Open Access

Open Journal of Internet of Things (OJIOT)
Volume 4, Issue 1, 2018
http://www.ronpub.com/ojiot
ISSN 2364-7108

Towards Adaptive Actors for Scalable IoT Applications at the Edge

Jonathan Fürst[A], Mauricio Fadel Argerich[A], Kaifei Chen[B], Ernö Kovacs[A]

[A] NEC Labs Europe, Kurfürsten-Anlage 36, 69115 Heidelberg, Germany,
{jonathan.fuerst, mauricio.fadel, ernoe.kovacs}@neclab.eu
[B] Computer Science Division, UC Berkeley, 387 Soda Hall, Berkeley, USA, kaifei@berkeley.edu

Abstract

Traditional device-cloud architectures are not scalable to the size of future IoT deployments. While edge and fog-computing principles seem like a tangible solution, they increase the programming effort of IoT systems, do not provide the same elasticity guarantees as the cloud and are of much greater hardware heterogeneity. Future IoT applications will be highly distributed and place their computational tasks on any combination of end-devices (sensor nodes, smartphones, drones), edge and cloud resources in order to achieve their application goals. These complex distributed systems require a programming model that allows developers to implement their applications in a simple way (i.e., focus on the application logic) and an execution framework that runs these applications resiliently with a high resource efficiency, while maximizing application utility. Towards such distributed execution runtime, we propose Nandu, an actor based system that adapts and migrates tasks dynamically using developer provided hints as seed information. Nandu allows developers to focus on sequential application logic and transforms their application into distributed, adaptive actors. The resulting actors support fine-grained entry points for the execution environment. These entry points allow local schedulers to adapt actors seamlessly to the current context, while optimizing the overall application utility according to developer provided requirements.

Type of Paper and Keywords

Regular research paper: *IoT, framework, programming model, edge-computing, adaptation*

1 Introduction

Internet of Things (IoT) applications arrive with a paradigm shift compared to traditional Internet applications: huge amounts of data are now generated at the edge of the network and flow towards the cloud to be processed and potentially combined with other data sources. While the first generation of IoT applications often followed a traditional client-server architecture, with thin, low complexity clients and high-performance cloud computing, recent applications are characterized by richer clients (smartphones, drones, cars) that use local sensor data as input for—possibly machine learning based—data processing to support *timely and local decision making* (e.g., UAVs) or user interaction—e.g., for augmented reality (AR) applications [12].

The requirement of a timely feedback loop conflicts with common approaches of data processing in the cloud due to the long round trip times, and is not reliable in a wireless setting. Further, advanced sensor modalities

This paper is accepted at the *International Workshop on Very Large Internet of Things (VLIoT 2018)* in conjunction with the VLDB 2018 Conference in Rio de Janeiro, Brazil. The proceedings of VLIoT@VLDB 2018 are published in the Open Journal of Internet of Things (OJIOT) as special issue.

like video or 3D data have high bandwidth requirements that can quickly congest (wireless) network links and become non-scalable in a cloud-centric solution for a wide deployment like envisioned for future smart cities [3].

Edge or Fog computing [37, 8] are principles that bring cloud resources closer to the user, which can greatly improve responsiveness, resilience, bandwidth usage and thereby the scalability of such applications. However, edge computing *adds more burdens on IoT developers*: programming distributed, dynamic and heterogeneous device-edge-cloud systems requires complex decision making on *(1) application partitioning* into tasks, *(2) task allocation* and *(3) task adaptation*. Application partitioning, meaning the structuring of application logic into components like packages, classes or functions, is a daily task for developers and is static during runtime (i.e., it needs to be done during implementation).

Task allocation and adaptation however, depend highly on the current context during execution time, often unknown to the programmer. As an example, an AR application running on a smartphone must place and adapt tasks differently when in proximity to an edge node , opposed to in a traditional smartphone-cloud environment. It also needs to adapt its behavior to changes in the network link (e.g., 3G vs. WiFi, network partition or downtime) and battery level for mobile devices. In all allocation and adaptation scenarios, the application must adhere to its individual requirements and constraints, such as responsiveness, accuracy, cost, energy consumption, CPU use, and storage.

In this work, we argue that these deployment and adaptation decisions should not be left solely to IoT developers , but happen in interplay with a distributed execution framework: Firstly, the complexity of these decisions moves the developer's focus away from the application logic that contributes to the business goals. Secondly, the number of combinations of different context factors and their implications are too large and possibly not fully known during implementation time. For example, network round-trip times have a high variance in a wireless and mobile scenarios, which is common for IoT applications [5]. Thirdly, programming distributed systems is hard and requires skills beyond the ones of many application developers.

We begin this work with the assumption that IoT programmers should focus on (monolithic) application logic, while the framework transforms this logic into adaptive, distributed components (see Figure 1). While there has been previous work with similar intentions, notably Sapphire [49] and Ray [35], they are targeting slightly different problems. Sapphire focuses on a framework for mobile-cloud systems. Ray focuses on highly parallelized data analytics and machine learning. Both assume homogeneous nodes, and in the case of Ray, in-data center computation. We envision that future IoT applications will span over a heterogeneous set of hardware platforms, from physical IoT devices over close-by edge-nodes to traditional cloud computing, and that such setting needs strong framework support.

Towards these goals, we propose **Nandu**, a first design and prototype of a distributed execution framework for IoT applications using adaptive actors [26], which enables highly dynamic, autonomous and edge-centric application partitioning and task scheduling. Nandu provides:

- Transformation of regular, sequential coding style programs into **distributed actors**.
- A **programming model** that allows developers to easily specify application wide Quality of Service (QoS) requirements and function-level developer hints, consisting of adaptable function parameters (or even multiple function implementations) and their expected utility towards the overall application goal.
- **Dynamic actor adaptation** that uses these developer hints and then optimizes actor adaptation throughout the lifetime of an application using strategies of (1) actor migration, (2) data degradation and (3) actor degradation.

In contrast to previous work [32, 49], Nandu considers different adaptation strategies to best conform to developer provided non-functional Quality of Service (QoS) requirements (e.g., latency, energy, memory use) under a changing execution context. Instead of only considering device-(edge)-cloud architectures, Nandu's worldview is informed by distributed nodes that may exchange tasks in a P2P fashion. The main contributions of this paper are:

- We derive the requirement of framework supported task adaptation for IoT applications from an exemplary use case (Section 2).
- A survey of popular actor systems, considering common IoT requirements (Section 3).
- The design of Nandu, an IoT tailored actor based distributed execution platform (Section 4).
- A prototype implementation and evaluation that show the advantages of Nandu (Section 5).

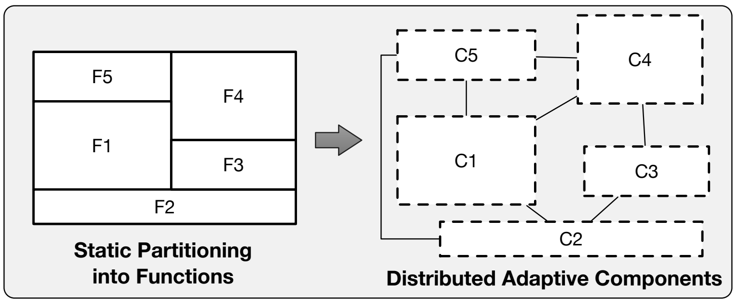

F5
F4
F1
F3
F2
Static Partitioning into Functions
C5
C4
C1
C3
C2
Distributed Adaptive Components

Figure 1: From monolithic application logic to distributed adaptive components

2 Motivation

Worldwide disasters have affected 1.7 billion people and killed 0.7 million between 2005 and 2014 according to numbers by the UN [43]. Information and communication technology (ICT) has been applied in academia and industry to help reduce these numbers. For example, by providing resilient ad-hoc network infrastructure for disaster response [23] or detecting natural disasters, like eruptions of a volcano by deploying wireless sensor nodes [45]. In the event of a disaster, drones are used to locate people in need for help or directly provide support, e.g., in the form of dropping defibrillator devices [15]. We consider a similar application scenario to motivate our work.

2.1 Application Scenario

Semi-autonomous drones, equipped with a monocular and an infrared camera are used by rescue teams to locate people in a difficult to access area in the aftermath of a natural disaster (see Figure 2).

This scenario might require multiple, concurrent running tasks:

T1 (Visual) Simultaneous Localization and Mapping. (V) SLAM based drone navigation requires near real-time sparse point cloud construction and vision-based localization [21].

T2 3D Model Reconstruction and Semantic Understanding. The 3D model and semantic understanding (e.g., road detection) is needed to help first responders to navigate to the disaster survivors. Pre-disaster infrastructure might not be accessible anymore (e.g., due to earthquake, flooding).[1]

T3 Infrared based Detection. Use thermal infrared camera to detect humans and animals.

[1] After the Haiti earthquake in 2010, volunteers used recent satellite data to manually (re-) create accurate mapping on OpenStreetMap (see `https://wiki.openstreetmap.org/wiki/WikiProject_Haiti`).

T4 People Detection. Detect people using computer vision algorithms on pictures taken with the drone's visual-optical camera.

Developing such systems is hard because of the diverse task requirements and dynamically changing networking and energy state:

Placement. T1 needs to run on the drone for real-time location feedback. T1, T2, and T3 can either run on the drone or be offloaded to the edge or cloud, because they are not as time critical as T1. When scheduling tasks, T4 should be a high priority task that happens only when T3 detects objects, while T2 is a continuous but low priority task.

Latency. Each task has different latency requirements. While T2 can be postponed until when resources are available, T1 is mandatory for a correct drone navigation and needs to be executed near real-time. To find survivors in a timely manner [15], T3 and T4 also require high responsiveness.

Accuracy. T1 requires high accuracy to avoid drone accidents, while the people detection performed in T3 and T4 can trade accuracy for a quicker response time.

Energy. Compared to the edge and cloud, the drone is restricted by its limited battery. It must find a good trade off for energy consumption between local computation and offloading.

Network Connection. Since wireless signal can be impacted by various factors (e.g., interference, distance), the system should constantly monitor the network conditions and adjust accordingly.

Ultimately, to work well, such distributed systems need to adapt and allocate concurrently running tasks dynamically during runtime. We derive the following key design goals.

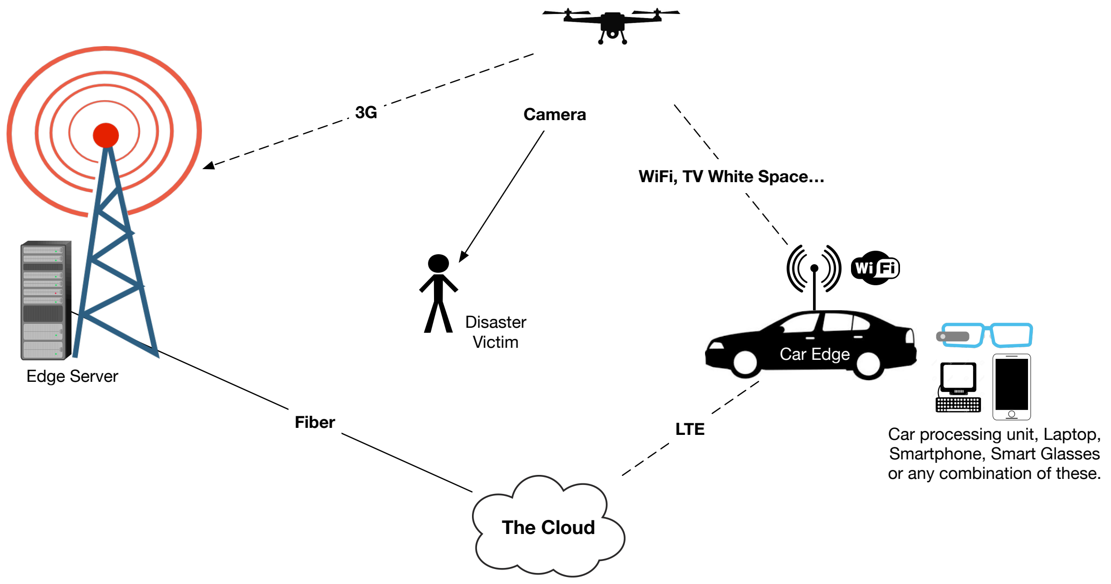

3G
Camera
WiFi, TV White Space...
Disaster Victim
Edge Server
Car Edge
Fiber
LTE
Car processing unit, Laptop, Smartphone, Smart Glasses or any combination of these.
The Cloud

Figure 2: Drone supported disaster management. (Drones are used to timely detect disaster survivors, provide support, and notify first responders. Mobile network infrastructure might be damaged or overloaded.)

2.2 Key Design Goals

(1) Dynamic discovery mechanism. Mobile IoT devices like drones or smartphones need to discover close-by available computational resources in order to distribute their tasks. E.g., multiple drones might distribute tasks among them according to their capabilities.

(2) Abstract execution mechanism away from application logic. To simplify development, programmers should be able to implement their application in a sequential, non-distributed way, as we envision most of them to be non-experts in distributed programming. Programmers are only required to make annotations (e.g., can this function be offloaded or not?) to their functions to transform the application into a distributed one.

(3) Dynamic adaptation. The runtime needs to dynamically adapt and migrate application tasks, depending on the current execution context. Adaptation mechanisms might require hints for the runtime in the form of annotations from the programmer that allow the application to dynamically adjust function parameters or choose different task implementations (e.g., slow and accurate or fast and less accurate people detection algorithm) to maximize the utility under the current context (e.g., node computational resources, node load, networking, battery level) and according to application requirements (e.g., latency).

3 Actor Model

In the actor model, actors are treated as the universal primitive of computation. They can communicate with each other only through message passing. When an actor receives a message it can perform some local processing, create new actors and send itself messages [26]. This simple model provides strong guarantees for highly concurrent systems and has been successfully applied to widely used distributed systems.[2] As such it provides a good foundation for a prospective execution model for highly distributed systems of IoT. We now briefly survey existing actor systems and their applicability for IoT systems (Section 3.1).

Erlang [22] is a functional programming language that follows the actor model. Actors need to be explicitly created at a specified location (i.e., either on the caller's server or on a remote server). Processes (actors) communicate with each other via asynchronous message passing both locally and remotely. Erlang does not support process migration. In traditional Erlang, scaling is constrained by the fact that processes need to be created explicitly and that connections are shared between all nodes, requiring data structures quadratic in the number of nodes at every machine. Chechina et al. [11] propose the scalable distributed Erlang library

[2] E.g., Spark relies on the actor model for distributed computing. Microsoft is scaling their actor based Orleans system up to millions of actors in a data center [7].

to overcome these bottlenecks by dividing nodes into smaller groups and providing a more implicit process placement.

Akka [1] is a popular actor framework on top of the JVM written in Scala. Since version 2.10.0, Scala uses Akka as the default actor library [27]. Like Erlang, Akka adopts the "Let it crash" model for resilience. Akka cluster provides a cluster membership service that relies on a Gossip protocol similar to Amazon's distributed key-value store Dynamo [19]. Akka cluster uses heartbeats in a 1 s interval and the Phi Accrual Failure Detector [25] to calculate the probability for a node being unreachable. Programmers can tweak failure detection by setting a threshold value. Lightbend has successfully run an Akka cluster with 2400 nodes on Google Compute Engine [36].

Orleans [7] adds in contrast to Erlang and Akka the abstraction of virtual actors (grains). The virtual actor abstraction removes the need for the programmer to explicitly create or destroy actors. Actors always exist virtually and their instances are dynamically created by the Orleans runtime on an available server when a message is passed to an actor. Likewise, actors are torn down when they are no longer needed. The virtual to physical actor mappings are stored in a one-hop distributed hash table (DHT) and a large local cache on every server to avoid the otherwise additional network hop for each message that is sent. Actors can be stateless with potentially many instances or restricted to a single activation. Stateless actors allow Orleans to automatically scale out hot actors by creating multiple instances.

3.1 Problems with Existing Actor Systems

Existing actor systems provide a simple model for abstracting concurrent computation. However, they are generally intended for in data center—or even for single machine—computation. Only recently, there are some efforts to extend them to geo-distributed data centers [6]. Actor based systems assume (mostly) reliable and uniform network communication. Last, actor programming frameworks require that programmers adjust their programming style to actor programming, a programming model unfamiliar to many developers.

Opposed to in data center computation, IoT systems often rely on wireless communication, which can be unreliable and not timely. Further, IoT and edge platforms are heterogeneous and are more limited in their resources compared to a server in a data center.

Actors of IoT systems need to adapt themselves according to the current context (network and computational resources). Part of this adaptation should be strategies such as data or task degradation. Further, we argue that because of the potentially wide group of IoT developers, to gain traction, a good framework should enable developers to use common sequential programming style and not expose them directly to actor programming.

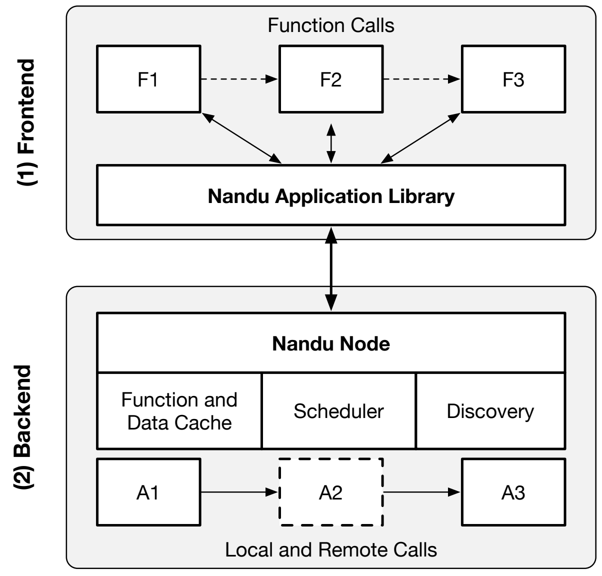


Figure 3: Nandu architecture overview. (Regular functions are intercepted by the Nandu application library and then scheduled as Nandu actors by a local scheduler, either locally or on a remote node.)

In the following, we port the actor model to edge-centric IoT systems by introducing the concept of *adaptive actors*.

4 Towards Adaptive Actors for Scalable IoT Applications

We now describe our design of Nandu, an actor based distributed execution framework and programming model, tailored for highly dynamic, autonomous and edge-centric IoT applications.

4.1 System Overview

Figure 3 depicts the overall proposed architecture of Nandu. Conceptually, we divide it into two parts (1) the application library (frontend) and (2) the node (backend), which are only loosely coupled. The main primitive of Nandu is a node. Each Nandu node can host a variable number of actors. Actors can communicate with other actors on the same node and with actors on remote nodes.

This communication is provided by the hosting node. Nodes communicate with each other through Remote Procedure Calls (RPC). Inspired by Orleans [7] model of virtual actors, our model abstracts the physical actor location away by using a distributed hashing table (DHT) to keep track of current node-actor mappings. This will allow actors to potentially migrate during the runtime of an application. If a function depends on local device I/O (e.g., camera stream, or an actuator), the corresponding actor is bound to the device as well.

To use Nandu, developers only need to import our library and then implement their application logic in a sequential, non-distributed way. They then can annotate (a subset of) functions to enable a conversion of these functions to adaptable actors. Practically, we achieve this transformation to actors by creating a Nandu object for each annotated function that contains the serialized function logic, the function input parameters and adaptation hints. Actors are next scheduled by Nandu either locally or on a remote node. The scheduler performs this decision based on the hints provided by the programmer, profiling of previous executions of the same function and based on its current state of available Nandu nodes (e.g., is there a node close-by with free resources). We plan to use a distributed hashing table to propagate the same state across distributed nodes.

We assume annotated functions to be stateless. In case of a node failure or network partition, we simply re-create the respective actor. For time-critical functions that require high reliability, Nandu can execute a function on multiple nodes and only take the first available result. Nandu uses a cache in form of an in-memory KV-Store to cache actors and input data.

4.2 Programming Model

Our programming model allows existing programs to be swiftly transformed into distributed, adaptive actors, while only exposing developers to a traditional, sequential coding style.

We achieve the transformation of ordinary functions to actors by enabling developers to annotate functions using the decorator design pattern, which provides an interception point to the Nandu application library. Annotated functions need to be self-contained, i.e., they need to contain necessary dependencies (e.g., necessary import statements) and not refer to variables outside the function scope. When an annotated function is invoked, instead of executing it as usual, we return a promise [31], that then serves as input for subsequent function invocations and thus implicitly enables us to link the output of one actor to the input of another. Promise constructs (also known as futures) are included in standard libraries of many major programming languages (e.g., C++. Java, Python). This makes our approach applicable to a wide range of implementation and systems. The function logic itself is scheduled locally or remotely and is executed when its input becomes available (i.e., the output of a previous function has been computed and thus the promise becomes available).

```python
from nandu import adapt
@adapt(offload = True,
    size = {640: 10, 480: 9, 320: 6, 240: 3})
def resize_image(img, img_id, size = 640):
    import imutils
    return(imutils.resize(
      img, width=min(size, img.shape[1])),
      img_id)
```

Listing 1: Nandu programming model. (Developers can annotate functions with hints for the runtime on what parameters to adapt dynamically. In this example, the image size is adapted and the function is potentially offloaded dynamically.)

To enable actor adaptations, function decorators allow developers also to specify *function utility* and *adaptation parameters* as *hints* to the Nandu scheduler. As an example, Listing 1 shows a code snippet using our Nandu prototype implemented in Python. The `offload=True` parameter explicitly allows the Nandu scheduler to place the actor remotely. The `size` dictionary parameter represents a selection of possible `size` parameter values together with their expected utility towards the application goal (i.e., greater image sizes will provide better accuracy). Nandu uses such hints to support its scheduling and adaptation decisions as we describe now.

4.2.1 Developer Hints

Developers assign adaptation hints for the overall application as well as for different functions.

Overall application hints. These hints represent non-functional requirements that should be achieved by the application or application components, e.g., “overall people detection pipeline should occur in $\leq 1\,\mathrm{s}$”. As such, they must define a measurable proxy that captures a requirement well.

Individual function hints. These hints indicate that a function is adaptable, which of its parameters can be adapted, and what the expected utility of different adaption values is. Utility represents the impact of different parameter values towards achievement of the overall application requirements, which cannot be measured by the system itself. For example, depending on the function logic and the specific parameter, this

Table 1: Execution context and possible adaptations

Task	Execution Context	Possible Adaptation
Search a suspect in a crowd using surveillance cameras and face recognition.	Many people in camera view. Face-recognition slow, latency outside requirements.	Degrade parameters of classifier to perform faster.
Predict maintenance of CNC machine in a factory in near real-time.	Required AI accelerator (GPU or dedicated ASIC) not available on edge.	Migrate to cloud-node equipped with AI accelerator.
Estimate room occupancy through wireless connected CO2 sensors to determine HVAC set-points.	Poor network bandwidth, it is not possible to send all the data in real time.	Reduce sampling rate or aggregate data within an interval, before sending it, to reduce bandwidth usage.

utility value can represent the accuracy of a predictive model; any Quality of Experience (QoE) measure, such as the user-perceived quality of different media compression levels; or any other performance measure. For the application scenario from Section 2, the developer might for example specify different classifiers and assign them utility values according to their expected accuracy results. Application hints together with individual function hints and online actor profiling allow us then to optimise the overall application utility through actor adaptation.

4.3 Adaptation

To capture different IoT execution contexts, Nandu uses three adaptation strategies: (1) actor migration, where actors are migrated to more powerful nodes, (2) data degradation, where actor input is degraded in order to adapt the application to changing network conditions or available resources and (3) actor degradation, where a different actor implementation is selected (e.g., a different classifier).

Any of these strategies might be a viable way to adapt a distributed application to the current execution context: Migration is intended for when a stable network link is available and data movement between actors is small; data degradation helps to mitigate varying network bandwidths, either during a single deployment (e.g., network bandwidth changes during the lifetime of an application) or for different deployment scenarios (e.g., the same application is deployed to wireless and wired devices); actor degradation is able to adapt a task to available computing resources, e.g., when additional load occurs or when a network link to a higher power node becomes unavailable and execution needs now occur on a lower power node (e.g., the drone from the scenario described in Section 2). Table 1 gives further examples of applications and their execution context dependent adaptations.

Nandu selects and executes these adaptation strategies by combining developer provided adaptation hints with online actor profiling, while using overall non-functional application requirements as a target for optimization (e.g., latency).

4.3.1 Optimization

Nandu optimizes adaptation by combining the developer hints introduced in Section 4.2.1 with online profiling to establish cost and utility of each actor execution. Nandu assigns a utility value for each actor execution. This utility value represents how well the actor performs according to non-functional application requirements that cannot be measured: accuracy, QoE or other performance measures. In addition, a cost value represents how much effort is involved in the task execution. This cost is monitored by Nandu through online profiling. In this work we focus on latency costs, however the same model can also be applied to other measurable cost factors depending on application wide QoS requirements: e.g. bandwidth, CPU or energy use.

The Nandu Optimizer is in charge of finding the best adaptation strategy, using the previously defined adaptation mechanisms (function annotations) to deliver the highest possible utility with the defined cost constraints in the current context (e.g. current available CPU or network bandwidth).

Finding the best strategy is a complex task due to the large number of combinations of adaptation mechanisms. To solve this problem, without adding a substantial overhead to the overall system, we currently use a simple heuristic value selection algorithm for

Algorithm 1 **Simplified parameter selection logic**

```
oldParams = {...}                              ▷ previous function parameters
windowSize = 20               ▷ tunable velocity parameter to fit application
i = 0

function DECORATOR(oldParams)
    if MMEAN(ObservedCosts, windowSize) > $Constraint$ then
        params = DEGRADE(oldParams); i = 0
    else
        if $i$ >windowSize then
            params = UPGRADE(oldParams); i = 0
        else
            params = oldParams; i++
        end if
    end if
end function
```

dynamic parameters (depicted in Algorithm 1). When an actor receives input and is executed, we monitor and save its execution time and input parameters. We then use these statistics to estimate the cost for future executions using a given set of parameter values. For example, in the scenario from Section 2, the cost is the time that it takes to process each image, from the moment the image is captured to the moment when the result of the people detection step is obtained.

When an actor is executing, Nandu chooses the parameter values that it expects will yield the best cost-utility trade-off (e.g., latency-accuracy). Selecting a parameter value not only affects the execution of the actor itself, but also the execution of subsequent actors. Our model thus estimates the cost for the whole execution pipeline. Practically, we estimate this cost by calculating the moving mean of the last K executions. If no profiling data is present for a given set of parameter values, 0 is returned to encourage the exploration of unknown sets of values. The window size is a parameter for our optimization model that can be configured to meet the expected velocity of the application (i.e. how swiftly Nandu should adapt to changing execution costs). More formally, the estimated cost is calculated as:

$$\sum_{f=0}^{N}\left(\frac{1}{WS}\sum_{k=0}^{WS} c_{f(params=P)}\right) \qquad (1)$$

Where N is the overall number of actors for the processing job, WS is the window size for which we calculate the moving mean (in our case $WS = 20$) and $c_{f(params=P)}$ is the cost of running function f with parameters P.

As described in Section 4.2, developers provide hints in form of parameter-value pairs to Nandu. During optimization, these hints are used to maximize the overall application utility. For instance, in our example, we specify different image resolutions and how they affect application accuracy. For the initial execution of an actor, we sort parameter values by descending accuracy, and start profiling the cost for each value in this order during runtime. In our example, Nandu will first select the image size that yields the highest accuracy because no profile data exists yet. During the next iteration, Nandu estimates the cost by using its profiled data; if this profiled cost (i.e., the latency in our example) is less than the cost target, Nandu continues to use this parameter value, otherwise it selects the next best value to lower the cost. This process continues until the cost target is met with the highest possible accuracy.

5 Implementation and Evaluation

We have implemented a prototype of Nandu in Python 3.6 to show that our programming model provides a simple abstraction for distributed IoT programming and that our adaptation strategies can support application requirements under dynamic conditions. Because we focus our evaluation on the benefit of our programming model and the benefit of dynamic adaptation, we currently use static host addresses instead of a DHT based and fully decentralized system. We use Python decorators to allow developers to specify adaptation hints and Python's future statement as placeholder for a function output. This allows us to schedule the different actors before their final input is available as described further in Section 4.2. To serialize function logic we use cloudpickle [16], which allows us to serialize Python objects not supported by pickle module in the Python standard library. Cloudpickle has been applied previously to Spark based big data processing

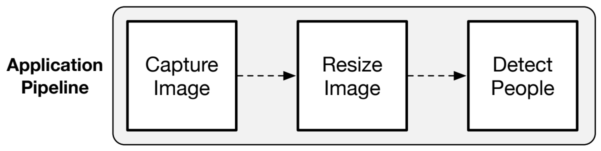


Figure 4: Simplified people detection pipeline

applications [47]. We have build an RPC server and client using the Python's asyncio module that runs on each Nandu node and abstracts message passing and serialization away from the Nandu application library. When a node receives a new message, we de-serialize it, and pass the resulting Nandu object to the scheduler, which decides on its execution.

We also implemented a simplified version of the Computer Vision-based people detection example introduced in Section 2 using OpenCV that consists of two devices: (a) the drone, which goal is to detect people in a stream of images in collaboration with (b) an edge device (see Figure 4).

The image processing pipeline is divided in three main steps: (1) capture the image, (2) resize the image, and (3) detect people in the image. Each step is performed by a different function; (1) and (2) are run on the drone while (3) might be migrated to the edge component. We use the pre-trained HOG + LinearSVM model from OpenCV [9] to implement the people detection step.

5.1 Experimental Setup

We use mininet [34], a simple network emulator, to simulate different network environments. Despite its simplicity, researchers have been able to reproduce the results of more than 40 networking papers with mininet [46]. Mininet does not explicitly simulate a wireless network setting, however our wireless application scenario is just an example of a changing network (e.g., recently, Zhang et al. [48] have shown wide bandwidth and RTT variances for Internet scale applications). This is why, without loss of generality, we only simulate different bandwidths to show how Nandu can adapt under different network conditions. As for dataset, we use a subset of the INRIA Person Dataset [18] with 410 images. The subset was created by selecting all the images with a size larger than 700 kB from the positive samples and images larger than 600 kB from the negative samples. The different thresholds were chosen to have a fairer split between positive and negative samples for our use case.

5.2 Latency-Accuracy Trade-offs

Using mininet, we simulate different network speeds: 5, 10, 20, 50 and 100 Mbps. To understand the latency accuracy trade-off, and to establish a baseline for Nandu's adaptive optimization, we then run experiments for static wireless link-image size pairs. Figure 5 depicts our measured latency values. As expected, latency is determined by two variables: (1) network link speed and (2) image sizes.

Moving on to the quality of the received results, Table 2 shows what accuracy we achieve with different image sizes and the HOG + LinearSVM classifier. For our application, we are only interested in detecting people, and not in the exact number of people. Therefore, we also classify results where the number of people is not exact as correct. From our results, we see that accuracy is relatively stable across image sizes and between 83 and 91 %.

Our experiments reveal two important insights: (1) applications can gain accuracy and retain latency requirements by adapting to different networking conditions, and (2) developers implementing their application without Nandu would need to manually pick an—application requirements dependent—adequate image resize parameter that is not optimal for all execution contexts or implement custom optimization strategies. With these insights, we now evaluate Nandu's performance for changing execution contexts.

5.3 Input Degradation

For evaluating Nandu's input degradation strategy, we annotate the resize function of our simple people detection pipeline with Nandu developer hints as shown in Listing 1, *adding only one additional line of code*. We further set the overall QoS requirement to 1 s latency. Then we run the application again for the same network speeds as in Section 5.2.

As we can see in Figure 6, the size, to which images were resized by Nandu, varies according to the available network bandwidth: as bandwidth grows, Nandu uses higher resolution images in order to keep the highest accuracy possible while not exceeding the target latency requirement of the overall application. Note that, because we are using a discrete set of parameter values as input to Nandu, the optimizer is not able to pick an optimal image size value for some network speeds. E.g., Nandu selects both, 480 px and 640 px for 100 Mbps, because perceived latencies of 480 px are 1 s, while latencies of 640 px are slightly larger than 1 s—an optimal value would be somewhere in between.

Figure 7 depicts the distribution of the completion times of our runs. Even with the sub-optimal, discrete

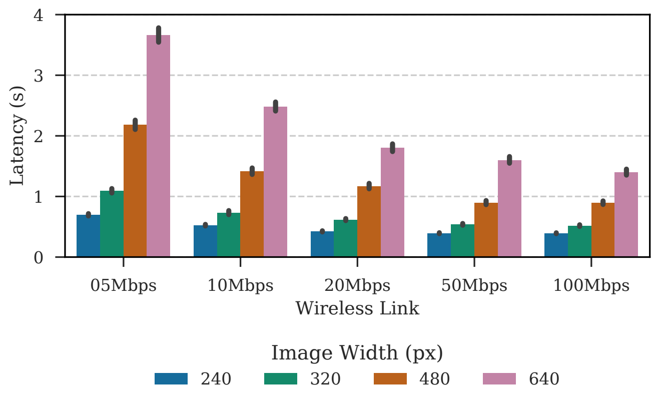


Figure 5: Network link speeds and experienced latency. (Different network speeds result in highly different latency results. No single image width works for all network speeds if application requirements should be met.)

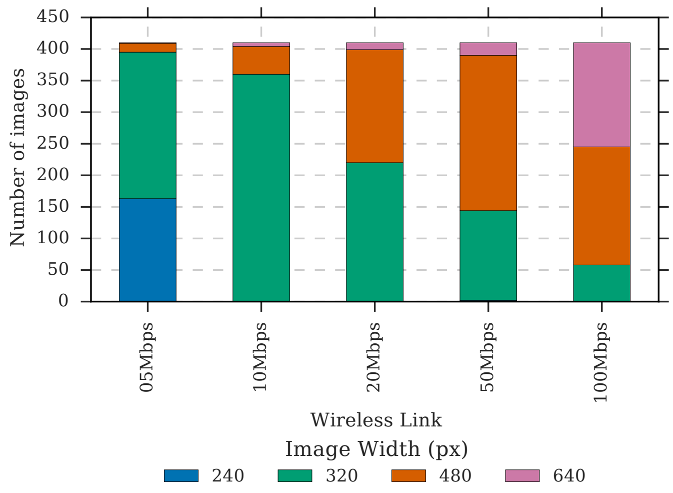


Figure 6: Selected image sizes for application runs. (Nandu selects the best possible size among the discrete set of supplied parameter values in developer hints according to application requirements and execution context.)

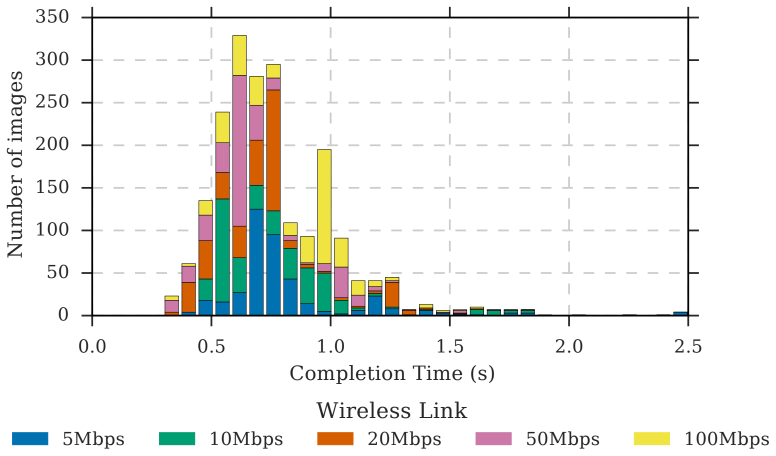


Figure 7: Stacked completion times of application runs. (The majority of images is processed in $\leq 1\,\mathrm{s}$ according to application requirements specified by the developer)

Table 2: Accuracy for different image sizes. (Larger image sizes result in higher accuracy.)

Image Size (px)	240	320	480	640
Accuracy (%)	86.1	88.3	88.8	90.73

Table 3: Mean latencies and accuracy results for different network speeds

Wireless Link (Mbps)	5	10	20	50	100
Mean Time (s)	0.82	0.76	0.72	0.69	0.85
Accuracy (%)	87.3	88.8	86.8	88.8	90.0

Table 4: Accuracy and proportional processing time of HOG descriptor with different scale values using 480px images

Classifier accuracy	Low	Medium	High
Scale	1.30	1.15	1.05
Accuracy (%)	80.7	85.7	88.8
Proportional proc. time (%)	100	145	351

size selection, the great majority of images were processed in $\leq$ 1 s. This is in strong contrast to our results in Figure 5, where some size values result in latencies above 3 s—a three-fold violation of application latency requirements.

Further, Table 3 shows that our mean accuracy over all bandwidth speeds is 88.34 %, which is higher than what we might achieve with a static parameter selection (e.g., Table 2 shows that picking an image size of 230 px, results in an accuracy of only 83.6 %). It also shows the accuracy increases with higher available bandwidth, because it can transmit more images with higher resolution.

5.4 Actor Degradation

Next we look actor degradation as an adaptation strategy by implementing a low, medium and high accuracy method of the HOG + LinearSVM classifier by modifying the value of the parameter scale, for the `detectMultiScale` method of the HOG detector. To be able to detect people close to the camera, as well as people further away, HOG calculates features for different scales of the same image. A smaller scale value results in more steps, while a larger value results in less steps. The resulting classifiers and their accuracy and latency results can be seen in Table 4. We then place the people detection actor on the edge node. We create additional load for two time intervals using `stress` (options: `-c 2`, spawning two `sqrt()` processes) a simple workload generator for Linux [44]. We set the overall QoS again to 1 s.

Figure 8 shows the result of this experiment. At the beginning, Nandu uses the high accuracy method. When CPU resources need to be shared with `stress`, it degrades first to the medium accuracy method and then to the low accuracy method to reach its latency goals. When additional load at the node is again reduced (at 180 s), Nandu returns quickly to its high accuracy method. Overall, Nandu achieves a mean accuracy of 84.88 % and a mean latency of 0.76 s.

5.5 Actor Migration

To evaluate Nandu's actor migration strategy, we again use `stress` to simulate additional load, this time on the drone (see Figure 9). This increases latency of the people detection task above QoS requirements and thus triggers an actor migration from drone to edge (40 s). When load is reduced, Nandu migrates the actor back to the drone dynamically (at 110 s and finally at 180 s). Overall, Nandu achieves a medium latency of 0.70 s across all of our experiments. Note, that accuracy is not influenced by actor migration, as we are solely adapting the placement of computation.

Nandu decides dynamically and fine-grained on migration, depending on available CPU resources on the Drone. To evaluate this, we again use mininet to evaluate different CPU shares by restricting available CPU bandwidth on the drone host (mininet relies on the CFS bandwidth control of the Linux kernel [42], which allows to explicitly set an upper CPU bandwidth limit for a process). Figure 10 depicts the actor placement distribution for different CPU shares. For smaller shares, actors are placed pre-dominantly on the edge server, while from 40 % upwards, actors are solely placed on the drone.

5.6 Adaptation Overhead

Finally, to evaluate the adaptation overhead of Nandu we compare the execution time of a Nandu actor with an ordinary Python function (1000 repetitions). This experiment results in a time difference of 4 ms for a single execution. The overhead is mainly due to the added execution time for the annotation decorator, which is invoked for each call. Despite that, this small overhead results in fewer lines of code (LoC).

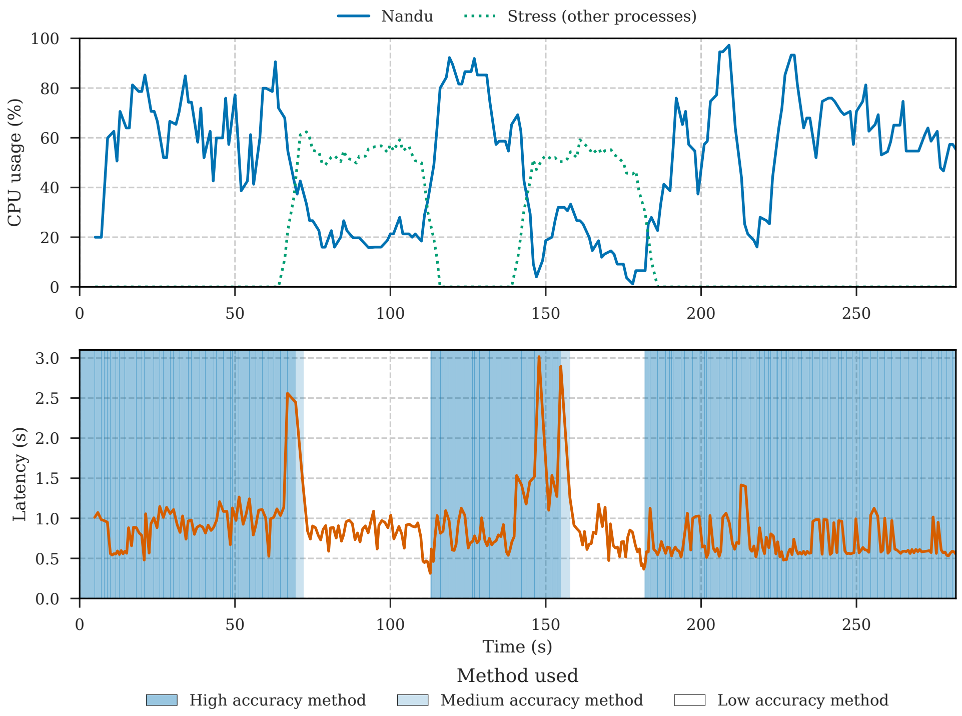


Figure 8: Actor degradation. (Increased load on the hosting node leads to increased latency after 60s and to dynamic actor adaptation to stay in QoS specified latency requirements.)

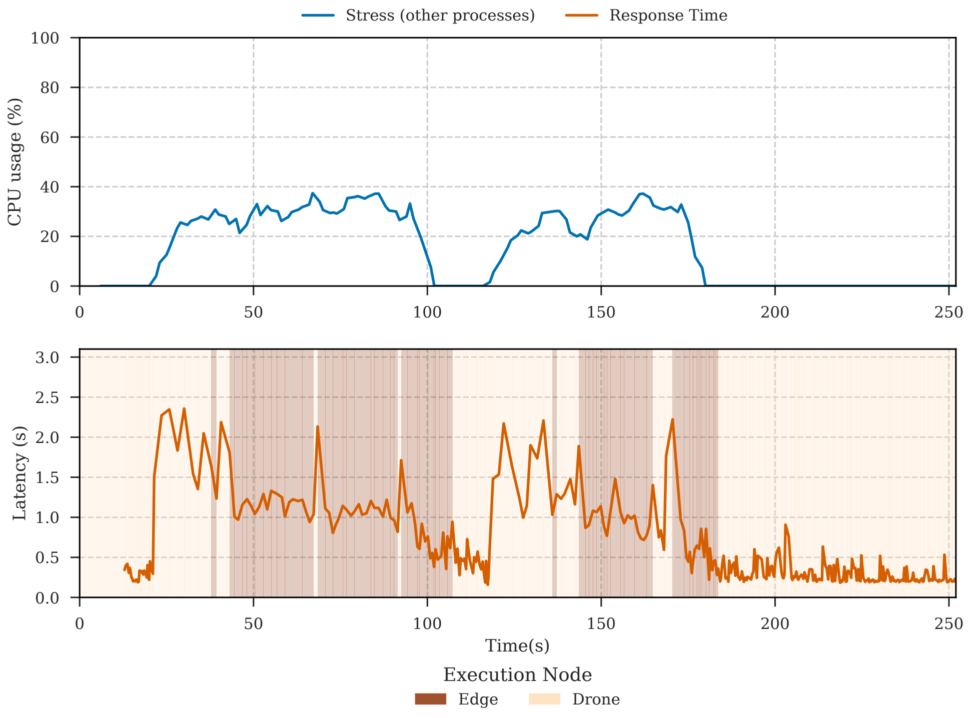


Figure 9: Actor migration. (Additional load on the drone dynamically triggers an actor migration to the edge.)

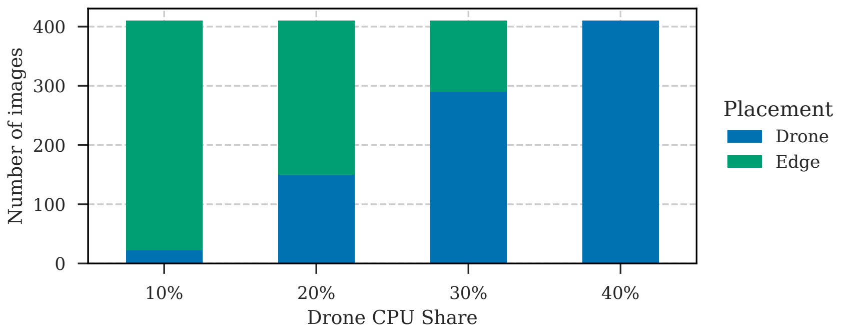


Figure 10: Placement distribution. (Nandu allocates actors dynamically on drone and edge according to available CPU resources.)

6 Discussion

Overall, these results show that Nandu is able to adapt to different networking and computing conditions by dynamically applying its adaptation strategy in unison with developer provided hints and overall application latency requirements. Because we use a discrete set of values for adaptation, the results are sub-optimal. Applying more advanced optimization strategies and a more fine-grained selection of input parameters (e.g., in our application for the image resolution) is a topic for future work. Specifically, we are currently exploring two different techniques: (1) Discrete Optimization and (2) Reinforcement Learning.

Because the decisions of task migration and degradation are discrete and the input degradation could be discrete or continuous, it is necessary to frame the problem in the discrete domain, as a discrete optimization problem. To define the adaptation problem as such, we consider the utility as our objective function, and our non-functional requirements as the constraints. Another approach that we consider is reinforcement learning. In this case, our system is seen as an agent; its state is seen as the current context and its own performance, in terms of utility and cost; an action is a change in its optimization strategy; and the reward for performing this action, is the improvement (or decrease) in its performance that this action had.

Our current model puts the responsibility of assigning utility values to different adaptation options on the developer. For different input degradations (e.g., image resolutions, frame rate) this can be achieved easily. However, the relationship between accuracy results and different ML based classifiers might sometimes be not obvious. We therefore consider to add an off-line profiling phase to our system, where we obtain accuracy values for different classifiers by using a testing data set as input (i.e., including ground truth values).

For node discovery, we are currently exploring several options: (1) Traditional service-discovery protocols like Avahi, Bonjour, Zeroconf for local network discovery; (2) a (local) message broker (e.g., MQTT); (3) Named Data Networking (NDN/CCN) principles for node discovery, similar to the work in [30], a Nandu node could simply send interest packets containing the name of the actor into the network to allow close-by nodes to perform the execution opportunistically and return the result in a NDN data packet.

7 Related Work

There are several related research fields to our work. We now discuss a selection of relevant works.

7.1 Distributed Execution Frameworks

In [32], the authors develop MagnetOS, a programming model for ad hoc networks where a thin distributed operating system layer makes the entire network appear to applications as a single virtual machine. MagnetOS then dynamically partitions a set of communicating Java objects in a sensor network with a focus on energy efficiency.

Sapphire [49] requires programmers to specify per-object deployment managers which aid in runtime object placement decisions, while abstracting away complexities of inter-object communication. Sapphire only focuses on mobile-cloud systems and leaves deployment decisions to the developer (application and deployment logic is split). In our system, deployment is decided by the runtime based on local and global system context.

Beam [40, 39] is a framework that simplifies IoT applications by letting them specify “what should be sensed or inferred”, without worrying about “how it is sensed or inferred.” Beam introduces the abstraction of an inference graph to decouple applications from the mechanics of sensing and drawing inferences.

Rivulet [4] is a fault-tolerant distributed platform for running smart-home applications that can handle typical failures like link losses, network partitions, sensor failures, and device crashes. It provides distributed platform for running smart-home applications on heterogeneous consumer appliances.

FogFlow [13] develops a programming model for edge computing where developers provide a directed acyclic graph and declarative granularity and stream shuffling hints for stream processing. FogFlow then manages task allocation of data processing tasks over cloud and edges for minimal latency and low bandwidth consumption.

In [38, 20] the authors develop the concept of tasklets, which are fine-grained computation units that are executed by a distributed run-time according to developer specified Quality of Computation (QoC) goals. Tasklets are similar to our approach in that they can be used to offload small computation tasks. However, our Nandu run-time further adapts the execution logic of a task dynamically according to QoS requirements and developer hints. Results are retrieved as promises opposed to blocking calls.

Ray [35] focuses on large-scale machine learning and reinforcement learning applications by allowing data scientists to easily parallelize existing applications. Dask distributed [2] is an effort out of the SciPy community with similar goals. Both approaches focus on simplifying and accelerating data science applications and an in-cluster computation. Our focus is on dynamic IoT-edge systems. Recently, [30] proposes NFaaS: named function as a service, in which close-by services (e.g., edge-hosted) are discovered by clients by the means of content centric networking principles (NDN). Their work is orthogonal to ours, as NDN can be used to discover close-by nodes more efficiently than using overlay networks.

7.2 Code Offloading

Code offloading approaches have been applied extensively in the mobile computing domain due to the limited on-device compute capabilities. [37] motivates the idea of cloudlets as means to overcome mobile resource limitations and latency limitations of cloud computing. Cloudlets are VM-based computing resources that can be used seamlessly by close-by mobile applications. MAUI [17] follows with an implementation of such a system for .NET on Windows mobile that shows superior performance compared to solely on-device computing strategies. Both CloneCloud [14] and COMET [24] use VMs to parallel execute tasks in the cloud. Our work is inspired by the performance gains achieved with such code-offloading techniques, but with Nandu we argue that in the IoT domain, tasks need not only be offloaded, but also adapted dynamically by the run-time.

7.3 Programming Abstraction

The goal of a good programming abstraction should be to enable programmers to express what they want their program to do, without specifying all details on how to. Senergy [28] is a framework for programming mobile applications. It supports programmers in automating common latency, power, accuracy trade-offs by letting them specify a priority among them. ENT [10] provides a type-based proactive and adaptive mode-based energy management at the application level, where developers characterize energy behavior of different program fragments with modes. In Nandu, we combine these ideas of run-time supported task adaptation with fine grained task migration.

Policy based network management [41] can provide further useful abstractions by applying policy based management to networking. Policies are defined as rules that define the states and behaviors of the network. This allows for simple, dynamic adaptations in large-scale distributed systems, without modifying the implementations. Nandu follows an orthogonal approach, where the execution framework has fine-grained control over distributed applications logic. To enable this fine-grained control, we have focused on IoT applications, the programming model and on framework support for different adaptation strategies (migration, task degradation and input degradation). Integrating policy based network management into our system can be an additional useful adaptation modality and create synergy between application and networking layer as has been shown in [29].

8 Conclusion and Future Work

We presented Nandu, an adaptive, actor based execution environment for distributed IoT applications. Nandu enables applications at the edge of the network to dynamically use available resources like envisioned in edge-computing, while freeing developers from the tedious and complex tasks of distributed system development. Applications dynamically adapt to the current execution context, like network link quality and available computational resources, through an interplay of developer provided hints and local schedulers at each participating node. Our prototype is able to adhere to application latency requirements while maximizing achieved accuracy dynamically.

Nandu is actively ongoing research in our group, and we see much potential for additions and future work. In similar fashion to [33], which applies machine learning to adaptive video streaming, we are actively investigating the application of reinforcement learning for scheduling decisions in the IoT space. Another aspect for our future work is to improve Nandu's security principles. We are looking in actor isolation techniques, e.g., through the use of individual actor unikernels. For node discovery, we are exploring Named Data Networking (NDN/CCN) principles for node discovery, similar to the work in [30], where the system sends interest packets into the network that contain the name of functions and returns the result in data packets.

References

[1] Akka, "Build highly concurrent, distributed, and resilient message-driven applications on the jvm," https://github.com/akka/akka, 2018.

[2] Anaconda, Inc, "Dask: Distributed computation in python," https://github.com/dask/distributed, 2018.

[3] G. Ananthanarayanan, P. Bahl, P. Bodík, K. Chintalapudi, M. Philipose, L. Ravindranath, and S. Sinha, "Real-time video analytics: The killer app for edge computing," *Computer*, vol. 50, no. 10, pp. 58–67, 2017.

[4] M. S. Ardekani, R. P. Singh, N. Agrawal, D. B. Terry, and R. O. Suminto, "Rivulet: a fault-tolerant platform for smart-home applications," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. ACM, 2017, pp. 41–54.

[5] N. Baccour, A. Koubâa, L. Mottola, M. A. Zúñiga, H. Youssef, C. A. Boano, and M. Alves, "Radio link quality estimation in wireless sensor networks: A survey," *ACM Transactions on Sensor Networks (TOSN)*, vol. 8, no. 4, p. 34, 2012.

[6] P. A. Bernstein, S. Burckhardt, S. Bykov, N. Crooks, J. M. Faleiro, G. Kliot, A. Kumbhare, M. R. Rahman, V. Shah, A. Szekeres *et al.*, "Geo-distribution of actor-based services," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, p. 107, 2017.

[7] P. A. Bernstein, S. Bykov, A. Geller, G. Kliot, and J. Thelin, "Orleans: Distributed virtual actors for programmability and scalability," *MSR-TR-2014–41*, 2014.

[8] 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*. ACM, 2012, pp. 13–16.

[9] G. Bradski and A. Kaehler, "Opencv," *Dr. Dobb?s journal of software tools*, vol. 3, 2000.

[10] A. Canino and Y. D. Liu, "Proactive and adaptive energy-aware programming with mixed typechecking," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2017, pp. 217–232.

[11] N. Chechina, P. Trinder, A. Ghaffari, R. Green, K. Lundin, and R. Virding, "The design of scalable distributed erlang," in *Proceedings of the 24th Symposium on Implementation and Application of Functional Languages (IFL 2012)*, 2012, p. 461.

[12] K. Chen, J. Fürst, J. Kolb, H.-S. Kim, X. Jin, D. E. Culler, and R. H. Katz, "Snaplink: Fast and accurate vision-based appliance control in large commercial buildings," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 1, no. 4, p. 129, 2018.

[13] B. Cheng, G. Solmaz, F. Cirillo, E. Kovacs, K. Terasawa, and A. Kitazawa, "Fogflow: Easy programming of iot services over cloud and edges for smart cities," *IEEE Internet of Things Journal*, 2017.

[14] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: elastic execution between mobile device and cloud," in *Proceedings of the sixth conference on Computer systems*. ACM, 2011, pp. 301–314.

[15] A. Claesson, A. Bäckman, M. Ringh, L. Svensson, P. Nordberg, T. Djärv, and J. Hollenberg, "Time to delivery of an automated external defibrillator using a drone for simulated out-of-hospital cardiac arrests vs emergency medical services," *Jama*, vol. 317, no. 22, pp. 2332–2334, 2017.

[16] C. contributors, "cloudpickle," https://github.com/cloudpipe/cloudpickle, 2018.

[17] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: making smartphones last longer with code offload," in *Proceedings of the 8th international conference on Mobile systems, applications, and services*. ACM, 2010, pp. 49–62.

[18] N. Dalal and B. Triggs, "Inria person dataset," *Online: http://pascal. inrialpes. fr/data/human*, 2005.

[19] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value

store," in *ACM SIGOPS operating systems review*, vol. 41, no. 6. ACM, 2007, pp. 205–220.

[20] J. Edinger, D. Schäfer, C. Krupitzer, V. Raychoudhury, and C. Becker, "Fault-avoidance strategies for context-aware schedulers in pervasive computing systems," in *Pervasive Computing and Communications (PerCom), 2017 IEEE International Conference on*. IEEE, 2017, pp. 79–88.

[21] J. Engel, J. Sturm, and D. Cremers, "Scale-aware navigation of a low-cost quadrocopter with a monocular camera," *Robotics and Autonomous Systems*, vol. 62, no. 11, pp. 1646–1656, 2014.

[22] Erlang, "Build massively scalable soft real-time systems," http://www.erlang.org/, 2018.

[23] S. M. George, W. Zhou, H. Chenji, M. Won, Y. O. Lee, A. Pazarloglou, R. Stoleru, and P. Barooah, "Distressnet: a wireless ad hoc and sensor network architecture for situation management in disaster response," *IEEE Communications Magazine*, vol. 48, no. 3, 2010.

[24] M. S. Gordon, D. A. Jamshidi, S. A. Mahlke, Z. M. Mao, and X. Chen, "Comet: Code offload by migrating execution transparently." in *OSDI*, vol. 12, 2012, pp. 93–106.

[25] N. Hayashibara, X. Defago, R. Yared, and T. Katayama, "The/spl phi/accrual failure detector," in *Reliable Distributed Systems, 2004. Proceedings of the 23rd IEEE International Symposium on*. IEEE, 2004, pp. 66–78.

[26] C. Hewitt, "Actor model of computation: scalable robust information systems," *arXiv preprint arXiv:1008.1459*, 2010.

[27] V. Jovanovic and P. Haller, "The scala actors migration guide," http://docs.scala-lang.org/overviews/core/actors-migration-guide.html.

[28] A. Kansal, S. Saponas, A. Brush, K. S. McKinley, T. Mytkowicz, and R. Ziola, "The latency, accuracy, and battery (lab) abstraction: programmer productivity and energy efficiency for continuous mobile context sensing," *ACM SIGPLAN Notices*, vol. 48, no. 10, pp. 661–676, 2013.

[29] V. Karagiannis and A. Papageorgiou, "Network-integrated edge computing orchestrator for application placement," in *Network and Service Management (CNSM), 2017 13th International Conference on*. IEEE, 2017, pp. 1–5.

[30] M. Król and I. Psaras, "Nfaas: named function as a service," in *Proceedings of the 4th ACM Conference on Information-Centric Networking*. ACM, 2017, pp. 134–144.

[31] B. Liskov and L. Shrira, *Promises: linguistic support for efficient asynchronous procedure calls in distributed systems*. ACM, 1988, vol. 23, no. 7.

[32] H. Liu, T. Roeder, K. Walsh, R. Barr, and E. G. Sirer, "Design and implementation of a single system image operating system for ad hoc networks," in *Proceedings of the 3rd international conference on Mobile systems, applications, and services*. ACM, 2005, pp. 149–162.

[33] H. Mao, R. Netravali, and M. Alizadeh, "Neural adaptive video streaming with pensieve," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017, pp. 197–210.

[34] Mininet Team, "Mininet," http://mininet.org/, 2018.

[35] R. Nishihara, P. Moritz, S. Wang, A. Tumanov, W. Paul, J. Schleier-Smith, R. Liaw, M. Niknami, M. I. Jordan, and I. Stoica, "Real-time machine learning: The missing pieces," in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. ACM, 2017, pp. 106–110.

[36] P. Nordwall, "Running a 2400 akka nodes cluster on google compute engine," 2013.

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

[38] D. Schafer, J. Edinger, J. M. Paluska, S. VanSyckel, and C. Becker, "Tasklets:" better than best-effort" computing," in *Computer Communication and Networks (ICCCN), 2016 25th International Conference on*. IEEE, 2016, pp. 1–11.

[39] C. Shen, R. P. Singh, A. Phanishayee, A. Kansal, and R. Mahajan, "Beam: Ending monolithic applications for connected devices." in *USENIX Annual Technical Conference*, 2016, pp. 143–157.

[40] R. P. Singh, C. Shen, A. Phanishayee, A. Kansal, and R. Mahajan, "A case for ending monolithic apps for connected devices." in *HotOS*, 2015.

[41] M. Sloman, "Policy driven management for distributed systems," *Journal of network and Systems Management*, vol. 2, no. 4, pp. 333–360, 1994.

[42] P. Turner, B. B. Rao, and N. Rao, "Cpu bandwidth control for cfs," in *Linux Symposium*, vol. 10. Citeseer, 2010, pp. 245–254.

[43] United Nations Office of Disaster Risk Reduction, "The economic and human impact of disasters in the last 10 years," www.unisdr.org/files/42862_economichumanimpact20052014unisdr.pdf.

[44] A. Waterland, "stress," https://people.seas.harvard.edu/~apw/stress/, 2014.

[45] G. Werner-Allen, K. Lorincz, M. Ruiz, O. Marcillo, J. Johnson, J. Lees, and M. Welsh, "Deploying a wireless sensor network on an active volcano," *IEEE internet computing*, vol. 10, no. 2, pp. 18–25, 2006.

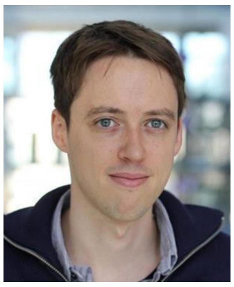
[46] L. Yan and N. McKeown, "Learning networking by reproducing research results," *ACM SIGCOMM Computer Communication Review*, vol. 47, no. 2, pp. 19–26, 2017.

[47] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[48] B. Zhang, X. Jin, S. Ratnasamy, J. Wawrzynek, and E. A. Lee. (2018) AWStream: Adaptive Wide-Area Streaming Analytics. [Online]. Available: https://nebgnahz.github.io/awstream-paper/awstream.pdf

[49] I. Zhang, A. Szekeres, D. Van Aken, I. Ackerman, S. D. Gribble, A. Krishnamurthy, and H. M. Levy, "Customizable and extensible deployment for mobile/cloud applications." in *OSDI*, vol. 14, 2014, pp. 97–112.

AUTHOR BIOGRAPHIES



Jonathan Fürst is a visiting researcher at NEC Laboratories Europe and a postdoctoral researcher at the IT University of Copenhagen. He has been working with software controllable micro grids, sensor networks, IoT systems and Building Management Systems (BMS) in non residential buildings. During 2014 he was a visiting researcher at UC Berkeley in the Software Defined Buildings (SDB) group. His current research focuses on developing better abstractions to program and run IoT applications efficiently from edge to cloud.

Mauricio Fadel Argerich is a research intern at NEC Laboratories Europe. He is a student from the Master's Degree in Data Science at La Sapienza, Università di Roma, and has previously obtained the degree of Information Systems Engineer at the Universidad Tecnologica Nacional in Argentina. He has worked as a software engineer with different technologies and carried out a research project in Automatic Link Generation as a visiting student at the University of Auckland in 2013. He is currently working on his thesis on leveraging machine learning for IoT applications.

Kaifei Chen is a Ph.D. candidate in Computer Science at the University of California, Berkeley. He is in the Building-Energy-Transportation Systems group. His research is focused on mobile computing, indoor localization, sensor networks, and ubiquitous computing. He obtained a bachelor degree in Computer Science and Technology from University of Science and Technology of China. He was a visiting student in Carnegie Mellon University Silicon Valley Campus in 2010, an intern in Microsoft Research Asia in 2011, and an intern in Microsoft Research Redmond in 2014.

Ernö Kovacs holds a Ph.D. from the University of Stuttgart. At NEC Laboratories Europe, he is a Senior Manager for "Cloud Services and Smart Things". His group works on Cloud Computing, IoT analytics, self-organisation and context-aware services. He was a leading architect in the SPICE and in the MAGNET Beyond project. He is currently contributing to the FIWARE, FIESTA and Mobinet projects. He was an advisor to the Singapore Smart Nation program in the Functional Specification round table and was leading NECs engagement in the Safe City Singapore test bed.