Towards Gateway to Cloud Offloading in IoT Publish/Subscribe Systems

Daniel Happ, Adam Wolisz
Technische Universität Berlin, Telecommunication Networks Group (TKN)
{happ, wolisz}@tkn.tu-berlin.de

***Abstract*—It is not uncommon today that sensor devices connected to the Internet solely send their data to Cloud-based servers for storage and processing. This does not only mean clients requesting data have to contact the Cloud-based service, even if the data is available in the local network, but also that data is sent to external services with unknown or ambiguous privacy policies. The great potential in using closer to the edge fog computing instead of Cloud computing to both enable faster and more privacy-aware processing locally has been recognized in the research community. In particular, on premise smart gateways can provide local low-latency storage and processing capabilities that are controlled locally and can be trusted. In this work, we outline a combined fog Cloud system that automatically selects a suitable execution location for processing tasks. We emphasize on the design challenges of such a system and further demonstrate a solution for the interplay between Fog and Cloud by showing how processing task can be migrated from one system to another on the fly without service interruption.**

I. Introduction

The upcoming Internet of Things (IoT) embeds sensors, software and network connectivity into physical objects. The insights gathered from these smart devices allow us to understand the physical world around us in real-time in ways never possible before. One common way to cope with the increasing processing demand is Cloud computing [1]. However, Cloud computing is not always suitable for the processing tasks commonly found in the IoT context, especially for latency and privacy sensitive applications [2]. Instead of forcing all processing to Cloud servers that are possibly located far away, in different judicial areas, fog computing proposes moving the intelligent processing of data to the edge of the network, in our case specifically to smart gateways [3].

Through leveraging local processing on smart gateway devices and migrating heavy computation to resourceful Cloud servers on demand, the resource constraints of local gateway hardware regarding CPU-, memory-, or network-intensive tasks can be overcome. In this work, we present our design of a flexible IoT processing relocation framework. Although the concepts provided here universally apply to process placement and migration in distributed IoT related systems using publish/subscribe, we consider the subproblem in the context of gateway to Cloud offloading as an example. The platform allows the definition of constraints for gateway providers and the definition of continuous IoT processing tasks along with metadata for users. The system dynamically and automatically determines if those processing tasks should be computed on a Cloud server rather than the local gateway device. As this is ongoing work, we highlight the open challenges that have to be met to implement the system.

II. Related Work

The ubiquitous availability, flexibility, reliability and usage-based cost model makes it feasible to send sensor data to the Cloud and to store and process it there [1], [4]. There have been several recent proposals to use pub/sub systems for IoT messaging [2], [5]–[7]. However, we also identify a trend to move data and data processing closer to the client, similar to content delivery networks (CDN). This approach is followed by Akamai's EdgeComputing [8], Cisco's Fog Computing [3], Intel's Intelligent Edge [9] and Microsoft's Cloudlet [10]; Especially the emerging Fog Computing concept has been proposed to overcome the challenges of the Cloud-based IoT approach [2], [3].

Previous work in the area of process migration can be divided into two distinct areas: mechanism and policy [11]. While the mechanism explores how to migrate processes between two hosts, the policy orchestrates which processes to be execute where and when to migrate. For example, policy related work includes [12] where the authors propose a joint control algorithm for mobile users and Cloud service provider in a unified mobile Cloud computing framework to minimize the overall financial cost. In [13] the author presents his findings regarding time- and energy-aware offloading decision making.

In the IoT context, the authors of [14] describe an integrated fog cloud IoT (IFCIoT) architectural paradigm, including application, analytics, virtualization, reconfiguration, and hardware layer. In [15], the authors propose a virtual machine migration mechanism for fog computing. Aazam and Huh [16] present a resource estimation and pricing model for fog-based IoT including resource prediction, estimation, reservation, and pricing. We complement this existing work and focus on the mechanism of gateway to Cloud offloading in the specific context of publish/subscribe and consider the sub-class of more simple processing tasks using this paradigm instead of virtualization.

III. System Model

We consider the overall architecture of today's IoT to be 3-layered, having a Cloud, fog, and device layer. This architecture is depicted in Figure 1. The device layer consists of sensor and actuator devices, which are severely constrained, mostly by processing power, memory and energy. Those devices

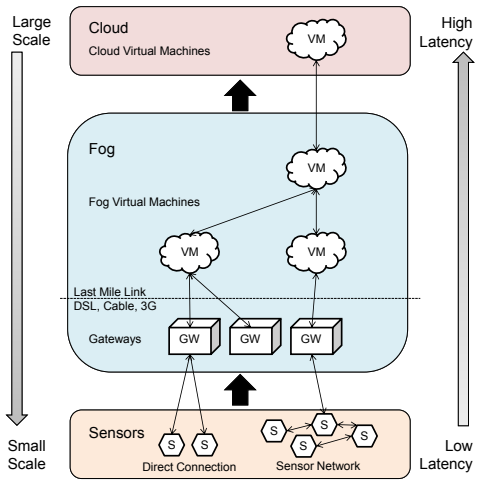


Fig. 1. Overall architecture of a Cloud-based IoT platform.

are directly connected to a gateway to interact with remote hosts. Gateways are themselves part of the fog layer. More general, fog instances provide the means to analyze and process data within the network closer to the end-user than in a centralized remote Cloud. The top layer is the remote Cloud layer, which provides ubiquitous and seemingly infinite access to storage and processing. While available resources and thus scalability increases towards the top, the latency and cost of communication also increases. Cloud services additionally may be provided from another legal area, making storage and processing of certain types of sensitive sensor data difficult or even prohibited by law.

We envision that all components use the publish/subscribe pattern to communicate [2], [5]–[7]. In this one-to-many pattern, the matching between consumer and producer of data is done by multiple dedicated message brokers. In the topic based scheme, the symbolic channel addresses are topics, usually in the form of strings, i.e. producers publish to and consumers subscribe to topics and messages are only delivered to matching subscribers.

While traditionally sensor devices used where very constrained in terms of processing power and storage, today's IoT hardware include powerful smartphones and embedded single-board computers. While most sensor devices are still energy-constrained, they will mostly use a gateway to connect to the public Internet. Those gateways, on the other hand, are usually mains-powered and powerful enough to take over some of the tasks Cloud services are providing today. A selection of suitable gateway devices is compiled in Table I. We argue that a lot of potential processing power and storage is available right on premise that is not fully utilized.

TABLE I
OVERVIEW OF SUITABLE SMART GATEWAY HARDWARE.

Device	CPU Clock	Memory	Storage	Price
Intel NUC	1.3 GHz	16 GB	SATA	∼$300
BeagleBone Black	1 GHz	512 MB	4GB EMMC, SD	$55
Raspberry Pi 3	1.2 GHz	1 GB	SD	∼$35
TP-Link TL-WR841N	650 MHz	32 MB	4 MB	∼$20

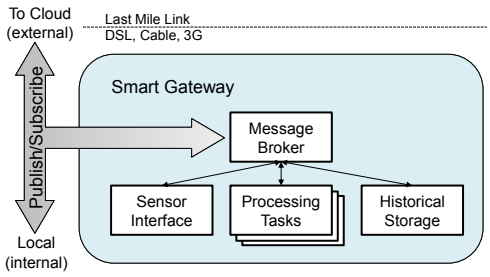


Fig. 2. Internal building blocks in IoT smart gateway device.

We envision the gateway to run the four main components depicted in Figure 2: The sensor interface offers the traditional gateway functionality of bridging low-power network and wide-area network. We introduce a pub/sub message broker that disseminates the sensor to processing or storage. The local pub/sub broker forwards subscriptions for external data of internal users and processing tasks to the Cloud. Data published locally is sent to the Cloud if a remote subscriber is present. Every processing task can therefore use internal as well as external data and reach any consumer of output data, local as well as remote.

A. *Offloading of Sensor Processing*

We distinguish two forms of sensor data processing: 1) batch processing 2) real-time processing. For the first, the data is stored and processed in a batch, e.g. every day at night an analysis is performed and the result saved as a summary of the data. For the latter, data is continuously processed and creates new streams of data. In this work, we consider real-time sensor data processing that follows the stream-based task model.

We define sensor processing as the computation of a set of output values on a set of input values. In the context of our system model, the input values are exclusively derived from subscriptions to a pub/sub system. The output is published on one or more output topics. We do not make any assumptions about the function provided, as it can be entirely defined by the user.

As opposed to the more traditional model of Cloud processing, we envision the system to default to local computation on the gateway device and only give tasks to Cloud-based services as an option. This has several advantages: Regarding performance, the local execution will not introduce additional networking delays to send data to remote Cloud services. Regarding the robustness of the system, local actuation loops can still run in case of network outages. Regarding privacy, we consider the local gateway to be fully under the control of the local operator and trusted, i.e. not leaking private data to third parties.

The task may also be offloaded to a Cloud-based service. Since different task have different requirements, not all tasks are suitable for remote execution. The task issuer gives additional constraints and requirement along with the processing task itself. The placement of the processing task must comply with the requirements given as well as the available resources on the gateway. We first identify the following broad classes for processing tasks requirements:

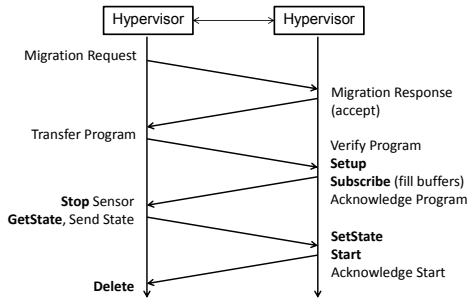


Fig. 3. Migration between two hypervisors.

- **Offloadable Tasks:** Those task can be processed either locally or remotely using Cloud infrastructure. The gateway is free to offload the task at any time as long as all specified requirements are met.
- **Unoffloadable Tasks:** Some task have to be executed locally, mainly because they cannot be run remotely, e.g. when they need local resources that cannot be transmitted to the remote infrastructure in time with the constraint bandwidth of gateways. Another reason might be privacy or juridical concerns that prohibit remote execution.

Because the system has to take into account available resources, task may also not be feasible to run at all, for instance if a lot of input data has to be processed fast and both processing power and upload bandwidth to the Cloud is severely constraint. For those task neither local nor remote execution of the task is possible, so that the task has to be rejected by the system.

Additionally, the system should dynamically adapt to the current state of both gateway and Cloud and offer migration in both directions, so that a processing task can also be relocated to the gateway. We see two sources of information required to make migration decisions. The first are static policies for the offloading. Those policies are given by the issuer of the processing task, e.g. regarding privacy requirements. We further assume profilers collect real-time performance metrics to help making offloading decisions. We identify CPU utilization, memory consumption and network traffic as relevant metrics for the offloading decision in the case of continuous precessing task on local gateways.

B. Migration of Sensor Processing

The migration framework consists of two identical parts on the local gateway and Cloud side. In a more general setting of processing placement in any distributed environment, the building blocks needed would be replicated across every system that would take part in the processing of sensor data, e.g. different Cloud instances or fog based intermediate systems. Those building blocks are a hypervisor that gets the offloading decisions from the offloading engine and put them into practice in the execution environment, which is the second building block, by controlling the running processing tasks.

To be migrated, every processing task has to offer a predefined set of operations to the hypervisor, which are given in Table II. Since we do not want to make any assumption

TABLE II
OVERVIEW OF SPECIFIC PROCESSING TASK METHODS.

Method	Desciption
setup	called before launch to initialize task
subscribe	called to subscribe to input topics and fill buffers
setState	called to set initial state to representation given
start	called to start processing
restart	called to restart processing
stop	called to stop processing
getState	called to get representation of the process state
delete	called for terminating the processing task

TABLE III
METHODS OFFERED BY HYPERVISOR.

Method	Desciption
start	called to register processing task
stop	called to stop processing task
startRemote	called to register remote processing task
stopRemote	called to stop remote processing task
migrate	called to migrate process to target

on the specific type of process, each process has its own constructor/de-constructor (setup/delete) methods to initialize the process. The task is initialized with a particular pub/sub broker to connect to and a set of input and output topics. After initialization, the hypervisor calls the subscribe method. This method starts to subscribe on every input topic given and fills the relevant queues with sensor data.

The setState method is called to set the initial state of the process. Likewise, the getState method is used to extract the current process state. Every processing task needs to provide those functions to serialize their relevant state to a byte stream and set their internal state when called with a suitable byte stream. The specific encoding is up to the task developer.

Please note that so far no command has actually started processing any data. This is triggered by a call to the start method. A process can likewise be stopped using the stop call. If necessary, a processing task can be restarted with the restart method.

For the hypervisor, we envision the interface given in Table III that can be called by the offloading engine. It can start or stop a processing task, both locally on the same gateway as well as remote. Additionally, it can instruct the hypervisor to migrate a processing task.

The hypervisors communicate with each other as described in Figure 3. When one of the hypervisors was triggered to initiate the migration, it contacts its counterpart with a migration request. The migration request contains the sensor metadata. The remote hypervisor consults its decision engine to check if enough resources are available and all policies would be met. The hypervisor would accept or decline the offloading request.

In the next step, the sensor is actually migrated. This includes the code as well as the current state. First, the code is submitted. The remote hypervisor executes the code and calls the setup method. It then instructs the processing task to subscribe to its input topics. We leverage the decoupling properties of the pub/sub system here, since at that point in the scheme we subscribe both the old and the new processing node to the same input topics.

The hypervisor then stops its local sensor and requests and

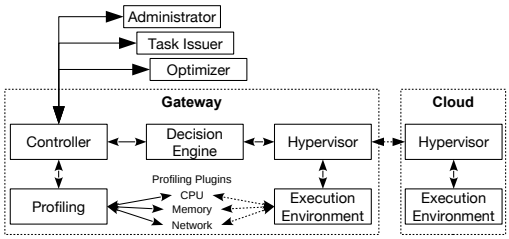


Fig. 4. Simplified gateway to Cloud offloading architecture.

sends the current state. The state includes an offset where the processing stopped and where it should continue. On receiving the state, the remote hypervisor is directed to start its processing task. Like with subscribing multiple processing tasks, we can seamlessly switch between the two processing tasks by publishing using the other task on the same topic. As long as the new processing task starts at the same offset, subscribers will not be able to identify which publisher issued a certain message and would not notice the migration. In case of an error, the original task can still be resumed on the original hypervisor at the last known offset. On successful migration, the original processing task is deleted.

The migration approach outlined above is similar to existing solutions, but innovates by taking advantage of the decoupling properties of the pub/sub system. Ongoing subscriptions of consumers do not have to be explicitly transfered between processing nodes, as is the case for traditional service migration. Consumers are not required to address a different host after migration; the migration is fully transparent to the subscribers.

IV. Gateway Offloading Architecture

We identify three distinct roles in our system:

- The gateway administrator defines constraints for the gateway device, for instance he can restrict the total CPU or memory consumption of all sensor processing tasks to reserve resources for other tasks that are not controlled by our system. The same applies to bandwidth constraints.
- The task issuer defines processes that are to be executed on sensor data. He has to provide the processing function and additional metadata, such as estimates of resource consumption and input and output topics as well as estimated output frequencies.
- The optimizer dictates the optimization goal in case multiple valid task allocations are found.

We do not consider consumers of processed data explicitly as a role, since they are decoupled from the system using the pub/sub network. The system can estimate the communications cost of the messaging middleware, if both publishers and subscribers are known to the messaging system, e.g. if advertisements are used before publishing on a topic. The offloading decision as such only takes into account local and remote traffic, but does not explicitly require knowledge of consumers. In future work, a joint processing and data storage optimization would be possible, where historical data is relocated close to processes needing that data.

Figure 4 depicts the overall architecture of our gateway to Cloud offloading system. The components of the system along with the typical workflow is given as follows: The system consists of a gateway and Cloud part. The gateway is the central entity of the system and the entry point in the definition of sensor processing tasks for task issuer, constraints for gateway administrator and optimizing target for the optimizer. Both parts have a hypervisor that can start, stop and migrate processing tasks and an execution environment wherein the processing tasks are run. The execution environment is instrumented with profiling plugins that monitor the overall and per task resource usage.

The system consists of the following building blocks:

- The controller in the gateway receives requests along with relevant metadata
- The profilers gather data to make an offloading decision.
- The decision engine get constraints, processing tasks, optimization goal and profiling data and decides on the placement of tasks.
- The local hypervisor either executes the task in the local execution environment or offloads the task to a remote hypervisor.

A. Policy and Processing Metadata

The three roles we identified have to be able to specify their requirements. The gateway administrator will have to specify the performance characteristics of the system along with certain constraints on his hardware to be able to reserve resources for unrelated other tasks running on the gateway. He has to give an estimate of the computing power of his processing node. He would further give an average percentage of CPU he is willing to use for processing as well as an additional allowance for bursts. The administrator further has to define the networking capabilities of his node, i.e. the upload and download bandwidth available to processing tasks, again separated into average and burst allowance.

In the proposed model, the optimizer would provide the optimization goal as an integral part of the development process. The optimization goal could either be the optimization on one particular metric, such as lowest external bandwidth consumption, lowest memory consumption or lowest CPU usage, or a more complex multi-objective decision problem. The optimizer could provide the optimization goal using a utility function that can be used to assign a numerical constant to every processing task. The system could provide a number of predefined template functions, such as CPU only, network only and memory only. Those templates would allow developers to specify goals without much additional effort. The offloading engine would then try to optimize the system with regard to the provided utility function.

The sensor processing task user would specify metadata related to its processing task. He would define the input and output topics of the processing tasks. Additional information, such as estimated CPU usage, memory usage and network usage, also have to be estimated. The CPU usage can be estimated in floating-point operations per second (FLOPS) per event using static program profiling. Likewise, the memory consumption can be estimated by analyzing the memory allocation calls per event on a certain input stream. All those

parameters heavily depend on the frequency of input events on the input topics, so those have to be estimated by the user as well. In case the user does not want to provide this data on his own, the system could alternatively assume some conservative default values and obtain the real data based on measurements.

The exact format to specify the metadata is the focus of ongoing work. We consider simple key-value structures, such as JSON, as well as more powerful semantic technologies. With the use of RDF ontologies, the exact meaning of requirements could be fixed. For instance, in the context of Cloud computing a similar approach based on RDF was proposed [17], which includes CPU, memory, networking and storage constraints already. Another benefit of RDF data could be the vast amount of knowledge and tools to automatically reason based on the provided data.

B. Profiling of Sensor Processing Tasks

We identify the need for the following profilers in our system: A static and dynamic program profiler is needed to provide an estimate of the runtime characteristics of the processing task. The static profile would be created by the user registering the processing task. Since the runtime characteristics heavily depend on the amount of input event-based sensor data, the user would have to estimate the frequency of input events. A dynamic profiler is needed to verify those characteristics at run-time per processing task and to monitor the overall system usage as a whole. A networking profiler is needed that measures both internal and external traffic. The distinction between internal and external traffic is important, since on task migration, internal traffic, which is associated with little cost, would become external traffic, which would introduce additional cost. However, external traffic would shift to an external node and would not need to be routed to the gateway anymore, thus reducing the cost on the gateway side. The profiler is needed for every processing task and the system as a whole, to also capture unrelated background processes that run on the gateway node.

Profiling is not particularly challenging on most systems. As an example, we focus on a Linux based system, but similar tools for other operating systems are also available. CPU usage and memory consumption on the overall system and of individual processes can be obtained using the `/proc` pseudo-filesystem. Networking information is also available via the `/proc` pseudo-filesystem. For the differentiation between internal and external traffic, the corresponding local pub/sub broker has to further be instrumented with internal and external message and size counters per topic. Considerations of suitable sampling intervals for the profilers are subject to future research.

C. Offloading Decision Engine

The decision the offloading engine has to take is whether to accept or reject a certain task based on the provided metadata and where to place the processing task. To determine if there is at least one allocation that would fulfill all requirements to be able to accept the request and to find a useful allocation strategy is subject to additional research. The problem is related to the knapsack problem and NP-hard.

We provide here as a first estimate an example strategy based on a greedy approach. If the task can be run locally without moving other processes, the processing task is accepted and run locally. Otherwise, the task is added to a temporary list of processing tasks along with all already running processes on the gateway. Out of this set, all processes that were determined to never be offloaded and have to run locally are copied to another set, the gateway set. If the resources on the gateway are not sufficient to locally simultaneously execute all processes in the gateway set, the new processing task has to be rejected. Otherwise, we iterate through the set of remaining processes and give priority to processing task according to the optimization goal to be run locally and allocate resources in a greedy fashion. If that does not yield a possible allocation, we act conservatively and reject the task, otherwise we know that there is at least one possible allocation, although it might not be optimal.

Since offloading decisions are always associated with trade-offs, the optimization will probably be based on multiple objectives at once. Therefore, we also plan to include in future work a solution based on multi-criteria decision analysis methods, such as Technique for the Order of Prioritisation by Similarity to Ideal Solution (TOPSIS) or VIKOR [13].

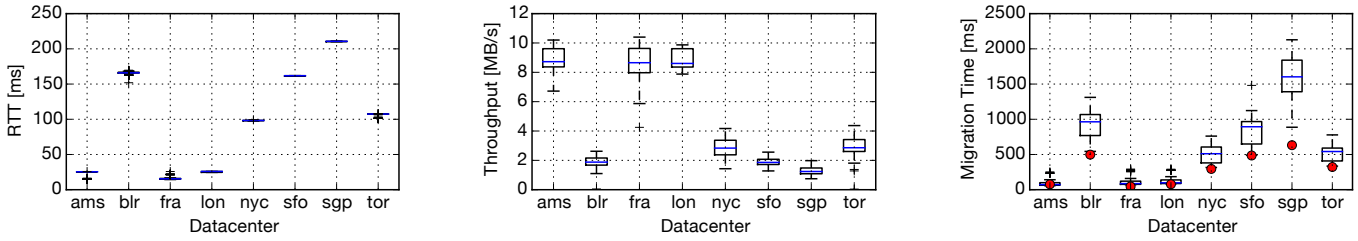
V. Migration Prototype

We have implemented the migration framework of the system outlined above along with a sample application that models a common processing task that might require migration. The prototype uses Message Queue Telemetry Transport (MQTT) as the messaging middleware between both the two hypervisor subsystems, as well as the actual processing tasks and the input data producers and consumers. MQTT is an example for a minimal topic based pub/sub protocol, only offering the publish, subscribe and unsubscribe primitives, showing that the proposed approach can be universally applied.

The hypervisor is written in Python. The execution environment is also a Python interpreter that is controlled by the hypervisor. As a broker we use the mosquitto MQTT broker, deploy it both on the local gateway and remote Cloud instances and bridge the topics between the brokers so that messages are forwarded between the brokers. All clients use the paho MQTT library.

The processing task is a simple image analysis task detecting movement in a camera stream. The processing task as such subscribes on a certain input topic and expects individual images in the JPEG format. It takes the first few images of a stream and stores them as the background. Every subsequent image is checked for movement by subtracting the greyscale representation of the image and the stored background and comparing the output with a certain threshold. The process additionally detects edges on regions that were identified to contain motion and marks those regions with overlay boxes. The original image with the detection boxes as an overlay is the output of the task.

The processing task is implemented using the cv2 library. The state, the implementation holds, consists of a the sequence number of the images that was processed last and the image the task considers to be the background.



(a) Round-trip-time between Berlin and DO. (b) Throughput between Berlin and DO. (c) Resulting migration time.

Fig. 5. Latency, throughput and migration time measurements between Berlin and Digital Ocean (DO) datacenters.

VI. Preliminary Evaluation

We use the prototype to evaluate the cost of a migration between a gateway node and a Cloud-based virtual machine in terms of migration time, which equals to the perceived service downtime. We use a beaglebone black as the local gateway located in Berlin, Germany and as the Cloud counterpart instances in every datacenter of the provider Digital Ocean. The size of the motion detection task is around 3kb. The size of the background image, i.e. the state of the task, varies depending on the background, but is in our case 21kb.

In Figure 5a we measure the latency to the datacenters offered by the Cloud provider. The results were obtained in 128 measurements with the ping utility, that is available on most unix-like systems. Figure 5b shows the throughput to the datacenters, which was also measured 128 times using a download with the wget command. We expect the migration time to be three round-trips and the time to transmit task state and code to the new system. Figure 5c shows the migration times of 64 migrations back and forth along with a red dot marking the estimate made beforehand.

The migration is sufficiently fast even to remote datacenters in Singapore or India, usually in the region under 1s and in the worst case still under 2.5s. The estimation marks the lower bound of the migration time, which can be easily derived using widely available tools. Due to the additional overhead and latency introduced by the messaging middleware, the actual average migration time shows to be about twice as high as the prediction. In conclusion, the migration time is small enough to be tolerable by the end-user for most sensor applications.

VII. Future Work and Conclusion

We argue that the full potential of globally interconnected sensor technology will only be fully utilized through increasing sensor data processing. To realize this vision we presented an initial concept and an architecture draft of gateway to Cloud offloading which enables fast and flexible definition of processing tasks as well as constraints and optimization targets. We outline the design challenges to provide simple and uniform access to processing on constrained gateway nodes as well as fog and Cloud nodes. We show that a common processing task found in IoT systems can be migrated in a reasonable short time in the order of seconds in the worst case. We intend to expand our research prototype and deploy a 3-layered testbed based on realistic gateway, fog and Cloud hardware to carry out a more in-depth performance evaluation.

References

[1] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of Things (IoT): A vision, architectural elements, and future directions," *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1645–1660, Sep. 2013.

[2] B. Zhang, N. Mor, J. Kolb, D. S. Chan, K. Lutz, E. Allman, J. Wawrzynek, E. Lee, and J. Kubiatowicz, "The cloud is not enough: Saving iot from the cloud," in *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '15)*. Santa Clara, CA: USENIX Association, Jul. 2015.

[3] 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*, ser. MCC '12. New York, NY, USA: ACM, 2012, pp. 13–16.

[4] T. Menzel, N. Karowski, D. Happ, V. Handziski, and A. Wolisz, "Social sensor cloud: An architecture meeting cloud-centric iot platform requirements," Apr. 2014, 9th KuVS NGSDP Expert Talk on Next Generation Service Delivery Platforms.

[5] A. Antonic, K. Roankovic, M. Marjanovic, K. Pripuic, and I. P. Zarko, "A mobile crowdsensing ecosystem enabled by a cloud-based publish/subscribe middleware," in *International Conference on Future Internet of Things and Cloud (FiCloud)*. IEEE, 2014, pp. 107–114.

[6] D. Happ, N. Karowski, T. Menzel, V. Handziski, and A. Wolisz, "Meeting IoT Platform Requirements with Open Pub/Sub Solutions," *Annals of Telecommunications*, vol. 72, no. 1, pp. 41–52, 2017.

[7] A. Rowe, M. E. Berges, G. Bhatia, E. Goldman, R. Rajkumar, J. H. Garrett, J. M. Moura, and L. Soibelman, "Sensor Andrew: Large-scale campus-wide sensing and actuation," *IBM Journal of Research and Development*, vol. 55, no. 1.2, pp. 6:1–6:14, Jan. 2011.

[8] A. Davis, J. Parikh, and W. E. Weihl, "Edgecomputing: extending enterprise applications to the edge of the internet," in *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*. ACM, 2004, pp. 180–187.

[9] Intel, "Intelligence From the Data Center to the Edge," 2014.

[10] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *Pervasive Computing, IEEE*, vol. 8, no. 4, pp. 14–23, 2009.

[11] F. Douglis, "Transparent process migration in the sprite operating system," Ph.D. dissertation, Berkeley, CA, USA, 1990.

[12] Y. Kim, J. Kwak, and S. Chong, "Dual-side dynamic controls for cost minimization in mobile cloud computing systems," in *13th International Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks (WiOpt)*, May 2015, pp. 443–450.

[13] H. Wu, "Analysis of offloading decision making in mobile cloud computing," Ph.D. dissertation, Free University of Berlin, 2015.

[14] A. Munir, P. Kansakar, and S. U. Khan, "Ifciot: Integrated fog cloud iot architectural paradigm for future internet of things," *CoRR*, 2017. [Online]. Available: http://arxiv.org/abs/1701.08474

[15] L. F. Bittencourt, M. M. Lopes, I. Petri, and O. F. Rana, "Towards virtual machine migration in fog computing," in *10th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC)*, Nov 2015, pp. 1–8.

[16] M. Aazam and E. N. Huh, "Fog computing micro datacenter based dynamic resource estimation and pricing model for iot," in *29th International Conference on Advanced Information Networking and Applications*, March 2015, pp. 687–694.

[17] D. Bernstein and D. Vij, "Using semantic web ontology for intercloud directories and exchanges." in *International Conference on Internet Computing*, 2010, pp. 18–24.