Mobile Fog: A Programming Model for Large–Scale Applications on the Internet of Things

Kirak Hong, David Lillethun, Umakishore Ramachandran
College of Computing
Georgia Institute of Technology
Atlanta, Georgia, USA
{khong9, davel, rama }@cc.gatech.edu

Beate Ottenwälder, Boris Koldehofe
Institute of Parallel and Distributed Systems
University of Stuttgart
Stuttgart, Germany
{ottenwaelder,koldehofe}@ipvs.uni-stuttgart.de

ABSTRACT

The ubiquitous deployment of mobile and sensor devices is creating a new environment, namely the *Internet of Things* (IoT), that enables a wide range of future Internet applications. In this work, we present Mobile Fog, a high level programming model for future Internet applications that are geospatially distributed, large–scale, and latency–sensitive. We analyze use cases for the programming model with camera network and connected vehicle applications to show the efficacy of Mobile Fog. We also evaluate application performance through simulation.

Keywords

fog computing; cloud computing; programming model; Internet of Things; future Internet applications; situation awareness applications

Categories and Subject Descriptors

C.2.1 [**Network Architecture and Design**]: Distributed networks; C.2.4 [**Distributed Systems**]: Distributed applications

1. INTRODUCTION

The growing ubiquity of mobile and sensor devices, such as smart phones and surveillance cameras, is enabling a wide range of novel, large–scale, latency–sensitive applications that are often classified as *Future Internet Applications*. For example, traffic applications can analyze recent location information shared by vehicles to detect traffic patterns, such as accidents or traffic jams, and use these patterns to enrich navigation systems in vehicles [5]. Smart surveillance applications can support police officers by displaying video streams on their smart phones showing suspicious people who were nearby within the last few minutes [3].

Platform as a Service (PaaS) clouds are attractive for developing large–scale applications due to their scalability and high–level programming models that simplify developing large–scale web services. However, existing PaaS programming models are designed for traditional web applications, rather than future Internet applications running on various mobile and sensor devices. Furthermore, public clouds, as they exist in practice today, are far from the idealized utility computing model. Applications are developed for a particular provider's platform and run in data centers that exist at singular points in space. This makes their network distance too far from many users to support highly latency–sensitive applications.

Cisco recently proposed a new computing paradigm called *fog computing* [2] that runs generic application logic on resources throughout the network, including routers and dedicated computing nodes. In contrast to the cloud, putting intelligence in the network allows fog computing resources to perform low–latency processing near the edge while latency–tolerant, large–scope aggregation can still be efficiently performed on powerful resources in the core of the network. Data center resources may still be used with fog computing, but they do not constitute the entire picture.

However, developing applications using fog computing resources is tricky because it involves orchestrating highly dynamic, heterogeneous resources at different levels of network hierarchy to support low latency and scalability requirements of applications. To ease such complexity, this paper presents a PaaS programming model, called Mobile Fog, that provides a simplified programming abstraction and supports applications dynamically scaling at runtime. Our contributions include the design of our programming model and the use case analysis of Mobile Fog. While Mobile Fog can support diverse applications, we use *situation awareness* applications as canonical examples in this work.

This paper is structured as follows: Section 2 discusses related work. Section 3 explains the details of the Mobile Fog programming model. Section 4 shows specific use cases for our programming model. Finally, Section 6 concludes with future work.

2. RELATED WORK

Cloud and data center systems provide highly scalable solutions for web-scale applications, but edge devices must communicate across the Internet to reach the cloud data centers. Satyanarayanan, et al. [7], show that WAN latencies can be high and that these latencies interfere with interactive applications. We argue that situation awareness applications are vulnerable to the same issue due to the

MCC'13, August 12, 2013, Hong Kong, China.

sense-process-actuate loop, as would be any other applications with feedback loops. In short, the could is too far from many mobile users for latency–sensitive applications.

Two systems that can provide resources for computing near the edge of the network are the MediaBroker [6] for live sensor stream analysis, and Cloudlets [7] for interactive applications. However, neither currently supports widely distributed geospatial applications.

Recent advances in software–defined networking (SDN) [4] allow programming in-the-middle network resources. However, these approaches only allow injecting routing logic into network elements, not generic application logic. Furthermore, SDN offers neither an elastic resource model nor a programming model for general–purpose applications.

Bonomi et al. [2] define the characteristics of fog computing and show that the fog is the appropriate platform for various applications, including connected vehicles and smart cities. This work complements our work since it presents the potential benefits of fog computing in terms of efficiency and latency while our work focuses on the right programming model for developing applications on the fog.

3. PROGRAMMING MODEL

As a programming model for large–scale situation awareness applications, Mobile Fog 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 allow applications to dynamically scale based on their workload using on–demand resources in the fog and in the cloud.

3.1 System Assumptions

In this work, we assume various devices are available to Mobile Fog for running large–scale applications. These heterogeneous devices include edge devices such as smartphones 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 Mobile Fog. Once computing instances are created in the fog, Mobile Fog 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 Mobile Fog, an application consists of distributed *Mobile Fog processes* that are mapped onto distributed computing instances in the fog and cloud, as well as various edge devices. While running, each process performs application–specific tasks such as sensing and aggregation with respect to its location and level in the network hierarchy.

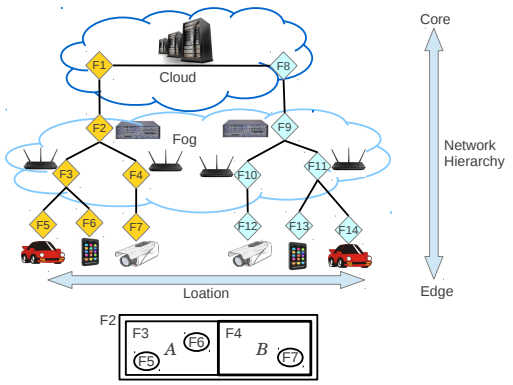


Figure 1: A logical structure of an application

Mobile Fog exposes the physical hierarchy of devices through a logical hierarchy of Mobile Fog processes that is described in Figure 1. As shown in the figure, a process on an edge device becomes a leaf node while a process in the cloud becomes the root node of a hierarchy, while processes on fog computing nodes become intermediate nodes. A connection between two processes indicates a communication path allowed through the Mobile Fog 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 F8 in the cloud).

Each Mobile Fog process handles the workload from a certain geospatial region. For instance, Figure 1 shows processes F5 and F6 mapped onto a connected vehicle and a smartphone within region A. These leaf processes are connected to process F3 running on a 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.

3.3 API

In Mobile Fog, application code consists of a set of event handlers that the application must implement (Table 1) and a set of functions that applications can call (Table 2). The event handlers are invoked by the Mobile Fog runtime system upon certain events. For example, *on_create()* is invoked when an application is started while *on_message()* is invoked when a message arrives. Mobile Fog 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.

Mobile Fog also provides detailed information about the underlying physical devices in order to allow application code to perform its task with respect to the geophysical location, available system resources, device capabilities, and network hierarchy level. In particular, application code can

Handler	Description
void on_create (node self)	Called when a new node is created.
void on_sense(type t, data d)	Called upon a new sensor reading from a passive sensor (e.g., a camera).
void on_message (tag t, node sender, message m)	Called when a new message arrived at a node. A tag specifies an application-specific type of the message. A sender specifies the identifier of the source node.
void on_new_child(node child)	Called at a parent node when a new child node is connected.
void on_new_parent(node parent)	Called at a child node when a new parent node is connected.
void on_child_leave(node child)	Called at a parent node when a child node is leaving.
void on_parent_leave(node parent)	Called at a child node when a parent node is leaving.

Table 1: Mobile Fog Event Handlers

Programming Interface	Description
void send_up (tag t, message m)	Sends a message from a child node to its parent node.
void send_down (tag t, message m, set<int>index)	Sends a message from a node to its child nodes.
void send_to (tag t, message m, node destination)	Sends a message to a specific destination node.
data sense (type t)	Retrieve a sensor data from an active sensor (e.g., temperature sensor).
int num_children()	Get the number of children of the calling node.
Location query_location()	Get the location of this device. Returns a location range (location coverage) if invoked on a fog computing node. If invoked on an edge device, returns a single location point.
int query_level()	Get the level of this device in the network hierarchy.
CapDesc query_capability(type t)	Query a specific type of capability such as sensing and actuation functionality on a device. As a result, a descriptor for the capability is returned if the device has the capability.
ResDesc query_resource (type t)	Query a specific type of resource (e.g., CPU and memory) on a device. As a result, a descriptor for the resource is returned.
set<object> get_object(key k, locationRange lr, timeRange tr)	Get application context data that matches a key, location range, and time range.
void put_object(object o, key k, location l, time t)	Put application context data associated with a key, location (range), and time (range).

Table 2: Mobile Fog Application Programming Interface

retrieve information about available system resources by invoking *query_resource()*. An application can also query the device capabilities through *query_capability()*. Depending on the capabilities of the physical device, the application code can invoke *sense()* to retrieve specific sensor data. However, if the sensor type is passive, the application code is notified of new sensor data through the *on_sense()* event handler. Mobile Fog also provides information about the location through *query_location()*. If a process is running on an edge device, *query_location()* returns the location of the underlying device, whereas it returns the geospatial coverage of a process running on a computing node in the fog or the cloud. Finally, an application can query its level in the network hierarchy through *query_level()*.

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 with mobile nodes while the connection between a mobile node and a computing instance in the fog changes as the mobile node moves.

Once the code is written, an application developer compiles the code to generate a Mobile Fog 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 Mobile Fog. To launch an application, a developer invokes *start_app()* with four 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 Mobile Fog 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. 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. We also allow defining a dynamic scaling policy, which will be explained in detail in Section 3.4.

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 Mobile Fog process, using its local system resources, that connects to the Mobile Fog process on a fog computing instance covering the location of the edge device.

If an edge device is mobile, the connection from the edge device to a fog computing node changes as the location of the edge device changes. When a mobile device moves from one region to another, a Mobile Fog process covering the new region becomes the new parent of the edge device. To acknowledge such a handover to the application, Mobile Fog invokes a set of handlers, *on_child_join()*, *on_parent_join()*, *on_child_leave()*, and *on_parent_leave()* on the mobile device, the old parent, and the new parent.

3.4 Dynamic Scaling

Many future Internet applications, especially sensor– and event–based applications, deal with dynamic workloads from the real world. To support these applications, Mobile Fog uses on–demand computing instances in the fog and the cloud to provide transparent scalability based on a user–provided scaling policy. The scaling policy includes a set of monitoring metrics, such as CPU utilization and bandwidth, and scaling conditions that trigger dynamic scaling (e.g., CPU utilization over 80%).

When a computing instance becomes overloaded due to the dynamic workload, Mobile Fog creates on–demand fog instances at the same network hierarchy level as the overloaded instance. To distribute the workload over multiple computing instances, Mobile Fog splits the geospatial coverage of the overloaded process into multiple smaller coverages. Similarly, when multiple nearby processes at the same network hierarchy level become underloaded according to the user–provided scaling policy, Mobile Fog merges their geospatial coverages into a single coverage area and terminates all the processes except one for the merged coverage.

To allow transparent scaling, an application has to store 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*. Upon dynamic scaling, Mobile Fog automatically partitions the objects onto multiple Mobile Fog processes according to the location of the objects and geospatial coverage of the processes.

4. APPLICATIONS

4.1 Vehicle Tracking using Cameras

A metro area may have hundreds of cameras monitoring the highways. These sensors could enable an application that uses traffic cameras to help police identify and track vehicles for which they have issued a search, which we refer to as a BOLO. Pseudocode for a simplified version of this application is given in Figure 2 to illustrate how our API provides the necessary abstractions to keep track of device capabilities and to build applications with sense-process-actuate patterns. For brevity, a few important parts of the application have been intentionally omitted.

There are three types of Mobile Fog processes in this application. Camera processes are the leaves of the tree and are responsible for sensing the environment and delivering

```
1:  Tracking ← ∅
2:  function on_sense(Type t, Data frame)
3:    if ¬query_capability(CAP_MOTION_DET) ∨
          detect_motion(frame) then
4:      send_up(MSG_VIDEO_FRAME, Message(frame))
5:    end if
6:  end
7:  function on_message(Tag t, Node sender, Message m)
8:    if t = MSG_VIDEO_FRAME then
9:      Interest ← ∅
10:     Vehicles ← detect_vehicles(m.frame)
11:     for ∀vehic ∈ Vehicles do
12:       if vehic ∈ Tracking then
13:         update_position(vehic)
14:         put_object(vehic, vehic.license_no,
                m.time, m.location)
15:       else
16:         license ← detect_license(vehic)
17:         if license then
18:           license_no ← read_license(license)
19:           if is_bolo(license_no) then
20:             put_object(vehic, license_no,
                    m.time, m.location)
21:             Tracking ← Tracking ∪ {vehic}
22:             send_up(MSG_FOUND_VEHIC, vehic)
23:           end if
24:         else
25:           Interest ← Interest ∪ {vehic}
26:         end if
27:       end if
28:     end for
29:     if |Interest| > 0 then
30:       ptz_cmd ← get_PTZ(choose_vehicle(Interest))
31:     else
32:       ptz_cmd ← get_PTZ(REST_POSITION)
33:     end if
34:     send_down(MSG_PTZ, ptz_cmd, {sender})
35:   else if t = MSG_PTZ then
36:     if query_capability(CAP_PTZ) then
37:       execute_PTZ(m.ptz_cmd)
38:     end if
39:   end if
40: end
```

Figure 2: BOLO Vehicle Tracking Algorithm

video frames to their parent processes. Lines 2-6 in Figure 2 show the handler that is executed every time a video frame is produced. Our API can be used to determine whether a "smart camera" has the ability to detect motion in the scene (line 3), and if present, it can be used to avoid sending video to the camera's parent unless motion is detected.

The lowest tier of fog instances are the immediate parents of the cameras and are responsible for identifying and tracking vehicles (lines 9-28) as well as controlling the cameras (lines 29-38). This allows the most intensive computation to be handled by the most widely distributed and lowest–latency resources. Video frames received by processes with camera children are processed by lines 9-34. First, a computer vision algorithm is used to detect_vehicles() in the video. If a vehicle is in the set of vehicles being tracked by this camera (i.e., it was previously detected), then the vehicle's position in space is updated and the location of the vehicle at that time is stored by *put_object*() on line 14. Otherwise it is a newly detected vehicle, and the algorithm attempts to identify it by its license plate. If the plate can be read clearly on line 16, then we can detect the license number (line 18) and determine if there is a BOLO for that vehicle (line 19). If so, we record the vehicle's location, add it to the set of vehicles being tracked, and notify this pro-

```
1: Requires: n //number of levels in hierarchy
2: function on_new_parent(node parent)
3:   if query_level()= n then
4:     da ← create_detection_algorithm(query_location())
5:     rq ← generate_range_query(query_location())
6:     send_up(MSG_DEPLOY,Message(da, rq, mobileId))
7:   end if
8: end
9: function on_message(tag t, node sender, message m)
10:   if t = MSG_SENSOR_DAT ∧ query_level()= n − 1 then
11:     put_object(sensor_data(m),"DAT",m.time,m.location)
12:   else if t = MSG_DEPLOY then
13:     if m.rq.range ⊂ query_location()) then
14:       put_object(m.da,"DETECTION",NULL,
              m.rq.range))
15:       location_multicast(MSG_RQ,
              Message(m.rq, nodeId),m.rq.range)
16:     else
17:       send_up(m)
18:     end if
19:   else if t = MSG_RQ then
20:     if query_level()= n − 1 then
21:       B ← get_object("DAT",m.rq.time,m.rq.range)
22:       send_to(MSG_BUFF, message(B), m.nodeId)
23:     else
24:       location_multicast(MSG_RQ,m,m.rq.range)
25:     end if
26:   else if t = MSG_BUFF then
27:     D ← get_object("DETECTION",NULL, NULL)
28:     P ← process_events(D, m.Events)
29:     send_to(MSG_PATTERN,message(P), mobileId(P))
30:   end if
31: end
```

Figure 3: Traffic Monitoring with MCEP

cess' parent that a new BOLO vehicle was detected (lines 20-22).

However, if the license is not sufficiently clear to read, we can use pan-tilt-zoom (PTZ) camera capabilities to zoom in for a better view. Line 25 adds the vehicle to a set of "vehicles of interest". Since a camera can only zoom in on one vehicle at a time, lines 29-33 select the vehicle to focus on using a choose_vehicle() algorithm and then sends the PTZ command to the camera (line 34). Lines 35-38 handle the PTZ command when it is received by the camera. The incorporation of PTZ creates a *sense-process-actuate* feedback loop that requires the low latency provided by the fog.

4.2 Traffic Monitoring using MCEP

Information from sensors on mobile devices, e.g., vehicles, can help detect traffic situations if the information is shared and aggregated. For example, a sequence of several similar movement and acceleration patterns from different vehicles on the same road can determine if an accident blocks the road. The *Mobility–driven distributed Complex Event Processing (MCEP)* system [5] enables deriving such patterns from sensor data. MCEP allows consumers to specify their interest in recent, nearby patterns with a detection algorithm (da) and an associated spatio–temporal range query (rq), which is parametrized with a *time* and *range*. For example, a range query could select relevant sensor data around a consumer that occurred in the last hour within a one kilometer perimeter around the consumer.

Figure 3 shows how a simplified version of the MCEP system is implemented using the Mobile Fog programming model. Vehicles are on the nth level of the Mobile Fog hierarchy and connect to the MCEP system by calling *connect_fog()* (not shown). Sensor data from those vehicles is buffered by Mobile Fog processes on level $n-1$, henceforth referred to as *edge processes* (Lines 17-18). This buffering allows efficiently answering spatio–temporal range queries after the vehicle has moved away from the edge process.

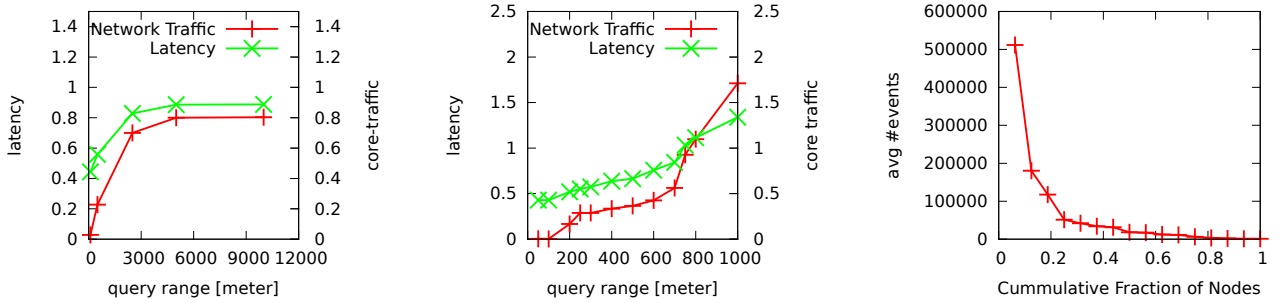
Each connected vehicle can post a query by sending a MSG_DEPLOY message containing a detection algorithm, range query, and its ID to its parent using *send_up()* (see Lines 3-8), e.g., the vehicle connected to $F5$ in Figure 1. The detection algorithm is pushed up the hierarchy to the Mobile Fog process that is associated with a geospatial range that fully contains the *range* of the query (Line 19-25), e.g., F_3 in Figure 1. This process is going to perform the detection algorithm and pushes the range query down, via *send_down()*, to all Mobile Fog processes that overlap with the *range* of the query (Lines 22 & 31), e.g., F_5 and F_6 in Figure 1. Edge processes buffer the relevant sensor data for the detection and thus process the range query. Results of the range query are sent via ID to the resource node that hosts the detection algorithm (Lines 27-29) and processed there. Detected patterns are sent via ID to the mobile device (Lines 34-36). With each location update of a vehicle, or when a vehicle connects to a new resource node, the callback interfaces *on_child_leave()* and *on_new_parent()* deploy a new detection algorithm and range query, and revoke the previous one (see Line 2). To automatically migrate detection algorithms and buffered sensor data when Mobile Fog scales up or down, they are put in the context store.

5. EVALUATION

To show the benefits of Mobile Fog over a cloud–based approach, in terms of communication cost, we conducted a network simulation using OMNeT++ [8] with realistic traffic patterns generated by SUMO [1]. We simulated a thousand randomly moving vehicles over fifteen minutes on the road network of an urban area (7.7×3.5 km). The entire area was covered by a Quadtree communication structure: one cloud as root, four core fog computing nodes, and sixteen edge fog nodes that directly communicate with vehicles. Fog nodes at each level cover uniformly divided parts of the entire area. The network latency between each pair of nodes is 20 milliseconds.

Based on the above setup, we simulated two large–scale application scenarios requiring low end-to-end latencies. The first application is vehicle–to–vehicle video streaming, where each vehicle randomly chooses another vehicle within a query range and starts streaming a fixed sized video to the target vehicle. In this application, a cloud–based approach streams all videos through the cloud while the Mobile Fog–based approach streams videos through intermediate fog computing nodes, similar to software defined networking. The second application is Mobile CEP, where a vehicle requests processing sensor data from a certain query range. In this application, the Mobile Fog–based approach stores sensor data in the nearby edge fog computing nodes, while the cloud–based approach stores all sensor data in the cloud. Both applications use the *send_up()* and *send_down()* API to handle user requests in the Mobile Fog–based approach, which means higher level nodes in the Quadtree will be utilized to handle larger query ranges.

Figure 4 (a-b) shows the relative end–to–end latency and core network traffic of the Mobile Fog–based approach compared to the cloud–based approach while varying the *query range*. The results show that the Mobile Fog–based ap-



(a) Vehicle–to–vehicle video streaming **(b)** Traffic monitoring using MCEP **(c)** Workload distribution of MCEP

Figure 4: Relative latency and network traffic of Mobile Fog compared to the cloud–based approach

proach significantly reduces both end–to–end network latency and core network traffic when query ranges are small, since Mobile Fog allows users to be served by nearby fog nodes. In the vehicle–to–vehicle video streaming application (Figure 4(a)), the Mobile Fog–based approach always outperforms the cloud–based approach since the video stream goes through the cloud only when a vehicle picks a far–away target vehicle. However, in the MCEP scenario (Figure 4(b)), the cloud–based approach outperforms Mobile Fog when the query range is very large because the Mobile Fog–based approach has to aggregate sensor data stored in edge fog computing nodes before serving a user query. If such a wide–scope aggregation happens frequently, the Mobile Fog–based approach could also push sensor data up to the cloud to achieve lower latency and lower network traffic for query processing at the expense of higher network traffic for normal operation.

Figure 4(c) shows the various workloads at different fog nodes. We measured the total number of events stored at each Mobile Fog process running on edge fog nodes in the MCEP scenario. The number of events stored at each process is directly related to the necessary storage capacity, as well as the processing needs since more events implies a higher chance of a user request. As shown in the figure, the workload distribution over different Mobile Fog processes is highly skewed, i.e., only a few processes are overloaded because certain regions have higher vehicle traffic. This result motivates dynamic scaling to handle the highly skewed and dynamic workload of applications.

6. DISCUSSION AND FUTURE WORK

In this paper we discussed the design of Mobile Fog, a programming model for large–scale, latency–sensitive applications in the Internet of Things (IoT). Since this is ongoing research, it creates interesting research problems for future work.

Runtime system implementation: We plan to develop a runtime system that implements the Mobile Fog programming model on real fog-enabled devices. The key challenge is to develop a distributed runtime system that can migrate Mobile Fog processes across different devices while providing reliability, security, and performance isolation for shared infrastructure resources.

Process placement algorithm: To achieve better network bandwidth utilization, latency, and load balance across distributed devices, we are currently working on an algorithm that can adaptively find a near-optimal placement of Mobile Fog processes. The key challenge in this direction is to find a better placement based on dynamic constraints including available resources, application workload, and migration cost for Mobile Fog processes.

Acknowledgment

This work is supported by contract research “CEP in the Large” of the Baden-Württemberg Stiftung.

7. REFERENCES

[1] M. Behrisch, L. Bieker, J. Erdmann, and D. Krajzewicz. SUMO - Simulation of Urban MObility: An overview. In *Advances in System Simulation*, SIMUL ’11, 2011.

[2] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. Fog computing and its role in the internet of things. In *Workshop on Mobile cloud Computing*, MCC ’12, 2012.

[3] K. Hong, S. Smaldone, J. Shin, D. J. Lillethun, L. Iftode, and U. Ramachandran. Target container: A target-centric parallel programming abstraction for video-based surveillance. In *International Conference on Distributed Smart Cameras*, ICDSC ’11, 2011.

[4] B. Koldehofe, F. Dürr, M. A. Tariq, and K. Rothermel. The power of software-defined networking: line-rate content-based routing using openflow. In *Workshop on Middleware for Next Generation Internet Computing*, MW4NG ’12, 2012.

[5] B. Koldehofe, B. Ottenwälder, K. Rothermel, and U. Ramachandran. Moving range queries in distributed complex event processing. In *Distributed Event-Based Systems*, DEBS ’12, 2012.

[6] D. J. Lillethun, D. Hilley, S. Horrigan, and U. Ramachandran. MB++: An integrated architecture for pervasive computing and high-performance computing. In *Embedded and Real-Time Computing Systems and Applications*, RTCSA ’07, 2007.

[7] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for VM-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8(4), October - December 2009.

[8] A. Varga and R. Hornig. An overview of the OMNeT++ simulation environment. In *Simulation tools and techniques for communications, networks and systems & workshops*, Simutools ’08, 2008.