Incremental Deployment and Migration of Geo-Distributed Situation Awareness Applications in the Fog

Enrique Saurez, Kirak Hong*, Dave Lillethun, Umakishore Ramachandran
Georgia Institute of Technology
Atlanta, GA, USA
{esaurez,hokira}@gatech.edu,
{davel,rama}@cc.gatech.edu

Beate Ottenwälder†
Institute of Parallel and Distributed Systems
University of Stuttgart, Stuttgart, Germany
beate.ottenwaelder@ipvs.uni-stuttgart.de

ABSTRACT

Geo-distributed Situation Awareness applications are large in scale and are characterized by 24/7 data generation from mobile and stationary sensors (such as cameras and GPS devices); latency-sensitivity for converting sensed data to actionable knowledge; and elastic and bursty needs for computational resources. Fog computing [7] envisions providing computational resources close to the edge of the network, consequently reducing the latency for the sense-process-actuate cycle that exists in these applications. We propose *Foglets*, a programming infrastructure for the geo-distributed computational continuum represented by fog nodes and the cloud. Foglets provides APIs for a spatio-temporal data abstraction for storing and retrieving application generated data on the local nodes, and primitives for communication among the resources in the computational continuum. Foglets manages the application components on the Fog nodes. Algorithms are presented for launching application components and handling the migration of these components between Fog nodes, based on the mobility pattern of the sensors and the dynamic computational needs of the application. Evaluation results are presented for a Fog network consisting of 16 nodes using a simulated vehicular network as the workload. We show that the *discovery and deployment* protocol can be executed in 0.93 secs, and *joining* an already deployed application can be as quick as 65 ms. Also, QoS-sensitive proactive migration can be accomplished in 6 ms.

CCS Concepts

•Computer systems organization → Distributed architectures; Cloud computing; •Software and its engineering → Distributed systems organizing principles; Distributed programming languages; **•Networks →** *Network architectures;* Sensor networks;

Keywords

fog computing, cloud computing, programming model, The Internet of Things, situation awareness applications

1. INTRODUCTION

The pervasive deployment of static and mobile sensors and actuators; and the computational fabric for gathering and processing sensor data are opening up opportunities for prototyping a variety of situation awareness applications including emergency response, disaster recovery, surveillance, and traffic congestion. Indeed, one could argue that we are witnessing a shift from the Information Age to the *Intelligence Age* [17]. One of the prime enablers for this shift is, of course, the ubiquity of connected hardware devices, often referred to as the *Internet of Things (IoT)*, the network of physical objects or "things" embedded with electronics, software, sensors, and connectivity. Situation awareness applications use the IoT fabric to convert information to intelligence as a natural evolution of the connected world imagined by the work in Mobile Information Access by Satyanarayanan [27] and the vision of ubiquitous computing imagined by Weiser [34].

The applications that can be built to leverage the IoT infrastructure are limited only by human imagination. However, there is a gap between the hardware connectivity at the physical level and the programming and dynamic resource requirements of the applications. Situation awareness applications are geo-distributed, latency-sensitive, data intensive, involve heavy-duty processing, run 24/7, and result in actuation with possible re-targeting of sensors. Further, they are bursty in resource requirements across space (e.g., cameras in the vicinity of an accident on the highway leading to congestion on the roadways) and time (e.g., rush hour traffic versus late night traffic); thus, the resource needs of the applications are not statically determinable across space and time. Although utility computing could be a cost-effective solution as an execution framework, today's cloud computing platforms have been designed, optimized and used to run traditional n-tier enterprise applications such as web apps, database servers, and batch apps such as MapReduce. Moreover, current cloud platforms have been purposely designed to be *opaque*, making it difficult if not impossible for a developer to specify and satisfy timeliness/latency constraints

*Kirak Hong is currently with Google, Seattle, USA.

†Beate Ottenwälder is currently with Bosch, Germany

DEBS '16, June 20 - 24, 2016, Irvine, CA, USA

ACM 978-1-4503-4021-2/16/06…$15.00

DOI: http://dx.doi.org/10.1145/2933267.2933317

in their applications through the available cloud interfaces.

Fog computing [7] is an infrastructure model for situation awareness applications running on the IoT fabric. The idea is fairly simple and intuitive. In order to meet the latency and scalability requirements of emerging latency-sensitive applications, the utility computing model offered by the cloud platforms should be extended to the edge of the network and the resources should become location-aware. Fog computing is a natural evolution to the utility computing model that recognizes that (a) the interfaces available in the cloud are not friendly to latency-sensitive applications, and (b) the communication latency from the sensing sources and actuation points to the cloud may be too prohibitive. It extends the cloud model by leveraging the computational resources in the network both for compute and storage. Although fog computing is a promising approach to address the latency constraints of situation awareness applications, it has to be augmented with the right distributed programming model, which is the focus of this paper.

For seamlessly dealing with the resource continuum offered by the IoT fabric, we need the right primitives that allow the placement of application components (generated by such stream programming models), data movement among the components and migration of computation and state commensurate with the mobility pattern of the sensors (e.g., self-driving cars). In this paper, we propose the *Foglets* programming model that facilitates distributed programming across the resource continuum from the sensors to the cloud. Foglets supports four main functionalities. Firstly, it automatically discovers fog computing resources at different levels of the network hierarchy and deploys application components onto the fog computing resources commensurate with the latency requirements of each component in the application. Secondly, it supports multi-application collocation on any compute node. Thirdly, it provides communication APIs for components of the application that are deployed at different physical levels of the network hierarchy to communicate with one another to exchange application state. Lastly, it supports both latency- and workload-driven resource adaptation and state migration over space (geographic) and time to deal with the dynamism in situation awareness application[1].

In the rest of the paper, we survey related work (Section 2), the details of the foglets programming infrastructure (Section 3), the system architecture (Section 4), the implementation details (Section 5), the evaluation results (Section 6), and concluding remarks (Section 7).

2. RELATED WORK

There is limited prior art in distributed programming models for situation awareness applications executing over the IoT fabric consisting of sensors, edge nodes, and the cloud. Proposals such as cloudlets [28] and Mobile Edge Computing (MEC) [20] help in off-loading latency-sensitive parts of the computation from the cloud to the edge computing node. Stream-oriented programming models [10, 21, 31, 26] have been proposed for small instances of situation awareness applications. The application is expressed as a stream graph consisting of computation vertices and communication edges. Once a programmer provides the necessary information including a stream graph, the underlying stream processing system manages the statically configured computational resources to execute the stream graph. In our prior research Hilley, et al. [13] have investigated programming idioms that elevate *time* as a first class entity at the application level, in the *persistent temporal streams (PTS)* system, which handles seamless integration of live and archived streams and automatic application-specified persistence of streams. Regarding the stream-oriented programming models, there have been other proposals for supporting edge computing for IoT applications. Belli, et al. [5] propose and implement a graph-based system architecture for processing real-time data coming from smart objects. Papageorgiou, et al. [25] present a framework for applying user-defined reduction function to data emanating sensors to reduce the downstream bandwidth requirements. Nisihio [22] presents a unified framework for heterogeneous resource sharing wherein the services are modeled as tasks and the resource requirements are modeled as time quanta to enable efficient execution.

Foglets is complementary to the aforementioned proposals for structuring IoT applications in that it can serve as the systems infrastructure for orchestrating the deployment, communication, and migration of the application components.

3. THE FOGLETS SYSTEMS MODEL

As a programming model for large-scale situation awareness applications, Foglets has two design goals: The first is to provide a high-level programming model that simplifies development on a large number of heterogeneous devices distributed over a wide area. The second goal is to provide an execution environment that enables incremental and flexible provisioning of resources from the sensors to the cloud.

3.1 System Assumptions

Foglets assumes the existence of a computational continuum that includes sensors and sensor platforms (such as cameras and connected vehicles), on-demand computing instances in a fog computing infrastructure, and a compute cloud with an Infrastructure as a Service (IaaS) interface (see Figure 1). The physical devices are placed at different levels of the network hierarchy from the edge to the core network. We assume each device is associated with a certain geophysical location through a localization technique such as GPS.

For the fog computing infrastructure, we assume physical devices called *fog computing nodes* are placed in the network infrastructure. For example, specialized routers can accommodate generic application computing in addition to packet forwarding, while dedicated computing devices can also be placed within the network for the sole purpose of fog computing. We further assume that the fog provides a programming interface that allows managing on-demand computing instances, similar to an IaaS cloud, including creating and terminating computing resources for a specific geospatial region and at a certain level of network hierarchy. Each computing instance has certain system resource capacities such as CPU speed, number of cores, memory size, and storage capacity, as specified by Foglets. Once computing instances

[1]A preliminary version of the Foglets programming model (dubbed "mobile fog") was presented in an earlier workshop [15]. The focus of the DEBS paper is the implementation of the APIs presented in our earlier work, extending it with discovery and migration algorithms, and conducting experimental evaluation.

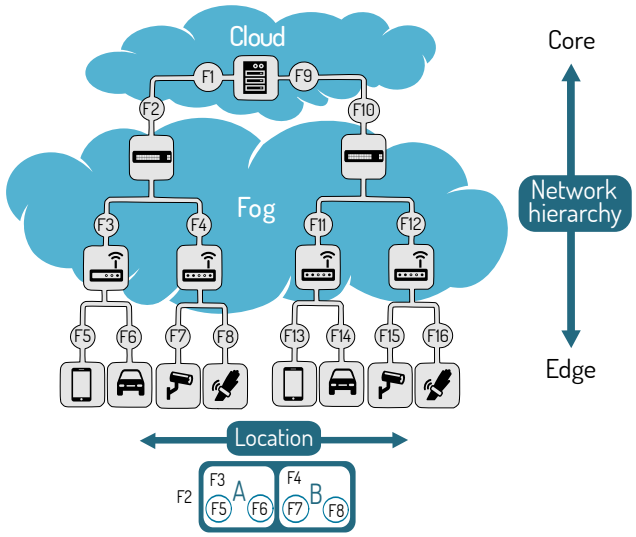

Figure 1: Fog+Cloud Infrastructure: Computational resources exist in the edge and the core routers in addition to the cloud. F<i> denotes an application component launched on a specific computational resource. A and B denote geographical regions.

are created in the fog, Foglets can use the instances to execute application code.

3.2 Application Model

Many large-scale future Internet applications require location and hierarchy-aware processing to handle the data streams from widely distributed edge devices. In Foglets, an application consists of distributed Foglet processes (each representing an application component that are mapped onto distributed computing instances in the fog and cloud, as well as various edge devices. Each process performs application-specific tasks such as sensing and aggregation with respect to its location and level in the network hierarchy.

Foglets exposes the physical hierarchy of devices through a logical hierarchy of Foglet processes shown in Figure 1. As shown in the figure, a process on an edge device is a leaf node while a process in the cloud is the root node of a hierarchy, while processes on fog computing nodes are intermediate nodes. A connection between two processes indicates a communication path allowed through the Foglets communication API, which is not necessarily a direct physical connection, but rather a logical one. Since communication costs within a data center are relatively inexpensive, root processes are connected to each other in a mesh network (e.g., F1 and F9 in the cloud).

Each Foglet process handles the workload for a certain geospatial region. For instance, Figure 1 shows processes F5 and F6 launched on a smartphone and a connected vehicle, respectively, within region A. These leaf processes are connected to process F3 running on an edge fog node that covers the region A. Similarly, processes F3 and F4 are connected to F2 running on a core fog node that covers a region encompassing A and B.

Interface	Description
void send up (message m)	Sends a message asynchronously from a child node to its parent node.
void send down (message m)	Sends a message asynchronously from a parent node to a child node.
void send_to (message m, node destination)	Sends a message to a specific destination node.
set<object> get(key k, location l, time t)	Get the application data that matches a key, location range, and time range.
void put object(object o, key k, location l, time t)	Put application data associated with a key, location, and time.

Table 1: Foglets API. The first three are communication primitives. The last two are for manipulating the local object store.

Handler	Description
void on_send_up (msg m)	Called when a new message arrives from a child node.
void on_send_down (msg m)	Called when a new message arrives from a parent node.
void on_receive_from (msg m, node source)	Called when a new message arrives from a peer node.

Table 2: Foglets Handlers. Application handlers that are invoked by the Foglets runtime on message arrival

3.3 API

In Foglets, application code consists of a set of event handlers that the application must implement (Table 2) and a set of functions that applications can call (Table 1). Running processes on different devices can communicate with each other using our hierarchical communication API, *send_up()* and *send_down()*, as well as a point-to-point communication API, *send_to()*. We provide the hierarchical communication API to encourage application developers to perform more efficient in–network processing. On the other hand, we provide the point-to-point API to support communication between application components hosted on peer Fog nodes. The event handlers are invoked by the Foglets runtime system upon certain events. For example, *on_send_down()* is invoked on a child node when when a message arrives from a parent. Foglets runs the same application code on various devices, including smartphones, smart cameras, connected vehicles, and the computing nodes in the fog and the cloud. This symmetric design simplifies large–scale application development since a developer does not need to write different programs for heterogeneous devices with different connectivity.

An application stores its application–specific data in a local object store called the *spatio–temporal object store*. An application can store its objects tagged by type, location, and time using *put_object()* and *get_object()*. For example, a traffic monitoring application may store detected license plate numbers, tagged by the detection time, camera location, and type *LicensePlateNumber*.

There are other API calls (not shown in Table 1) to query the available system resources at a fog node, as well as location-specific information. Similarly, there are other han-

dlers (not shown in Table 2) mainly for application initialization, pre-processing prior to migration at a source fog node, post-processing at the destination fog node upon migration, triggering setup/cleanup activities when a new child joins/leaves a parent, etc.

Once the code is written, an application developer compiles the code to generate a Foglet process image that can be deployed with an associated unique identifier called an *appkey*. With the appkey, a developer can manage the application using the management interfaces provided by Foglets. To launch an application, a developer invokes *start_app()* with five parameters. The first parameter, *appkey*, specifies the application code to deploy. *Region* specifies a geospatial region where the application will run. *Level* specifies the total number of levels in the hierarchy of Foglets processes. For instance, a video surveillance application may need three levels of hierarchy to perform motion detection at smart cameras, face recognition at fog computing instances, and aggregation of identities at a cloud computing instance. The *Capacity* parameter specifies the class of on–demand computing instances at each level of the network hierarchy. The last parameter, *QoS*, indicates communication latency requirements at each level of network hierarchy. Foglets uses this parameter to find appropriate upper-tier computing resources for hosting upper-level application components. In the previous example, each fog computing instance requires high CPU capacity for face recognition while the cloud computing instance requires high storage capacity to record the location of each individual over time.

Unlike the computing instances in the fog and the cloud, edge devices join an application by invoking *connect_fog()* with the appkey. As a result, the edge device creates a local Foglet process, using its local system resources, that connects to the Foglet process on a fog computing instance covering the location of the edge device.

Foglets runtime automatically discovers the appropriate fog computing node to host an application component for processing sensor data from an edge device. Further, if an edge device is mobile, the connection from the edge device may have to be changed from one fog computing node to another as the location of the edge device changes. Both the discovery and migration of application components (including the state transfer of the object store from one fog node to another) are automatically managed by the Foglets system architecture to be described next.

4. SYSTEM ARCHITECTURE

4.1 Design Space Exploration

We first explore how we can leverage existing prior research in the design of Foglet system architecture.

4.1.1 Hosting Application Components

An application may consist of several components. Besides, there may be multiple applications that are simultaneously using the IoT infrastructure. Resource isolation (especially, memory) across components of different applications is a must for the security and integrity of the individual applications; even within an application such isolation between the application components is good from the point of view of bug proliferation and performance tuning. We will explore the pros and cons of Virtualization technology, specifically full blown *virtual machines* and *containers*, from the point of view of hosting application components.

Application Components hosted in Virtual Machines.

Virtualization is a staple of data centers and is the basis for accountability and containment of resource usage. Additionally, virtual machine migration enables load balancing and resource provisioning in data centers. More recently, Xen [18] and VMWare [3] have implemented *live* migration of VMs with downtimes ranging from tens of milliseconds to a second. satyanarayanan, et al., [28] propose the concept of *cloudlet* to exploit standard VM technology in edge computing. Ha, et al. [11] discuss the limitations of live VM migration for use on edge devices. Both Ha, et al. [11] and Yao, et al. [36] discuss approaches for efficient cloudlet migrations. There are newer approaches to VM migration with less involvement of the hypervisor [19] or with a reduction in the startup time by using delta encoding between an original VM and the changes that occurred during execution [29]. However, despite such advances in VM migration techniques, given the latency requirements of situation awareness applications, full blown virtualization may be impractical for hosting application components in the Foglets environment. Recent advances in container-based virtualization simplify the deployment of applications while isolating them from one another [8].

Application Components hosted in Containers.

Container-based virtualization [6, 30, 24] is an alternative to full-blown virtual machines. It is finding increasing adoption in mainstream operating systems as a means of providing isolation and resource control. Docker [16] has emerged as a standard for Linux containers. Google, IBM/Softlayer, and Joyent are all examples of extremely successful public cloud platforms using containers [6].

In the container technology, applications (contained in the containers) share the OS (and where appropriate the binaries and libraries). Consequently, the memory footprint of containers is significantly smaller than in a hypervisor environment, allowing hundreds of containers to be hosted on a physical host. Since the containers use the host OS as a base for system services, restarting a container (upon container migration) does not necessitate restarting the OS [6].

Further, once the Linux-based container is installed, only the extra difference layers, such as additional binaries and libraries, need to be migrated to correctly execute the handlers in the context of Foglets. Docker uses the *union file system* [35] to combine these layers into a single image [9]. For all these reasons, containers are ideal for hosting applications components in the Foglets programming model.

4.1.2 Selection Strategies for Migration

There is prior work in selecting *where* and *when* to migrate an application component (or the entire application) in a mobile environment. Ottenwalder, et al. [23] present a placement and migration method for cloud and fog resources, ensuring application-defined end-to-end latency restrictions and reducing the network utilization by planning the migration ahead of time. Urgaonkar et al. [32] model the migration problem as a Markov Decision Problem (MDP). They decouple the initial MDP into two independent MDPs that allows the problem to be solved using a Lyapunov optimization. Wang, et al. [33] propose further refinements to this solution developing a polynomial time algorithm with

some relaxation in the system assumptions regarding the error bounds on the costs of hosting and migration. In the Foglets implementation, any of these strategies could be used for deciding on when and where to migrate. Additionally, we take the resource pressure on the nodes as in important component in taking the migration decisions.

4.2 Foglets Runtime

There are four entities in the Foglets runtime system:

1. **Discovery Server:** This is a partitioned name server that maintains a list of fog nodes available for hosting application components at different levels of the network hierarchy for a given geographic area.

2. **Docker Registry Server:** This is a server (replicated for different geographical areas) that contains the binaries for the applications that have been launched on the Foglets infrastructure. As we mentioned earlier (Section 3.3), every application has a unique *appkey*. Upon launch, the registry maintains a key-value store for the binary images of the application components for each level of the network hierarchy. The key is the *appkey* and the value is the set of application binary images for the different levels. Both the discovery and registry servers may be hosted on the same physical machine.

3. **Entry Point Daemon:** This daemon is started on each non-leaf fog node at system boot time. This daemon executes directly on top of the host OS in the fog node and awaits requests from the immediately lower level in the fog hierarchy to host a parent (application component) for a child (application component). This daemon periodically sends "I am alive" message to the Discovery server so that the information in the Discovery server remains up to date. It participates in the discovery and migration protocols to be described shortly.

4. **Worker Process:** This is the process that will carry out the functionality contained in a particular application component assigned to it. For each leaf node of the launched application, the Foglets runtime will spawn a worker process to execute on the edge device (e.g., a camera) assigning the appropriate application component that has to be hosted on the device. For each non-leaf node, worker processes will be incrementally created with the dynamics of the applications as will be described shortly.

4.3 Launching an Application

To make the discussion concrete, let us continue with the simple example application we mentioned in Section 3.3: a simple video surveillance application needing three levels of hierarchy to perform motion detection at smart cameras, face recognition at fog computing instances, and aggregation of identities collected over space and time at a cloud computing instance. The roadmap for developing and launching the application using Foglets is a 2-step process as follows: (1) The developer writes the application logic for each of detector, face recognizer, and aggregator, as well as the handlers for each of the three levels. (2) The developer creates the binary images for each of the application component (detector, face, aggregator), and using a system-wide unique *appkey* registers the appkey and the images with the Docker registry server.

To launch the application, as we mentioned before in Section 3.3, the developer would use the API *start_app* specifying the five parameters (appkey, region, level, capacity, QoS). The Foglets runtime will ensure that Fog computing resources at the different levels are up in the region specified by the application. Further, the registry server will retrieve the detector process image from its key-value store (using the appkey) and start up worker processes to host the detector on all the cameras in the region specified. Note that neither the resources nor the worker processes for the upper-tiers are provisioned for this application at the time of launch. Such provisioning will occur incrementally based on the application dynamism, which is the crux of the dynamic discovery and deployment protocol to be discussed next.

Algorithm 4.1 Discovery and Deployment Protocol

1: $candidates \leftarrow$ from Discovery Server
2: Send Discovery ping to each candidate
3: The candidate responds back its Id n and state s,
4: $s \in$ {READY-DEPLOYED,READY,BUSY}
5: Let R be the set of responses (n, s) $\forall$ candidates with n the Id and s the state.
6: Let $S = \{s | s = state(r), \forall r \in R\}$, be the set of all the received states.
7: **if** READY-DEPLOYED $\in$ S **then**
8: run the Join Protocol in Algorithm 4.2
9: **return**
10: **else** ▷ Application is not deployed in area
11: **if** READY $\in S$ **then**
12: $R_{READY} \leftarrow \{c | c \in candidates, state(c) =$ READY$\}$
13: Obtain the best candidate $c_{closest}$ from R_{READY}
14: Send a DEPLOY container message to $c_{closest}$
15: **else** ▷ No available fog node in area
16: Restart the Discovery and Deployment algorithm, increasing the geographical area to be queried from the Discovery Server.
17: **end if**
18: **end if**
19: $r \leftarrow$ response from join to bc.
20: **if** $r = Accept$ **then**
21: Select bc as the new parent
22: **return**
23: **else**
24: $R_{READY} \leftarrow R_{READY} \setminus \{c_{closest}\}$
25: **if** R_{READY} is not empty **then**
26: **go to** 13
27: **else** ▷ No fog node with available resources
28: Restart the Discovery and Deployment algorithm, increasing the geographical area to be queried from the Discovery Server.
29: **end if**
30: **end if**

4.4 Discovery and Deployment Protocol

Discovery has to do with finding the fog computing nodes (matching the capacity constraints) at the right level of the computational hierarchy for hosting an application component. Deployment has to do with spinning up a *Docker container* in a fog node to run a *Worker* process that will carry

out the work of the application component to be hosted on this node. The EntryPoint daemon in each fog node maintains the state of that node, which can be one of: *READY - DEPLOYED* (RD), *READY* (R), and *BUSY* (B). Both RD and R indicates that the fog node has the capacity to host the application component at this level. RD additionally says that the required application component is already launched at this node (i.e., a container with that application component is already present at this node), which would be advantageous in reducing the latency for resource provisioning. The B state, on the other hand, indicates that the fog node does not have resource capacity to accommodate new hosting requests. Resource discovery and provisioning occur incrementally in Foglets. The first time an application component attempts to do a *send_up*, the Foglets runtime at that fog node contacts the Discovery server to obtain a list of fog nodes in the upper-tier commensurate with the level and the capacity requirements stated by the application. The child node then executes a 2-phase join protocol to choose a parent from the list (Figure 2). The pseudo-code for the discovery and deployment algorithm is shown in Algorithm 4.1. Upon getting the list of possible candidate parent fog nodes, the child sends a ping message to each of them. Each of the candidate parents sends back a response with their node-id (n) and state (s). There are three possibilities:

1. The set of potential parent nodes that have the application component already deployed (s = READY-DEPLOYED) in a container is non-empty (Line 8 of Algorithm 4.1). In this case, the Join protocol is called (Algorithm 4.2).

2. There are no nodes with the application component already deployed. However, there are nodes that are ready (s = READY) to accept a child (Line 12 of Algorithm 4.1). In this case, choose the best candidate node from the set of READY nodes to deploy a container hosting the application component as follows:

$$R_{READY} = \{c | c \in R, response(c) = READY\}, \quad (1)$$

choose the closest fog node,$c_{closest}$, using equation (2),

$$\min_{\forall e \in R_{READY}} distance(e, child) \quad (2)$$

and initiate the second phase of the Deployment protocol (Line 14 of Algorithm 4.1), wherein the child node sends a DEPLOY message to $c_{closest}$ and waits for the response, which could be either $ACCEPT$ or $REJECT$. If the response is $ACCEPT$, then the child has successfully joined the parent. If the response is $REJECT$, then the child chooses the next closest candidate node in the set $R_{READY} \setminus \{c_{closest}\}$ and sends a DEPLOY message to it. If all the candidates send $REJECT$ responses, then the Discovery algorithm is reinitialized with a bigger geographical area (currently we double the radius on each iteration).

3. All the candidate nodes are loaded (s = BUSY) and cannot accept any more children. In this case, the Discovery algorithm is reinitialized with a bigger geographical area.

Algorithm 4.2 Join Protocol

```
1: function JOIN PROTOCOL(W_READY-D)
2:     Obtain best candidate bc from the set W_READY-D = {c|c ∈ R, state(c) = READY − DEPLOYED}
3:     Send a Join message to bc
4:     Wait for response w
5:     if w is ACCEPT then
6:         select bc as the parent and start connection
7:     else
8:         W_READY-D ← W_READY-D \ {bc}
9:         if |W_READY-D| > 0 then        ▷ Set is not empty
10:            Choose the next best candidate bc in W_READY-D
11:            go to 3
12:        else
13:            Restart the Discovery and Deployment Algorithm
14:        end if
15:    end if
16: end function
```

4.5 Join Protocol

The join protocol is shown in Figure 2. Algorithm 4.2 gives the pseudo code. The child chooses the best candidate *bc*, which is geographically close to it to send a join message per the following equation:

$$\min_{\forall e \in W_{READY-D}} distance(e, child) \quad (3)$$

If there are many candidates with equivalent distances, then their current load conditions could be taken into account in the choice. When the candidate node receives the join request for an application component that is already deployed, it queries the worker process in the container associated with that application component. If the Worker process is ready to accept this child, then this node become the parent and an $ACCEPT$ response is sent to the child. If the load conditions have changed since the first phase of the Discovery protocol, the Worker process may deny the join request, in which case the candidate node would reply with a $REJECT$ response to the child. In this case, the child node would try the next best candidate (Lines 8-11 of Algorithm 4.2). It is conceivable that the network state may have changed during the execution of the join protocol in which case the Discovery and Deployment protocol is started all over again (Line 13 of Algorithm 4.2).

Such an incremental application deployment ensures highly adaptive and elastic resource utilization driven by application dynamics and QoS needs from the edge of the network. Application components are deployed at any level of the logical network hierarchy, only if a node in a lower-level executes a *send_up* message. Using the dynamic resource discovery protocol, Foglets incrementally maps the logical hierarchy of an application onto the physical hierarchy of the network infrastructure. The EntryPoint in a fog node deploys an application by launching a Docker container with the Worker process in it to carry out the functionality of the application component for that level.

4.6 Migration

In Foglets, migration of an application component from one fog node to another (at the same level, usually the edge

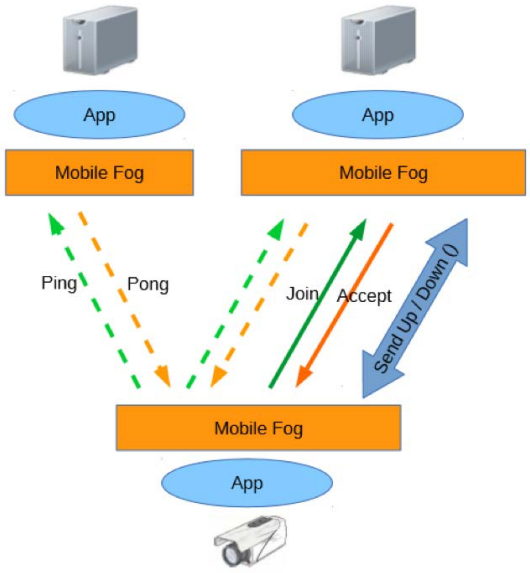

Figure 2: Join Protocol: The Discovery server gives a list of fog nodes in the upper-tier to the requesting child node. The protocol pictorially shown above results in the child choosing a parent to join from the list.

nodes close to the sensors) may be warranted due to one of two reasons:

1. **Meeting QoS expectations:** We expect in the future many situation awareness applications would involve mobile sensors and sensor platforms (e.g., self-driving cars). In this case, the edge fog node hosting the application component that processes the sensor data cannot be statically defined but should be dynamically provisioned to adjust to the mobility pattern of a given application. Thus, meeting latency constraints for processing sensor data could be a trigger for migration of application components.

2. **Load balancing considerations:** In the Foglets system, a given fog node could be hosting application components of multiple situation awareness applications using the same IoT sensing infrastructure. Depending on the situation (e.g., traffic incident, riot, etc.) a particular fog node could experience resource pressure. Thus, resource pressure at a fog node could be a trigger for migration of application components.

There are two inter-related aspects with respect to migration in the Foglets system:

1. **Computation Migration:** This is related to changing the parent of a given child in the network hierarchy (usually the fog node close to the sensors) either due to QoS or load-balancing considerations. To facilitate computation migration, Foglets expects the application to provide two handlers (Table 3), one to be executed at the current parent and the other at the new parent.

   The handler, *on_migration_start(child)*, allows the current parent to package the volatile state of the computation of the parent node with respect to the specific child node. The handler, *on_migration_end(s)*, allows the new parent to initialize its local state using the transferred state for the child. Foglets calls the first handler on the current parent node and ships the state to the new parent node. Once the transfer is complete, it is safe to switch the child to the new parent. The new parent node will start processing *send_up* calls from this transferred child as soon as the initialization of its local state is complete.

2. **State Migration:** This relates to the persistent data that is generated by an application component (in the object store mentioned in Section 3.3), which has to be made available in case the application component is migrated to a new fog node. The state migration is done in parallel with execution in the new node.

Handler	Description
state on_migration_start (child)	Called before a migration process starts. Application code running at the original parent node provides a stream context by returning a state object.
void on_migration_end (state s)	Called after a migration process ends. Application code running at a new parent node can recover a stream context from the provided state object.

Table 3: Foglets Migration API.

4.7 QoS-driven Migration

In Foglets, QoS is specified as the upper bound T on the latency between an application component and its parent. We discuss two mechanisms that exist in Foglets for supporting QoS-driven migration: *proactive*, and *reactive*.

4.7.1 Proactive Migration

There are two parameters associated with proactive migration in the Foglets system: α and β (both between 0 and 1), with $\alpha < \beta$. The values of α and β are chosen commensurate with the QoS needs of an application. The runtime system uses default values for these parameters if an application does not explicitly specify them. The Worker process in the Foglets runtime has a one-to-one relationship with an application component. It is aware of the QoS requirements (latency bound T) of the application component bound to it in the container. The Worker process continually monitors the actual latency experienced on *send_up* and/or ping messages from its children. When the latency from a given child goes above a threshold, $\alpha \cdot T$, proactive migration starts. The Worker process at the current parent will find a suitable candidate parent node from the list of available neighbors at the same level obtained from the Discovery server. The choice is facilitated by the fact that each Worker is periodically exchanging ping times and resource utilization information with its neighbors at the same level. First only the *object store state* associated with this application component will be migrated gradually in anticipation of a change of parent. We call this *state replication*. If and when the actual latency goes above the second threshold, $\beta \cdot T$, then a decision is made to migrate the parent itself. In either case, the choice of the parent to migrate to is based on a number of factors including the geo-location of the child

relative to the future parent, measured ping latencies, and the capacity constraints of the future parent (measured by available uncommitted resources such as CPU and memory, which can be obtained from statistics maintained by the host OS of the parent).

Upon a decision to migrate the parent, the Worker process sends a *Start Migration* message to the candidate future parent. Upon receiving an $ACCEPT$ message from this future parent, migration will proceed in parallel to move the computation and the object store state. On the other hand, if a $REJECT$ message is received for the migration request, then the Worker process will initiate *Start Migration* with the next best candidate in the list of potential future parents.

Once a new parent has been identified to migrate, the parent migration proceeds as we described before with the invocation of the application-specific handlers on the old and new parents to transfer the child to the new parent. In parallel with moving the computation, Foglets also initiates moving the object store state of the application from the old to the new parent. Some of this state may have already been replicated proactively before the parents were switched. In any event, now that the new parent is alive and well, the state can actually be *moved* to the new parent instead of being replicated. There are several opportunities for optimization. First, not all state needs to be moved to the new parent. Since situation awareness applications tend to work mostly with recent data, it may be sufficient to move the most recent historical data to the new parent. If there is a need, the new parent can demand-load older historical data[2].

4.7.2 Reactive Migration

It is conceivable that due to the system dynamics, the proactive migration detailed above does not happen in a timely manner to adhere to the latency requirements of the application. It could even be that a parent fog node has become unresponsive due to overload. In this case, reactive migration may be triggered by the child. That is, the child will decide to find a new parent by going through the Discovery protocol (Algorithm 4.1). Once a new parent has been found, the process of transferring the computation and object store state from the old parent to the new parent will proceed exactly as in proactive migration. The only difference is that the new parent will contact the old parent to initiate the migration.

4.8 Workload-driven Migration

Multiple applications may be collocated in the same fog node. Bursty resource requirements of one application could detrimentally affect the performance of other applications in meeting their respective QoS constraints. Foglets provides a mechanism for workload-driven migration.

To support workload-driven migration, the EntryPoint daemon in each fog node keeps statistics on resource usage by all the containers deployed at this node. The Worker processes associated with these containers periodically report their respective resource usages (CPU, memory, missed deadlines if any) and geospatial information of the children that they are communicating with. Foglets runtime uses the stats provided by the EntryPoint daemon to make migration decisions if it finds the fog node is overloaded. While the QoS-conscious migration techniques deal with individual children migration, workload-driven migration does entire container migration when possible. The system sorts the containers by the number of children each container is connected to. It chooses the top γ containers from this sorted list as candidates for migration. Using the geo-location information of the children catered to by the containers, Foglets chooses to migrate containers with children farther away from the location of the fog node. The intuition is that such children are drifting away and are likely to move to a different parent anyhow. Similar to proactive migration, an overloaded fog node pings its neighbors at the same level in different geo-locations to find a home for the containers it wants to migrate. Migrating whole containers will help reduce the load on the depressed fog node rapidly.

[2]For the sake of conserving space, we do not elaborate on all the details of state movement from the old to the new parent. There are also corner cases to be handled in the runtime system, wherein a sequence of parent moves could result in the object store state being fragmented in multiple previous parents' nodes. Foglets does all the book-keeping to make sure that it has the trail of previous parents to retrieve historical items in any object store in the presence of such migration.

5. IMPLEMENTATION DETAILS

The Foglets system is implemented using C++ with the operating system Ubuntu 12.04. The communication protocols are implemented using the ZMQ [14] library for message delivery and the protobuf library from google [1] to serialize the data. The Foglets implementation uses docker containers [16] and RocksDB [2]. The base image used to develop the container with the Foglets library and the Worker process runtime is *ubuntu:latest* (version 14.04). Instantiating and creating a new instance of a container is fast, given that the images created by a developer will use an image developed on top of the Foglets framework as a base image. This allows a developer to test the different characteristics of the system locally in their development system, without the need to deploy their application on top of the real hardware. Further, if the Foglets container images are pre-installed in a fog node, then to start an application component on that node only the delta (the specific application) needs to be pulled from the Registry server, reducing the overhead. RocksDB has two main features that help in an efficient implementation of the object store: *prefix iterator*, and *read-only access mode*. The prefix iterator allows fast access to the object store (using *time* as the prefix in our implementation), and the read-only access mode eliminates the need for locks during migration of the object store state. Finally, the Discovery server can easily be replicated for scalability since it only maintains connection endpoints that are updated infrequently, namely the fog nodes (and not the IoT devices themselves).

6. EVALUATION

In this section, we begin our evaluation by measuring the costs, in time, associated with launching a container in the Foglets runtime system. Using a workload derived from SUMO traffic simulation to drive our system, we conduct experiments to measure the time for the incremental deployment operations of Foglets (namely, Discover-Join and Discovery-Deployment) for launching application components in the fog infrastructure. Next, we conduct experiments on the migration component of the Foglets system.

The first experiment is to measure the basic cost of switching from one parent to another. We then conduct experiments to show the efficacy of the migration operations of Foglets under two conditions: (a) dynamic workload driven reactive migration, and (b) proactive migration when latency constraints are not met.

6.1 Platform

Our experimental platform is a set of four Penguin Relion 1752 nodes interconnected by 10 Gbps Ethernet. Each Penguin node contains a 2-socket, 6-core, 2.66GHz Intel X5650 hyper-threaded processor, and 48GB RAM. We emulate 16 fog nodes on top of these four very powerful hardware platforms. We run Ubuntu 12.04 on all the nodes as the base OS. The containers we launch on these platforms will use the kernel services of the base OS. We also emulate the Discovery Server and Docker Registry Server on these machines. We use an auxiliary node that serves as the workload generator for our emulated fog infrastructure. This auxiliary node sends messages to the fog nodes emulating the sensor inputs (position information and video) from automobiles plying in the city of Atlanta.

Docker Image	Time (s)
Debian	8.6
Ubuntu	1.07
Foglets base	1.1
Foglets application w/base	18.36
Foglets application already deployed	0.42

Table 4: Startup times for different configurations of Docker images

6.2 Bringing up Foglets System

The Foglets system implementation uses container technology. In this section, we measure the times for "booting up" Foglets. The Foglets base Docker image (containing the runtime for the API calls, the communication libraries, and the Worker process to run the application) is built on top of the Ubuntu official docker image. The Foglets base image will then be used by the developers to host their application component (including the communication handlers) to be deployed at each level of the logical network hierarchy. We measured the times to spin up each layer of this software stack needed at a fog node. As a baseline value, we use the time taken to download and execute two images from the official Docker container, namely, Ubuntu and Debian.

The numbers shown in Table 4 is the average of 100 runs. Here are the base costs for "booting up" Foglets:

1. Pulling the full Debian image from the repository and starting its execution costs 8.6 seconds on an average. With the Debian image already in the system, the docker run-time only needs to download the additional layers to form the Ubuntu image with an average cost to download and start of 1.07 seconds.

2. The Foglets system base image is developed on top of the Ubuntu image, it contains all the required libraries and the runtime system for implementing the Foglets primitives. If the Ubuntu image is already present in the host system, pulling the image and booting up the system takes on an average 1.1 seconds.

3. The reference application we use in our evaluation is vehicular traffic simulation on Atlanta roadways. The application consists of cars sending their positional information and video to their associated fog nodes. The application uses OpenCV for processing the video feeds. Consequently, the library needed for the application is large and serves as a good reference application for our experimental studies to show the efficacy of our system for dealing with large application images to be launched on fog nodes.

   The application with all the libraries and the Foglets handlers is 2.1 GB. Due to its size, downloading and starting the image, even with the Foglets base image present, takes up to 18.36 seconds.

   Downloading the application image needs to happen only once when the application is started, or during the bootup of the Foglets system.

4. If the application image is already downloaded then launching it in the container takes only 0.42 seconds.

5. To bootup Foglets, we first distribute the binaries of the Worker process and EntryPoint daemon to all the fog nodes. Then the system bootup can proceed in parallel to initiate the Worker process in all the leaf nodes and the EntryPoint daemon on the other nodes. On an average, starting the EntryPoint has a latency of 4.17 ms and starting the Worker process has a latency of 40 ms.

6. The Worker process binary size is 45 MB and the EntryPoint binary size is 43.4 MB. We measured a raw throughput of 50 MB/s between the nodes of our experimental platform. Thus, it would take approximately 1 second to send the binaries to their corresponding nodes. Our measured times for bootup is in agreement with the raw throughput measurement. It takes only 1.04 seconds for the Worker processes to be distributed and started in the leaf nodes, and 30 seconds for the EntryPoint daemon deployment and container download (which is a summation of all the numbers in Table 4), both of which can be done in parallel.

6.3 Microbenchmarks

This subsection quantifies the cost of the main operations in the Foglet system, by simulating the movement of cars as leaf nodes.

6.3.1 Workload

As we mentioned in the previous subsection, we use vehicular simulation as the driver application for our experimental study. We simulate the movement of the cars using the traffic simulator SUMO [4], which enables us to model realistic traffic patterns of vehicles on an Atlanta OpenStreetMap graph [12]. Our simulation is a snapshot of the traffic in a rectangular grid of the city of Atlanta (7.7 km x 5.7 km) for 10 minutes using the road network graph, and 600 vehicles for each simulation run (on an average). For each simulation run, we observed 110,166 events, meaning that each vehicle sends 184 events on an average, including both locations and images to be processed. At any instant of time in the simulation, there are on an average 101 vehicles in the area covered by the simulation. Using a grid structure, we

divide the area covered by the simulation into 16 geographic sections and assign a fog node to cover the sensor inputs for each section. As we mentioned earlier, we emulate the 16 fog nodes on the 4 Penguin machines. In order to emulate the movement of the cars into the execution of the Foglets system, the distance of the car to the location of the fog node is included into the latency calculation as shown in equation (4)

$$latency = measured\ latency + \epsilon \cdot distance, \quad (4)$$

where epsilon is chosen such that if a car moves out of the grid section that it is currently in, it is most likely violating the QoS requirement (expressed as a latency constraint).

6.3.2 Discovery, Deployment, and Join Operations

Figure 3 shows a comparison between the cost of the discovery-join and discovery-deployment. We limit the number of candidates to four each time we try to either deploy or join the system; if unsuccessful, the child tries with the next best node in the list or increases the geographical area to be queried. As a reference, the round-trip network latency is around 4 ms for performing a null RPC (a send_up followed by a send_down, shown in Figure 4).

As can be seen in Figure 3, there is considerable variance in the discovery and deployment measurement, where the lower 25% of the deployment operation takes less than 636 ms and the higher 25% of the deployment operation has a latency of more than 1,097 ms. The lowest observed delay is close to the minimum possible of 42 ms, the time required to start a container as shown in Table 4. This huge variance is due to the state machine used for implementing the discovery protocol, in which it waits for responses from all the candidates. If the system is not loaded, then a faster response can be obtained from the nodes involved. The long tail that could happen in the deployment algorithm is upper-bounded by the timeout mentioned in Section 4.4.

The Discovery-Join operation (wherein the application is already running in the container) is much faster. The median for this is 72 ms, with the 75% percentile at 240 ms, which mainly depends on how fast the EntryPoint daemons can respond to the new child in the first phase of the algorithm. The measurements include situations in which a JOIN request is returned with a rejection.

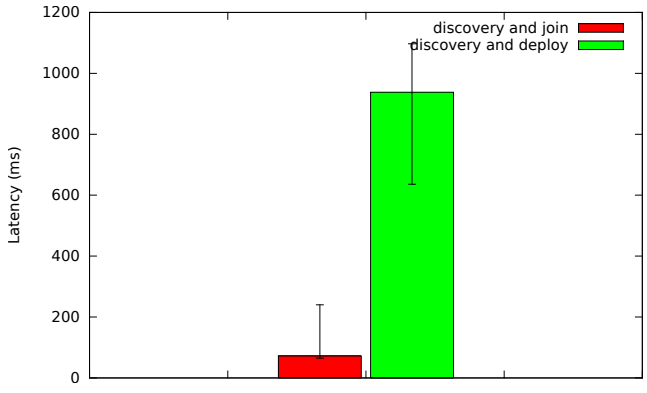

Figure 3: Comparison of Discovery-Join and Discovery-Deployment Operations. Error bars represent the 25% and 75% percentiles

6.3.3 Migration Operation

Figure 4 shows the average cost in milliseconds for selecting and changing to a new parent for a child in proactive migration. The measurement is taken when there is no application state to be packaged and sent to the new parent (i.e., the on_migration_start handler is a null handler), and the application is already deployed in the new parent. As we can be seen from Figure 4, this operation takes roughly three times compared to a round-trip message. The time required for reactive migration is the same as the discovery protocol measurements shown in Figure 3, since it is initiated by the child node.

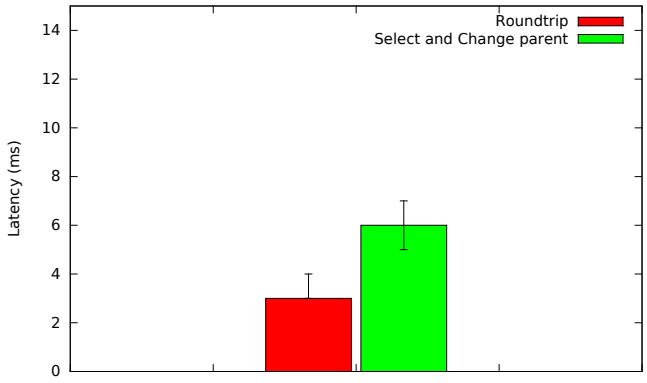

Figure 4: Proactive Migration Operation. As a point of comparison we show the network round-trip time. Error bars represent the 25% and 75% percentiles

6.4 Dynamic workload driven migration

In this experiment, we demonstrate qualitatively and quantitatively the ability of Foglets to do workload driven migration. The experiment uses two fog nodes geographically close to each other. A leaf node could in principle choose either of them as the parent. We allow a new car to enter the geographical area every 0.5 seconds and just stay put (i.e., it is stationary). The first car to appear would join one of the two fog nodes (*fog node 1* in Figure 5). Since now fog node 1 has a container already launched with the application, the subsequent cars entering the same geographical area will prefer to join the same fog node. This is seen in Figure 5. The capacity limit for fog node 1 is reached when 39 cars have joined it. Subsequent join requests will be rejected resulting in the next car joining fog node 2. At this point, we stop introducing new cars into the experiment.

Fog nodes 1 and 2 exchange ping times and capacity information to each other. Due to the resource pressure, fog node 1 starts transferring its children to fog node 2. Since the cars are stationary, the transfer is gradual but in the steady state it can be seen that both fog nodes have roughly an equal number of children.

6.5 Proactive migration

In this experiment, we want to show both qualitatively and quantitatively how Foglets does QoS-sensitive proactive migration. The scenario is a traffic jam. A number of cars are reporting to one fog node initially. But as the traffic jam clears, the cars move in different directions towards

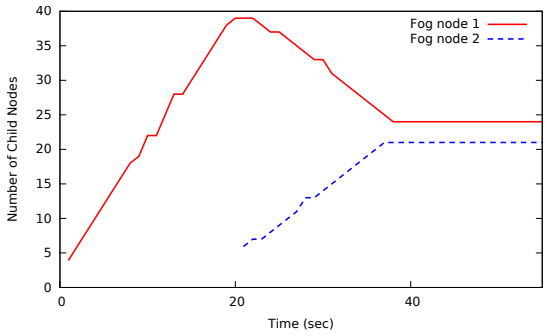

Figure 5: Workload Driven Migration. Over time fog node 2 accepts more children to offload the work from fog node 1.

their respective destinations. We simulate this situation by first spawning a number of cars close to *fog node 1* and let them stay put (i.e., they are stationary). This can be seen in the measured results shown in Figure 6 for the first several tens of seconds. Then we make the cars move away in three different directions starting around 70 seconds into the simulation. Due to the increasing distance from fog node 1, the children start experiencing increasing latencies triggering proactive migration of the children to the fog nodes corresponding to the areas that the cars are moving into. As can be seen in Figure 6, Foglets is able to quickly migrate the nodes, reacting to an increase in the latency experienced by the children. It can be seen that the number of cars stays constant in the graph which is an indication that no messages are missed by the fog nodes and all the leaf nodes are successfully transferred from fog node 1 to other geo-local fog nodes close to the moving cars.

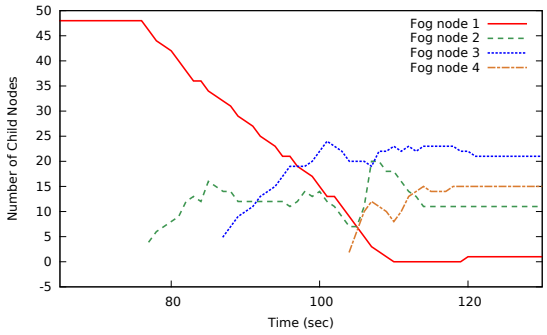

Figure 6: QoS-driven Proactive Migration. Over time the children are migrated to the fog nodes that are geo-local to the children moving in different directions.

7. CONCLUSION

Situation awareness applications are emerging as important drivers of the IoT infrastructure. Provisioning computational resources close to the sensor sources in the IoT infrastructure is crucial to meet the latency constraints of such applications. Fog computing is a good utility computing model to cater to the edge computing needs of situation awareness applications. In this paper, we have proposed a programming infrastructure for the computational continuum extending from the sensors to the cloud called *Foglets*. It provides APIs for app development as a dataflow graph whose nodes can be placed in the different levels of the computational hierarchy commensurate with their latency properties. Foglets provides primitives for communication between the application components, and also embodies algorithms for the discovery and incremental deployment of resources commensurate with the application needs. It also provides mechanisms for QoS-sensitive and workload-sensitive migration of application components due to sensor mobility and application dynamism. Foglets is implemented using container technology and we present performance results to showcase its effectiveness for situation awareness applications. Our immediate future plans include using the Foglets programming model to develop a large-scale surveillance application on top of the camera network installed on the Georgia Tech campus, and conduct *in situ* studies of the scalability of the proposed migration mechanisms.

Acknowledgments

This work was funded in part by an NSF CPS program Award #1446801, and a gift from Microsoft Corp. We thank the members of the embedded pervasive lab (EPL) for their helpful feedback on this paper.

8. REFERENCES

[1] protobuf: Google's data interchange format. https://github.com/google/protobuf. Accessed: 2016-01-02.

[2] Rocksdb. http://rocksdb.org/. Accessed: 2016-01-12.

[3] Vmware. http://www.vmware.com/products/esxi-and-esx/overview.html. Accessed: 2015-09-30.

[4] M. Behrisch, L. Bieker, J. Erdmann, and D. Krajzewicz. Sumo - simulation of urban mobility: An overview. In *in SIMUL 2011, The Third International Conference on Advances in System Simulation*, pages 63–68, 2011.

[5] L. Belli, S. Cirani, G. Ferrari, L. Melegari, and M. Picone. A graph-based cloud architecture for big stream real-time applications in the internet of things. In *Advances in Service-Oriented and Cloud Computing*, pages 91–105. Springer, 2014.

[6] D. Bernstein. Containers and cloud: From lxc to docker to kubernetes. *Cloud Computing, IEEE*, 1(3):81–84, Sept 2014.

[7] F. Bonomi, R. Milito, J. Zhu, and S. 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.

[8] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 161–174, Berkeley, CA, USA, 2008. USENIX Association.

[9] Docker. Understanding docker. https://docs.docker.com/introduction/understanding-docker/. Accessed: 2015-09-30.

[10] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. SPADE: the System S declarative stream

processing engine. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1123–1134, New York, NY, USA, 2008. ACM.

[11] K. Ha, Y. Abe, Z. Chen, W. Hu, B. Amos, P. Pillai, and M. Satyanarayanan. Adaptive vm handoff across cloudlets. Technical report, Technical Report CMU-CS-15-113, CMU School of Computer Science, 2015.

[12] M. M. Haklay and P. Weber. Openstreetmap: User-generated street maps. *IEEE Pervasive Computing*, 7(4):12–18, Oct. 2008.

[13] D. Hilley et al. Persistent temporal streams. In *Proceedings of the ACM/IFIP/USENIX 10th International Middleware Conference (Middleware '09)*, December 2009.

[14] P. Hintjens. *ZeroMQ: Messaging for Many Applications*. " O'Reilly Media, Inc.", 2013.

[15] K. Hong, D. Lillethun, U. Ramachandran, B. Ottenwälder, and B. Koldehofe. Mobile fog: A programming model for large-scale applications on the internet of things. In *Proceedings of the Second ACM SIGCOMM Workshop on Mobile Cloud Computing*, MCC '13, pages 15–20, New York, NY, USA, 2013. ACM.

[16] S. Hykes. What is docker? https://www.docker.com/whatisdocker/. Accessed: 2015-09-30.

[17] Internet of Things Consortium. Intelligence age. http://iofthings.org/intelligence-age/.

[18] Z. Liu, W. Qu, W. Liu, and K. Li. Xen live migration with slowdown scheduling algorithm. In *Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2010 International Conference on*, pages 215–221, Dec 2010.

[19] P. Lu, A. Barbalace, and B. Ravindran. Hsg-lm: Hybrid-copy speculative guest os live migration without hypervisor. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, pages 2:1–2:11, New York, NY, USA, 2013. ACM.

[20] Y. H. Milan Patel, P. Hede, J. Joubert, C. Thornton, B. Naughton, J. R. Ramos, C. Chan, V. Young, S. J. Tan, D. Lynch, N. Sprecher, T. Musiol, C. Manzanares, U. Rauschenbach, S. Abeta, L. Chen, K. Shimizu, A. Neal, P. Cosimini, A. Pollard, and G. Klas. Mobile-edge computing âĂŞ introductory technical white paper. Technical report, Mobile-edge Computing (MEC) industry initiative., https://portal.etsi.org/portals/0/tbpages/mec/docs/mobile-edge_computing_-_introductory_technical_white_paper_v1%2018-09-14.pdf, sep 2014.

[21] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *Proceedings of the IEEE International Conference on Data Mining Workshops*, 2010.

[22] T. Nishio, R. Shinkuma, T. Takahashi, and N. B. Mandayam. Service-oriented heterogeneous resource sharing for optimizing service latency in mobile cloud. In *Proceedings of the First International Workshop on Mobile Cloud Computing and Networking*, MobileCloud '13, pages 19–26, New York, NY, USA, 2013. ACM.

[23] B. Ottenwälder et al. Migcep: Operator migration for mobility driven distributed complex event processing. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 183–194, New York, NY, USA, 2013. ACM.

[24] C. Pahl and B. Lee. Containers and clusters for edge cloud architectures–a technology review. In *Future Internet of Things and Cloud (FiCloud), 2015 3rd International Conference on*, pages 379–386. IEEE, 2015.

[25] A. Papageorgiou, B. Cheng, and E. Kovacs. Real-time data reduction at the network edge of internet-of-things systems. In *Network and Service Management (CNSM), 2015 11th International Conference on*, pages 284–291, Nov 2015.

[26] P. S. Pillai, L. B. Mummert, S. W. Schlosser, R. Sukthankar, and C. J. Helfrich. Slipstream: scalable low-latency interactive perception on streaming data. In *Proceedings of the 18th international workshop on Network and operating systems support for digital audio and video*, NOSSDAV '09, pages 43–48, New York, NY, USA, 2009. ACM.

[27] M. Satyanarayanan. Mobile information access, 1996.

[28] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for vm-based cloudlets in mobile computing. *Pervasive Computing, IEEE*, 8(4):14–23, Oct 2009.

[29] S. Simanta, K. Ha, G. Lewis, E. Morris, and M. Satyanarayanan. A reference architecture for mobile code offload in hostile environments. In *Mobile Computing, Applications, and Services*, pages 274–293. Springer, 2012.

[30] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev.*, 41(3):275–287, Mar. 2007.

[31] W. Thies, M. Karczmarek, and S. P. Amarasinghe. Streamit: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 179–196, London, UK, 2002. Springer-Verlag.

[32] R. Urgaonkar, S. Wang, T. He, M. Zafer, K. Chan, and K. K. Leung. Dynamic service migration and workload scheduling in edge-clouds. *Performance Evaluation*, 91:205 – 228, 2015. Special Issue: Performance 2015.

[33] S. Wang, R. Urgaonkar, K. Chan, T. He, M. Zafer, and K. K. Leung. Dynamic service placement for mobile micro-clouds with predicted future costs. In *Communications (ICC), 2015 IEEE International Conference on*, pages 5504–5510. IEEE, 2015.

[34] M. Weiser. Some computer science issues in ubiquitous computing. *Commun. ACM*, 36(7):75–84, July 1993.

[35] C. P. Wright and E. Zadok. Unionfs: Bringing File Systems Together. *Linux Journal*, 2004(128):24–29, December 2004.

[36] H. Yao, C. Bai, D. Zeng, Q. Liang, and Y. Fan. Migrate or not? exploring virtual machine migration in roadside cloudlet-based vehicular cloud. *Concurrency and Computation: Practice and Experience*, 27(18):5780–5792, 2015.