight the shortcomings of current application-centric edge-based middleware, such as computational redundancy and limited sensor sharing amongst edge peer IoT devices. These shortcomings drive the main purpose of our work: Building an edge-based middleware which can; 1) enable complete decouplement of applications, sensors, and processors in IoT devices, 2) allow seamless access to processed information across IoT entities, and 3) act as an omnipotent entity that can optimize these interactions for maximal resource utilization.

To fulfill these objectives, we build on our prior work [22] and propose the Hive, an edge-based middleware system which maximizes resource sharing among a group of heterogeneous IoT devices dubbed *Bees*. The Hive's objectives take shape in the three stages of its architecture: Data, Processing, and Core *(Section IV)*. The data stage is responsible for the decouplement of applications and sensors while remaining completely oblivious to the nature of both. Its first component, the *Seeker Interface*, manages all supported applications. Its second component, the *Provider Interface*, allows concurrent access to connected sensors residing at different IoT Bees. Together, the two interfaces allow data exchange between any number of applications and any number of sensors. At the Processing stage, Bees share compute information to maximize utilization of computational resources. The components at this stage perform heavy computations at the middleware layer so that their results can be shared by all applications and allow computational offloading so that the computations can be performed by any arbitrary processor in the network. The decisions for data-exchange and Computational offloading are made by The Core Stage of the architecture. At the core, a single central *Queen Bee* is distributively elected to gather data from all other *Worker Bees* to make the aforementioned pairing decisions and accordingly connect Bees.

To manage the heavy interaction required in our architecture, we develop the Hive Protocol (*Section V*). This protocol

*These authors have contributed equally to this work.

978-1-7281-5943-0/20/$31.00 ©2020 IEEE
DOI 10.1109/SEC50012.2020.00034

enables and governs communication between the components of the architecture across all Bees allowing for interoperability between different implementations of the Hive on different devices and platforms. We create a generic hive packet construct which carries data needed for identifying Bees, separating data streams, specifying computational algorithms, and authenticating messages. We then describe how these packets behave to support four message classes: one class for interfacing with Hive-supported applications (Class200), and three that correspond to the three architectural Hive stages: data (Class300), processing (Class400), and core (Class100). The protocol also implements the distributed Queen election algorithm which ensures that the system converges to a single Queen during bootstrapping or queen failure.

We evaluate a prototype implementation of the core and data functionalities of the Hive and its impact on two sample client applications *(Section VI)*. We demonstrate the robustness of the queen election algorithm and find that our system reaches steady-state in at most 3.5s. Then, we test the impact of the Hive on a sound-based emotion recognition app (Vokaturi) and identify the hive-induced improvement in its overall performance. Next, we carry out an extensive analysis of the costs of the Hive both, in a real-life setup for realistic results, and in an emulated environment for at-scale analysis *(Section VII)*. We achieve these results through a number of case-studies, testing many sensors streaming to a single application and a single sensor streaming to many applications. We also run tests with up to 10 bees on virtual machines with different network conditions and data rates to observe the impact of the Hive at scale in different environments. Overall, our evaluation uncovered some insightful conclusions. First, The Hive can seamlessly provide processed data from one provider to n seekers while cutting overall CPU utilization down to around $\frac{1}{n} + c$ where c is a constant that depends on the rate of the input data. In addition, the Hive achieves a delay well below one second under reasonable data rates. Finally, the number of Bees/IoTs we can add to a given Hive is only limited by bandwidth.

Overall, we summarize our contributions as follows:

- We design and implement an edge-based middleware IoT system, the Hive, that enables IoT ensembles to seamlessly share resources in order to improve application experience while increasing overall effectiveness and efficiency.
- We create a dedicated Hive protocol that enables these IoT ensembles to easily participate and intercommunicate within the Hive.
- We practically demonstrate the promise of the Hive by integrating it with two representative sample IoT applications, namely, audio-based emotion recognition and video-based facial detection, and show how both applications greatly improve in performance as a result of this integration.
- We extensively evaluate the video-based Hive prototype, and show its promise and impact on different metrics at scale.

II. Related Work

A. Cloud-based Solutions

Traditional IoT Middleware solutions such as those in [23]–[25] address many of the common IoT challenges such as interoperability, data management and limited storage. Similar solutions have been in industry such as Cisco's Jasper, Amazon's AWS IoT Core and PTC's ThingWorx. These solutions, however, rely on cloud-based infrastructures. This makes them non-ideal for real-time IoT applications because communication to cloud servers can cause delay that is much higher than the latency requirements of such applications as discussed in [26], [27] and empirically shown in [28], [29]. Therefore, recent work in the area has been considering edge-computing as an alternative for providing IoT services.

B. Edge-based Solutions

Moving Middleware operations closer to the Edge has been shown to decrease delay, increase privacy and even address multiple resilience issues [30]–[37]. The trend has been reflected on proposed IoT architectures across different domains in literature [38], [39] and industry [40]–[42]. Some solutions, such as [43], even present an architecture that recognizes the trade-off between edge and cloud-based management and dynamically shifts between using the two paradigms in real-time according to the system's varying needs. While this prior work addresses many of the shortcomings of the traditional cloud-based approach, it has been solely focused on providing computational resources at the Edge server to avoid the large network bottleneck of utilizing the cloud. It does not consider harvesting non-computational resources from the IoT devices themselves.

C. Resource Sharing Solutions

State-of-the-art solutions impose a level of application containment on the system, which makes sharing context and data across different applications running on the same Network a non-trivial task. Some previous work has proposed creative solutions for sharing storage space by distributing data across Edge IoT devices [44]–[48], but we were not able to find a fit as good for sharing context and computational resources. When we investigated the prominent industrial Iot Edge frameworks (e.g. Microsoft Azure and Apache Edgent), we found that they do not support sending data over dynamic P2P paths which keeps them from supporting dynamic IoT scenarios like the ones discusses in this paper. Context-sharing in and off itself is not a new problem in ubiquitous computing. [49] considers 50 different research and commercial projects over the last decade in the field of context-aware computing. Recent work builds on the general rule and modules suggested, such as [50] which proposes a solution for context-sharing across applications in smart buildings, and [51] which presents an approach for remote health monitoring. But not unlike modern edge IoT solutions, these solutions enable context sharing only for a limited set of predefined applications within a given domain, which is not flexible enough for today's dynamic and rapidly evolving IoT applications.

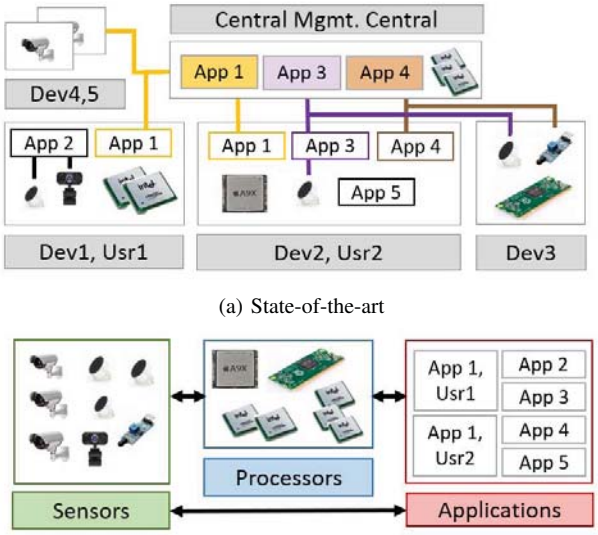
III. Hive Motivation and Objectives

In this section, we demonstrate the shortcomings in current state-of-the-art IoT solutions with regards to expected future IoT use cases. We use a sample smart-home scenario where users have various IoT systems running and expect them to work seamlessly together to provide a better experience. The scenario presented in Fig. 1 is that of a household containing 5 devices: Dev1 is Usr1's laptop, Dev2 is Usr2's phone, Dev3 is a raspberry-pi equipped with a microphone and a temperature sensor, and Dev4 and Dev5 are off-the-shelf CCTV camera. The house has App1, which is a smart surveillance system like the one offered by SierraOne [52] and many other companies. These systems recognize individuals and objects to recognize threats, which entails heavy and real-time image processing. App2 is a typical video conferencing tool that does some image processing to change the user's background like the case in Zoom [53]. App3 is a speech-based emotion recognition solution such as those offered by Affectiva [54] and Vokaturi [55], aimed to continually monitor the mental health of residents. App4 is a room temperature monitoring app and App5 is a computationally demanding phone game.

Today's IoT solutions as in Fig 1(a), meets each application's basic requirements, but fails to wield the advantages of the IoT ensemble as a whole. Consider these examples:

1) Usr1 wants to talk and walk while on a video call via App1, but despite having enough cameras and microphones around, App2 cannot access App1's cameras nor App3's microphones because they were not directly configured to serve it.
2) Assuming 1 is solved, App1 and App2 may both be running similar feature-extraction algorithms on the same video stream, yet the user will have to pay the energy cost of this demanding computation twice because there is no straight forward way to avoid redundant computations.
3) App5 cannot leverage processors on the management node or laptop to reduce on the phone.
4) A mental health monitoring system can make more insightful conclusions given the temperature of the room, but unless App3 is pre-configured to talk to App4, it cannot benefit from the temperature readings made by the sensor on Dev3.

To overcome these limitations, we designed the Hive, a system consisting of a pool of *Bees*, a group of smart devices running a middleware which abstracts all resources in the network into a pool of applications, a pool of sensors, and a pool of processors that can seamlessly access one another (see Fig 1 (b)). To fully address the gaps identified, the Hive must fulfill the following objectives: 1) Data can be seamlessly and simultaneously collected from any single sensor by an any subset of applications and from any subset of sensors by any single application. 2) An application can access information that is collected and processed by other applications. 3) All applications and sensors can run their computations on arbitrarily any processor in the network. 4) The omniscient middleware gets to optimize all resource sharing related decisions (e.g. which sensors should serve a given application, which bee should be doing the processing, etc.). The Hive aims to enable these objectives through a comprehensive system that we detail in the upcoming sections of this paper.



(a) State-of-the-art

(b) Our Vision

Fig. 1: Communication between IoT components of a given smart-home scenario given two Middleware models

IV. Hive System Architecture

The Hive consists of a system daemon and an accompanying API, this allows it to work across different devices while expecting minimal integration efforts from application developers. The architectural breakdown of the daemon depicted in Fig. 2 enables the desired decouplement by orchestrating a data-exchange flow that is compatible with any arbitrary type of data, device, and application. This flow is divided into three stages: data, processing, and core. At the data stage, applications and sensors on a device are abstracted into applications and sensors pool respectively. Any application pool can collect data from any sensors pool that is connected to the Hive. At the processing stage, data collected can be offloaded to and processed on any edge-connected device. This effectively abstracts computational units on all devices in the network into a processor pool. Finally, the core stage creates and manages connections between these pools of applications, sensors, and processors (as per our vision shared in Fig. 1). Fig. 2 shows all architectural components serving the applications pool in red, those serving the sensors pool in green, those serving the processors pool in blue, and components concerned with connecting all devices in grey. We now describe the roles these components play in each stage in more detail.

A. Data Stage

In this stage, we decouple applications and sensors while allowing arbitrarily many applications to collect data from

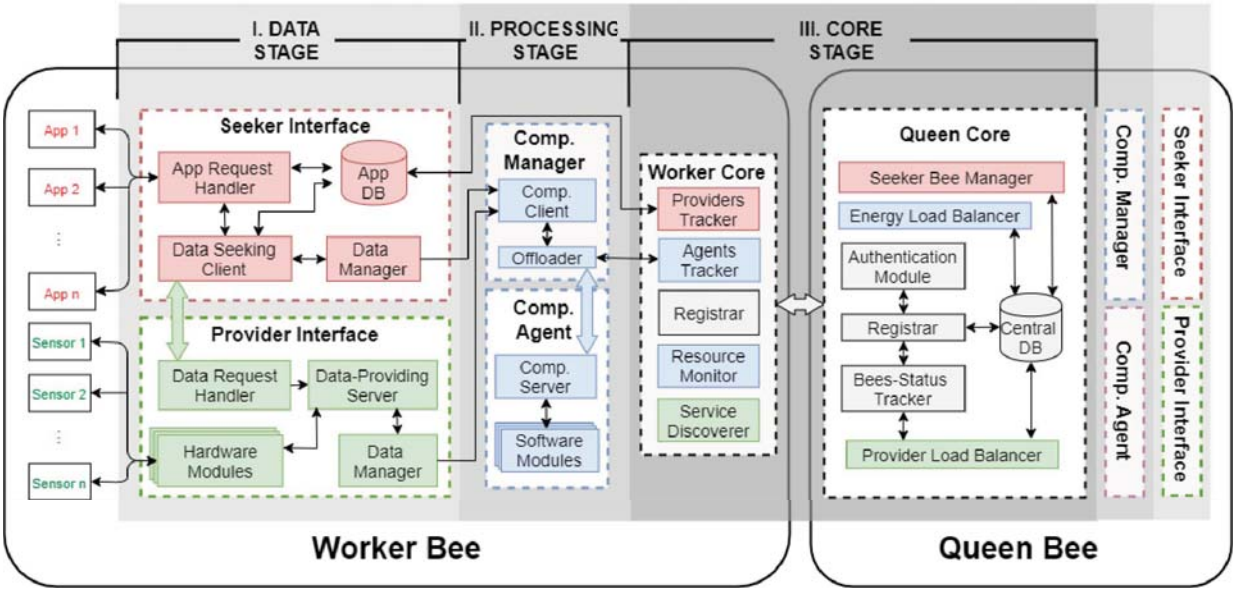


Fig. 2: Hive Architecture

many sensors. In a given data-exchange, we refer to a Bee where an application has requested data as a *Seeker Bee*, and one where a sensor is providing data as a *Provider Bee*. A device can simultaneously act as a seeker in one exchange and a provider in another. The architectural components on each side of the exchange are described in more detail below.

1) The Seeker's Side: The *Seeker Interface* allows arbitrarily many Hive-supported applications to access the Hive resources without exposing the details of the Hive to the application nor the opposite, ensuring complete decouplement of applications. The Hive API on the supported application contacts the local middleware instance through the *App Request Handler*. The Handler is responsible for authenticating applications, updating the local database with their information, and forwarding their data requests to the *The Data Seeking Client*. This client collects the data requested by an application from the top *Provider Bees* specified in the local App Database. After the data is collected, *The Data Manager* is responsible for any operations that need to be done on it such as filtering or aggregating data streams from multiple providers, or passing the data to be processed in the second stage before it is passed back to the Request Handler then to the application. The Data Manager can allow context sharing since it has the processed data known by all applications (such as results of localization or feature extraction).

2) The Provider's Side: The *Provider Interface* allows arbitrarily many sensors to serve the hive with no awareness of which or how many applications their data will be serving, ensuring sensory decouplement. It receives data requests from the Hive on *The Data Request Handler* which parses and passes the requests to the *The Data-Providing Server*. The server collects data from the *Hardware Module* which controls the intended server and passes the results through the Data Manager if needed. The Provider's Data manager is responsible for any operations that need to be done on the data before sending it back such as checking it for a certain threshold or passing it to be processed in the second stage. This capability can reduce redundant computations in the network because it can ensure that common algorithms (such as feature extraction) are run only once on the data instead of redundantly on multiple Seekers. (This is discussed further in Section V).

B. Processing Stage

After data is collected either at a Provider (pre-exchange) or a Seeker (post-exchange), it can be processed further before it is passed back from the Hive to the application. Pushing heavy processing from applications level to the middleware allows for better utilization of computational resources in the hive via computational offloading, with the added benefit of making development easier for IoT Engineers. In a given computational offload, we refer to the offloading device where the data was collected as *Manager Bee* and to the offloadee where the algorithm will run as *Agent*. The architectural components on each side of the computational offload are described in more detail below.

1) The Manager's Side: The *Computational Manager* allows the Seeker and Provider Interfaces to access the Hive's processors without exposing the details of the Processing Stage's operations to the Interfaces nor the opposite, effectively decoupling the processors pool from the other two. It receives processing requests and data streams on *The Computational Client* then passes them to *The Offloader* which then contacts the appropriate Agent bee and collects results as a derived data stream that it passes back to the Data Interfaces.

2) The Agent's Side: The *Computation Agent* allows a processing unit on any given bee to perform computations for any Manager bee and return results. The data and requests are received from the Manager on *The Computational Server* which is responsible for receiving the processing request from the *Manager*, invoking the relevant Software Component that would process it, then passing the results back. *Software Components* are the contained code snippets that contain different computationally demanding algorithms (e.g. video feature extraction).

C. Core Stage

The Core of the Hive is the glue that brings everything together; it enables and manages connections between the different components to maximize robustness and utilization. Core functions are centralized at a single distributively elected node, referred to as the *Queen Bee*, which manages all other connected devices referred to as *Worker Bees*. The core architectural components at the *Queen Bee* and their counterparts at each *Worker Bee* are described below.

1) The Worker's Side: The Worker Core assists the Queen in maintaining the robustness of the system and optimizing its resource usage. Robustness is achieved through the *Registrar*. Upon boot-up, it authenticates the Worker Bee with the Queen. After that, it is responsible for responding to Queen Heartbeats and restarting the Queen election process in case the Queen dies. In case the current node is elected as Queen, the Registrar replaces the Worker Core with a Queen Core.

To assist the Queen in optimizing resource usage, the *Service Discovery* module discovers and monitors available sensors and the *Resource Monitor* monitors the state and health of the local resources. This information is periodically passed from all Workers to the Queen. The Queen is then able to answer queries from the Worker's *Providers Tracker* about the optimal Providers for a given datatype and *Agents Tracker* for optimal Agents that this Bee should offload to. They then update the Seeker Interface's database and the Computational Manager's offloader respectively.

2) The Queen's Side: The *Queen Core* authenticates and tracks all Worker Bees in the system, collects relevant information from them, and centrally decides on optimal Seeker-Provider and Manager-Agent pairings. The *Registrar* handles requests from the Worker's Registrar, verifies their validity via the *Authentication Module*, adds them to the global database then continues to track their status via the *Bees Status-Tracker*. The tracker sends registered Worker Bees periodic Queen Heartbeats and uses the Heartbeat ACKs to verify the Bees' connection status. If a Bee is found dead, the Tracker immediately updates the database.

The Queen makes two types of decisions to optimize utilization of resources: Seeker-provider pairings (which sensor would each application use) and Manager-Agent pairings (which Agent would run the code). To make valid Seeker-Provider pairings, the *Provider Load Balancer* keeps the central database updated with information about available sensors in the network which it collects from the workers' Service Discovery Modules. The *Seeker Bee Manager* then accesses this data and constructs a prioritized pairings table in the database that it then uses to answer queries from any Worker's Provider Tracker. Similarly, the *Energy Load Balancer* collects data from Workers' Resource Monitor and uses it to balance processing load across devices and accordingly answer queries made by Workers' Agents Trackers. The Load Balancer is most effective in event-triggered load spikes which require more demanding processing. The exact distribution of load will depend on the energy capabilities of online devices and the nature of the task at hand. Exact task scheduling heuristics are covered and tested in prior work [56].

V. Hive Protocol

This section describes the Hive Protocol which enables seamless and robust communication between the components of the hive and is similarly divided into data, processing and core functionalities. We first discuss the design logic underlying the communication flow overall. Next, we present how this flow is manifested in the protocol's four main classes of packets. Then we present the general Hive packet construct that enables this protocol. Last, we describe the main core functionality: Distributed Queen Election.

A. Communication Flow

The hive achieves its unique level of flexibility and careful deliberation of the underlying communication flow. Below we describe the most significant points of flexibility and how they were accomplished:

Processing Request flexibility: Without assuming a specific kind of algorithm that will be requested, it is difficult to decide whether computational requests should be made by the Provider, thus avoiding computational redundancy across seekers; or by the seeker, thus avoiding unnecessary networking cost. The hive mediates this by allowing both Data Managers to make processing requests. The Data Seeking Client decides -based on the algorithm requested- whether the Seeker or the provider should place the request. The computation client is responsible for multiplexing concurrent ones.

Device Oblivion: To maintain complete device oblivion the Seeker and Provider always exchange data across Hive Protocol even if they are on the same device. This ensures that the Provider Load Balancer maintained by the Queen is always up to date since the Seeker Interface would always go through it. Given this up-to-date input on how many Seekers each Provider is serving, the Queen makes more optimal decisions on Provider-Seeker pairings.

Processing Unit Oblivion: Similarly, computation managers and agents will always exchange data across the Hive Protocol even if they are on the same device. This ensures that the Energy Load Balancer is always up to date.

B. Hive Packet

The Hive Packet consists of seven headers with a total size of 20 bytes followed by packet body that can carry extra

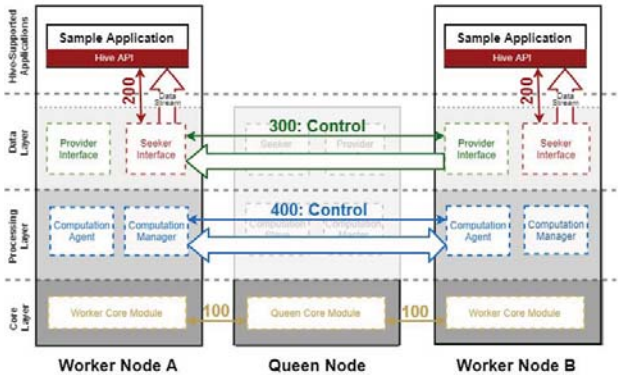


Fig. 3: Exchange of Different Packet Classes in the Hive

information, and ending with a 4-bytes-long hash. The typical usage of the headers and the hash is described below:

Packet Type (2 bytes): The packet type is a three-digit integer. The first integer depends on the packet class (see 4.B). The second digit defines the packet function. The third digit indicates the state of the function: a request, an accept (ACK) or a reject (Error).

Hive ID (2 bytes): This unique ID, along with the Authentication Hash, allows two Hives to be colocated without interference. Devices first check if their Hive IDs match before they try to resolve the Hash.

Source and Destination Bee IDs (8 bytes): These define the packet source and destination using the Hive's addressing scheme. By handling addressing in the Hive layer, we decouple the Hive from its underlying network technologies.

Data Type ID (2 bytes): This is the ID of the Hardware Module that will be handling this request. It differentiates between different data streams. SOUND, VIDEO and TEMP are examples of Hardware Component IDs.

Algorithm ID (2 bytes): This identifies the *Software Component* that will be used on this stream, if any. This is part of the data level since a Seeker specifies software components in its data request to the Provider whenever applicable to allow provider to potentially run an algorithm once for potentially many Seekers (see 5.D). It is also a part of the processing level since it is how the Data Managers communicate with the Computational Manager (and then the Manager communicates with the Agent) which Software Component to run on the data.

Body length: The length of the packet body in bytes.

Authentication Hash: A 32-bit integer which uses the Hive ID and a user-defined password to ensure that packets exchanged in the Hive are from user-authenticated devices.

C. Protocol Classes

The Hive Protocol is divided into four functions addressing the 3 levels of operation of the hive depicted in Figure 3. The protocol's three main functions are determined by the 100, 200, 300 and 400 class packets. In all packets, the packet type is indicated by the first two digits: the first is the class and the second in the function in that class. The third digit is either 0 or 5 for the initial packet itself, 1 or 6 for an error and 2 or 7 for an acknowledgement.

The class-100 packets are for control packets carrying out core functionalities. These packets are used to coordinate between the different bees and exchange management information. They are responsible for the boot-up process, Queen election and re-election, address resolution and pairing of Seeker and Provider Bees. They are exchanged over UDP.

The class-200 and class-300 packets are for data functionalities. 200-class packets are for IPC between the Hive layer, which runs as a system daemon, and the applications running on the device. Whenever an application registers, it starts a TCP connection with the Hive layer over which class-200 packets are exchanged. They are used to confirm the registration, authenticate and serve the application. The class-300 is for control over actual data exchanged between a Seeker and a Provider. For Data exchange, two direct channels are established between Seeker and Provider bees (without interference of the Queen). One of them is a TCP connection for data control packets and the second is a UDP stream carrying the actual data.

The class-400 packets are for task-offloading between a Computational Manager and Agent. Three channels are established for computational offloading between Manager and Agent bees (without interference of the Queen). One of them is a TCP connection for class-400 control message exchange between both parties, the second is a UDP stream from the Manager to the Agent with the data, and the third is a UDP stream from the Agent to the Manager with the computational results. The results-stream is passed as is from the Seeker Interface to the Application, in case of post-exchange processing, or from the Provider Interface to the Seeker Interface to the Application, in case of pre-exchange processing.

D. Queen Election Algorithm

We use a variation of a classic distributed leadership election algorithm [57] to bootstrap the Queen election and ensure the robustness of the system. The algorithm ensures that the system will always converge to a single Queen. Then, the Queen can track all Worker Bees to ensure the robustness of data connections.

The process is summarized in the State Machine illustrated in Figure 4. As shown in the figure, when a device boots up it enters a unique boot-up status. It will wait for a heartbeat, which is a periodic broadcast from the Queen. If this new device hears a heartbeat, it becomes a Pending Worker; otherwise, it assumes there are no queens around and declares itself as the Queen. If two devices boot up at the same time, they may both declare as queens, but eventually one would hear the heartbeat of the other. When that happens, the two *compete*. Competition between Queens is a comparison between up-times. The Queen with the longest up-time wins because this indicates that it is a more stable device in the house. Stability is important because we want to avoid the delay introduced by Queen re-election. After the competition, one device stays as Queen and the other demotes itself to Pending Worker.

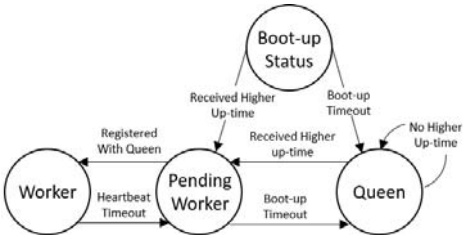


Fig. 4: Queen Election Process State Machine

Pending workers become Workers after they register all their data and synchronize with the Queen. Finally, the device remains as a worker bee unless it misses 3 heartbeats in a row from the Queen bee. It then it assumes the Queen is dead, elects itself as Queen and the election process restarts.

We argue that this algorithm will eventually converge to a single Queen with any number of nodes and after any number of Bees leaving or joining the hive. [57] presents a proof that given an n-node network, composed of a connected component of size $m < n$, and of $n - m$ isolated nodes, for every $k < n - 1$ node disconnections and $q < n - m$ nodes joining, exactly a single node in the network will have equal number of *cand* messages (Queen Heartbeat) sent and *accept* (ACK) messages received. We also note that this proof does not account for message loss due to congestion, which is very likely in our system. However, the Hive Protocol accounts for it by sending the Queen Heartbeats periodically. Namely, our protocol does not guarantee that there will be a single Queen at any given second, but it guarantees that eventually the system will converge to a single Queen. Given the role of the Queen in the system, this leniency is tolerable.

VI. Preliminary Assessment

In this section, we evaluate the functionality of a prototype implementation of the Hive. We begin by testing the core functionality of the system: the boot-up process responsible for electing the Queen. Then, we test the prototype on a sample emotion recognition application to demonstrate how the hive can improve the quality of the application.

A. Queen Election

We begin by testing the robustness of the Queen Election Algorithm implemented in the core module against various failures. The set-up has 5 Raspberry Pi's (RPI2 model B+) with IDs 001 to 005. All Bees are connected via Wi-Fi over a Cisco Linksys E900 router. We observe the bees' status over time and in different scenarios. We trigger the algorithm by booting up all the devices, then again by killing the elected Queen. Figure 5 shows the Queens declared over time.

We observe that, initially, not all bees declare themselves as Queens. Bee001, which was first to boot-up, declares itself as Queen and is followed by Bee002 and Bee003. Bee004 and Bee005 hear Bee001's broadcast (Queen Heartbeat) before their boot-up timeout and directly declare themselves as workers. Bee002 and Bee003 eventually hear the broadcast and after competing with Bee001, demote themselves leading

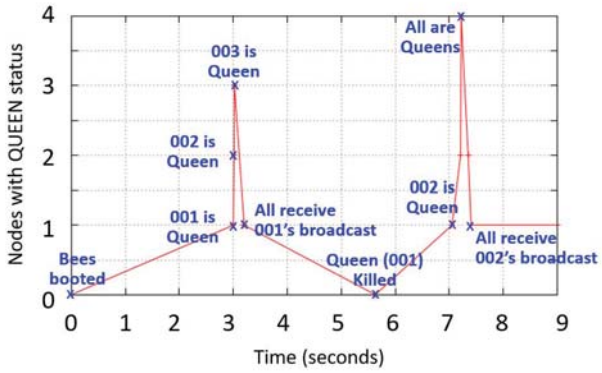


Fig. 5: Changing number of Queens during Boot-up

the system to converge to a single Queen after approximately 3.2 seconds. Afterwards, we kill Bee001, the Queen, at approximately 5.5, this can be seen in Figure 5 as a dip where the system declares that it momentarily has 0 queens. All other bees detect the loss of a queen after missing three consecutive heartbeats, and declare themselves as Queens at approximately 7.25; they then compete and converge to a single new queen by second 7.5.

After observing and plotting the behavior of the system during queen election, we switch to 3 nodes setup and run further tests, 10 times each. The system consistently passes those tests. Below, we describe each test, the delay we measure in it and the average of this delay over the 10 runs.

- Sequentially turn on all devices: First device to boot-up becomes the queen and the other two register as workers.
- Turn on devices at the same time: Each device declares itself as queen, then they compete and elect a single queen. Delay from boot-up to the last worker registration is **3.21s** on average.
- Disconnect Queen: One or both workers detect that a Queen is gone and election restarts. Delay from the Queen's death to the last device's registration with the new queen is **2.35s** on average.
- Disconnect a Worker: Queen detects that a worker disconnected and clears up its resources from the Hive. Resources are cleared after an average of **3.5s** from the worker's death.

These tests also demonstrate the Hive's behavior in case of a network partition. The partition that has the Queen (Bee001) will detect Workers' disconnection while the disconnected Workers in the other partition will detect Queen disconnection and elect their own. Once connectivity is retrieved between the partitions, the Queens will compete and converge in less than **3.21s** as per the second experiment described above.

B. Sound

Context-aware event-driven sensing is one of the features that motivated our system design. Given the importance of this feature, we implement and test it in our preliminary prototype using the following scenario: in case a user is being recorded

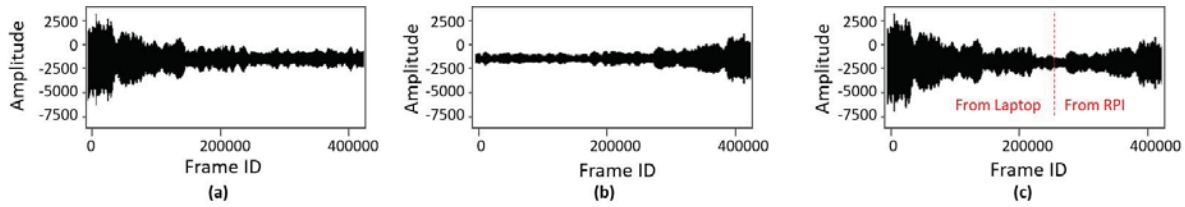


Fig. 6: Left: Sound collected without using the hive on (a) laptop and (b) RPI. Right: (c) Sound collected using the hive

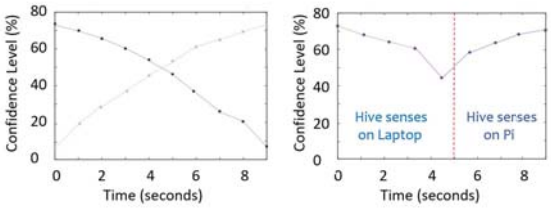


Fig. 7: Left: Emotion recognition from both bees without using the hive. Right: Emotion recognition from a single bee using the hive.

by any hive-based application, the event in which they move between devices triggers the hive to start collecting sound from the microphone that is closer to the user. There are countless potential IoT applications that can use this feature. This feature achieves collaboration between devices in two ways: first, devices can produce triggering events for other devices, in this case the location of the user. Second, sensors can collaborate to collect an optimal data stream, in this case toggling between microphones to collect the clearest sound from the microphone on the device closest to the user.

We enable this feature by implementing a simulated localization software component LOC that returns user location. The Data Request Handler accepts a request if the user is in the same room as the device and rejects it otherwise. If the Bee accepts the request, it collects the sound stream using an ALSA-based sound component that we implement.

We test this implementation with two experiments in which a source of sound is moving from one room to the other. Each room has a different IoT device equipped with a microphone. The objective is to have the Hive automatically select one of the Bees to capture the sound stream. In the first experiment, we run a simple sound capturing application to track a travelling sound source and show how this feature can provide a higher and clearer sound stream. In the second experiment, we run a sound-based emotion recognition application that uses the Vokaturi library [55] to show the impact of the Hive on a real potential smart home application. We describe both experiments in more detail below.

The sound stream used is a melody that we played from a moving device. The sound is collected and printed by a hive-supported application on Bee003. We first move the source device from the proximity of Bee001 to Bee002 and collect the sound samples from each bee, without defining the triggering event. We achieve this by only allowing one of them to register to the hive at a time. Then, after defining LOC as the algorithm used, since it is a stream-defining algorithm, we allow both Bee001 and Bee002 to register and collect the sound from both microphones. The results of this experiment are presented as sound waves (amplitude over time) in Figure 6.

Figure 6 (a) and (b) shows the two sound streams collectable without event-triggered sensing. Without event-triggered sensing, the application can either use the microphone on Bee001, where sound is loud then fades away, or the microphone on Bee002, where it is low then rises. We experimentally verify that the difference in sound intensity between 01 and 02 is because 01 is a laptop with a stronger microphone than the Pi. As seen in Figure 6 (c), after defining the trigger (shown in red), the hive only collects sound from Bee001 until the sound emitting device is close enough to Bee002, then it starts picking up the sound stream from Bee002 instead. This provides a smoother sound stream with appropriate volume.

We repeat the previous experiment using an emotion recognition application on top of our Hive prototype to demonstrate ease of integration. We use a 9-second human sound recording instead of the melody and create two copies of the recording, each chopped into 9 one-second pieces. One copy, running on the laptop, had sound that was reduced in intensity (volume) by 10% after each second. The other copy, running on the RPI1 had sound that increased in intensity by 10% after each second. This setup was developed to emulate user movement between two devices without introducing other variables that can impact the system performance. The hive collects the sound stream in an identical manner to the previous experiment and feeds it to the emotion recognition application. The application uses the Vokaturi [55] open source API to classify the sound recording as one of five main emotions. The library also reports the classification's confidence level.

Figure 7 shows the confidence levels reported with and without the hive's event-triggered sensing feature. The graph on the left shows the reported confidence levels from the laptop (Dark purple) and pi (light blue). As seen in the graph, without the hive, the application developer would have to choose between rising or falling confidence. However, as the graph on the right shows, hive empowered the application with higher quality data; confidence level stayed consistently above 40%. We note that the confidence levels is a function of the library used, not the Hive; the experiment is rather concerned with the variance of confidence with the movement of the sound source.

VII. Video Integration and Evaluation

We extensively evaluate our Hive prototype to better understand the costs, benefits, and bottlenecks of using it. We first demonstrate two case-studies on two real-life scenarios using physical hardware, then study both scenarios on-scale using virtual machines on a simulated network. In our first scenario, we test for a Seeker's (application) ability to seek data from arbitrarily many Providers (sensors); and in the second, we test a Provider's ability to provide data and processed information to arbitrarily many Seekers. Together, these case studies demonstrate the decouplement that we are after: we can have arbitrarily many applications dynamically access arbitrarily many sensors and processors. The case studies also serve to show the cost-benefit balance of the hive in a real-world demonstration. The scale tests demonstrate the Hive's versatility at-scale and under many environments. All of our testings are done using real-time video-based applications since video is a more intensive resource to help stress-test the Hive and evaluate the data exchange process.

A. Video Implementation

For our testing, we developed a Face Detection Module, a sample Hive software module in python for processing the video streams. The module uses OpenCV's built-in Haar cascade classifier to compute face detection confidence scores for each video frame and make it available at the provider as "video metadata" data type. To aid our measurements, it also puts a timestamp on the data for each frame. Providers use this module to provide meta-data streams with video to one or more seekers. A seeker can then combine video streams from all the providers it is receiving from and output a stream consisting of the frames with the highest confidence scores. Note that the video input at a provider is encoded and compressed so that the data transmitted over the network to seekers is minimal. The streams are decoded on a seeker to select the frames from all streams. The frames are then delivered decoded, i.e. raw, to the apps. The fact that the software module is written separately in a different language from the Hive demonstrates the system's modularity and its ability to plug-in components as necessary.

We also wrote three hive-supported video applications that require face-detection. App1 represents a video conferencing app that displays the stream with the best frontal face. It also outputs the delay of each frame by calculating the time since the frame's timestamp as recorded by the Face Detection Module, this constitutes the delay of the hive from provider to application. App2 represents a crowd counting application, reading the metadata stream from the hive and counting the number of faces detected in the streams. App-3 represents a surveillance app that also receives the metadata from the hive and monitors the confidence scores, outputting a warning message if a face is detected with threshold confidence that exceeds a certain threshold. For each of the three apps, we also wrote a non-hive version, which reads a video source, runs the face detection itself, and outputs the same result. The three applications, together with the sound application in the last section, demonstrate the flexibility of the Hive and its impact on applications' performance.

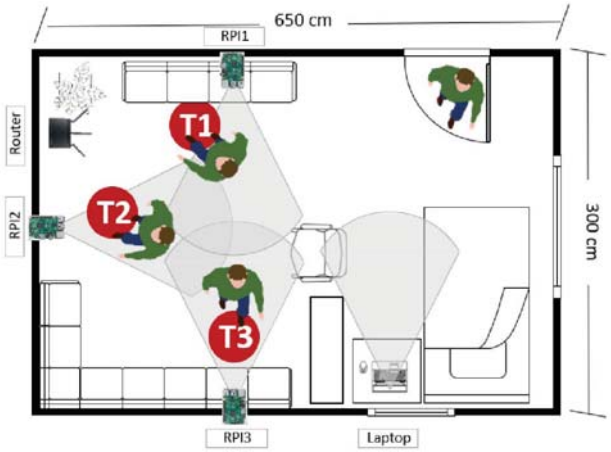


Fig. 8: Experimental set-up for Video-based Case Study

Parameter	case study values	scalability tests values
Video resolution	640x360, 854x480, 1280x720	
Frame rate	6, 12	24
Delay (ms)	1.5	5, 20, 50
Bandwidth (Mbps)	45	10, 20
Providers/Seekers	1…3	1…9

TABLE I: Evaluation Parameters

B. Video Integration Tests

1) Experimental Set-up: To test the performance of video-based applications on the hive in a real-life environment, we use the experimental set-up shown in Figure 8. The set-up has 4 bees: 3 Raspberry Pi's (RPI2 Model B+) each equipped with a camera module (v2) and a laptop running Fedora 31 with a core i5-8250u processor. The Pi's are fixed close to one another in one half of the room and the laptop is sitting behind an obstacle as shown in Figure 8. All devices have both the hive and non-hive versions of all three applications above. All devices are connected via Ethernet over a Cisco Linksys E900 router, except for the Laptop which is connected over Wi-Fi. We measured the link between the laptop and the Pi's at 45 Mbps bandwidth and 3 ms round-trip time.

In our experiments, we measure delay, bandwidth, and CPU utilization. Minimized delay is an essential quality for real-time IoT applications, reduced CPU utilization caters to low-energy devices allowing higher heterogeneity of Bees and bandwidth verifies the system's effectiveness in different realistic environments. We vary the video quality and the number of seekers as in Table I. The 720p resolution is not used in the experiments involving Pi's because they are incapable of running face detection consistently at high resolution. We did not implement the code offloading module of the Hive in our prototype so the *Computational Manager* can run any software modules only locally.

2) Scenario A: One Seeker, Many Providers: This scenario demonstrates how the Hive can improve the performance of an application on a Seeker Bee by allowing seamless access to

Fig. 9: Video as observed by different providers vs stream as displayed on seeker

data from multiple sensors (potentially on multiple Provider Bees). The Hive is boot-up with cameras on the three Pi's and App1 running on the laptop. As specified in the application's data request, the hive collects data from multiple provider Bees and uses the face detection meta-data from the provider to make a post-exchange decision on which stream it should feed the application.

We study how The Hive can allow any application requesting an optimal data stream, in this case, an optimal face video stream, to receive it from any sensors around the network, for example, surveillance cameras around the house. The scenario we mimic is a person having a video call while moving around the room. As shown in Fig. 8, the Pi's cover a part of the room that the laptop, the Seeker on which the application is running, has no visual coverage. The movement of the subject is also shown in the figure. The subject first faces Pi1 at time T1, then moves to face Pi2 at T2 and finally Pi3 at T3. Figure 9 shows the frames as captures by each Pi individually and the final stream given to the seeker application (laptop). Notice that there are instances where the subject is visible by more than one provider as in T1, but the Hive chooses the highest confidence of a forward face. The Hive's value lies in enabling the application to access data that would otherwise be unattainable. The opportunities that this unlocks for application developers are endless.

Next, we use a pre-recorded video, for better control, to evaluate the performance under this setup in terms of delay and bandwidth usage. Figure 10 shows how delay changes with different video qualities. The bandwidth results are as expected; it increases linearly with the number of seekers. The usage increases by less than 100% going from 6 fps to 12 fps because of video compression optimizations. Delay almost halves as we double the frame rate while increasing resolution and the number of seekers causes minimal change.

To make better sense of the empirical data from our experiment, we can model the hive costs abstractly based on the description of our implementation. The are several components

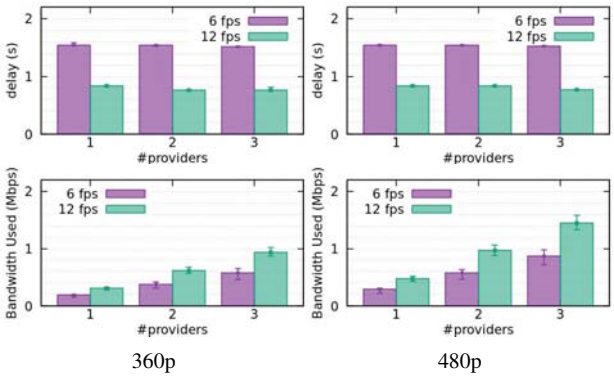


Fig. 10: One seeker, many providers performance

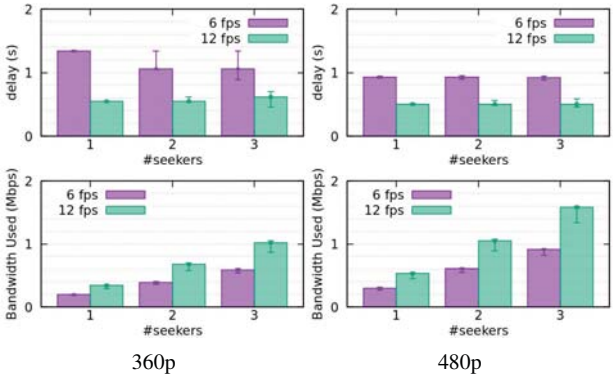
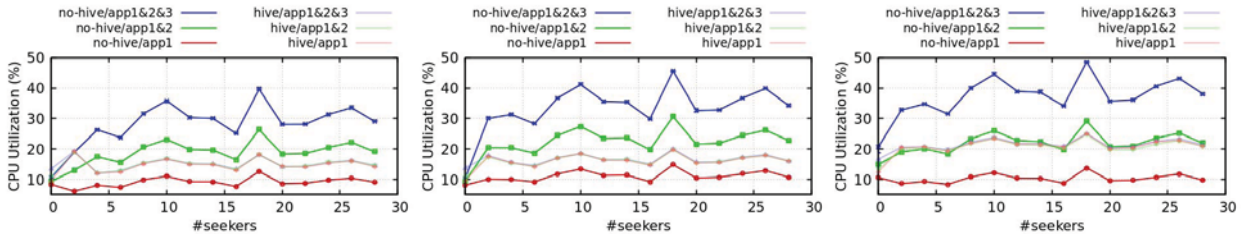


Fig. 11: Many seekers, one provider performance

to hive delay d_{hive}. First, we feed sensory data to the hive using ffmpeg as a "hardware module." The delay introduced by ffmpeg is not included in our measurements because it is a function of the hardware module rather than a reflection of the actual hive performance. We timestamp the frame just before running face detection, which is the first component of delay d_{detect}. Frames are scaled down to a fixed dimension before running the model, hence d_{detect} does not depend on video resolution. Next, the frame and its metadata are copied over through the hive pipeline from the provider to the data manager at the seeker that combines the streams and finally to the app, potentially going through a network in the middle. These copy operations incur a cost d_{copy}, which depends on data size which is a function of video resolution. The multiple copy operations cause a frame to be buffered multiple times before reaching the destination. For example, the seeker software module combines frames from all providers before handing the complete result frame to the data manager. Ignoring all other sources of delay, this causes the frame at the app to be some constant b frames behind the new frame being generated at the source. At a frame rate of r, a frame is generated every $1/r$ interval which causes a buffering delay of b/r. In total,

$$d_{\text{hive}} \approx d_{\text{detect}} + d_{\text{copy}} + b/r \tag{1}$$

Looking back at the result, b/r is the delay component that



(a) 360p (b) 480p (c) 720p

Fig. 12: Many seekers, one provider CPU utilization

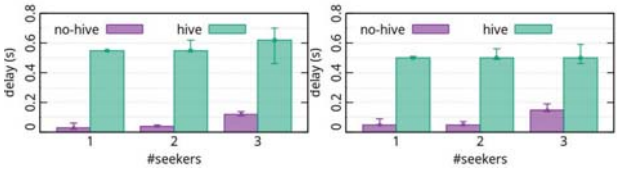
causes delay to decrease with a higher frame rate. On the other hand, d_{copy} is the only component that depends on resolution, which we deduce to be small based on the results as the delay is virtually the same with both resolutions.

3) Scenario B: Many Seekers, One Provider: This scenario demonstrates how the Hive can increase the utilization of resources on the network by enabling a single sensor to concurrently provide video data to multiple applications while sharing computations. We first, test performance using the same configuration as before with the laptop as the provider and the three Pi's as seekers running App1. The results are shown in Figure 11. These results match the ones for one seeker, many providers case with the exception that delay is overall reduced by around one third. This is attributable to the fact that the bulk of processing happens on the provider side, and in this case, the provider is a laptop that has a more powerful CPU than the Pi's which are the providers in the previous case.

4) Hive Cost-Benefit Analysis: We utilize our video setup to assess the costs and benefits of the hive. Two fundamental costs associated with the hive are CPU utilization and delay. A key advantage of the hive is avoiding redundancy by performing computations once and sharing results. This reflects on the CPU utilization when the same app is run on different machines using the same resources. On the other hand, the hive requires coordinating devices, which we expect to add some delay overhead compared to a setup with fixed, hard-coded data paths.

To evaluate the save in CPU utilization we achieve by using the hive, we test two configurations, local and networked. In the local configuration, we run a single instance of the hive on the laptop and run the hive apps also on the laptop and compare it against running the non-hive versions of the apps on the laptop. In the networked configuration, We run the applications each on a Pi, running the laptop as the only provider. In both cases, we use 12 fps recorded video and vary the video resolution and the number of applications (seekers). Since the face detection components are run on the provider, the face detection algorithm now only runs once with the hive and the results are streamed to all the different seekers. Figure 12 shows the results for the local configuration.

Similarly to equation (1), we can model the CPU utilization with and without the hive to explain the results. Denoting the set of providers with $\mathbb{P}$ and the set of application with $\mathbb{A}$ the



(a) 360p (b) 480p

Fig. 13: Hive delay cost

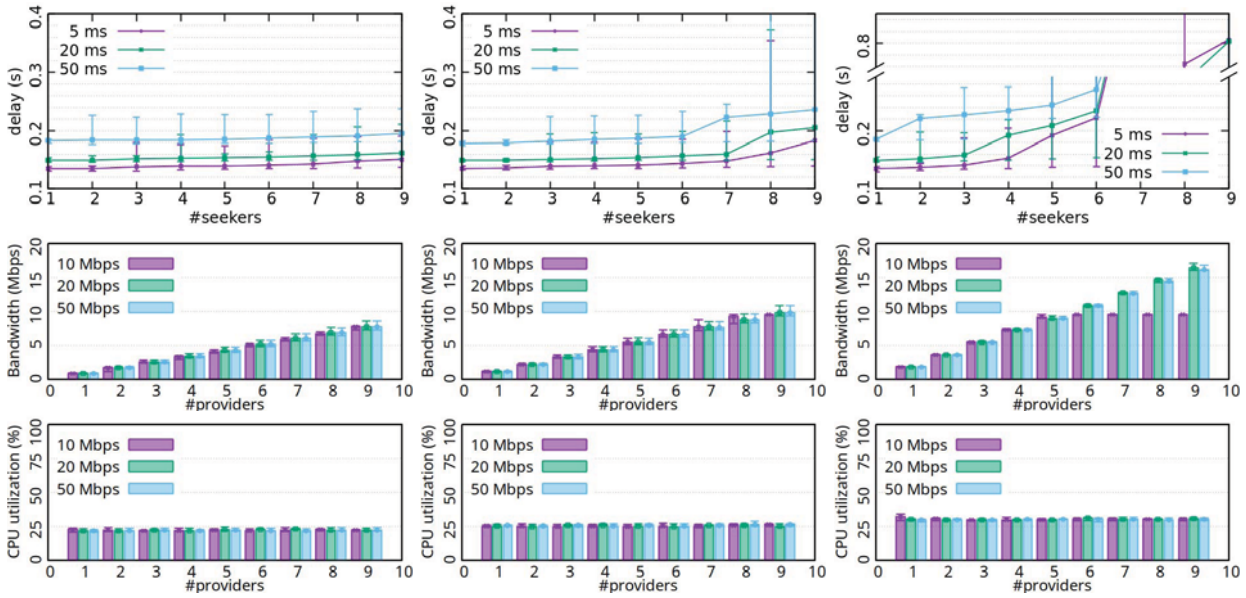
CPU cost of a single instance can be expressed as follows:

$$c_{\text{hive}} \approx c_{\text{decode}} + c_{\text{detect}} + \left(\sum_{\mathbb{P}} c_{\text{decode}}\right) + c_{\text{copy}} \tag{2}$$

$$c_{\text{no-hive}} \approx \sum_{\mathbb{A}} \left(c_{\text{decode}} + c_{\text{detect}}\right) \tag{3}$$

In the hive case, the video stream is decoded once in the provider interface to run face detection then as many times as $\mathbb{P}$ in the seeker interface to decode all received streams, and finally there is the cost of copying the frames through the hive pipeline including the network. We ignored the costs of the applications themselves since, in the hive case with our particular applications, they are very simple and negligible compared to the other costs, e.g. App2 just adds up the counts of face detection scores above a threshold in the provider-computed metadata. In the non-Hive case, each application decodes the video stream and runs the face detection independently.

The local configuration shows the computational gain of the hive achieved through the elimination of redundancy. Looking at Fig. 12 in light of equations (2) and (3), for the no-hive case, c_{detect} is independent of resolution. The cost of decoding c_{decode} should be dependent on resolution, but we see that cost increases only minimally with the resolution, which indicates c_{detect} is dominating. In the hive case, note $|\mathbb{P}| = 1$. There is a noticeable overhead compared to the 1-App no-hive case, but adding applications causes virtually no change, verifying our assumptions that application cost is negligible. On the other hand, increasing video resolution increases the cost of the hive. Given that c_{detect} is independent of resolution and c_{decode} is minimal as shown by the no-hive case, it seems that c_{copy}, the cost of copying the video stream through the stages of the hive pipeline, is the main overhead compared to non-hive. In summary, with n seekers, where the no-Hive case costs some



360p 480p 720p

Fig. 14: Many seekers, one provider results

kn, the hive costs a$k + c$ where c is a function of data rate (video resolution). This is significant save in CPU utilization.

To ensure the Hive is advantageous even if we introduce network overhead, we also run the networked configuration and observed a similar pattern. Comparing CPU utilization in this case, however, is tricky as comparing laptop CPU to Pi CPU is not possible directly. We weighted the results by benchmarking each CPU and observed the same trend as above but omitted the graphs for brevity. Nevertheless, We note that in this scenario, the Bees do not have a local video source, so without middleware like the Hive, they would not even function simultaneously.

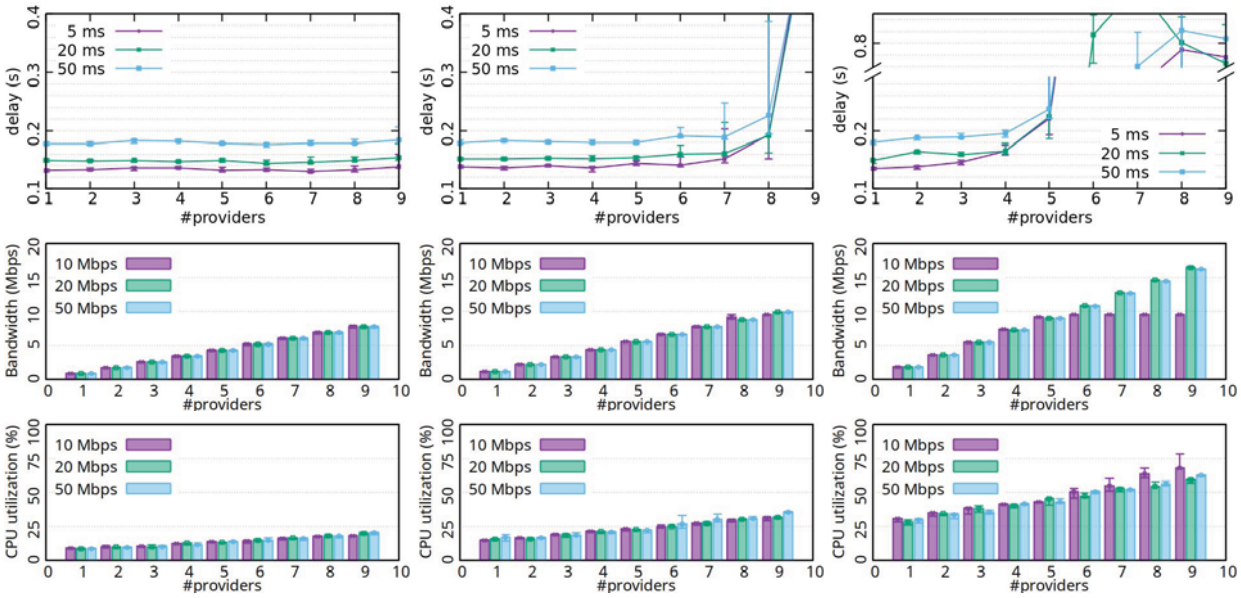
Next, we evaluate the impact of the hive on delay. We repeat the setup in scenario B but fixing the frame rate at a nominal value of 12 and comparing the hive delay with non-hive delay. The non-hive setup is the same as the networked configuration we discussed earlier. The results are summarized in Figure 13. We notice our hive prototype adds a delay overhead ranging between 0.3 s and 0.4 s which are attributable to the flexibility and agility the hive provides, in contrast to the non-hive setup which is hardcoded to stream to fixed targets. Also, we observe the hive delay stays relatively constant as we vary our parameters, which demonstrates that the bulk of the delay is a one-off cost that diminishes with scale.

C. Scalability Testing

1) Experimental Set-up: For scalability, we run 10 instances of the Hive on 10 virtual machines running on Intel Xeon X5690 each assigned two threads and with interconnections of at least 1 GB/s and less than 1 ms round-trip time. The scalability tests allow us to test the versatility of the hive under many different networking conditions and on a larger scale than with physical devices. We use the Linux tool `tc` with the `netem` queuing discipline to simulate the different network conditions and use a pre-recorded video as the hive video input. The parameters are listed in Table I.

2) Scenario A: Many Seekers, One Provider: We examine the impact of n seekers on one provider in terms of delay and CPU load on the provider. In Fig. 14 we plot the average of delay of all n streams, over 10 measurements across n for the 10 Mbps case as well as bandwidth usage and CPU utilization at the provider. We notice the hive has a base delay of about 120 ms which increases with the induced network delay. The delay does not change noticeably as we increase seekers until the total bandwidth exceeds the available bandwidth at which point, it abruptly jumps to much higher values. This is consistent with our modeling of delay in (1). CPU utilization also stays almost constant. Looking at (2), the only component that should be affected is c_{copy} as receivers are added, but the portion of this component on the provider uses a compressed stream, and the branching happens at one point only: at the providing server, hence the change is minimal. We did not plot delay for 20 and 50 Mbps because they do not saturate the bandwidth at 9 seekers and the delay virtually stays the same.

3) Scenario B: One Seeker, Many Providers: We now examine the impact of n providers on one seeker in terms of system delay and CPU load on the seeker. In Fig. 15 we plot the average delay of all n streams, over 10 measurements across n for each combination of network parameters. We also plot the CPU load and bandwidth utilization at the seeker. Results are similar to the many seekers, one provider case, except for CPU utilization which increases uniformly with the number of providers. From (2), we can see that the sum $\sum_{\mathbb{P}} c_{\text{decode}}$ increases and more videos are decoded which is



360p 480p 720p

Fig. 15: One seeker, many providers results

a significant cost. Besides, the portion of copy operation (c_{copy}) on the seeker side copies raw frames which is more expensive than the compressed stream copy which happens on the provider side. Nevertheless, at the scale we were able to test, only exceeding the bandwidth was a limiting factor.

VIII. Conclusion and Future Work

In this paper, we proposed the Hive, an Edge-based IoT middleware that completely decouples applications, sensors and computational units in a manner that allows IoT systems to utilize access to all resources available on the network. The architecture achieves this decouplement on three different stages: data exchange, processing and core. The supporting protocol governs and organizes communication between entities. We then demonstrated the impact of the middleware using audio and video apps in cases of many sensors to one application and one sensor to many applications using physical hardware. We verified the system's ability to handle up to 10 nodes in each scenario under different network conditions using a virtual set-up. We found that the Hive cuts CPU utilization by a factor of $1/n + c$ where one sensor feeds n applications while maintaining a delay below 1s under most conditions. To the extent of our tests, bandwidth was the only factor limiting delay. We conclude that the edge-based design approach that we adopt for the hive increases the performance of IoT real-time applications while improving their overall CPU utilization and having negligible impact on delay. In the future, we want to enhance the scalability of the system by implementing and comparing different load-balancing and computational offloading schemes to find ones that are most suitable for the Hive.

References

[1] N. G., "How many iot devices are there?" 2019. [Online]. Available: https://techjury.net/blog/how-many-iot-devices-are-there/

[2] O. Hamdan, H. Shanableh, I. Zaki, A. R. Al-Ali, and T. Shanableh, "Iot-based interactive dual mode smart home automation," in *2019 IEEE International Conference on Consumer Electronics (ICCE)*, Jan 2019, pp. 1–2.

[3] H. Abdelnasser, K. Harras, and M. Youssef, "A ubiquitous wifi-based fine-grained gesture recognition system," *IEEE Transactions on Mobile Computing*, vol. 18, no. 11, pp. 2474–2487, 2018.

[4] H. Abdelnasser, K. A. Harras, and M. Youssef, "Magstroke: A magnetic based virtual keyboard for off-the-shelf smart devices," in *IEEE SECON*, 2020, pp. 1–9.

[5] M. A. Shah, K. A. Harras, and B. Raj, "Sherlock: A crowd-sourced system for automatic tagging of indoor floor plans," in *IEEE MASS*, 2020.

[6] H. Abdelnasser, M. Youssef, and K. A. Harras, "Magboard: Magnetic-based ubiquitous homomorphic off-the-shelf keyboard," in *IEEE SECON*, 2016, pp. 1–9.

[7] O. Hashem, M. Youssef, and K. A. Harras, "Winar: Rtt-based sub-meter indoor localization using commercial devices," in *IEEE PerCom*, 2020, pp. 1–10.

[8] M. Ibrahim, M. Gruteser, K. A. Harras, and M. Youssef, "Over-the-air tv detection using mobile devices," in *IEEE ICCCN*, 2017, pp. 1–9.

[9] M. A. Shah, B. Raj, and K. A. Harras, "Inferring room semantics using acoustic monitoring," in *IEEE MLSP*, 2017.

[10] G. C. Nobre and E. Tavares, "Scientific literature analysis on big data and internet of things applications on circular economy: a bibliometric study," *Scientometrics*, vol. 111, no. 1, pp. 463–492, 2017.

[11] M. Aazam, K. A. Harras, and S. Zeadally, "Fog computing for 5g tactile industrial internet of things: Qoe-aware resource allocation model," *IEEE Transactions on Industrial Informatics*, vol. 15, no. 5, pp. 3085–3092, 2019.

[12] A. Madushanki, M. Halgamuge, H. Wirasagoda, and A. Syed, "Adoption of the internet of things (iot) in agriculture and smart farming towards urban greening: A review," *International Journal of Advanced Computer Science and Applications*, vol. 10, pp. 11–28, 04 2019.

[13] P. Gupta, A. Pandey, P. Akshita, and A. Sharma, "Iot based healthcare kit for diabetic foot ulcer," in *Proceedings of ICRIC 2019*. Cham: Springer International Publishing, 2020, pp. 15–22.

[14] M. Aazam, S. Zeadally, and K. A. Harras, "Health fog for smart healthcare," *IEEE Consumer Electronics Magazine*, vol. 9, no. 2, pp. 96–102, 2020.

[15] M. F. Al-Sa'D, M. Tlili, A. A. Abdellatif, A. Mohamed, T. Elfouly, K. Harras, M. D. O'Connor *et al.*, "A deep learning approach for vital signs compression and energy efficient delivery in mhealth systems," *IEEE Access*, vol. 6, pp. 33 727–33 739, 2018.

[16] A. Emam, A. A. Abdellatif, A. Mohamed, and K. A. Harras, "Edge-health: An energy-efficient edge-based remote mhealth monitoring system," in *IEEE WCNC*, 2019, pp. 1–7.

[17] A. Saeed, A. Abdelkader, M. Khan, A. Neishaboori, K. A. Harras, and A. Mohamed, "Argus: realistic target coverage by drones," in *ACM/IEEE IPSN*, 2017.

[18] Ó. Blanco-Novoa, P. Fraga-Lamas, M. A. Vilar-Montesinos, and T. M. Fernández-Caramés, "Towards the internet of augmented things: An open-source framework to interconnect iot devices and augmented reality systems," in *Multidisciplinary Digital Publishing Institute Proceedings*, vol. 42, no. 1, 2019, p. 50.

[19] A. Saeed, A. Abdelkader, M. Khan, A. Neishaboori, K. A. Harras, and A. Mohamed, "On realistic target coverage by autonomous drones," *ACM Transactions on Sensor Networks (TOSN)*, vol. 15, no. 3, pp. 1–33, 2019.

[20] A. Saeed, M. Ammar, E. Zegura, and K. Harras, "If you can't beat them, augment them: Improving local wifi with only above-driver changes," in *IEEE ICNP*, 2018.

[21] R. Shakeri, M. A. Al-Garadi, A. Badawy, A. Mohamed, T. Khattab, A. K. Al-Ali, K. A. Harras, and M. Guizani, "Design challenges of multi-uav systems in cyber-physical applications: A comprehensive survey and future directions," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 4, pp. 3340–3385, 2019.

[22] A. Essameldin and K. A. Harras, "The hive: An edge-based middleware solution for resource sharing in the internet of things," in *MobiCom Smart Objects Workshop*, 2017, pp. 13–18.

[23] N. Sinha, K. E. Pujitha, and J. S. R. Alex, "Xively based sensing and monitoring system for iot," in *2015 International Conference on Computer Communication and Informatics (ICCCI)*. IEEE, 2015, pp. 1–6.

[24] E. Latronico, E. A. Lee, M. Lohstroh, C. Shaver, A. Wasicek, and M. Weber, "A vision of swarmlets," *IEEE Internet Computing*, vol. 19, no. 2, pp. 20–28, 2015.

[25] J. Soldatos, N. Kefalakis, M. Hauswirth, M. Serrano, J.-P. Calbimonte, M. Riahi, K. Aberer, P. P. Jayaraman, A. Zaslavsky, I. P. Žarko *et al.*, "Openiot: Open source internet-of-things in the cloud," in *Interoperability and open-source solutions for the internet of things*. Springer, 2015, pp. 13–25.

[26] P. Hofmann and D. Woods, "Cloud computing: The limits of public clouds for business applications," *IEEE Internet Computing*, vol. 14, no. 6, pp. 90–93, 2010.

[27] M. Aazam, S. Zeadally, and K. A. Harras, "Deploying fog computing in industrial internet of things and industry 4.0," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 10, pp. 4674–4682, 2018.

[28] S. Mubeen, P. Nikolaidis, A. Didic, H. Pei-Breivold, K. Sandström, and M. Behnam, "Delay mitigation in offloaded cloud controllers in industrial iot," *IEEE Access*, vol. 5, pp. 4418–4430, 2017.

[29] P. Ferrari, A. Flammini, E. Sisinni, S. Rinaldi, D. Brandão, and M. S. Rocha, "Delay estimation of industrial iot applications based on messaging protocols," *IEEE Transactions on Instrumentation and Measurement*, vol. 67, no. 9, pp. 2188–2199, 2018.

[30] A. Mtibaa, K. A. Harras, and A. Fahim, "Towards computational offloading in mobile device clouds," in *IEEE CloudCom*, vol. 1, 2013, pp. 331–338.

[31] K. Habak, E. W. Zegura, M. Ammar, and K. A. Harras, "Workload management for dynamic mobile device clusters in edge femtoclouds," in *ACM/IEEE SEC*, 2017, pp. 1–14.

[32] J. Wang, S. Pambudi, W. Wang, and M. Song, "Resilience of iot systems against edge-induced cascade-of-failures: A networking perspective," *IEEE Internet of Things Journal*, vol. 6, no. 4, pp. 6952–6963, Aug 2019.

[33] V. Prokhorenko and M. Ali Babar, "Architectural resilience in cloud, fog and edge systems: A survey," *IEEE Access*, vol. 8, pp. 28 078–28 095, 2020.

[34] K. Habak, M. Ammar, K. A. Harras, and E. Zegura, "Femto clouds: Leveraging mobile devices to provide cloud service at the edge," in *IEEE CLOUD*, 2015, pp. 9–16.

[35] K. Habak, C. Shi, E. W. Zegura, K. A. Harras, and M. Ammar, "Elastic mobile device clouds: Leveraging mobile devices to provide cloud computing services at the edge," *Fog for 5G and IoT*, p. 159, 2017.

[36] H. K. Gedawy, K. Habak, K. Harras, and M. Hamdi, "Ramos: A resource-aware multi-objective system for edge computing," *IEEE Transactions on Mobile Computing*, 2020.

[37] H. Gedawy, K. A. Harras, K. Habak, and M. Hamdi, "Femtoclouds beyond the edge: The overlooked data centers," *IEEE Internet of Things Magazine*, vol. 3, no. 1, pp. 44–49, 2020.

[38] A. R. Elias, N. Golubovic, C. Krintz, and R. Wolski, "Where's the bear? - automating wildlife image processing using iot and edge cloud systems," in *2017 IEEE/ACM Second International Conference on Internet-of-Things Design and Implementation (IoTDI)*, April 2017, pp. 247–258.

[39] A. M. Rahmani, T. Nguyen gia, B. S. Negash, A. Anzanpour, I. Azimi, M. Jiang, and P. Liljeberg, "Exploiting smart e-health gateways at the edge of healthcare internet-of-things: A fog computing approach," *Future Generation Computer Systems*, 02 2017.

[40] "Cisco edge intelligence." [Online]. Available: https://www.cisco.com/c/en/us/solutions/internet-of-things/edge-intelligence.html

[41] "The future of computing: intelligent cloud and intelligent edge." [Online]. Available: https://azure.microsoft.com/en-us/overview/future-of-cloud/

[42] "Apache-edgent," 2016. [Online]. Available: https://edgent.incubator.apache.org/

[43] A. Javed, K. Heljanko, A. Buda, and K. Främling, "Cefiot: A fault-tolerant iot architecture for edge and cloud," in *2018 IEEE 4th World Forum on Internet of Things (WF-IoT)*, 2018, pp. 813–818.

[44] A. Elgazar, M. Aazam, and K. Harras, "Edgestore: Leveraging edge devices for mobile storage offloading," in *IEEE CloudCom*, 2018, pp. 56–61.

[45] A. Saeed, M. Ammar, K. A. Harras, and E. Zegura, "Vision: The case for symbiosis in the internet of things," in *Proceedings of the 6th International Workshop on Mobile Cloud Computing and Services*. ACM, 2015, pp. 23–27.

[46] A. E. Elgazar and K. A. Harras, "Enabling seamless container migration in edge platforms," in *ACM CHANTs*, 2019, pp. 1–6.

[47] A. E. Elgazar, M. Aazam, and K. A. Harras, "{SMC}: Smart media compression for edge storage offloading," in *2nd {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 19)*, 2019.

[48] A. Elgazar and K. Harras, "Teddybear: Enabling efficient seamless container migration in user-owned edge platforms," in *IEEE CloudCom*, 2019, pp. 70–77.

[49] C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos, "Context aware computing for the internet of things: A survey," *IEEE Communications Surveys Tutorials*, vol. 16, no. 1, pp. 414–454, 2014.

[50] C. Kamienski, F. Borelli, G. Biondi, W. Rosa, I. Pinheiro, I. Zyrianoff, D. Sadok, and F. Pramudianto, "Context-aware energy efficiency management for smart buildings," in *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, 2015, pp. 699–704.

[51] R. Fallahzadeh, Y. Ma, and H. Ghasemzadeh, "Context-aware system design for remote health monitoring: An application to continuous edema assessment," *IEEE Transactions on Mobile Computing*, vol. 16, no. 8, pp. 2159–2173, 2017.

[52] "Sierra one technologies." [Online]. Available: https://www.sierraonetech.com/

[53] "Meet from literally anywhere: Zoom virtual background," 2020. [Online]. Available: https://blog.zoom.us/wordpress/2016/09/23/zoom-virtual-background/

[54] "Introducing affectiva's emotion recognition through speech." [Online]. Available: https://blog.affectiva.com/introducing-affectivas-emotion-recognition-through-speech

[55] "Vokaturi emotion recognition by speech." [Online]. Available: https://vokaturi.com/

[56] H. Gedawy, K. Habak, K. Harras, and M. Hamdi, "An energy-aware iot femtocloud system," in *2018 IEEE International Conference on Edge Computing (EDGE)*. IEEE, 2018, pp. 58–65.

[57] M. Franceschetti and J. Bruck, "A leader election protocol for fault recovery in asynchronous fully-connected networks," California Institute of Technology, Tech. Rep., 1998.