UNIVERSITY OF CALIFORNIA

Los Angeles

Adaptive Distributed Systems Spanning Cloud-Edge Networks

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Joseph George Noor

2021

ABSTRACT OF THE DISSERTATION

Adaptive Distributed Systems
Spanning Cloud-Edge Networks

by

Joseph George Noor
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2021
Professor Mani Srivastava, Chair

The edge computing domain is notably diverse in its composition. Capabilities including sensing, actuation, and mobility are often constrained by limitations such as those in energy, storage, and hardware-specific implementations. Leveraging the diversity of cloud-edge systems when building applications pose fundamental challenges that constrain expressivity, portability, and reconfigurability. Given this complexity, system tooling is necessary to aid application development and deployment in order to provide a stable runtime foundation by which environment dynamics can be accounted for. This dissertation explores the notion of incorporating *self-awareness* into autonomous systems spanning the cloud and edge, such that they might observe the constantly changing state of their applications and resources to implicitly adapt for optimized behavior.

Key thrusts include (1) DDFlow, a macroprogramming abstraction that organizes distributed applications into a visual dataflow representation, (2) Portkey, an adaptive key-value store that reconfigures data placement based on access patterns, and (3) EdgeRM, a distributed resource manager that unifies a heterogeneous computing cluster spanning high-performance servers and low-power sensors, which leverages a WebAssembly execution model and system interface extension for unified sensing.

Ultimately, this research seeks to lift IoT and embedded devices into the general-purpose computing domain, thereby enabling a carry-over of technical contributions pioneered by the distributed systems community. As systems researchers continue to introduce new paradigms in how to design, deploy, and support applications spanning cloud-edge, our community can help manage the difficulty in harnessing the capabilities of these networks.

The dissertation of Joseph George Noor is approved.

Harry Xu

Ravi Netravali

Carlo Zaniolo

Mani Srivastava, Committee Chair

University of California, Los Angeles

2021

May we all achieve the fullest expression of our creation.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Thank you to my advisor, Mani Srivastava, for your inspiring intellect, wise words, thoughtful guidance, provocative questions, and strict regiment.

Chapter 2 is a version of Noor, Joseph, et al. ”DDFlow: Visualized Declarative Programming for Heterogeneous IoT Networks.” Proceedings of the International Conference on Internet of Things Design and Implementation. 2019. Chapter 3 is a version of Noor, Joseph, et al. ”Portkey: Adaptive Key-Value Placement over Dynamic Edge Networks.” Proceedings of the Twelfth ACM Symposium on Cloud Computing. 2021. Chapter 4 is a version of work under preparation with co-author Joshua Adkins, UC Berkeley. Chapter 5 is a version of Noor, Joseph, et al. ”The Case for Robust Adaptation: Autonomic Resource Management is a Vulnerability.” MILCOM 2019-2019 IEEE Military Communications Conference (MILCOM). IEEE, 2019.

Thank you to CONIX and IoBT, funded by SRC and the Army Research Lab, for your enabling me to pursue my research; furthermore to the United States and California for providing the foundation for this institution which serves as the fruit of modern civilization. Research in this dissertation was sponsored in part by the Army Research Laboratory (ARL) under Cooperative Agreement W911NF-17-2-0196, and by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the ARL, DARPA, SRC, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

Finally, thank you to my parents, for imparting in me a burning passion for advancing human intellect, the technique necessary to absorb and create knowledge, and the spiritual intensity to relentlessly pursue that to which I set my mind.

VITA

2009	Programmer Intern, JPL Space Radiation Lab, Caltech.
2012	Software Engineering Intern, Center for Domain-Specific Computing, UCLA.
2013	Research Assistant, Physical Chemistry Lab, UCLA.
2013	B.S. (Computer Science & Engineering), Magna Cum Laude, UCLA.
2014	GPU Architect Intern, NVIDIA.
2013–2015	Teaching Assistant, CS31, CS32, CS244, Computer Science Department, UCLA.
2015	M.S. (Computer Science), UCLA.
2015–2018	Graduate Student Researcher, Scalable Analytics Institute, Computer Science Department, UCLA.
2018–present	Graduate Student Researcher, Networked & Embedded Systems Lab, Computer Science Department, UCLA.

PUBLICATIONS

I Always Feel Like Somebody's Sensing Me! A Framework to Detect, Identify, and Localize Clandestine Wireless Sensors. 30th USENIX Security Symposium (USENIX Security 21). 2021.

How Can I Explain This to You? An Empirical Study of Deep Neural Network Explanation Methods. Advances in Neural Information Processing Systems (NeurIPS). 2020.

Time Awareness in Deep Learning-Based Multimodal Fusion Across Smartphone Platforms. 2020 IEEE/ACM Fifth International Conference on Internet-of-Things Design and Implementation (IoTDI). IEEE, 2020.

Exploiting Smartphone Peripherals for Precise Time Synchronization. 2019 IEEE International Symposium on Precision Clock Synchronization for Measurement, Control, and Communication (ISPCS). IEEE, 2019.

DDFlow: Visualized Declarative Programming for Heterogeneous IoT Networks. Proceedings of the International Conference on Internet of Things Design and Implementation (IoTDI). 2019.

The Case for Robust Adaptation: Autonomic Resource Management is a Vulnerability. MILCOM 2019-2019 IEEE Military Communications Conference (MILCOM). IEEE, 2019.

mCerebrum: A Mobile Sensing Software Platform for Development and Validation of Digital Biomarkers and Interventions. Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems (SenSys). 2017.

Apache REEF: Retainable Evaluator Execution Framework. ACM Transactions on Computer Systems (TOCS) 35.2. 2017.

Optimizing Interactive Development of Data-Intensive Applications. Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC). 2016.

CHAPTER 1

Introduction

Edge computing is a tremendous shift from the decades of advancements in cloud computing. The emergence of sensor networks and the Internet-of-Things has led to a world where an incredibly diverse collection of sensors, actuators, hardware accelerators, and mobile devices are connected to enable complex and emerging applications, including those spanning smart cities and military domains. As the number of IoT and edge devices rapidly increases [Col18], the complexity and capability of hardware continues to evolve, resulting in an unprecedented amount of information residing at the edge. Given the sheer volume of latency-sensitive data that is continuously collected, mechanisms that remove the edge-to-cloud from the critical path are necessary, as it places unacceptable restrictions on availability, throughput, and latency, with projected trends only serving to further exacerbate these limitations [Fut19]. In an effort to aid in the development and deployment process, new classes of distributed system tooling are specializing to focus on the problems arising in this domain.

The fabric onto which IoT and edge computing applications run possess fundamentally unique characteristics. First, embedded devices have traditionally demanded low power constraints with limited computational and networking capabilities. Second, many of these devices, especially sensors and actuators, are supplanted with the intention of capturing and interacting with the surrounding environment – the physical location of a device plays an essential role in its purpose within the sensor network. Third, due to the inherent juxtaposition of embedded devices within the physical world, privacy and security implications become a primary concern. These devices are not only exposed to a new physical attack vector, but are able to capture highly sensitive information which must be carefully managed.

1.1 The Problem: A Heterogeneous, Ad Hoc, and Dynamic Edge

In recent years, new and emerging capabilities have further exacerbated the problem of system management in the IoT and edge computing space. First, new enterprise and research devices have created even more heterogeneity and hardware isolation. Enterprise frameworks such as Samsung SmartThings and Apple HomeKit attempt to stitch together disjoint devices into a fully integrated system; however, these solutions are closed source and limited in both capability and supported devices, resulting in redundancy and isolation. Second, new accelerators and hardware, especially in support of deep neural networks and machine learning, have further increased the variation in device efficiency. Third, entirely new sets of capabilities have emerged at the edge, including actuation (e.g., control operators for industrial systems, assembly line automation, valve operation), and device mobility, where the location of a particular device is no longer fixed (e.g., camera PTZ, phones, drones, and vehicles). Finally, and most importantly, the edge is no longer one that is reliably instrumented and statically pre-configured. Instead, applications are being deployed over ad hoc and constantly changing networks, with devices entering and exiting a system at runtime, for example in first responder and military scenarios. Capturing and accounting for environment dynamics in these domains pose significant and previously unexplored challenges.

The nature of the edge network is increasingly different, discordial, and dynamic. In this transition, many of the previous paradigms for systems spanning the edge fail to map to this new world; a static deployment cripples application behavior. In essence, there is a fundamental need for a systematic approach to building high-quality and adaptive distributed applications. To this end, I posit the following three open and unsolved problems constitute the essential elements of a robust autonomic cloud-edge system:

Req 1 A means to dynamically assemble a heterogeneous collection of devices into a resource pool that implicitly exposes the individual and diverse capabilities of its composition.

Req 2 An approach to efficiently program non-trivial (i.e. more complex than IFTTT) applications which are able to take advantage of the diverse capabilities of the underlying devices while providing portability across networks with different compositions.

Req 3 Resource management mechanisms to provide efficient execution and resilience to changes that occur in environment dynamics at runtime, all while preserving application semantics.

The evolution of edge computing towards an ad hoc and dynamic network inherently requires an adaptive distributed system; applications deployed in a static fashion can no longer maintain quality-of-service for practical deployments. In this dissertation, I introduce a unified vision for enabling cloud-edge system *self-awareness*, such that an underlying system runtime may be empowered to (1) prepare and configure a generic application over a diverse yet unified resource pool, (2) track or infer network and environment state during runtime, in order to (3) reconfigure an application execution to preserve semantics while maximizing performance. The components of this system serve to bring us one step closer to a future of interconnected and widely-accessible IoT.

1.2 The Future: A World of Interconnected IoT

The future is a world where the hardware that comprises today's edge network is no longer isolated through special-purpose programming abstractions and custom-built application stacks. Instead, these networks are simply a logical extension of the existing cloud computing infrastructure that is readily accessible to developers today. When designing IoT applications of the future, leveraging sensors and actuators will be just as straightforward as leveraging scalable computing infrastructure or virtual private servers that accelerate generically-written applications based on their hardware-optimized specification. The emerging trend of

"Infrastructure-as-Code" is one such realization of this definitive pattern [Mor16], and its application to IoT is all but certain.

Consider the smart city of the future, where autonomous vehicles, municipal sensor fleets, and user deployed smart home systems are woven together into a unified, communicable, and shared platform. Given the current state of IoT design, this is simply impossible, as each vendor isolates their devices with a custom application stack and incompatible networking interface. As a result, distributed system administrators undergo an enormous undertaking simply to coalesce the hardware *owned by a single entity.* The most practical solution is far from the most ideal: each device is treated either as (1) a complete, standalone (and therefore limited) system (e.g. autonomous vehicle), or (2) as a dummy data-forwarder (e.g. sensor fleet), with system intelligence confined to the cloud. Moreover, these incompatibilities restrict the sharing of infrastructure across users and applications, ultimately inhibiting the natural growth of IoT into an extension of general-purpose computing that can uniquely sense and interact with the physical world.

To illustrate this point more clearly, let us consider three tangible use cases of distributed IoT systems: (1) developing hyper-local traffic-aware routing algorithms, (2) empowering first responders during disaster management, and (3) detecting and mitigating violent crime. Firstly, and most obviously, municipal sensors such as those deployed at street lights, traffic intersections, cellular towers, and public service vehicles offer an obvious means to serve each of these three use cases, whether it be in the generation of live traffic activity, providing high-fidelity AR information to first responders, or detecting and tracking any public display of weapons. Additionally, user homes, smartphones and autonomous vehicles offer a incredibly rich suite of sensor data that can be voluntarily contributed to aid in the enhanced performance of each of these applications.

Yet the practical achievement of the three aforementioned use cases is limited by the isolated and disjoint state of IoT. Deploying multiple applications against these distributed systems is currently infeasible due to the lack of shared IoT infrastructure. With each device group or vendor confined to their own isolated silo, there exists no means to simultaneously multiplex these use cases over the same hardware, even if wholly owned by a single entity, not to mention incorporating voluntary and ephemeral crowd-sourced devices. The only way to enable multi-tenancy is to coalesce application intelligence at the nearest general-purpose computing environment. That is, as of today, the cloud. This ultimately results in one of two scenarios: either applications posing low-latency QoS requirements are dismissed, or a dedicated system is custom built specifically for that application. This disappointing outcome is fundamentally limiting the practical adoption and applicability of IoT. Without designing for generic compatibility, IoT will continue to remain a niche environment for wealthy users and embedded developers. On the other hand, if we were to design a practical means for sharing infrastructure with multi-tenant usage of flexible and diverse computational capabilities, perhaps IoT can ascend to it's utopian image of a sensor-rich and highly capable environment accessible to the standard developer with reasonable development tools and general-purpose computing resources.

Bridging the Gap. The broad vision of this research is to unify IoT through a shared resource manager with access to a persistent shared data layer. As opposed to treating devices as single-purpose data forwarders, IoT devices will be capable of hosting multi-tenant applications with access to peripheral devices and compute pushed out to the microprocessor itself. Whereas prior systems aimed to design frameworks that serve only a subset of applications, we argue that a novel layer of abstraction is needed to truly enable *general-purpose* and *interactive* computing hosting multiple frameworks at the edge, similar to the general-purpose capabilities offered by current cloud computing platforms. The core idea proposed is a self-aware adaptive system, aiming to project one step toward bridging the gap between the limitations of the current state of IoT and a realization of the interconnected future.

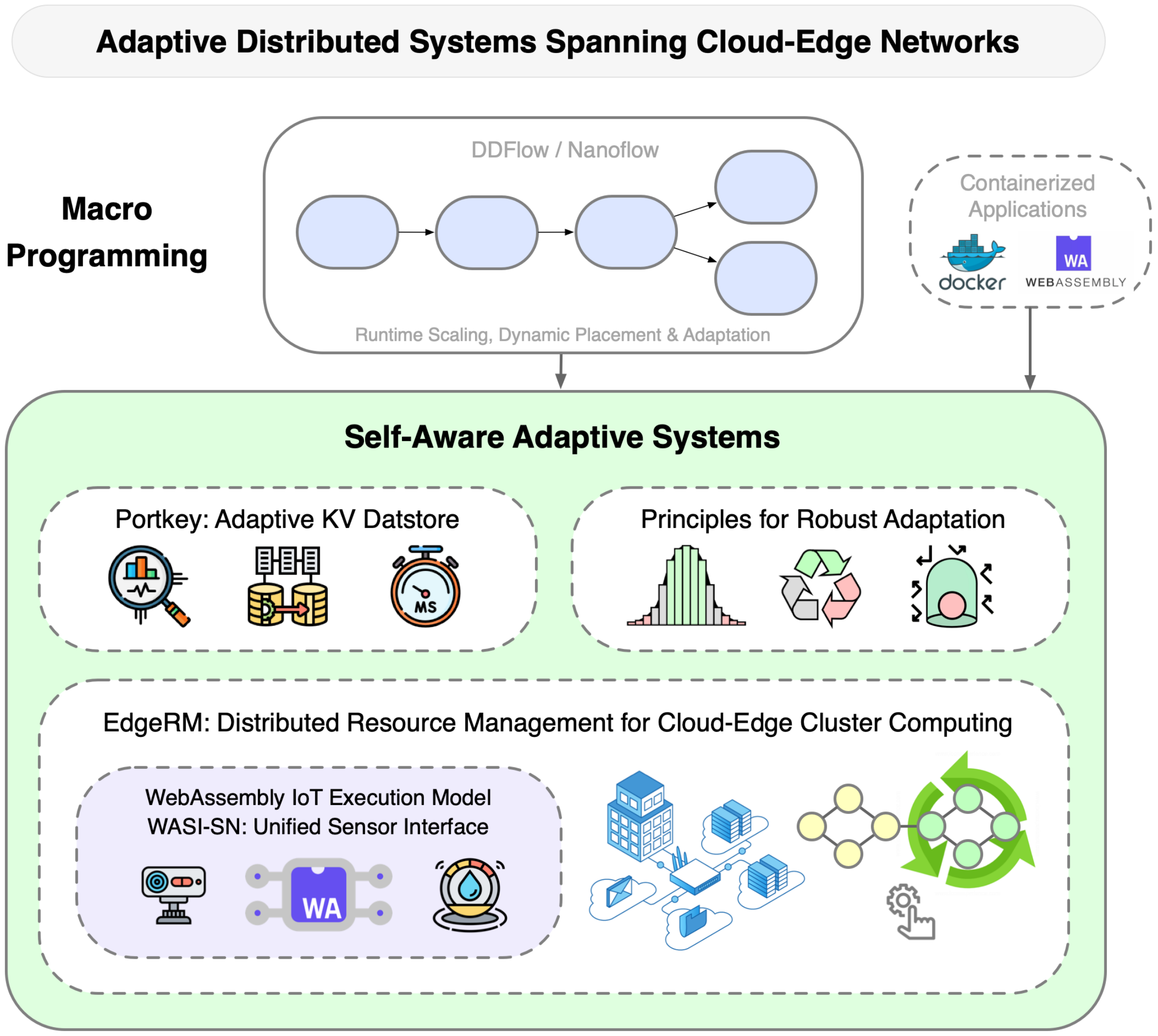

Adaptive Distributed Systems Spanning Cloud-Edge Networks
Macro Programming
DDFlow / Nanoflow
Runtime Scaling, Dynamic Placement & Adaptation
Containerized Applications
docker
WA
WEBASSEMBLY
Self-Aware Adaptive Systems
Portkey: Adaptive KV Datstore
MS
Principles for Robust Adaptation
EdgeRM: Distributed Resource Management for Cloud-Edge Cluster Computing
WebAssembly IoT Execution Model
WASI-SN: Unified Sensor Interface
WA

Figure 1.1: The Vision Illustrated

1.3 The Vision: Self-Awareness for Adaptive Systems

Self-awareness refers to the notion that an autonomous system has the means to model itself and evaluate its ability to manage application execution. The model encompasses a set of possible layouts and/or configurations; at any given moment, applications are mapped to the system using one instance of the possible configurations. Given this instance, the system tracks quantifiable performance metrics. The key enabling feature of self-awareness is the ability to simulate the results of altering its configuration, potentially shifting to another instance of possibly layouts. This "what if" thought experiment provides a means for the system to move towards optimal performance by deciding on desirable configuration changes, justifying its decision, committing an update, consulting the impact on performance metrics, and re-evaluating its configuration. Achieving self-awareness requires defining (1) a flexible system model, (2) the space of possible configurations, (3) the performance metrics to track, and (4) a means for the system to collect, evaluate, and act on this information.

Figure 1.1 depicts the high-level elements comprising the vision for the self-aware autonomous system, with each component serving to satisfy one of the essential requirements introduced in Section 1.1. Each chapter of this dissertation highlights an element of this vision, taking a top down approach from macroprogramming to core system contributions.

1.3.1 (Macro) Programming

The *Macroprogramming* components provide a hierarchical planning and programming abstraction that specifies application tasks in a declarative manner. It aims to avoid designing applications specifically for a particular deployment or resource pool configuration. Instead, the end-user should describe *what* a system is meant to accomplish, without explicit regards to the *how*. This flexible specification empowers systems to make decisions about how to efficiently distribute and execute a particular application while taking advantage of the available resources and capabilities inherent to a particular deployment. Furthermore, a declarative programming abstraction enables portability across networks, such that the same specification can be effectively accomplished across varying hardware and network configurations.

DDFlow introduces a declarative dataflow-style programming abstraction that dynamically assembles a list of capabilities over a resource cluster. Each device deploys a manager that implicitly collects and reports its available tasks to a central coordinator. In this manner, hardware-specific implementations are abstracted for a particular task; for example, two different devices may offer the same camera sensing task with differing underlying implementations. Applications are constructed visually by organizing Nodes into a streaming set of tasks, with support for user-defined functions. Upon issuing a DDFlow application to the coordinator, a scheduler dynamically scales up the application based on Node constraints and Wire specifications, issues the relevant task requests, monitors each device in the application deployment, and reissues tasks if needed based on environment dynamics.

Subsequent work on EdgeRM addressed the strict definition of constructing DDFlow Nodes with JavaScript wrappers, which limits the overall portability and universality of application expression. To this end, EdgeRM supports application definition as a set of containerized tasks via Docker and WebAssembly. These tasks can then be dynamically issued and managed across a resource cluster through the EdgeRM messaging protocol.

1.3.2 Execution Environment

The core of the system comprises three thrusts serving as the autonomic execution environment over which applications are managed. The execution model consists of a distributed resource manager spanning high-performance servers and low-power sensors with access to a low-latency external state management platform augmented with principles guiding robust adaptation. When enabled by macroprogramming, these key elements provide scope and mechanisms by which an adaptive system utilizes information gained via self-awareness.

EdgeRM provides a unified resource management interface for applications spanning cloud-edge network deployments. An aggregate view over the entire cluster lifts IoT and embedded devices, previously confined to the "sensor network," into the general purpose computing cluster. Applications defined as sets of Docker containers and WebAssembly modules can be issued across the cluster via a shared interface. To support portability across embedded devices, an execution model supporting WebAssembly for IoT integrates the WASI-SN extension to the WebAssembly System Interface for unified sensing across varying device drivers. EdgeRM provides a foundational platform over which complex application frameworks can be developed, such as our Sensor MapReduce Framework, while enabling multi-tenant resource sharing. The scheduling interface offers a means for custom-built application logic while supporting dynamic deployment, monitoring, and adaptation of task bindings over the resource pool.

For cloud-edge applications requiring a distributed storage platform, *Portkey* provides low-latency access to a persistent key-value store. Whereas previous works have attempted to design a one-size-fits-all approach to data placement, Portkey customizes key-value placement to a deployment based on its particular network and access patterns. Client accesses are tracked, summarized, and reported to the Portkey placement engine, which uses information about application and network state to continuously optimize KV placement to minimize client request latency. This allows networked deployments to implicitly learn data hosting without requiring any external inputs from system administrators or domain experts.

Finally, adaptation in-and-of-itself offers an attack surface by which adversaries may negatively influence system decisions. Guarding against these potential threats led to the introduction of the ADAPT^2 principles of secure adaptation. The incorporation of state estimation, moving target defense, and location obfuscation helps protect a self-aware system from the risks of naive adaptation. While not a system in its own right, the ADAPT^2 principles are widely applicable to most resource management designs, and help guide systems developers into incorporating preventative measures that limit adverse configurations.

CHAPTER 2

Enabling Adaptive Systems via Declarative Macroprogramming

The most fundamental problem limiting the ability for adaptive systems in autonomic computing is an application description and implementation which binds itself to a particular suite of devices within a particular network deployment. The ability for a system to decide how to place and manage an application requires a specification that is abstract and high-level enough to enable such freedom. A designation of computation to a particular device or subset of devices inherently restricts the possible space of execution solutions.

How might applications be constructed to express complex tasks spanning cloud, edge, and IoT networks, while simultaneously allowing for the maximal set of placement options? An obvious first step is move away from defining tasks as they relate to a particular device. Instead, one should describe the desired end-to-end behavior over the entirety of the network. This notion of global application specification is broadly referred to as macroprogramming.

Describing an application as a macroprogram prevents a tangible binding to any particular set of devices, thereby offering portability and resilience across diverse network characteristics. Previous work in macroprogramming abstractions (e.g. [GGG05, NMW07, KWA03, HLR13]) achieve this through various techniques, including SQL-style languages or event-driven callbacks, while including necessary spatiotemporal primitives to capture the physical properties of sensors and edge devices. However, these systems were designed in an era that precedes the modern edge environment, which now includes actuation and mobility over a diverse and dynamic network. As such, they lack the sufficient mechanisms to enable current applications. A problem emerges in providing an efficient means to specify coordinated behavior across the cloud-edge while exploiting device heterogeneity and enable system adaptation/reconfiguration, all while providing portability across varying networks.

In recent work published at IoTDI, we introduce DDFlow [NTG19, NSG19], a macroprogramming abstraction and accompanying system runtime which offers the appropriate primitives to properly isolate application semantics from an arbitrary deployment environment. Application specification is accomplished through a declarative and visual user interface, implemented as an extension of the Node-RED IoT system [Fou18]. Developers visually describe what the application should accomplish, without explicit regards to the how. This allows for effective visualization, programmability, and resusability. Furthermore, this provides an opportunity for a system runtime to have flexibility in decision-making, thereby enabling adaptive systems at the edge.

This chapter introduce the macroprogramming abstraction that provides an initial prototype enabling adaptive systems. I begin with an example motivating application that represents a real-world latency-constrained scenario. After touching upon related work, I then introduce the DDFlow macroprogramming abstraction for declaratively programming distributed applications spanning a diverse and dynamic edge.

2.1 Motivating Application

Emerging applications at the edge range from those spanning smart homes, smart cities, military and medical domains. In order to ground the discussion, I will introduce an example scenario highlighting the key requirements and issues emerging from this environment. This motivating application, while simply one example, illustrates the latency-constrained, heterogeneous, and dynamic nature of modern applications that are now actively deployed over IoT networks.

With a sensor network that spans college campus cameras, drones, a speaker, and cloudlet

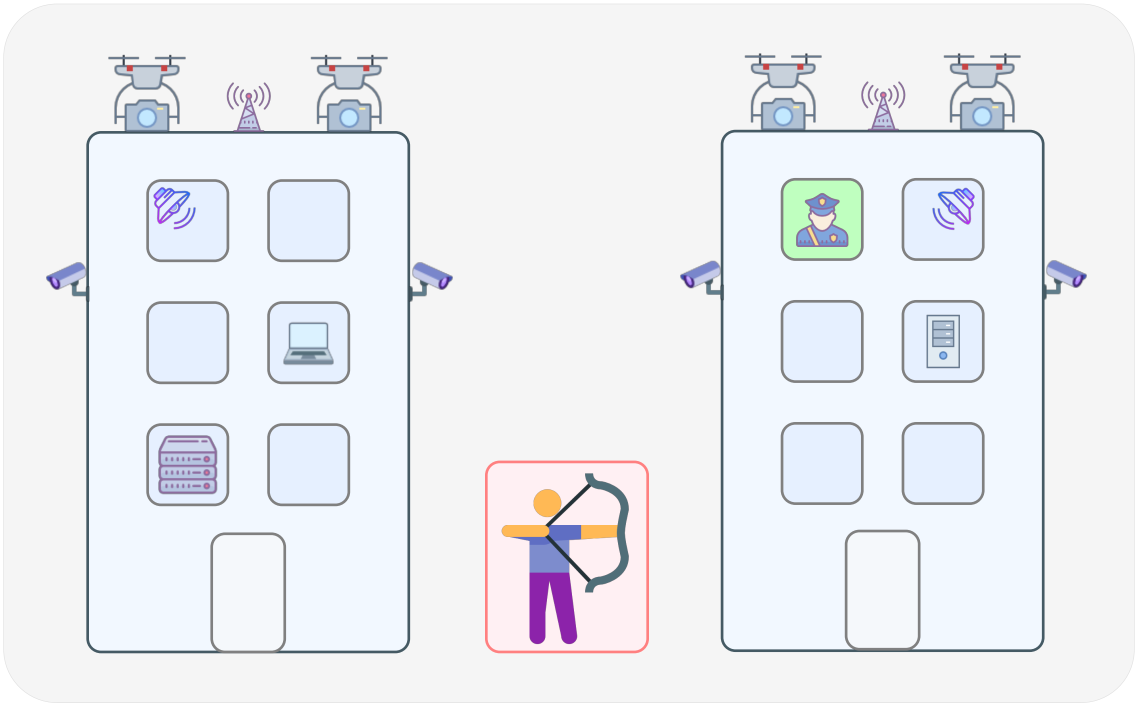


Figure 2.1: Example Motivating Application

servers, the objective is to achieve the following:

1. Recruit cameras in a specified region to identify a target or object-of-interest

2. Classify the captured image frames from the cameras to detect the target

3. Upon initial detection, play a sound on the speaker at a command center in order to alert those nearby that the target has been identified

4. Available nearby drones are tasked to pursue the target and provide a live video feed

This example application, illustrated in Figure 2.1, showcases many of the pain points associated with system management and application development over a cloud-edge network.

First, minimizing end-to-end latency is a critical quality-of-service requirement; similarly, due to the sensitive nature of the application, maximizing up-time availability is essential.

Second, the devices comprising the system have fundamentally different and diverse capabilities, spanning sensing, actuation, mobility, and computational capacity. Some devices, for example cameras, may offer similar high-level functionality, but across varying hardware and quality metrics. There exists large heterogeneity across devices serving the same functionality. For example, different devices may offer image classification, but with varying accuracy, granularity, and throughput.

Third, the network over which the systems operates is prone to significant change at runtime; devices may be moving, devices may enter/leave the network, and overall network conditions (e.g. link latencies, bandwidth) are prone to fluctuating. Configuring and reconfiguring in spite of the ad hoc nature of this network is nontrivial. Variable network characteristics significantly affect optimal computation placement, for example whether to stream video to a nearby classifier, or accept less accurate local classification to maintain acceptable throughput. Coordinating and reconfiguring devices to maintaining application semantics while minimizing latency and maximizing availability given these constraints is a challenging problem.

Fourth, new devices, for example cameras and drones, are constantly being created and integrated into modern systems. Accounting for, inserting, and implicitly exposing these capabilities without having to fully rearchitect the system or recompose the application specification poses an additional burden.

Fifth, providing an abstraction which enables portability, such that moving to a new campus or enviornment does not require rebuilding the entire distributed system every time. Reconfiguring an application to a new environment should be a minor modification, and require minimal pre-instrumentation.

Finally, providing an abstraction which is powerful enough to express complex applications of this nature, as they span across varying hardware and sensors. An abstraction which addresses the previously defined pain points without capturing a rich suite of nontrivial applications is insignificant. Is it possible to design an abstraction which enables this flexibility and expressivity, write once, and deploy everywhere?

The key idea in DDFlow is to take a declarative approach to application specification. Before introducing the abstraction and its inspiration, I will briefly touch upon previous related work in macroprogramming for edge and IoT networks, how they attempt to address many of the problems introduced by this motivating application, and where they fall short.

2.2 Background and Related Work

The notion of macroprogramming refers to a field of research that aims to provide centralized specification for applications spanning distributed sensor networks [WM04]. The goal was to enable complex coordinated activity without forcing developers to configure individual device or account for a particular network. The typical approach is to define a language (often with an accompanying runtime framework) that allows efficient description of global application behavior. There have been many proposed approaches, abstractions, and languages introduced in order to accomplish this task [GGG05, HLR13, KGM07, KWA03, NW04]. Each approach makes various assumptions on the set of target applications, and in doing so adopts differing design decisions for the proposed abstraction; some examples include whether to focus on local node behavior or global computations, whether to offer a network independent or dependent abstraction, and how to express computation as a function of the network topology [MP11]. Aside from the convenience of centralized specification, the key contribution of macroprogramming is the flexibility it grants to the runtime system in how to efficiently manage the execution of the application.

DFuse[KWA03] is a macroprogramming framework focused on aggregating sensor data into higher order evaluations; that is, simple streaming operators over collected sensor streams. Applications are defined as dataflow graphs composed of sensors, fusion point operators, and outputs. The runtime determines where to place computation in a heterogeneous network. Given the flexibility in placement, application adaptation occurs with devices suggesting the transfer of a subset of their current compute tasks to a nearby device. While effectively capturing data fusion applications, DFuse lacks support for external actuation, mobility, and scalability; tasks such as sensing should often be subdivided across a set of available devices.

Kairos [GGG05] is a programming language to define global behavior of distributed computation by expressing pairwise interactions between neighboring nodes. The language primitives involve accessing neighboring devices and writing shared data values. The notion of pairwise expression places a constraint on application portability and expressivity, potentially leading to unexpected behavior in the mobile IoT environment.

Regiment [NMW07] offers a functional macroprogramming language that groups data streams into regions based on spatial locality, allowing an expressive language that captures many sens-compute applications in sensor networks. Due to the high-level of the language abstraction and its basis on the Token Machine Language programming model [NW05], it is generally ineffective at expressive heterogeneous network capabilities. With no language support for mobility or actuation, it falls short in control applications.

Mobile Fog [HLR13] presents a programming model for IoT applications spanning the fog (cloud-edge) network heirarchy. Applications are constructed using event-driven message-passing callbacks. With support for dynamic scaling, complex event processing, and distributed key-value storage [MGS17], Mobile Fog offers a flexible programming interface that offers single-file centralized specification. This abstraction works well for simple, constrained tasks. However, for complex applications spanning heterogeneous edge devices, grouping all devices' application logic into a unified callback function can become unwieldly and difficult to understand.

The most directly related work to DDFlow is the *D-NR* system described in [GBL15]. Their work provides an initial implementation of a distributed extension to Node-RED. Developers define a master flow composed of sub-flows which each deploy to different devices in the overall network. Heterogeneity is accomplished by explicitly categorizing devices into *edge*, *IO*, and *compute*, with inter-device communication leveraging MQTT. Ultimately, D-NR provides an interesting prototype for visual programming. However, with a lack of declarative specification, scaling, fault tolerance, and dynamic adaptation, it falls short in delivering a robust system framework for IoT application extending outside a controlled home environment.

2.3 Capturing the Sense $\Rightarrow$ Compute $\Rightarrow$ Actuate Paradigm

A useful programming abstraction for building distributed applications across the distributed cloud-edge must have the ability to capture a complex set of applications while maintaining an abstraction that is high-level enough to empower an underlying runtime to enact adaptive system decision-making. In this way, it can provide portability across networks with a unified framework by which distributed systems can be easily assembled. Key shortcomings of related work include a lack of support for actuation, and a lack of implicit scaling, where the optimal scaling of an application cannot be known until deployment, and as such should not be explicitly stated as part of the application specification. An added bonus would be a programming model that is intuitive to understand; building complex applications by encoding all application logic in unified callback functions will quickly become unmanageable.

Applications spanning the cloud-edge, especially those involving sensing and actuation, typically follow a universal paradigm structure which can be referred to as the IoT app archetype. Regardless of domain, this paradigm is prevalent across a wide variety of applications. The paradigm is as follows:

1. Applications begin with sensing, or some other form of event trigger.
2. Sensing data is fed into some sort of machine learning model to extract higher-level deductions.
3. Aggregates (e.g. windowing, grouping) and filters are typically performed on extracted

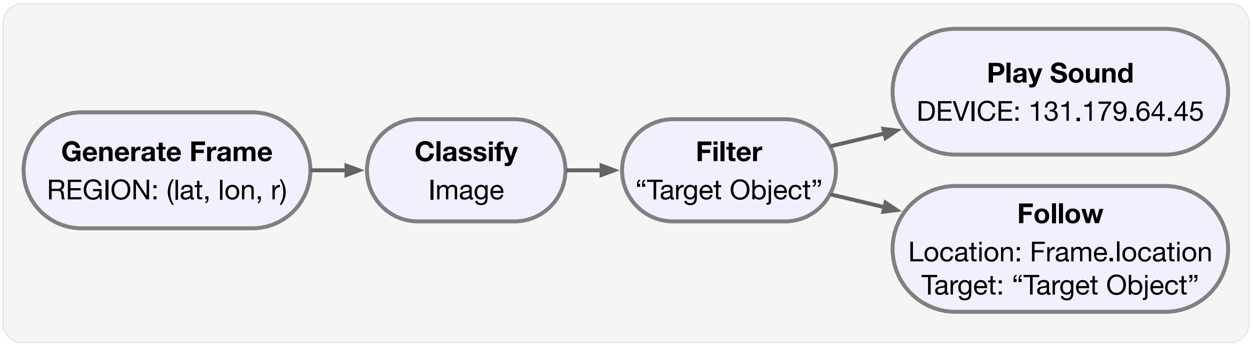


Figure 2.2: The Motivating Application in DDFlow

features before and/or after the ML model is applied.

4. Downstream results are presented to the end-user and/or used to trigger actuation(s).

Dataflow is a natural choice for both visualizing and describing applications spanning IoT networks. It is a programming model that has been previously validated by the community (e.g., [ABC15, KWA03]), and is a logical choice for an event-driven abstraction that begins with sensing and ends with actuation, capturing the perception $\Rightarrow$ cognition $\Rightarrow$ actuation paradigm that is pervasive across edge networks. Applications developed in this manner are highly visual and intuitively understood, providing an interface that allows developers, project managers, and future engineers to fully grasp an application at a glance. Furthermore, dataflow decouples application specification from the deployment network, enabling both portability between networks, and an opportunity for a system runtime to provide optimization. However, dataflow in-and-of-itself lacks the sufficient primitives necessary to fully capture application expression of this nature. This essential insight led to the creation of DDFlow.

2.4 The DDFlow Macroprogramming Abstraction

Figure 2.2 depicts a DDFlow specification for the motivating application described in Section 2.1. It begins by generating image frames in the region of interest. These frames pass through an image classifier, which is filtered to focus solely on identification of the target. Once identified, a speaker is notified to play a sound, and drones are deployed to the last observed location of the target, initiating a follow sequence.

There are a set of abstraction primitives provided by DDFlow extending dataflow to support a declarative application programming interface. More specifically, dataflow lacks the sufficient descriptors necessary to enable runtime scaling. In deploying an application to diverse environments with varying regions, devices, and capabilities, the precise scale of an application is often not known a priori. DDFlow aims to allow an application to be implicitly and dynamically scaled depending on the resources available at runtime and the constraints of an application. Existing work generally lacks this notion; not only should multiple tasks be assignable to the same device, but an individual task may be collectively accomplished by replicating across a dynamic set of devices. DDFlow achieves runtime scaling via the *Node* and *Wire* fundamental primitives.

2.4.1 Node

DDFlow applications are defined as a sequence of actions, or Nodes in a dataflow graph. A Node is a computational abstraction representing a stateful function that maps inputs to outputs, either of which are optional. Each Node corresponds to at least one instantiation of a task that must be deployed onto a device in the network (e.g., generating camera frames, classifying images, playing a sound). Inputs and outputs are key-value dictionaries (i.e., JSON messages) that contain application data as well as metadata including timestamp and sender.

Nodes are constrained via a set of parameters relevant to a particular task (e.g., Filter contains a key-value to filter incoming messages). Due to the spatiotemporal nature of IoT applications, two fundamental parameters underlying all Nodes are *Region* and *Device*, optional parameters restricting the deployment of a task to a particular spatial region or set of devices. In Figure 2.2, the Node generating camera frames is associated with a circular region (lat, lon, r). Only devices capable of generating camera frames within the specified region of interest are potential candidates during runtime scaling. Regions can be described as a bounding box, with other structured location information, or as a list. Dynamic and moving regions can be specified via the *input* keyword, which monitors a given Node's inputs for a *region* value to update the existing deployment region. The *Device* parameter supercedes the *Region* parameter and allows for precise Node placement.

To deploy the motivating application to a new environment, a developer needs only to change the *Region* parameter for generating camera image frames and the *Device* parameter for playing a sound. During deployment, the system runtime will dynamically scale the application to the available devices in the regions of interest that are capable of accomplishing the specified Node tasks.

2.4.2 Wire

In defining a sequence of actions, Nodes are connected via *Wires*, representing a connection in a dataflow graph. Each Wire carries a key-value dictionary from the output of one Node to the input of the downstream Node. Due to the elastic runtime scaling of Nodes, WIres definitions follow one of three forms: *Stream* (one-to-one), *Broadcast* (one-to-many), and *Unite* (many-to-one).

In the motivating application, all devices that are performing the Follow task should receive updates to target location regardless of which camera generated the detected frame. As such, the connection between Filter and Follow is a *Broadcast* Wire. On the other hand, only one device is responsible for playing a sound upon target identification; as such the connection from Filter to Play Sound is a *Unite* Wire. Finally, in order to take advantage of motion detection capabilities, each camera generates a stream of frames to an independent Classify instance; as such, Generate Frame and Classify are connected via a *Stream* Wire.

2.5 Considerations

Defining applications in a declarative manner using DDFlow enables application portability and provides an opportunity for a system runtime to make decisions regarding application management for a particular deployment. This abstraction can be used to express a wide array of applications spanning cloud-edge networks, especially those involving sensing and acutation across heterogeneous devices. In the next section, I describe the initial work on building a system to support the abstraction, which serves as an initial prototype for the self-aware adaptive system. However, the work in application expression still has opportunity for further exploration. As applications are developed, intricacies requiring further constraints may require changes to the existing primitives (e.g. a new Wire type). Furthermore, this work in Macroprogramming is targeted towards an audience of individuals with basic knowledge of programming; selecting the appropriate nodes and wires require a basic understanding of how the application will scale and deploy. While this visual programming interface provides a flexible declarative means to design applications, a higher-order programming interface is also desirable; "Command-by-Intent" aims to achieve this via verbal programming.

One can view programming systems of this nature across multiple levels: systems developers typically require low-level control, whereas application developers look to abstractions to simplify specification. Furthermore, for certain classes of end users, for example police commanders, specification must be accomplished at a very high level, with abstractions similar to those given to digital assistants such as Google Home and Amazon Alexa. Previous work in macroprogramming does not suffice in this regard; end users may not be willing to sit and wire up dataflows in high-stress environments. Instead, macroprogramming serves as yet another intermediary representation between application developers and physical code, offering a functional means of declaratively issuing tasks over a distributed collection of devices. To provide an end-user solution, we can look to *Command-By-Intent*, a military concept that reduces commander orders to a desired outcome or end-state instead of explicitly defining the plan sequence. This concept can be mapped to adaptive systems by defining a repository of macroprograms that can be dynamically recalled and issued using oral commands, thereby increasing overall utility by offering a level of abstraction suitable to wider set of end-users.

2.6 The DDFlow System Runtime

To provide an adaptive system prototype, the work published at IoTDI provided an initial runtime implementation accompanying the DDFlow macroprogramming abstraction presented in [NTG19]. This runtime provides a first step towards self-awareness and a foundation for the adaptive system elements to be explored during my PhD research. This chapter briefly introduces the basic elements of this system, comprised of a programming interface, system architecture, dynamic runtime scaling, and dynamic adaptation/reconfiguration, such that further research direction may have a grounding system onto which implementations can be evaluated.

2.6.1 Programming Interface

A key aspect of the DDFlow macroprogramming abstraction is its visual composition. To support this notion of visual and declarative programming, the DDFlow system runtime repurposed the Node-RED [Fou18] IoT programming tool, which provides a single-device programming environment. In essence, the front-end visual "wiring" of single-device flows is now used to express DDFlow applications that are composed of DDFlow Nodes connected by DDFlow Wires. Figure 2.3 presents a screenshot of the motivating application expressed in the DDFlow interface.

Nodes themselves are implemented as lightweight JavaScript classes bootstrapped into the DDFlow system. An example implementation of a basic filtering Node is provided in

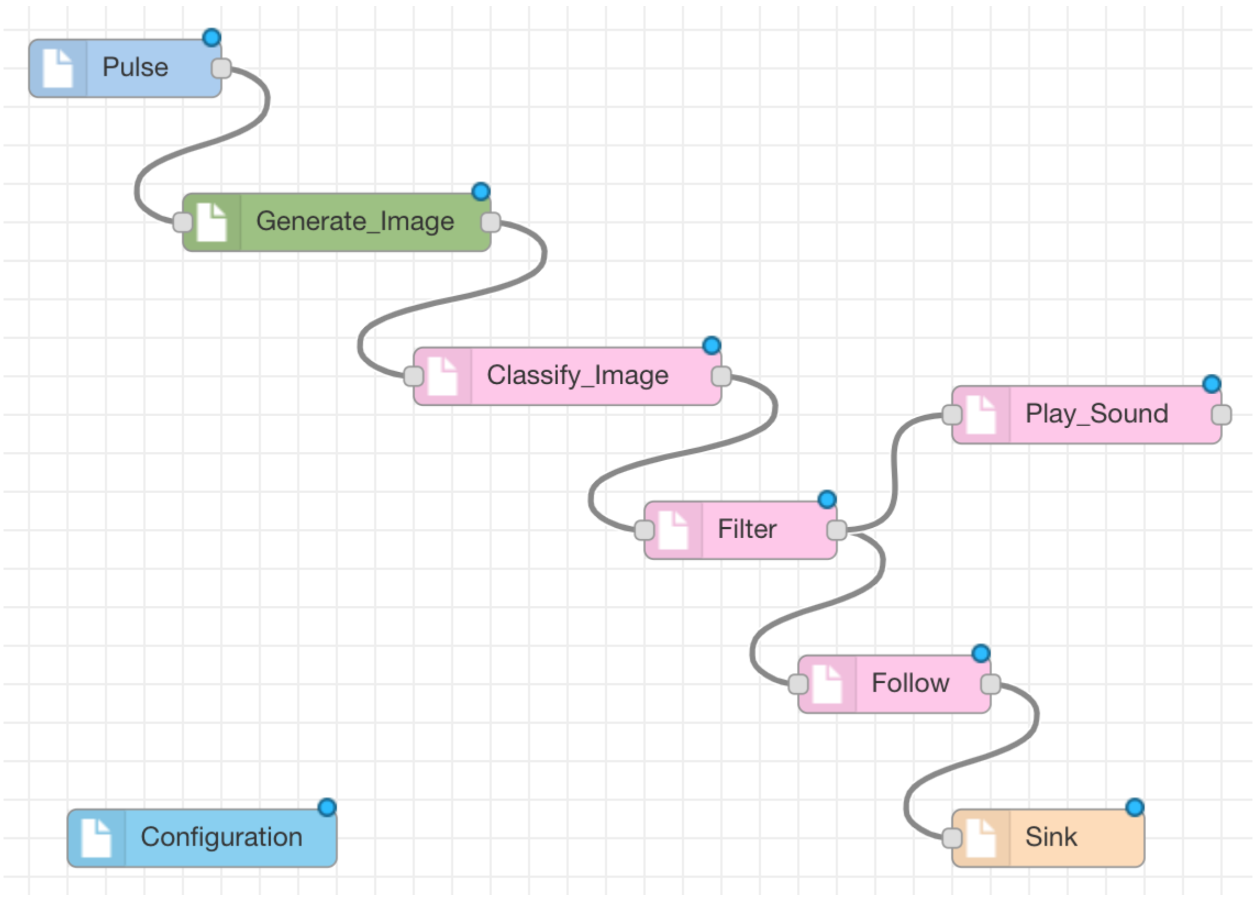


Figure 2.3: Screenshot of DDFlow Programming Interface

Listing 2.1. A Node implementation consists of three primary functions: `init` upon node instantiation, `receive` upon message recieval, and `terminate` upon node termination. The `params` value optionally provides a list of configuration parameters to request in the visual DDFlow interface for a particular node, provided to the init function as a `config` object. Finally, a Node instance may have local state, in this case `filter_key` and `filter_value`, stored as key-values within the class.

Listing 2.1: Filter Node Example

```
module.exports = {
    filter_key: "default",
    filter_value: "default",
    params: ["fkey","fvalue"],
    init(config) {
        this.filter_key = config.fkey;
        this.filter_value = config.fvalue;
    },
    receive(data) {
        if(data[filter_key].includes(filter_value)) {
            this.send(data);
        }
    },
    terminate() {
    }
}
```

2.6.2 System Architecture

The DDFlow system follows a service-oriented architecture, a proven architecture for dataflow and IoT systems [IBG16, KWA03], depicted in Figure 2.4. Each device offers a distinct set of *Services*. Services represent a particular implementation for a Node in the DDFlow abstraction (e.g. image classification, camera frame capture). In a heterogeneous environment, different physical devices have have different underlying implementations, but offer the same Service to DDFlow via the same high-level Node interface.

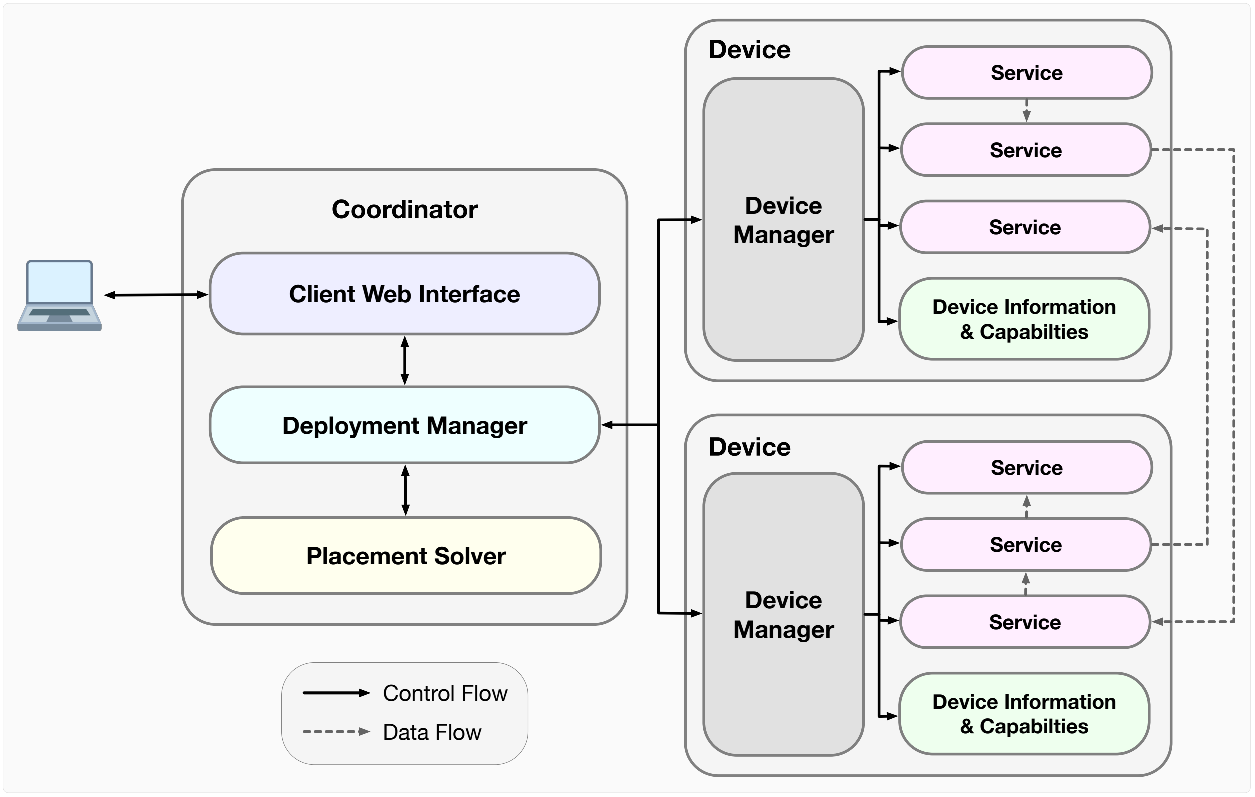


Figure 2.4: DDFlow Runtime System Architecture

Intra-device coordination is achieved through the *Device Manager*, a lightweight web server that is deployed on every device participating in the system. It is responsible for activating and deactivating service instances, and providing device information (e.g. available resources, utilization, location) to the Coordinator. Resource constrained hardware that cannot deploy a Device Manager expose themselves to the system through a more capable nearby proxy device.

Inter-device coordination is accomplished through a Coordinator, a web server that accepts and manages DDFlow applications as they are issued onto the available network. The Coordinator is composed of three main components: a web interface, a deployment manager, and a placement solver. The web interface, presented in Figure 2.3, is hosted by the Coordinator. Developers design DDFlow applications and submit them to the system via this interface. Upon acceptance, the Deployment Manager contacts the available devices and retrieves up-to-date information regarding capability and availability. Using this information, the DDFlow application is scaled to a particular task graph that fits the given network. Using the Placement Solver, tasks are assigned to particular devices, and the Deployment Manager issues the relevant service startup requests. The Coordinator monitors deployed applications to detect significant network changes, such as a disconnected or failing node, and adjusts the deployment mapping as needed. For resilience, the Coordinator can be replicated onto many device; placement of the Coordinator is recommended on device with high availability.

2.6.3 Runtime Scaling and Placement

When an application is deployed, the *Coordinator* contacts all *Device Manager*s to obtain updated state information and decide which devices to map an application task graph. In doing so, it may map many services to the same device, such as a powerful server with accelerators, and it may map the same service to many devices, such as a fleet of drones or sensors performing a group task.

Each *Device Manager* provides the *Coordinator* with information including location, utilization, estimated service and network latency, and devices within wireless range. From this information, the *Coordinator* constructs a network topology graph and a task graph. The topology is modeled with wired devices connected to a backbone network and wireless devices connected to other devices within range. The task graph is generated from the DDFlow application graph by scaling Nodes based on the region, availability, and capability constraints.

Given a network topology, task graph, and device capabilities, the *Coordinator* formulates computation mapping as a linear programming problem with the objective to minimize the longest path's end-to-end latency in the task graph. While admittedly a simplified metric, latency serves as baseline by which a system can begin to compare relative network speeds and model network characteristics. The solver will find the best solution to the objective function given the following constraints: (1) Neighbors in the task graph must also be accessible from each other in the network graph. (2) Devices must possess the necessary Node implementations for all assigned tasks. (3) Devices must have available the necessary resources required to execute the assigned task. The *Coordinator* solves this linear programming problem and issues task requests to all the relevant *Device Manager*s. This placement algorithm, described in detail in [TS18], is only one such algorithm for assigning tasks to devices. It is trivial to swap for another placement solver.

2.6.4 Dynamic Reconfiguration

To enable dynamic adaptation and recovery, the *Coordinator* probes devices in the network at an application-defined periodicity to monitor for environmental changes (e.g., disconnected node, overloaded device). Upon detection of a significant deviation of system characteristics (i.e., compute or network latency), the *Coordinator* computes a new placement mapping of an application. This new mapping is evaluated with respect to the objective function (e.g., end-to-end latency). In the case of projected improvement greater than a threshold, a remapping is triggered. Any failed or disconnected device will additionally trigger a remapping.

The *Library* system service provides a key-value data storage API for services to preserve local state. Thus, when the *Coordinator* issues the pertinent task activation/deactivation requests, all task-relevant key-values are forwarded from the terminating service to the device launching the new task instance.

Communication adapts at a finer granularity. The *Router* system service hides network-specific details by providing transparent messaging to devices. For a given device-service destination, a forwarding table identifies the optimal next-hop, either pushing the packet via TCP/IP infrastructure mode or via peer-to-peer Wi-Fi ad hoc mode.

2.6.5 Case Study

To showcase the benefits of a declarative programming interface with an adaptive runtime, we developed a simulation testbed. Devices are inserted into the Airsim [SDL18] environment simulator and connected via the Mininet [FAB15] network emulator. The main system components simulated in this evaluation, to represent a simplified version of the motivating application, are the following:

1. A camera used to identify the target
2. Three servers that are capable of providing image classification: a server with a GPU, a server without a GPU, and a camera-local accelerator
3. A client device with accompanying speaker used to alert upon target identification
4. A drone that follows the target after identification
5. Three wireless access points that provide the drone with communication to the backbone network.

Single-hop wired network links are modeled with a 2ms link latency, to account for processing, queuing, transmission, and propagation delays [MZP08]. Due to the inherent variability, wireless links are modeled as varying from 30-50ms [SZL16].

Three devices are capable of providing classification. The first is a server using an NVIDIA Titan X GPU and the YOLOv3 model (˜20ms per frame) [RF18]. The second is a server relying on its Intel Xeon CPU E5-2620 v3 @ 2.4GHz with the YOLOv3-tiny model (˜880ms per frame). Finally, a Google Vision Kit camera comes equipped with an Intel Movidius VPU [Goo18]. It contains support for constrained TensorFlow Lite models, with its default image classifier requiring ˜3.2s per frame.

In the first application scenario, the camera is streaming image frames to an available classifier. Upon successful target classification, the speaker is notified to play a sound. A

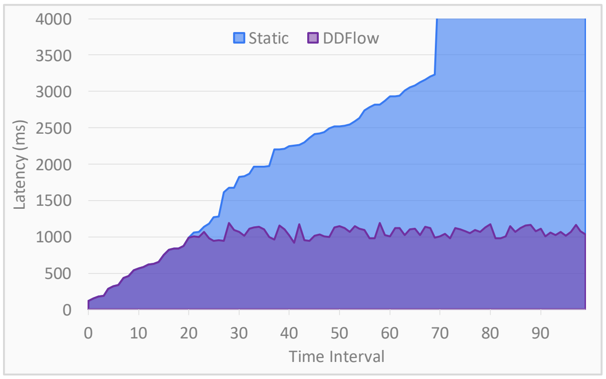


Figure 2.5: Adaptation during device overload

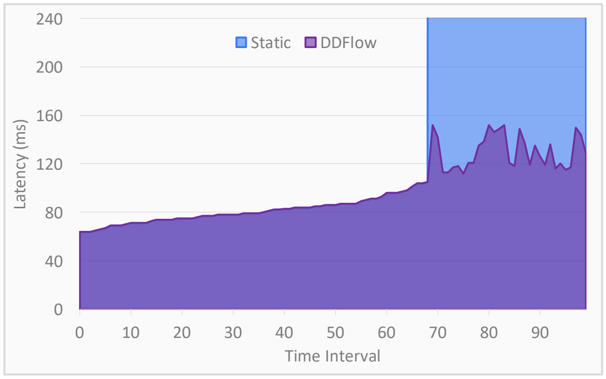


Figure 2.6: Adaptation during access point failure

static deployment streams frames to the fastest classifier, the GPU server. If that server becomes overloaded, performance degrades. DDFlow is able to switch to another available device to minimize impact on end-to-end latency and preserve application semantics. This is shown in Figure 2.5. As the GPU server becomes overloaded, both static mapping and DDFlow see end-to-end latency increase. After a certain threshold it becomes advantageous to switch to the CPU server, and as such DDFlow is able to maintain minimal impact to end-to-end latency. Eventually, the GPU server fails, crashing the statically deployed application, but the DDFlow application continues.

The second application scenario illustrates adaptation during network over-utilization and access point failure. The objective is to stream live video from the drone to the client. As the drone moves in physical space, it switches wireless access points. When a backbone access point fails, the static deployment becomes unable to establish a routing path from drone to client. In DDFlow, upon wireless access point failure the networking system service dynamically switches to a Wi-Fi ad hoc peer-to-peer communication protocol. With only a single peer-to-peer hop, we can re-establish a routing path to the client and preserve the application. The results are shown in Figure 2.6.

CHAPTER 3

Portkey: Adaptive Key-Value Placement over Dynamic Edge Networks

3.1 Before there was Portkey, there were Pebbles

In my initial research on how to design advanced mobile sensing applications, we stumbled upon a pervasive problem when collecting high-frequency data on smartphones: there is no mobile data storage engine that can support the rate of collection needed for modern health applications, aside from writing straight to the file system.

SQLite is the de facto datastore layer on mobile devices including Android and iOS, but it is unsuitable for storing high-frequency raw sensor data streams. Its design of saving data as a flat file presents limitations on throughput, particularly as the overall database size increases. For workloads that are write-dominant, such as sensor data collection, data is seldom deleted or updated (e.g., sensor samples) and is often small in record size e.g., a single message record could be a few hundred bytes.

Writing data streams to SQLite can be prohibitively expensive due to SQLite database journaling and its update-in-place semantics i.e., records reside at a particular location in stable storage, and updates mutate the record directly. Furthermore, flash memory (the dominant stable storage medium in mobile devices) is page-oriented, which means that each record write corresponds to read and write of an entire page [OKL15]. Common page sizes for NAND Flash memory chips today are around 8KB, which further increases write ampli-

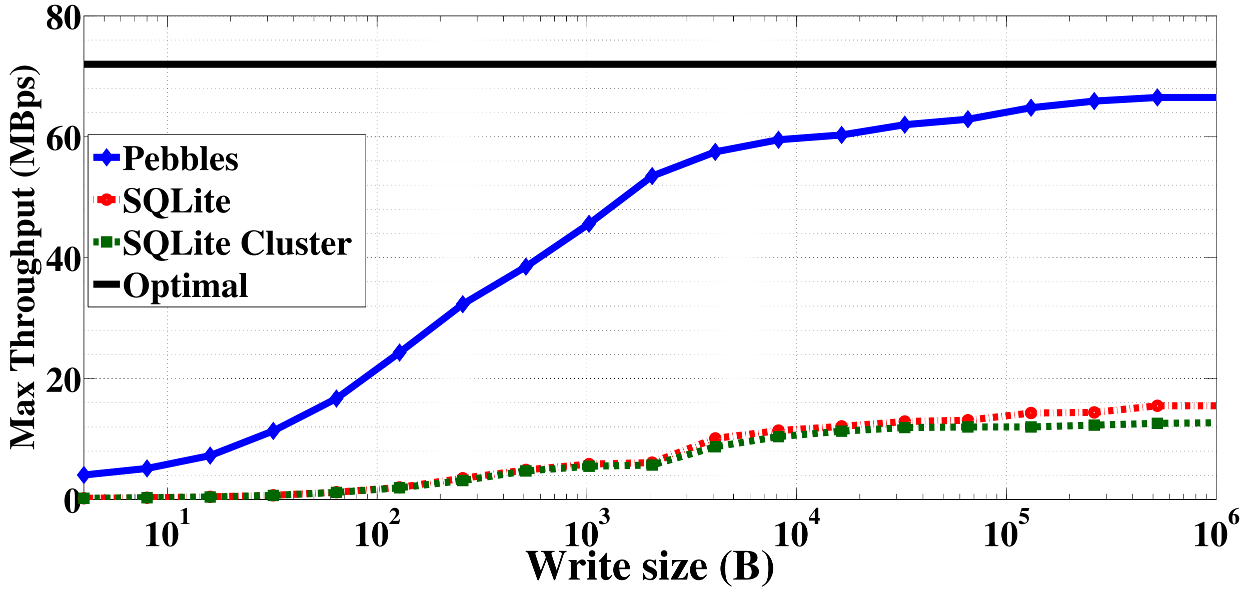


Figure 3.1: Maximum write throughput with increasing write size. Pebbles achieves 92% of the optimal write throughput while SQLite and SQLite cluster (used in AWARE) achieve 22% and 18% respectively at their steady states while writing large blocks of data.

fication[1] for small records that our target applications exhibit. In general, a single record inserted into a table with k indexes results in $2 \times (k+1)$ pages written under SQLite [OKL15].

Consequently, when using SQLite to store raw sensor data, as data size grows, the query performance begins to degrade and fall behind the rate necessary for real-time computation of biomarkers. For the mCerebrum [HHS17] system, after about 8 hours of data collection, biomarker computations begin to timeout due to growing query response time.

3.1.1 Scalable Storage of High-rate Sensor Data

Log-structured storage systems such as RocksDB [roc17] may provide an alternative to SQLite; however, RocksDB aims to support general RDBMS workloads and lacks the minimized overhead necessary to support mobile devices. To address the specific requirements

[1]Write amplification refers to the actual amount of data that is rewritten for a given record e.g., if records are stored in 8KB pages, then writing a 12 byte record results in writing at least an 8KB page.

of mobile sensor data workloads, we have developed a custom log-structured storage layer called *Pebbles*, which is optimized for high-frequency append-only writes of data arriving in batch or record streams. Pebbles also provides transparent data sync, allowing applications to offload data to the cloud for further processing and data archive. On the mobile device, data is stored in a circular log to maximize the throughput of flash memory. To support fast queries, Pebbles maintains a lightweight index on a logical timestamp and topic, which is used to identify data streams.

Figure 3.1 shows the max write throughput by varying data write sizes of Pebbles versus SQLite and a cluster of multiple SQLite databases (used in AWARE [FKD15]). This benchmark was performed on the internal flash memory of a Samsung Galaxy Tab S2 SM-T713. Each system was configured with an 8MB in-memory buffer (split across database instances for the cluster with round-robin writes) and performed a total of 4GB writes. The optimal throughput of 72 MBps was determined by performing one large consecutive write to the internal memory.

At lower data write sizes, such as those exhibited by typical mCerebrum [HHS17] applications, Pebbles outperforms SQLite by more than 20x. The performance gain of Pebbles is directly related to the lower write amplification relative to SQLite. In the lower data write sizes, the CPU becomes the bottleneck, preventing Pebbles from saturating maximum storage bandwidth. Nevertheless, the achieved throughput is sufficient for mCerebrum.

At large data writes, such as those to be exhibited by the mCerebrum batch data workloads, Pebbles is able to saturate storage bandwidth and outperforms SQLite by more than 4x. SQLite is not capable of saturating the storage bandwidth at these large write sizes due to system overhead, including primary key constraints and index maintenance, which attribute to increased write amplification. The SQLite cluster suffers even more performance due to its reduced ability to perform sequential writes. In Pebbles, write amplification is minimized through the use of a circular log that is clustered with the primary index i.e., both are append-only on new data writes and garbage collection is performed, on both, sequentially with an optional cloud data sync.

3.2 Portkey Introduction

The expansion of IoT and mobile systems has resulted in deployments of high-volume data producers and consumers residing at the network edge. Example applications include autonomous vehicular networks [SBS17, AQE20, QAB18, NSB18, WGM20, STA19], federated learning [BEG19, KMY16, ZWH19], drone-assisted disaster management [EN16, Cro19, EKN17], and AR/VR [CKY20, BBC17, EPB18]. Key to these applications are the inherent mobility and resource constraints of clients and (potentially) servers, as well as the requirement of low-latency data accesses [TWB20, ZWH19, REF16]. Consequently, these applications typically employ lightweight datastores (e.g., key-value (KV) stores such as Redis [Red20b] and Apache Cassandra [LM10]) *entirely* at the edge [MBL18, RG18, CLB18, MGS17], in order to avoid high edge-to-cloud communication latencies [NAE18].

Unfortunately, existing distributed KV stores are ill-suited for edge settings, and instead were designed for datacenter environments with relatively uniform client-server latencies, e.g., when servers reside on the same rack. Accordingly, KV stores typically opt for randomized KV assignment strategies [DCB19] such as consistent hashing and hash slot sharding that prioritize load balancing and fault tolerance, but ignore client mobility and the resulting client-server latencies (§3.4.1). The result is that retargeting KV stores to the edge computing context can yield largely inefficient data placements. For example, using our dataset for an autonomous vehicular application (§3.3.2), we find that existing placement strategies yield 1.7-11.9$\times$ higher access latencies than an optimal strategy that explicitly incorporates client mobility (§3.4.2); Figure 3.2 illustrates the intuition behind this suboptimality.

To fill this void, we present **Portkey**, the first distributed KV store that explicitly incorporates the time-varying mobility and latency patterns experienced by edge applications. Portkey formulates data placement as an online optimization problem, whereby data access patterns and client locations/latencies are continuously tracked and used to tune KV placements in a manner that globally minimizes access latencies (e.g., average, tail). To realize this, Portkey must overcome two challenges with regards to efficient data collection and fast placement decisions. The underlying insight to both of our solutions (described below) is that, due to the very nature of dynamic systems, an optimal placement now is likely to become stale in the near-future. Thus, Portkey prioritizes rapid but potentially suboptimal decisions over delayed optimal ones.

Challenge 1: efficient data collection. At its core, the ideal placement for a given KV pair (or simply KV) at any time is impacted by two factors: which clients access that KV, and what is their latency to each available datastore server. The former can be logged by transparent request proxying, while the latter involves generating a logical network distance matrix across distributed clients. Though conceptually straightforward, collecting latency information is difficult under the tight resource constraints imposed by edge networks (i.e., bandwidth [WZZ17]), as well as edge devices such as IoT gateways and sensor nodes (i.e., energy and memory [SCZ16]).

To solve this, Portkey generates succinct latency sketches [MRL19, GDT18] using a series of lightweight techniques inspired by Network Tomography [Vou14]. First, to generate a holistic view, Portkey profiles the end-to-end latency of the datastore from the perspective of each client by (1) passively profiling application-generated requests, and (2) judiciously inserting active probes to servers that are not accessed by client workloads. Then, in subsequent time windows, Portkey employs *locality-aware reprofiling* such that latency information is recollected only if there exists sufficient evidence that a client's motion *might* affect KV placements. Importantly, due to the short time windows that Portkey operates over, reprofiling decisions consider only proximal servers whose client-server latencies would be most affected by short-term mobility. The same servers should already house the KVs that a client accesses, resulting in low reprofiling overheads (i.e., few active probes).

Challenge 2: fast placement decisions. Even with the necessary information, making

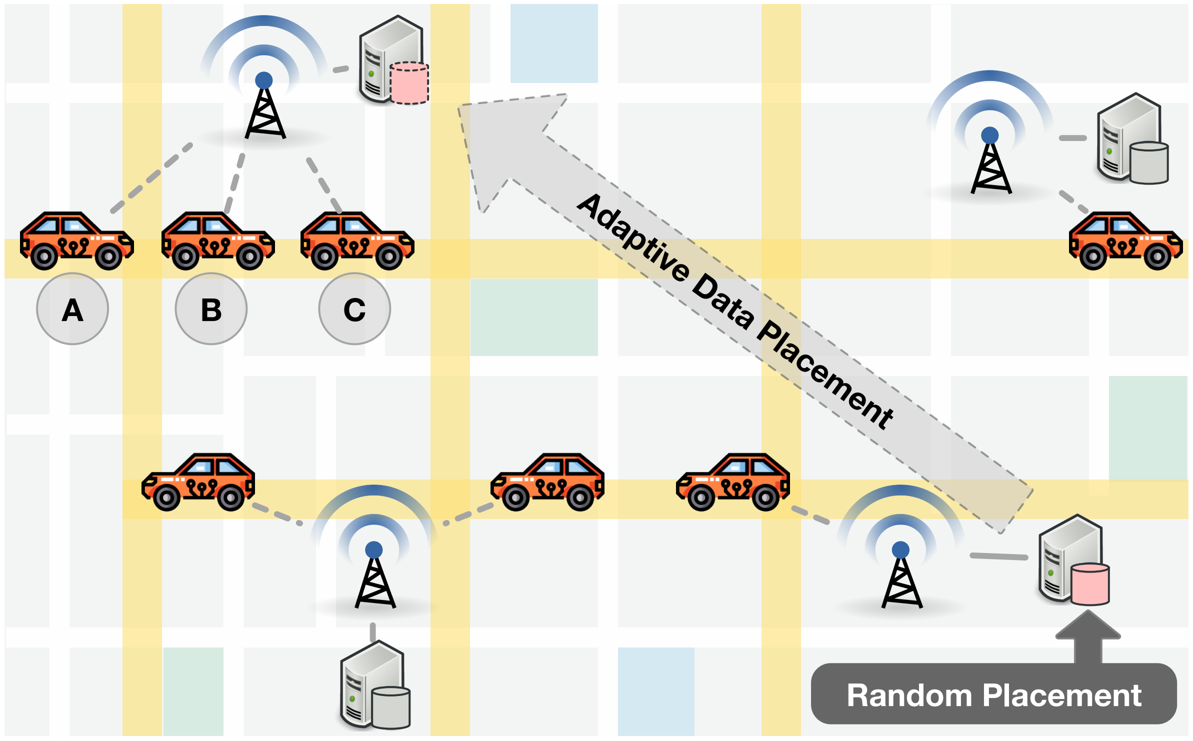


Figure 3.2: An example smart city deployment for coordinating autonomous vehicles. Distributed datastore servers are attached at access points spread throughout the edge network. The primary replica for a key-value (KV) pair shared by clients *A*, *B*, and *C* should intuitively be placed at their nearest host, which may vary over time as the clients move. The randomized placement used in existing systems ignores this locality, resulting in potentially large request latencies.

placement decisions involves solving a computationally hard optimization problem (reducing to the NP-hard Partition problem [KK03, BRS08]) that incorporates all of the influencing factors including client location, server capacity, network state, and workload patterns (§3.5.2). Worse, the ideal placements can change as any of the aforementioned properties change, which can occur at very short time-scales in settings with high client mobility. For example, in our mobility trace of taxis moving through Rome [BBL14], there is a 72% probability that at least one client will switch access points every 10 seconds.

To generate real-time placement decisions in response to changing system dynamics, Portkey's adaptive solver operates on keys independently, and handles host storage constraints by using a greedy assignment that prioritizes KVs with the largest marginal impact on overall datastore performance, i.e., balancing storage requirements with access frequency. Overheads are further reduced by having non-contentious keys, e.g., those that are accessed frequently but only by a single client, skip the formal solver. This greedy heuristic foregoes optimal placement in exchange for rapid placement of the most important KVs at any time. However, suboptimalities (e.g., from ignoring the impact of colocating KVs) only persist for short time scales, and subsequent profiling and solver iterations will reveal the missed latent effects, allowing for timely readjustment.

We implement Portkey as an immediately deployable modular extension to the Redis KV store that can *transparently* adapt data placements to arbitrary workloads and network characteristics. Unfortunately, to our knowledge, there does not exist a public dataset for our target edge applications that includes the associated client mobility. Thus, to evaluate Portkey, we first developed representative datasets for an autonomous vehicular application that incorporate *real* taxi mobility traces over public distributed KV benchmark workloads (§3.3.2); our datasets and testbed cover a wide range of values for parameters that affect datastore performance including data locality and client-server latencies. We also deployed Portkey in two (small-scale) smart building and crowd sourcing applications that use Raspberry Pis and live mobile networks. In comparison to the predominant randomized placement policy and a variety of locality-aware heuristic strategies, Portkey reduced average and tail (95th percentile) request latencies by 21-82% and 26-77%, while delivering low network (1-3%) and memory ($<$ 1MB for thousands of servers and KVs) overheads. We will open-source Portkey and our datasets post-publication.

3.3 Target Applications

In this section, we first describe our target edge applications and their intrinsic properties (§3.3.1), and then describe the representative workloads (§3.3.2).

3.3.1 Edge Applications and Goals

Autonomous Vehicular Networks. Applications over vehicular networks commonly require access to a distributed data management platform. For example, rapid (10s-100s of ms) updates to 3D feature maps (e.g., CarMap [AQE20]) offer real-time information needed for localization and route-planning over varying traffic and road blocks. These feature maps are naturally disaggregated based on location, while the sets of accessing vehicles depend on current location and active route. In contrast, enhancing situational awareness by expanding line-of-sight perception through inter-vehicle data sharing (e.g., AVR [QAB18]) can reduce accidents and improve driving experience. Unlike stationary 3D feature maps, AVR information is inherently attached to mobile vehicles whose locations vary over time. For practical collision avoidance, latencies on the order of tens of milliseconds are needed [JS14, BTD06]. Finally, future ride-hailing services that coordinate a large fleet of autonomous vehicles must perform rapid data access and decision making to determine optimal route planning and customer-vehicle matching [NSB18, WGM20, STA19].

Disaster Management. Embedded technologies have dramatically impacted our ability to predict and respond to natural disasters at the edge. These data processing pipelines are fundamentally latency-sensitive; faster detection provides a better response opportunity. For example, early earthquake and wildfire detection sensor systems leverage crowdsourced data across mobile clients (e.g., IMU or inertial measurement unit values, locations) to determine disaster areas (e.g., earthquake hotspots) and offer advanced notification enabling vital preparation [KLA19, MBG15, Pay19]. Incorporating UAV and drone deployments further enhances situational awareness and augments disaster response, ranging from firefighting to search & rescue [Cro19, EKN17, EN16]. These systems commonly rely on a distributed storage platform for unified data collection, fusion, sharing, coordination, and localization.

Federated Learning. Federated learning is a novel approach to distributed machine learning over multiple edge devices that does not need to exchange or centralize local data samples [BEG19]. As it involves continual data updates between edge devices and shared parameter servers, early implementations have been shown to suffer from inefficient network communication [KMY16]. These delays are further exacerbated by the mobile and disparate nature of clients in edge settings. Accordingly, solutions have noted the importance of low-latency datastore accesses for federated learning [ZWH19].

Augmented and Virtual Reality. Avoiding "VR sickness" is a difficult challenge in AR/VR applications. Although a number of sources contribute to this effect, reducing round-trip latency is a primary factor impacting application integrity and user discomfort [CKY20, BBC17]. To this end, researchers have noted the need for a low-latency data management platform for AR/VR applications, with a gold standard of 15–20ms [EPB18]. Use cases for such a datastore include persisting and quickly sharing virtual environment information (e.g., locations, textures, state) to enable faster rendering and real-time updates for end users.

3.3.1.1 Key Workload Properties

Although emerging applications in the edge network setting are diverse in nature, they share key workload characteristics that distinguish their datastore requirements from traditional cloud applications.

1. **High Latency Sensitivity.** Low-latency data access (i.e., no more than tens of milliseconds) is critical for all of the aforementioned applications. As application data is commonly produced and consumed at the edge, this motivates data storage to also occur entirely at the edge (to avoid costly cloud-edge network latencies [NAE18]). Note that this is true even for data streaming systems that use a data management

broker for pub/sub messaging with data persistence (e.g., for fault tolerance and recovery) [Red20a].

2. **Large Client-Server Latency Discrepancies.** The networks onto which these applications are overlaid do not offer the relatively uniform client-server latencies of a cloud cluster. Instead, geo-distribution results in certain edge datastore servers residing "closer" to a client than others (with respect to latency).

3. **Device Mobility.** A unique aspect of these applications is the inherent mobility of the client devices. From smartphones to autonomous vehicles, locations change over time. The movements of a given client can also directly affect its client-server latencies. Accounting for and adjusting to this volatility is essential in optimizing for access latency.

3.3.2 Representative Dataset and Testbed

To the best of our knowledge, there are no openly available datasets representative of the workloads in the aforementioned edge applications. Thus, to provide a baseline for testing and evaluating improvements to data placement policies, we developed an in-house edge-KV dataset; we will open-source this dataset post publication. Our dataset is inspired by the autonomous vehicular network application, and incorporates realistic client mobility patterns and distributed data access characteristics. More specifically, our dataset leverages *real* mobility traces of taxis moving through Rome [BBL14], and its data accesses are derived from the de-facto KV benchmark, YCSB (augmented with its distributed extensions [PPR11, CST10]). Of course, precise data access/locality patterns and network latencies play a large role in edge datastore performance; we next describe how our setup and datasets cover wide ranges of values for these different properties.

Network Setup. Given an autonomous vehicular network, deployments can range from zero-infrastructure peer-to-peer routing to a those that rely on wireless access points and/or

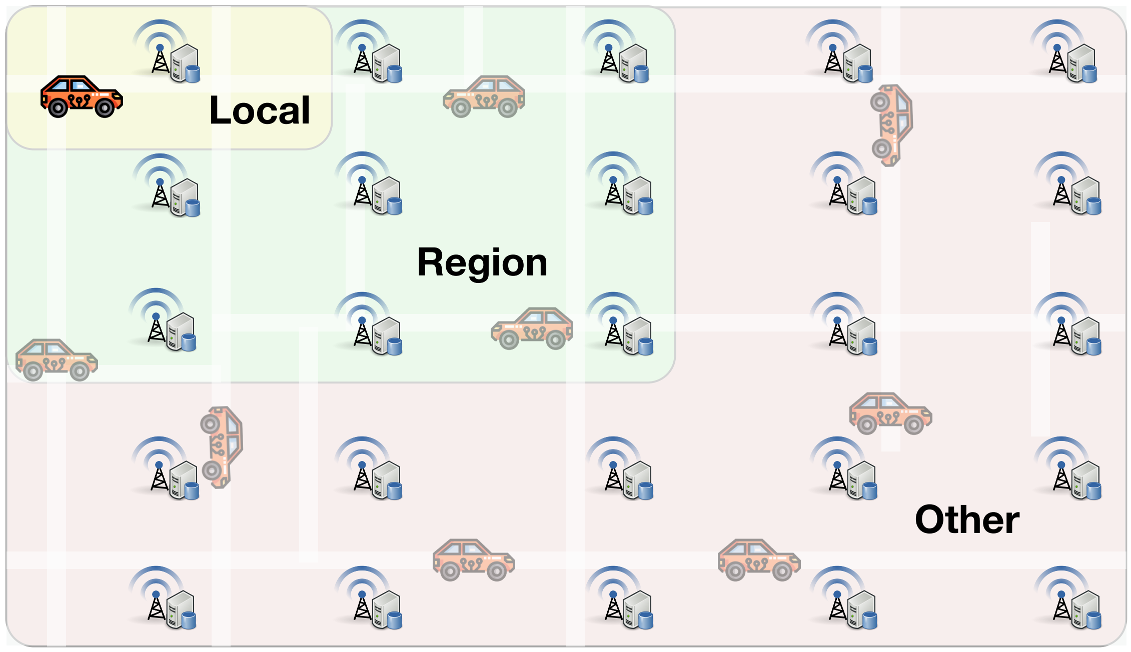


Figure 3.3: *Testbed Setup.* 25 access points, each housing a datastore server, are overlaid onto a 5x5 city grid. 25 autonomous vehicles move throughout the city according to real taxi mobility traces. Each vehicle acts as a datastore client, deploying one of the representative workloads. Network latencies are assigned in a three-tier approach, such that contacting farther access points involves longer delays.

base stations connected via a wired backhaul [JSF18, FH08]. Portkey is explicitly designed to operate across this spectrum of potential network topologies. While its operation is agnostic to the particulars of the network setup, its advantages are most pronounced when there is a large latency discrepancy between clients and datastore servers. For the purposes of evaluation and analysis, we opted for a setup as depicted in Figure 3.3, with 25 Wi-Fi access points (APs) each housing a datastore server, overlaid onto a 5x5 grid with an inter-AP distance of 400m (approximately 2x the outdoor range of 802.11n [AMO15]). Vehicles act as datastore clients as they move throughout the region based on their corresponding taxi mobility trace. For a flexible environment enabling an evaluation of cases where the benefits of adaptive data placement are minimized, we emulated this network using Mininet [min20, FAB15], with datastore clients and servers running on a shared server.

Given a network setup that most closely resembles a wireless last-hop to wired backbone network, we opted to assign network latencies in a three-tier approach guided by the reported end-to-end vehicular network latencies as described in [CBM17, NF13, SMA04, PNS18]. In our default configuration, clients access their *local* AP with a random latency ranging from 5-10ms [SZL16, MRS18], *regional* APs with a random latency between 20-30ms, and all *other* APs with a random latency between 40-50ms. In §3.7.2, we evaluate different latency ranges for each tier, including cases where the benefits of Portkey are minimized (i.e., when there is minimal client-server latency discrepancy). Additionally, §3.7.4 presents results from two small-scale applications running over real (not emulated) networks and edge devices.

Application Workloads. Edge applications vary in terms of the relationship between data locality and data access patterns. To incorporate these properties, we decomposed the YCSB benchmark suite into 6 workload traces that holistically cover both regional locality and the local vs. global nature of KV ownership:

- **per-client**: comprises KVs that are owned and modified by singular clients.
- **regional**: KVs are assigned to individual regions, and clients only access (read or write) KVs for the region they are currently located in.
- **group**: splits KVs and clients into random groups (irrespective of region). Clients only access the KVs within their group.
- **global**: contains global KVs accessed and updated by all clients, e.g., full dataset scans.
- **all-RW**: combines all previous workloads, with request type evenly split across KV reads and writes.
- **all-R**: contains the same KV access patterns as "all-RW," but all accesses are reads.

Each workload has a data access trace per client, and each trace consists of 20,000 read or write (i.e., get or set) requests to a subset of 1,000 keys. All KVs are sized at 1KB. Inter-request delays are sampled from a Generalized Pareto distribution with a shape parameter of 0.155, and are scaled to an average delay of 100ms in line with prior work [AXF12].

3.4 Background and Motivation

Here, we provide a brief overview of existing KV datastore placement strategies (§3.4.1), and present measurements illustrating why these placement strategies are suboptimal for deployments spanning edge networks (§3.4.2). Note that while KV stores such as Redis can be used as a front-end cache for another backing datastore, our focus is on distributed implementations (e.g., Redis Cluster) that serve as a persistent (i.e., durable) store tolerant to machine failure via replication, fault tolerance, and failover mechanisms.

3.4.1 Existing Placement Strategies

Random KV Assignment. The vast majority of distributed KV stores employ hash-based sharding when partitioning keys across a cluster [DCB19]. Figure 3.4 depicts two popular implementations: consistent hashing (used by Cassandra [LM10] and memcached [Dor20]) and hash slot sharding (used by Redis Cluster [Red20b] and MongoDB [Mon20]). Both of these approaches result in a *random* assignment of KVs to datastore servers; the difference is in the granularity of assignment. In consistent hashing, each key is hosted and replicated at its nearest servers in a hash ring. On the other hand, hash slot sharding groups keys into slots, with each slot having an individual assignment to replica servers. Although such randomized placements ignore the effects of non-uniform client-server latencies (§3.4.2), they do offer desirable load balancing properties.

Data Replication. Consistency and fault tolerance across a replica set are accomplished through either quorum consensus or primary-secondary replication [DCB19]. Primary-secondary replication allows consistent data accesses to be optimized by only focusing on the

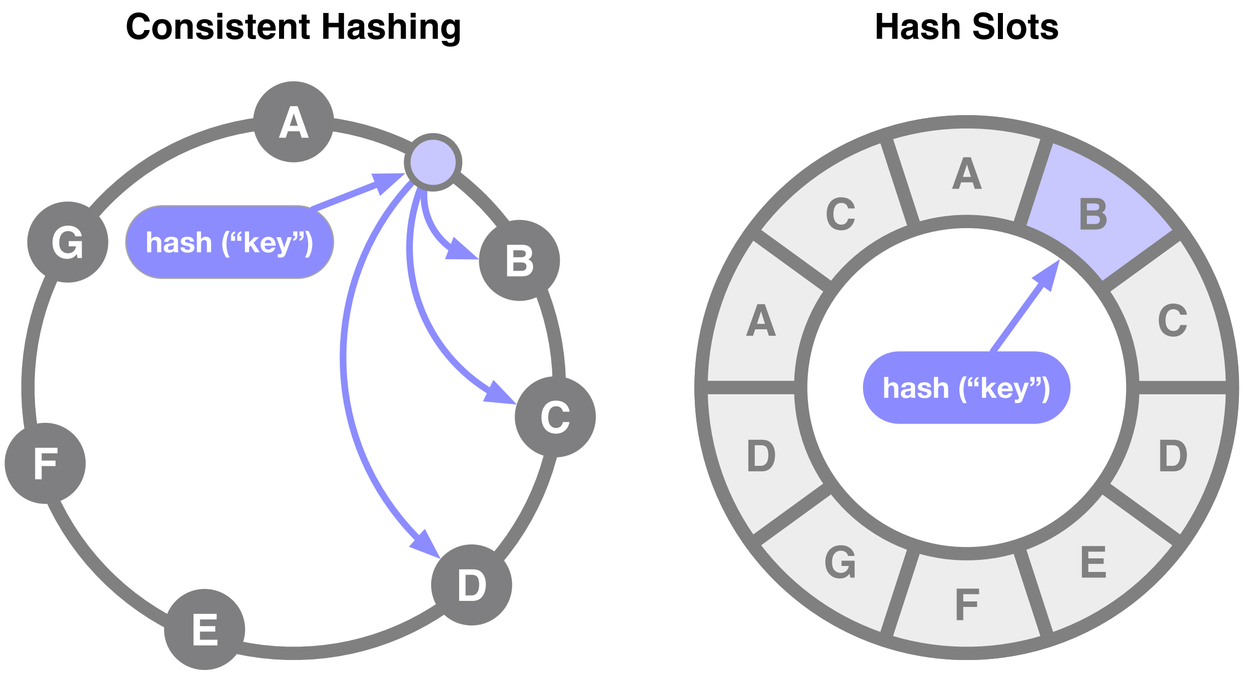


Figure 3.4: In consistent hashing, servers and keys are hashed onto a ring. Replicas are selected by traversing the ring. In hash slot sharding, the ring space is divided into equal slots, with each slot assigned to datastore servers.

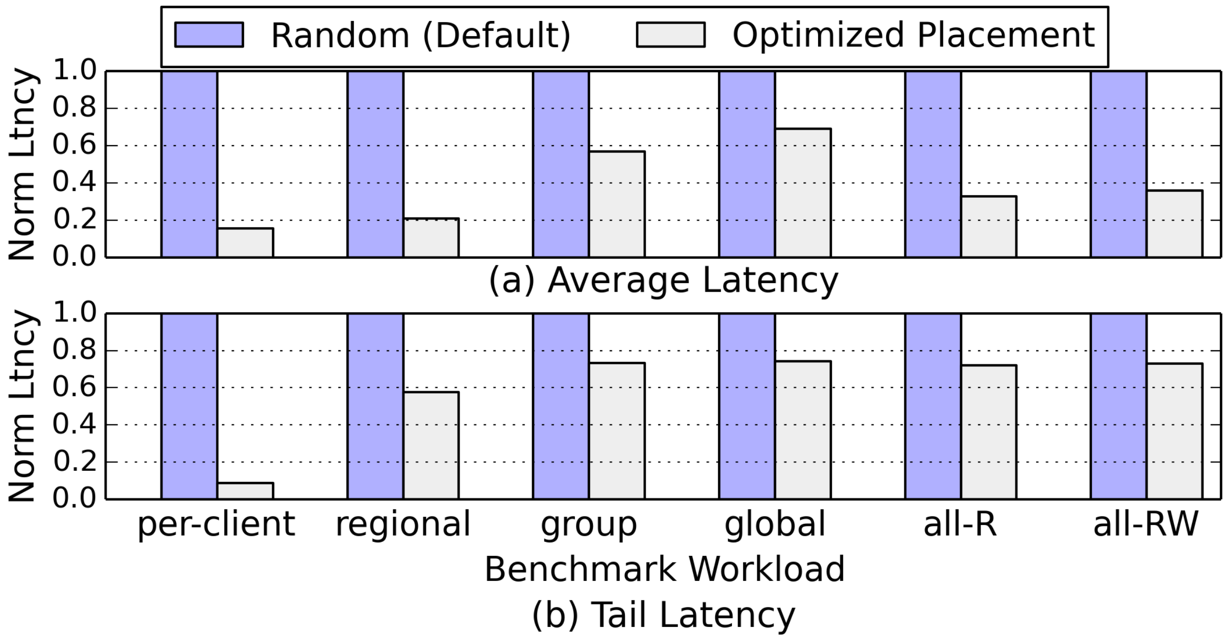


Figure 3.5: Performance impact (for average and 95th percentile tail latency) of an optimal KV placement policy that explicitly considers client mobility, versus the standard randomized placement policy. Results are normalized to those with the randomized policy.

primary replica placement. In a quorum, request latency is based on the slowest reply, thus requiring the colocation of a quorum near the accessing clients. We target primary-secondary approaches and focus on optimizing placements of a key's primary replica. However, we note that Portkey's placement strategy can be directly applied to secondary or quorum replicas as well. Further, as discussed in §3.6, Portkey adopts the same fault tolerance and consistency guarantees as the datastore it runs atop.

3.4.2 The Case for Adaptive KV Placement

To illustrate the performance benefits of adaptive placement, we compared the *Random* placement strategy used by existing KV stores to an *Optimal* placement strategy that leverages future (perfect) knowledge of client locations and data accesses. Using this information, the Optimal strategy predetermines the ideal placement for each KV over a short (near-instantaneous) time window. More specifically, each KV is hosted at the datastore server that minimizes average or tail (95th percentile) request latency across all client accesses in the current window. §3.5.2 formalizes the optimization problem that underlies the Optimal strategy.

Figure 3.5 shows the results for both strategies across all of our workloads (§3.3.2). As shown, the Random strategy results in average request latencies that are 1.7-5.4× worse than the Optimal across the workloads; suboptimalities are 1.4-11.9× for tail latencies. The main issue with Random placement is that it ignores the (time-varying) locality of datastore clients and their KV accesses, and thus potentially places KVs far away from their accessing clients. Of course, the impact of this omission is more pronounced in certain workloads than others. For example, when considering average request latency, the inefficiency is most pronounced (4.8-6.4×) for the *per-client* and *regional* workloads where client KV accesses are inherently localized (to the client's location or its encapsulating region). In contrast, the discrepancies between Random and Optimal placements are lower (1.5-1.8×) for the *group* and *global* workloads where KV accesses are not explicitly centered around spatial locality. However, even in these cases, the Optimal strategy outperforms the Random one, primarily by hosting KVs at the servers nearest the majority of (potentially dispersed) accessing clients.

Takeaway. These results suggest that incorporating knowledge of client mobility and data access patterns into KV placement decisions can substantially improve overall datastore performance (as measured by client-perceived request latency). Of course, the aforementioned Optimal placement strategy presents a loose upper bound and is unrealistic in practice given its oracle-like knowledge of future data accesses and client locations. In the next section, we describe how Portkey realizes many of these benefits in a practical manner, by using continuous (but efficient) workload and network profiling, and an online migration strategy to achieve real-time, dynamic KV placement.

3.5 Design

This section details the system design that Portkey uses to practically realize adaptive KV placement with near-optimal performance. Doing so requires addressing two key questions. First, how can the essential information needed to determine an optimal placement (i.e., data accesses, time-varying client-server latencies) be efficiently collected across resource-constrained edge devices and networks? Second, despite the associated computational complexity, how can the collected information be used to make rapid but effective placement decisions that keep pace with time-varying networks and client mobility?

The intuition underlying Portkey's design is to lean into the client mobility and dynamism intrinsic to edge settings. More specifically, without an oracle, Portkey must rely on recent client access patterns and locations/latencies to predict future accesses and latencies. Of course, accurate predictions become more challenging to obtain over long time horizons. Further, in our target settings, recent accesses and current predictions are likely to become outdated quickly as clients move around and client-server latencies adjust accordingly. Thus, Portkey prioritizes fast (and frequent) approximate decisions over slow optimal placements. Accordingly, Portkey opts for an iterative, fast-correcting approach to KV placement for real-time adaptation to edge system dynamics.

Figure 3.6 illustrates Portkey's processing pipeline and workflow. Portkey is incorporated as a modular extension atop existing datastore systems. The client datastore library is augmented to track each KV access and judiciously monitor end-to-end client-server latencies (§3.5.1); §3.6 discusses why we opt for client-side profiling. This data is uploaded to the Adaptive Placement Engine at an application-defined *window size*.[2] Upon reception from all clients, the engine first computes a global network distance matrix. Along with the aggregate sets of client-key accesses, the Placement Solver performs fast global approximation

[2]The frequency of upload affects both the agility with which Portkey adapts to system dynamics, and the network overheads imposed by shipping profiling information. We use a default window of 10 seconds.

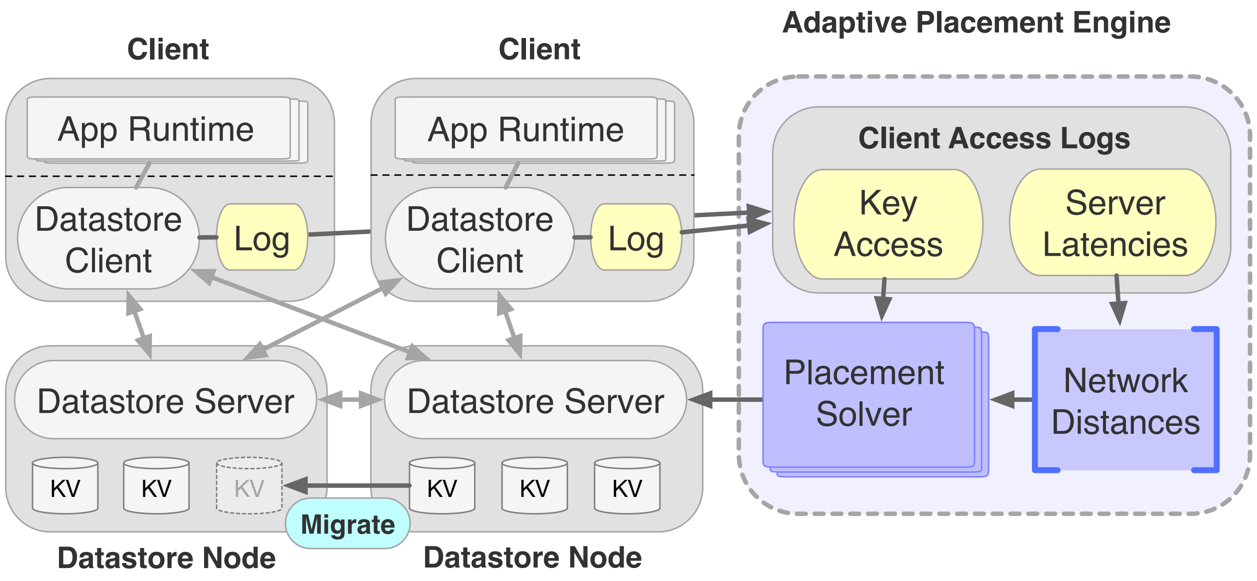


Figure 3.6: Overview of Portkey. Data accesses and latency information are collected by clients and periodically uploaded to the Adaptive Placement Engine, which determines near-optimal placements and issues the corresponding migration instructons to datastore servers.

of optimal KV placements and issues migration instructions to the appropriate datastore servers (§3.5.2).

3.5.1 Efficient Data Collection

Efficient profiling of information that influences optimal KV placements can be broadly decomposed into two categories: minimizing network (and accordingly, device energy) overheads with judicious client-server latency probing (§3.5.1.1), and succinctly storing latency information and KV access statistics to minimize device memory overheads (§3.5.1.2).

3.5.1.1 Judicious Network Probing

Identifying the optimal KV hosts requires an understanding of the latency delays that each client would incur in contacting each potential datastore server. This latency should encapsualate both the network delays in contacting a server, as well as the processing delays that the server imposes in serving a requested key. A naive strategy to collecting this information would be for each client to simply, in each time window, issue a probe request to each server to log the necessary information. However, this additional (per client-server pair) traffic would add undue stress to servers and edge networks, as well as client devices that must expend energy to support such network transfers.

Instead, Portkey employs a lightweight, end-to-end probing technique that is inspired by Network Tomography [Var96]; importantly, we eschew approaches that rely on support from intermediary network nodes as we target general edge applications with varying administrative policies. At a high level, tomography seeks to infer network internals using only end-to-end measurements that take one of two forms: *passive* tomography leverages data from traffic generated naturally by users, while *active* tomography inserts probes into the network to glean measurements. The goal of Portkey's approach is to leverage its operation over short time-scales (i.e., short windows) to minimize the number of active probes required to obtain accurate and holistic client-server latency information, despite client mobility.

Approach. To develop a comprehensive latency profile to all servers in a given time window, Portkey's client datastore library passively profiles every application-generated request to log the accessed server, as well as the incurred round-trip latency (including server processing delay). Of course, a client's natural data accesses may result in incomplete latency information by failing to contact certain datastore servers; this is especially true with adaptive placement, as distant servers should be minimally contacted. To fill in the missing information, Portkey actively injects requests targeting only the excluded servers, thereby limiting overheads.

Given the short windows over which Portkey makes placement decisions, regenerating an entirely new set of latency values in each window is impractical. This is true even with Portkey's judicious injection of active probes, as clients are unlikely to contact many servers in a short period of time. To handle this, Portkey uses *locality-aware reprofiling*, depicted in Figure 3.7, to recollect client-server network information only if there is sufficient evidence

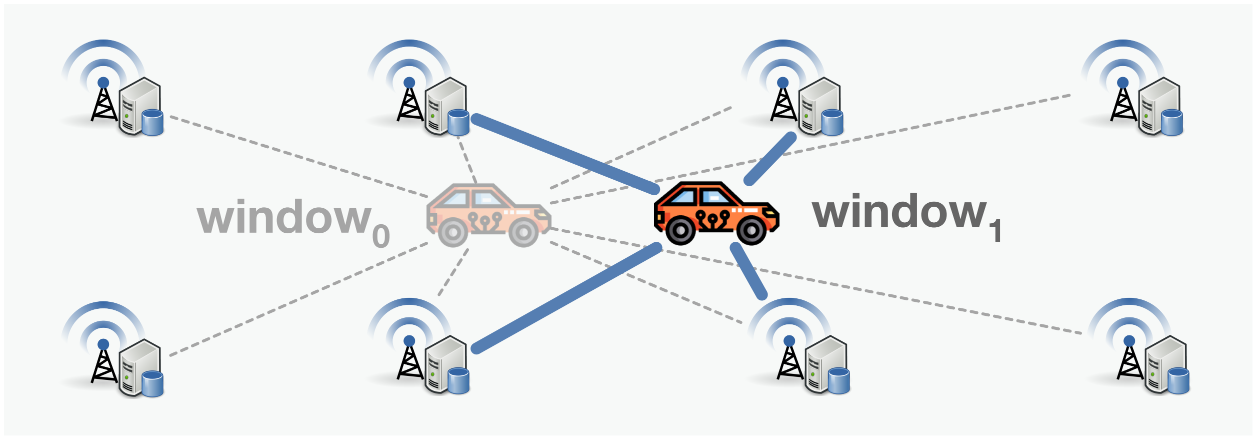


Figure 3.7: Portkey's locality-aware reprofiling. In subsequent profiling windows, clients only contact their nearest datastore servers, and use the observed latency values to determine if placement-altering motion has occurred; if so, clients then collect latencies to the remaining servers.

that latency values have changed enough to *potentially* alter KV decisions. In each time window, a client collects latency information (preferably passively, but if not, actively) only to the k-nearest datastore servers; k is a configurable parameter that is set to 5 by default in our experiments. If the relative ordering of latency values amongst those servers changes, *or* if any latency values change by more than a configurable threshold (20% by default), then a client is deemed to have moved enough to warrant full reprofiling, and active probes are injected to any remaining datastore servers that are not passively contacted.

The guiding intuition is that a client's mobility is inherently localized over short durations, and latencies to the nearest servers are the best indicators of how much motion has occurred. Importantly, with adaptive placement, the nearest servers to a client are the most likely ones to host the KVs that the client accesses. Thus, latency information to the k-nearest datastore servers will often be passively profiled, resulting in low overheads.

3.5.1.2 Efficient Storage.

During normal operation in a window, a client may passively collect multiple latency values per server. Given the potential memory constraints on edge clients, this information must be stored efficiently. To do this, Portkey provides a succinct latency sketching framework that enables applications to specify which part of the latency distribution they would like to consider. When tracking average request latency, each client stores the number of accesses and total aggregate latency for a server, which can be used to derive average request latency while requiring only an 8-byte memory footprint per server. If the application instead wishes to optimize for a metric that requires the full latency distribution (e.g., tail latency), Portkey employs the DDSketch [MRL19] and moments [GDT18] sketching techniques to track approximate quantiles with a minimal memory footprint.

Tracking Data Accesses. In addition to latency information, Portkey clients must also track KV accesses. In particular, for each data request, Portkey must record which KV was accessed, as well as the corresponding payload size. Payload sizes must be collected because they dictate which KVs can fit on a given server, and they provide a mechanism with which to compare the relative importance of different key placements; indeed, Portkey's Placement Solver (§3.5.2.2) relies on payload sizes to scale each KV placement to a marginal per-byte cost benefit.

The process of logging client data accesses is fairly straightforward: Portkey transparently proxies each request/response in the client datastore library. Instead, the primary consideration here is efficient storage, particularly since each client can access a given KV multiple times in a given window. Portkey relies on a key sketch, depicted in Figure 3.8, to bound the memory overhead at each client. Key access counts and aggregate payload size are stored to infer an average payload size for a KV. Given the sparse nature of client-key accesses, Portkey's implementation only stores non-zero key counts, consuming 8 bytes each.

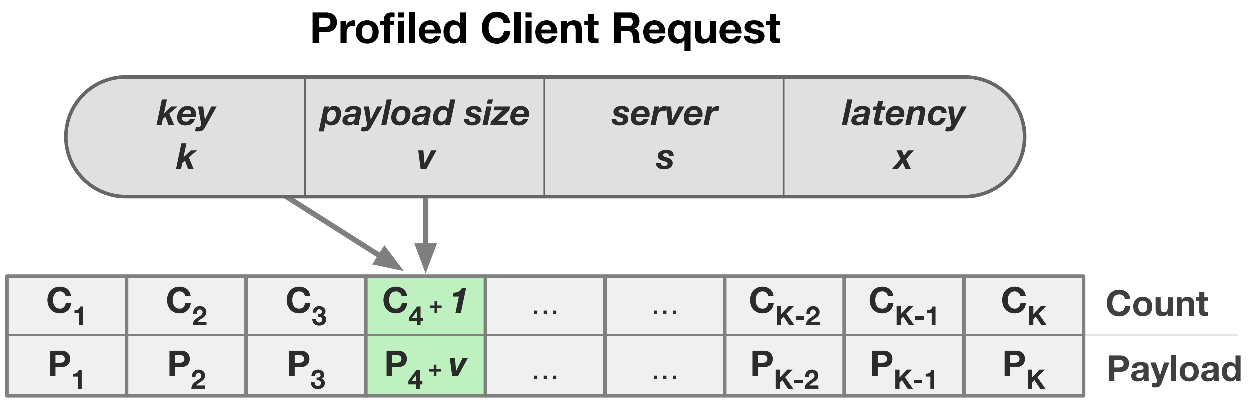


Figure 3.8: Portkey's sketches for efficiently tracking client data accesses. Access counts and aggregate payload size capture individual client workload patterns.

3.5.2 Fast Placement Decisions

Given the information collected in §3.5.1, the Portkey Placement Solver is responsible for making adaptive placement decisions. Solving for optimal data placement is known to be NP-hard, as it reduces to the Partition problem [KK03, BRS08]. Worse, an optimal solution is likely to change with client mobility, which can occur at very short time-scales in the edge network setting. This section begins by formalizing the placement optimization problem (§3.5.2.1) and then describes Portkey's greedy approximation to enable fast placements that closely resemble optimal decisions (§3.5.2.2). We center the discussion on optimizing average request latency, and conclude with a description of modifications to support tail latency optimization (§3.5.2.3).

3.5.2.1 Problem Formalization

Given a datastore spanning N nodes, the solver's objective is to identify the best host servers for the K keys contained in the system. The best host for a particular key k is chosen by minimizing the overall cost $C(k)$ across all candidates. The specific cost metric depends on the desired performance objective, e.g., minimizing average or tail request latency; we focus on average latency for now. Accordingly, the host is chosen by minimizing

$$C(k) = \min_{n} \ C_n(k) \quad \forall\, n \in N$$

where $C_n(k)$ is the cost (i.e., latency) of hosting a key k at node n. This latency cost can be computed as an average over every client's distance from the candidate node weighted by the frequency of client access to the given key,

$$C_n(k) = \sum_{i=1}^{N} f_i(k) \cdot d_{in}$$

where d_{in} refers to the distance between nodes i and n (e.g., the average request latency in serving a client i from host n), and $f_i(k)$ represents the relative frequency that client i accesses key k. Client access patterns for each key can be modeled as a frequency access vector

$$\overrightarrow{f(k)} = \begin{bmatrix} f_1(k) & f_2(k) & \dots & f_N(k) \end{bmatrix}$$

which is dictated by each client's access count for the specified key in its data access sketch. Network distances can be represented via the following distance matrix:

$$\mathbf{D} = \begin{bmatrix} d_{11} & d_{12} & d_{13} & \dots & d_{1N} \\ d_{21} & d_{22} & d_{23} & \dots & d_{2N} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ d_{N1} & d_{N2} & d_{N3} & \dots & d_{NN} \end{bmatrix}$$

Such a distance matrix can be extracted from the statistics collected across each client's network sketch. Given this formulation, the key's cost vector can be computed as a vector-matrix multiplication:

$$\overrightarrow{C(k)} = \overrightarrow{f(k)} \cdot \mathbf{D}$$

This process can then be independently repeated for each key. If each key's frequency access vector were to be instead represented as a row in a key access matrix, multiplying the key access matrix with the distance matrix derives a cost matrix containing cost vectors across all keys with a single matrix-matrix operation. The computational complexity in computing this cost matrix scales linearly with the number of keys, and in polynomial time with the number of nodes; that is, $\mathcal{O}(kn^2)$. The $k \times n$ cost matrix must then be scanned to identify the best candidates for placement. An optimal assignment of KVs over this cost matrix is NP-hard when considering the limited storage capacity at each server [BRS08].

3.5.2.2 Portkey's Adaptive Placement

Portkey's solver is predicated on the notion that an optimal assignment in any given moment is likely to become stale over time, especially with moving clients. As such, Portkey employs an approximate solver that can quickly and iteratively respond to system dynamics using the latest client-provided information. This is accomplished via a three-fold approach. First, KV costs are treated as independent. While this ignores the impact of colocating KVs onto the same server, subsequent request profiling and solver iterations will account for this effect, readjusting if needed. Second, to address host storage limitations, a greedy assignment prioritizes the KVs with the largest marginal impact on performance. This sacrifices optimal placement in exchange for a rapid adjustment of the most important KVs accessed at any given moment. Finally, to ensure efficiency with data-intensive systems, a fallback heuristic is selectively applied to the least contentious KVs; for example, a KV accessed only by a single client can forego the solver and quickly be placed at its nearest host.

While this approach deviates from the optimization formulation in §3.5.2.1, it allows for fast and self-correcting placement decisions that closely mimic optimal decisions (§3.7). We next detail the three principles of Portkey's solver in turn.

Independent KV Cost Analysis. An optimal placement assignment should consider the impact of colocating KVs on the same server. This results in an exponential decision space; reassigning a KV affects server load and latency, thereby requiring a cost matrix update for all remaining KVs. The Portkey placement engine opts to treat costs as independent, ignoring the impact of KV assignment. The guiding intuition is two-fold. First, we predict that in many cases the larger contributor to round-trip request latency is network delays as opposed to server processing time. More importantly, given the iterative nature of Portkey's placement decisions, subsequent profiling (which incorporates both network and server delays, as per §3.5.1) will quickly reveal this latent effect, allowing for prompt self-correction.

Greedy Assignment. To greedily place KVs, Portkey first computes a utility score per KV, indicating the marginal benefit of prioritizing its reassignment. Specifically, the utility score is calculated as the difference in cost (i.e., latency) between the current assignment and the optimal assignment, scaled by the KV payload size. This provides a per-byte marginal cost benefit. These utility scores are then sorted from largest to smallest impact, and KVs are greedily assigned in order of importance. Once a server reaches its storage capacity, keys are assigned to the next best host. When including the cost matrix computation, this assignment process results in overall solver complexity of $\mathcal{O}(kn^2 + k \log k)$.

Skipping the Formal Solver. For very large systems (with many KVs) and small window sizes, even a polynomial-time solution may be insufficient. To this end, Portkey selectively employs a locality-aware heuristic that attempts to skip the formal solver. To do so, the placement engine maintains a notion of the max number of keys that can have the full cost matrix computed and analyzed in sufficient time, based on the (1) number of datastore servers, (2) window size, and (3) reusability of computational profiling from prior iterations. If the number of keys accessed in a given window exceeds this max value, a first pass over the key access matrix sorts the keys by the number of accessing clients. Keys accessed by the largest number of clients are solved with cost vector analysis, and the remaining ones use a *dominant-node* heuristic where keys are assigned to the server nearest the most frequently-accessing client. The idea is that the formal solver is most helpful in balancing placement across a large number of (potentially dispersed) accessing clients; keys accessed by a single client are optimally placed nearest that client.

3.5.2.3 Optimizing for Tail Latency

Portkey's solver is not inherently tied to optimizing for average latency across clients and accesses. Supporting other optimization metrics simply involves altering the generation of the cost matrix used by the adaptive placement solver. For example, to optimize for tail latency, each client uploads its tail latency (instead of average latency) to every datastore server. Then, to compute the cost for hosting a given key on a given candidate server, we generate a distribution of accesses to that key where a client's latency to that server appears a number of times equal to the number of times that client accessed the key. The latency component of the cost is then set to be the tail (e.g., 95th or 99th percentile) of that distribution, rather than a weighted average across all clients' accesses to that key. The remainder of the Portkey solver then operates in the same way as above.

3.6 Implementation

We implemented Portkey as a modular extension to the Redis Cluster distributed KV store. Aside from being one of the most popular data management systems [Sol20], Redis natively supports deployment over lightweight embedded devices, such as Raspberry Pis and Android smartphones [Red20d], thereby offering compatibility with our target edge applications. As Redis Cluster uses hash slot sharding (§3.4.1), we implement adaptive placement by dynamically adjusting the slot assignment map and migrating the associated keys. This coarsens the granularity of adaptive placement from individual KVs to hash slots; however, we discuss placements in terms of keys for ease of disposition.

Our implementation of Portkey includes (1) altering the Redis client library to support the collection of datastore usage statistics (we consider the redis-clustr npm package [GoS20]), and (2) developing a standalone program encompassing the placement solver engine. In total, our implementation required ≈1,000 new LOC. A key benefit of client-side modification is that it allows for adaptive placement on unmodified servers, thereby supporting future versions without needing to fork the code base. Further, clients can optionally prioritize which requests are latency-sensitive with selective logging, and in doing so forego modification to the request protocol.

Consistency. As our implementation required no internal modifications to the Redis Cluster, the consistency of Portkey is the same as that of Redis Cluster. According to its specification [Red20b], Redis Cluster cannot guarantee strong consistency, as the primary replica will acknowledge writes before ensuring propagation to a quorum of secondary replicas. As such, primary replica failure could potentially result in the loss of writes that have been acknowledged.

Fault tolerance and recovery. Portkey leverages the standard Redis Cluster mechanisms for fault tolerance and recovery. In the case of a primary node failure, a failover mechanism promotes a secondary replica to replace the primary, adjusting the cluster as needed [Red20b]. This failover is maintained even if a node fails during key migration.

Ensuring consistency during migration. The migration mechanism used in our implementation follows the recommended Redis Cluster protocol typically used to redistribute keys for nodes entering or exiting the system during deployment [Red20c]. The main benefits of this approach are that (1) no data loss can occur in the case of migration or node failure, and (2) concurrent updates are allowed during migration. This ensures that client workloads remain uninterrupted during migration; adaptive placement can be transparently performed with the only noticeable effect on clients being a reduction in access latency. However, a notable drawback in the Redis Cluster reassignment protocol is a restriction that limits bulk migration. This is due to specifics in the Redis Cluster epoch mechanism used to propagate assignment map updates. In order to maximize the immediate benefit of adaptive placement, Portkey's Placement Solver sorts key assignments based on marginal cost reduction before issuing migration commands.

3.7 Evaluation

To evaluate Portkey, we primarily use the autonomous vehicular workloads and experimental setup described in §3.3.2. §3.7.4 additionally describes results from two real, small-scale application deployments. Throughout the evaluation, we consider two versions of Portkey that optimize for either average request latency or 95th percentile tail latency.

3.7.1 Request Latency Speedups

We compared Portkey with four alternative placement strategies. **Random** refers to the randomized placement strategy used by default Redis and most other existing KV stores (§3.4.1). **Accessing Node** is a heuristic that selects the nearest server to a random client that accesses a given KV. **Dominant Node** is an alternative heuristic that places KVs closest to their most frequently accessing client. Both heuristics provide simple yet realistic locality-aware placements based on past accesses. Finally, **Optimal** presents the unachievable lower bound described in §3.4.2 that uses perfect knowledge of future client-server latencies and data accesses.

As shown in Figure 3.9, Portkey delivers 21-82% and 16-45% lower average request latencies than the Random and locality-aware heuristics; Portkey's tail latency improvements are 26-77% and 4-75%. Perhaps more importantly, despite lacking oracle-like knowledge of future latencies and KV accesses, the resulting performance with Portkey's placements are always within 15% for average latency, and almost always within 4% for tail latencies. The one exception for the latter is the per-client workload. The reason is that suboptimalities with Portkey stem from delays in learning workload/latency information and shifting KVs to their ideal servers (the optimal strategy knows ideal placements a priori). This is more pronounced in the per-client workload since the latency discrepancy between the ideal (i.e., local) server and all other servers is large, crossing a tier in our testbed. In contrast, for the regional workload, although there is still a single optimal server, the latency discrepancy with several other servers (in the same region) is low.

Of course, performance with each approach varies based on workload characteristics. For example, when considering average request latency, in the *per-client* workload, all locality-aware placements perform substantially (70-81%) better than the random assignment strategy. On the other hand, the performance discrepancies between the locality-aware heuristics and Portkey was most noticeable in the *group* and *global* workloads. In those cases, the Portkey solver was able to more intelligently balance placements across the large number of accessing clients, resulting in a 2.1-2.6x relative performance improvement.

3.7.2 Varying Edge Settings

In addition to the workload characteristics considered in §3.7.1, Portkey's performance is affected by datastore server density and client-server network latencies. Here, we present results showing Portkey's performance as these properties vary.

Impact of Datastore Server Density. Figure 3.10 presents the performance impact of increasing the fraction of edge APs that serve as candidate datastore hosts. For instance, 20% corresponds to 5 of the 25 APs supporting a Redis instance. As shown, Portkey's speedups grow as the density of datastore servers increases. For instance, peak speedups grow from $2.3\times$ to $5.2\times$ when the server density jumped from 40% to 100%. The reason is that a higher server density enables regional locality to be exploited: KVs can more often be placed such that requests are commonly served within the local region. Accordingly, in a deployment where limited resources are available, spreading out the datastore instances will maximize the regional coverage and available locality.

Varying Network Latency. As described in §3.3.2, our testbed follows a three-tier approach to setting client-server latencies. More specifically, *local*, *regional*, and *other* servers are randomly assigned client-server latencies between 5-10ms, 20-30ms, and 40-50ms, respectively. To understand how Portkey performs under different network settings, we considered

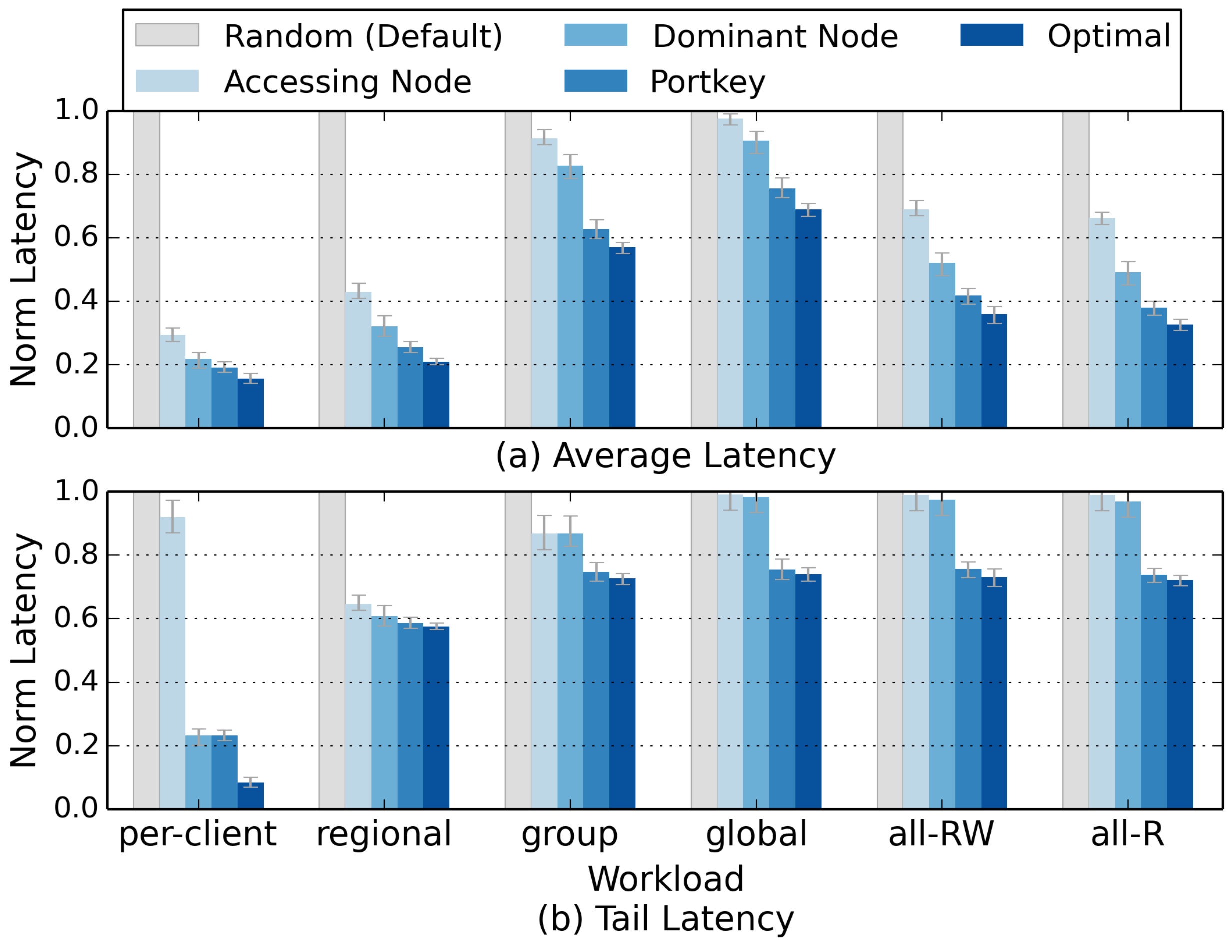


Figure 3.9: Portkey's average and tail (95th percentile) latency speedups over existing random placement strategies and locality-aware heuristics. Results are normalized to the random approach. The median and entire range for five runs of each workload and placement strategy are plotted.

four variants for latency assignment: **fast** follows the strategy from §3.3.2, **slow** increases the minimum latency in each tier by 5× while keeping the width the same, **spread** increases the maximum value in each range by 5× while keeping the minumum values the same, most akin to expected wireless latencies from peer-to-peer network routing [AS18], and **collapse** reduces all client-server latencies across all tiers to the same value (i.e., the latency to all servers is the same), as might be observed in a wide-area cellular network setup

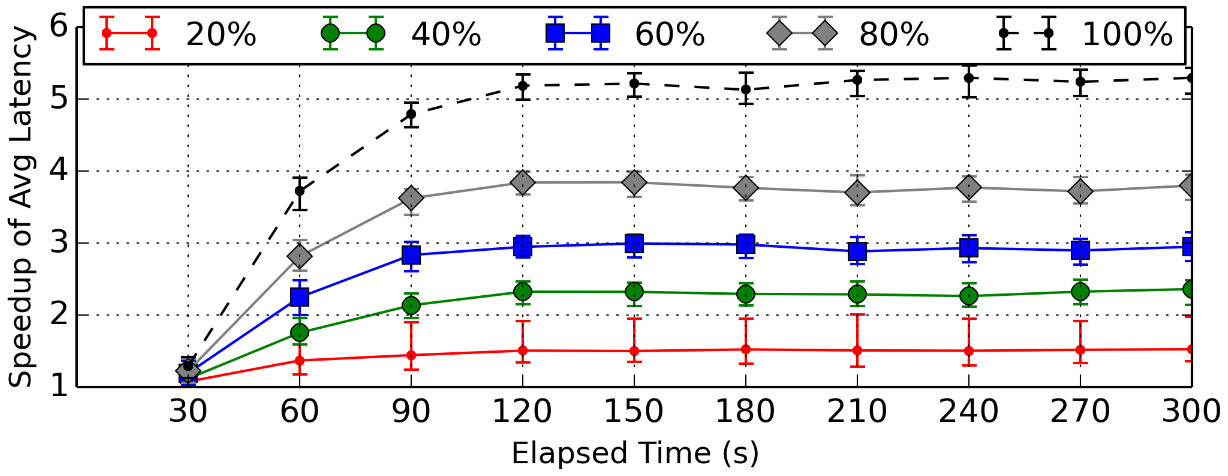


Figure 3.10: Performance impact of varying the percentage of our testbed's 25 edge APs that can serve as datastore servers. Results use the *per-client* workload, and points represent medians with error bars covering the spread across five runs. APs were randomly selected before each run. Portkey's speedups grow as datastore server density grows.

with a very high-speed wired backhaul. As shown in Figure 3.11, altered latency values do not significantly affect the speedups that Portkey delivers. Instead, the key determinant to Portkey's wins is the existence of latency discrepancies between servers, which in turn lead to speedups when KVs migrate close to their accessors. Consequently, Portkey offers little advantage in the *collapse* scenario; placement becomes unimportant as client-server latencies are all uniform.

3.7.3 Profiling Portkey

Convergence. Figure 3.12 presents Portkey's performance over time when optimizing for tail latency. Results are windowed and averaged over thirty second intervals. These temporal results indicate the pattern in Portkey's approach to extracting client workload information. As shown, the first iterations of the adaptive placement engine result in the largest number of migrations. Over time, the datastore is able to converge to an asymptotic performance baseline once sufficient data about client workload patterns and network state has been

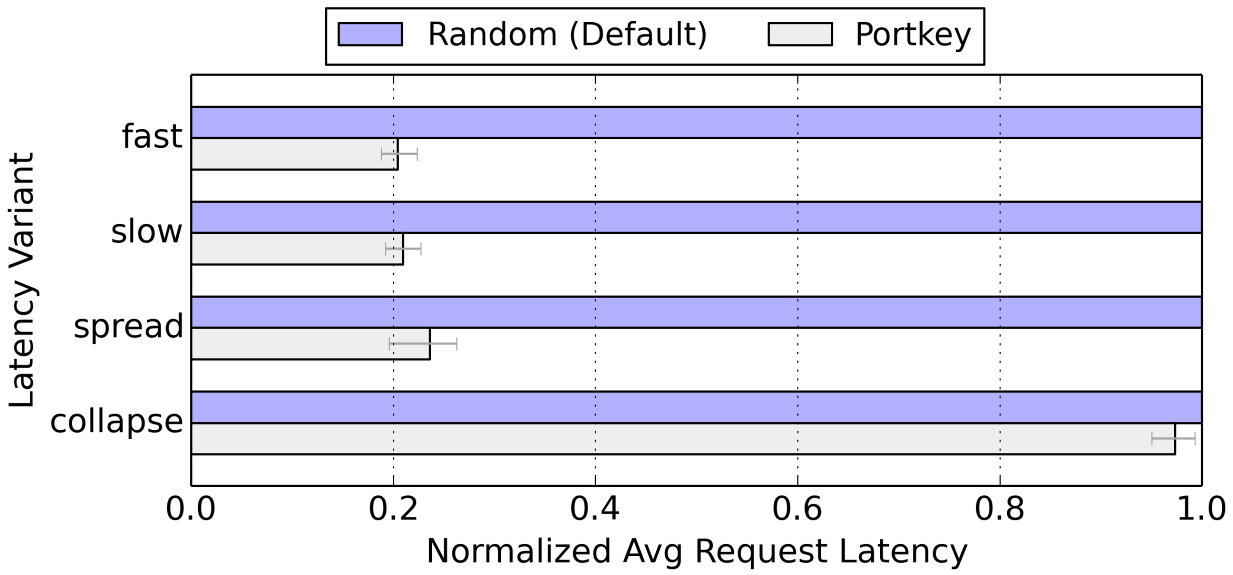


Figure 3.11: Impact of edge network latencies on Portkey's performance; §3.7.2 defines the four scenarios. Results use the *per-client* workload and are normalized to random placement. Portkey's performance is largely unaffected by latency values, other than when all client-server latencies are equivalent (eliminating the importance of placements).

inferred. In other words, after an initial warm up period of approximately 1-2 minutes, performance remained relatively stable and varied primarily in response to continual client mobility.

Placement Solver Scalability. An important aspect of the Portkey placement solver is its ability to scale up to large workloads and deliver fast decisions. Figure 3.13 presents the scalability of a solver instance computing placement of (a) 1024 keys over a varying number of servers, and for (b) a varying number of keys over a 1024-server cluster. Profiling was done on a 2019 Macbook Pro. The full placement solver as described in Section 3.5.2.2 scales linearly with the number of keys, and quadratic to the number of nodes. The heuristic-based approaches scale linearly across both dimensions. Portkey's adaptive solver leverages this notion to selectively fallback to heuristic placement for large cluster and key sizes. For example, with a window size of 10 secs, a single solver operating on a 1,024 node cluster will

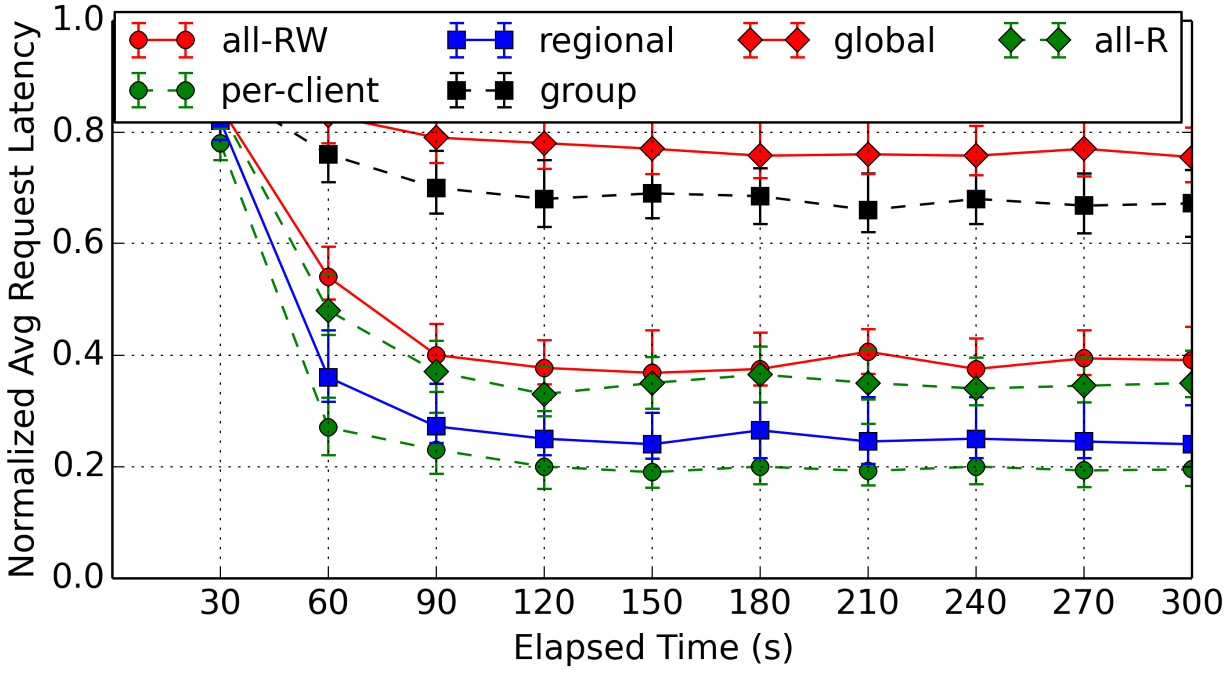


Figure 3.12: Tail latency improvement over time for Portkey. Results are a snapshot of windowed performance over the first five minutes and are normalized to randomized placement. Placements converge after approximately two minutes, when client workload patterns and network perspective have been sufficiently inferred. Further adjustments are mostly in response to client mobility.

perform full cost matrix analysis for up to 65,536 keys (consuming 1 sec) and use heuristic placement for additional keys.

Memory Overhead. The memory overhead for the data access log at each client is determined by the number of KV accesses and servers within the system. The key sketch and network sketch each grow linearly with the cardinality of client-key accesses and servers in the system, respectively. Each key access requires 8 bytes to store the client access count and aggregate payload, while each server consumes 8 bytes to store its access count and aggregate latency. Thus, a datastore spanning thousands of nodes and tens of thousands of keys consume less than 1MB at each client.

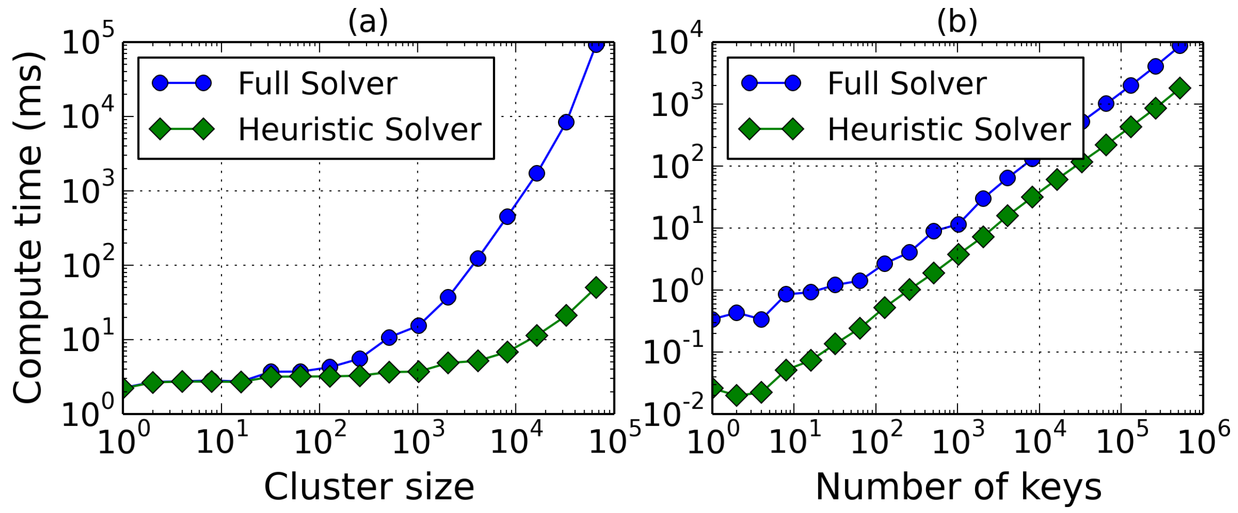


Figure 3.13: Scalability of Portkey's placement solver when varying (a) cluster size and (b) key set size.

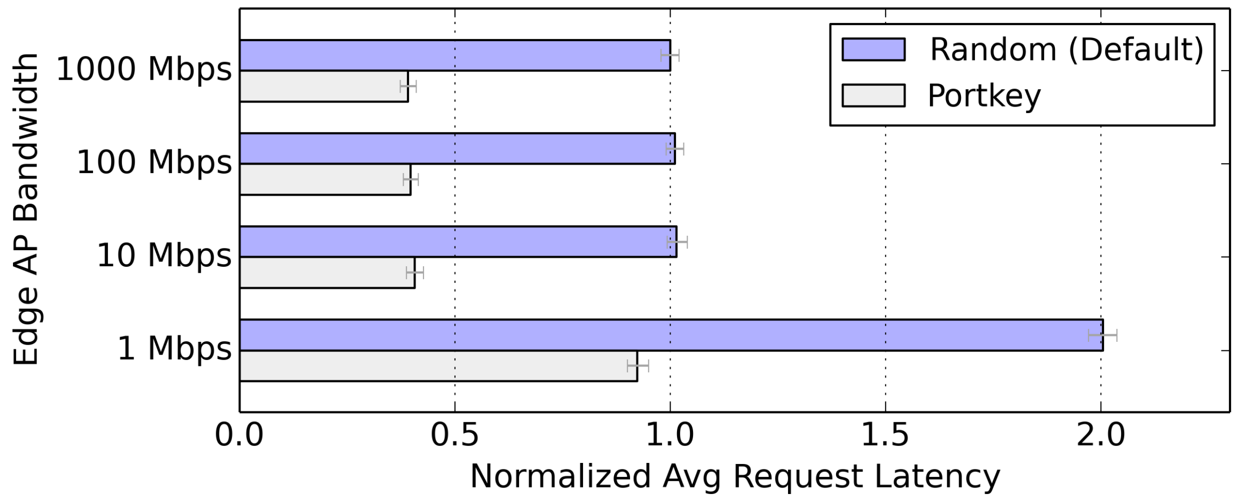


Figure 3.14: Impact of restricting edge bandwidth for the *all-RW* workload. Portkey's relative advantages persist across the considered bandwidths. Bars list medians (normalized to the 1000 Mbps random placement values) with error bars spanning the range of values across 5 runs.

Network Overheads. We profiled the aggregate requests and associated payloads issued by the cluster with and without Portkey for our workloads. Bandwidth overheads ranged from 0.75% to 3.11%, depending on the magnitude of KV migration. To ensure that the added bandwidth requirements do not result in a cost of migration that outweighs the benefits, we ran an experiment that increasingly restricted the amount of AP bandwidth. As shown in Figure 3.14, the most notable impact of restricting bandwidth is the drop from 10 to 1 Mbps, where both Portkey and the default Redis Cluster experience significant slowdown. However, Portkey retains its relative speedup, which consistently falls within 2.1-2.5$\times$ across all considered bandwidths.

3.7.4 Small-Scale Deployments

We deployed two small-scale applications to validate Portkey over real networks and edge devices. The primary objective in these experiments is to highlight how Portkey's benefits in these conditions are similar to those observed in our testbed.

Smart Building Interface. The first application consisted of a Redis Cluster deployment over ten Raspberry Pis (RPi) spread throughout a building and connected over a wireless network, with a single RPi running the Portkey placement engine. Each RPi updates the datastore with (1) frequent updates to the latest reading from attached sensors (e.g., camera image, ambient noise level, wireless network signal strength), and (2) infrequent updates to a key corresponding to device status. An accompanying smartphone application provided user access to view device status and the latest published values. The difference in read-write dominance of each key resulted in varying placements; in particular, the frequently updated sensor value keys migrate to each RPi's local Redis instance, enabling faster system writes and a 5x average latency improvement over randomized placement.

Crowd-Sourced Data Collection. A wide-area Redis Cluster was deployed over nine RPis spread across three campuses, with one RPi designated to host the Portkey engine. User smartphones ran an app that passively updated each user's current GPS coordinates within the datastore to provide crowd-sourced live traffic information. This live activity map could be optionally accessed and viewed on each smartphone. User mobility, including changing

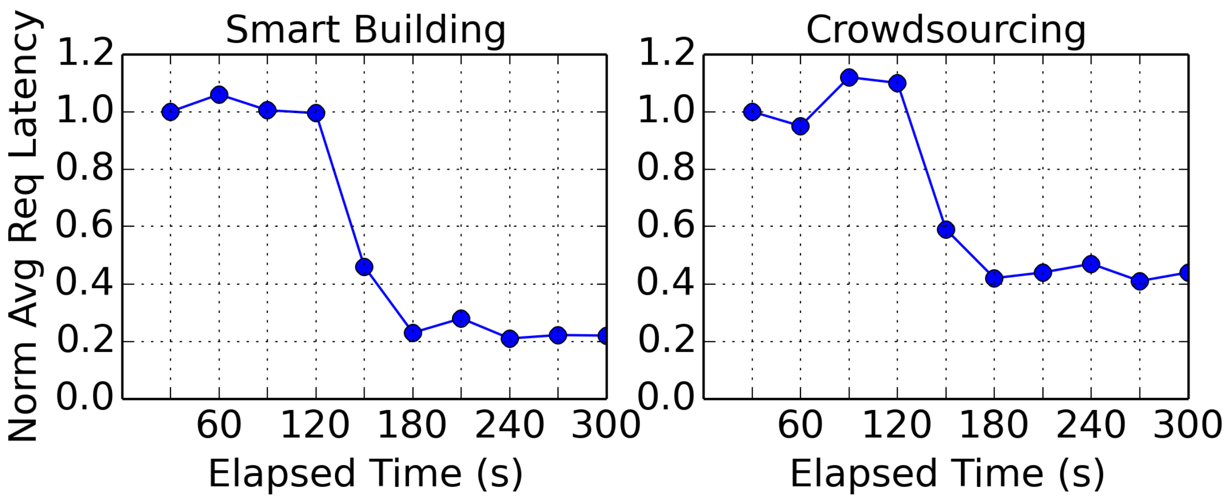


Figure 3.15: Portkey's speedups in two real deployments. Portkey was enabled after 120 secs of random placements.

physical location or network connectivity (Wi-Fi vs cellular), resulted in the continual migration of user data to their nearest host. With Portkey, a user's location data migrated to their nearest RPi within their currently occupied campus, yielding a 2.5x improvement in request latency over randomized placement.

Takeaway. Portkey's request latency speedups for both applications are summarized in Figure 3.15, and closely match those from our emulated testbed. Importantly, without any developer-specific input providing insight into the particular network deployment or workload, Portkey was able to optimize for the most frequent datastore accesses. The designated RPis executing these extensions consumed additional memory footprints of less than 20MB, with less than 2% increase in overall CPU utilization.

3.8 Discussion

Distinction from caching solutions. Caching techniques are often used to bring data closer to the primary producers and consumers,

Caching and secondary replicas for eventually consistent reads. In support of reads that do not require strong consistency, secondary replicas or content caches can improve access latency and/or reduce the load on the primary replica. Such approaches are orthogonal to this work, which focuses on primary replica placement to optimize consistent data accesses to the underlying datastore.

Security risks with client-side logging. Trusting statistics reported by client libraries inherently poses security risks. A compromised client may negatively influence placement by misreporting or falsely injecting unnecessary requests. To this end, Portkey supports an optional configuration that limits the overall impact of a compromised client. When enabled, key access vectors (§3.5.1.2) are scaled to a unit vector, resulting in each client equally contributing to placement decisions (§3.5.2.1), thereby mitigating the effect of nefarious clients. This comes at the expense of a potentially superior placement decision if clients honestly report genuine data accesses.

Migration overhead. Portkey currently does not consider the bandwidth overhead of migration when reassigning KVs. However, this is not fundamental to the placement solver, and can be incorporated by adding a loss component to the utility score computed in §3.5.2.2, e.g., with a minimum threshold that scales with the KV payload size. Further, we note that even with this omission, Portkey's network overheads (including from migration) are consistently low, between 1-3% (§3.7.3).

Spanning geographic regions. Deployments spanning geographic regions pose unique constraints when considering an idealized datastore architecture. Quorum architectures enable load balancing across an individual key at the expense of an increased number of client-side system requests. Alternatively, a primary-secondary architecture reduces the required number of requests for consistent access at the risk of overburdening the primary. This work aims to mitigate the downsides of a primary-secondary architecture by migrating the primary replica such that consistent requests can be made fast. Meanwhile, load balancing is achieved by spreading key placements across alternative datastore hosts.

When is adaptive placement not beneficial? The ability to place data at the best available host is directly related to the frequency by which the network state and workload locality change. In the case of a static network and consistent workload, the system will asymptotically stabilize placement. In contrast, if the network were randomly shuffled immediately after client profiling data is uploaded, system performance would be effectively equivalent to a randomized placement. The same is true for client workloads where previous KV accesses offer no indication to future ones. Ultimately, adaptive placement is valuable only if there is some inherent pattern that can be learned from network state and client accesses.

Tuning the window size. The window size provides a mechanism enabling the datastore administrator or application developer to tune the tradeoff between latency speedups and network overheads based on preferences and perceived notions of dynamism. In particular, larger window sizes imply less responsiveness to changing dynamics, but also lower network overheads from shipping client profiling information. Thus, relatively static deployments require less agility, thereby maintaining high performance with minimal overhead by using large windows. In contrast, rapidly changing networks should configure Portkey for increased agility (i.e., small windows) to more quickly respond to changes.

3.9 Related Work

Improving Data Locality. Previous work seeking to improve the locality of clients accessing a distributed datastore can be broadly categorized into approaches that either (1) intelligently select across a set of existing replicas, or (2) explicitly migrate data across datacenters. C3 [SCS15] enables each client to determine the best subset of replicas to issue datastore requests to. Pando [UHG20] built upon this notion by splitting object data such that the effective spread across datacenters is increased, thereby increasing available locality. Volley [ADJ10] migrates data between datacenters by cross-referencing a reverse-IP look-up of user location derived from application logs with known datacenter locations; their iterative solver then triggers application-specific migration mechanisms at a relatively course granularity across weeks and months. Tuba [AT14] increased the frequency of reconfiguration to hours in order to adjust replication parameters based on which datacenter regions are most active. Akkio [ARS18] groups small shards of KVs based on developer insight into joint client accesses, migrating shards across datacenters depending on which regions are currently serving a client. Physalia [BCP20] uses infrastructure-aware KV placement and migration to maximize availability across availability zones. TripS [OCW17] included the storage tiers offered by cloud providers as an additional migration parameter. More generally, custom hashing methods including locality-sensitive hashing [LJW07] and Social Hash [SKK16] leverage application-specific information to collocate jointly accessed data with clients. Finally, in the context of cellular networks, historical work on call handoff techniques take an analogous approach of maximizing QoS by dynamically adjusting allocated cellular tower bandwidth based on user locality and range [ESK05, TRV98].

In focusing on the cloud computing setting, placements across individual hosts within a cluster have been left relatively ignored. In contrast, Portkey focuses on retargeting these datastores to edge contexts and adapting to the inherent mobility and latency variations.

Edge Datastores. Recent works have explored datastore platforms that are customized to the inherent variability and distribution of edge networks. FogStore [GR18] sought to provide a datastore spanning the near-edge by incorporating the notion of a context-of-interest, defined by a system administrator or domain expert, that dictates regional constraints on a replica quorum placement. All data and clients are then associated with the notion of a location; data accesses performed within the context-of-interest receive strong consistency, while those outside resort to eventually consistent access. Nebula [ROC14] introduced an HDFS-like storage system spanning the edge-cloud, with large immutable files spread across participating storage devices. PathStore [MBL18] proposed an edge datastore that uses a write-back cache hierarchy with eventual propagation to enforce session consistency. Application-specific KV systems, such as those for computer vision, have also been proposed [RG18].

Each of these datastore platforms sacrifice certain flexibility in application support. They require prior knowledge of application or network characteristics, either by explicitly associating a location with data, sacrificing consistency, or focusing on domain-specific workload patterns. In contrast, Portkey offers a generalized solution to improve the locality of clients accessing a datastore. The objective is to customize a given deployment to specific and variable client accesses while offering the equivalent consistency guarantees of the underlying system, all without necessitating developer input. To the best of our knowledge, none of the previously proposed systems are openly available for use; our Redis Cluster extensions serve to provide a tangible implementation.

3.10 Conclusions

This research presents Portkey, the first distributed KV store that explicitly targets the intrinsic mobility and time-varying client-server latency profiles experienced in edge applications. Unlike prior datastores that opt for randomized data placement policies, Portkey dynamically adapts data placements according to periodically-profiled latencies and data access patterns. Key to Portkey is its treatment of mobility as a first-class primitive, and its prioritization of rapid (but approximate) placement decisions over slow optimal ones. These insights enable efficient profiling strategies that adhere to edge device and network constraints, as well as greedy placement heuristics that are self-correcting over short timescales. Results with an autonomous vehicle dataset, as well as two small-scale application deployments, show that Portkey reduces average and tail request latencies by 21-82% and 26-77% compared to existing placement strategies.

CHAPTER 4

EdgeRM: Practical Cluster Computing for Modern Edge Devices

Generality, interactivity, and multi-usability defined the rise of modern computing. The success of early operating systems like Multics and Unix relied on these ideals—generality enabled early adopters to discover the full set of capabilities of this nascent domain; interactivity fostered iteration and sped up this discovery; multi-usability was required to efficiently use the expensive and difficult-to-deploy physical resources. While these had long been goals in early computing, they were not broadly realized until system designers had access to sufficient memory, storage, and networking, which, for Unix was represented by the PDP-11/45 with 144 KB of memory and 1 MB of fixed disk.

The low-power processors which drive the sensors and actuators at the edge of today's IoT do not yet live up to the ideals of early operating systems. In enterprise and research IoT programming systems alike these processors are considered fixed-function devices, data-forwarders to more general and interactive Linux-class machines. They are so ignored that despite quite literally being at the edge of the network, the literature has instead come to explicitly define the nearby cloudlets and gateways as the IoT's "edge." But these processors are now being released with 128-256 KB of memory, 1 MB of flash, and off-the-shelf IP-based networking. They should be capable of achieving those early ideals and realizing a more complete role as the IoT's edge, and we claim that now is the time for this ascension.

IoT ascension would have marked and lasting impacts on our ability to collect, process, and apply data from the physical world. Due to the low cost of computation relative to communication, enabling general computation on these devices would allow us to write, and, more importantly, discover applications that would otherwise not be feasible within the device's resource constraints, such as our Sensor MapReduce framework (§4.4.1.3). Interactivity would enable a new set of programmers to access devices that are currently reserved for embedded experts. Multi-usability would allow us to share infrastructure, amortizing the difficulty of deploying devices in the physical world. Furthermore, it would help unify the fragmented approach of deploying every IoT system with its own vertical stack, custom gateway, and cloud service.

This paper, therefore, proposes EdgeRM, a novel resource manager which extends the general purpose computing cluster to include the resource constrained devices that comprise the true edge of the IoT. We are not the first to propose the use of cloud architectures to program Linux-class IoT devices, nor are we the first to propose the interactive programming of resource-constrained IoT devices en masse (macroprogramming), however we are the first to propose a unification of these two visions. Recent increases in the memory available on the low-power processors and developments in memory efficient and safe virtual machine technologies such as Web Assembley (WASM) now allow us to create a resource manager which can extend the reach of existing edge programming frameworks and facilitate the operation of multiple macroprogramming frameworks simultaneously.

EdgeRM is architecturally similar to Apache Mesos, aggregating resources from agent nodes and offering them to multiple programming frameworks, however it must aggregate new *sensor and actuator resources* while embracing heterogeneous and mobile devices under significant resource constraints. To aggregate sensor and actuator resources in addition to compute resources, we create a new resource type which exposes not just the amount of a resource, but also the supported APIs by which that resource can be accessed. To support heterogeneity and mobility, we propose the use of an *Active Scheduler* to profile the capabilities of compute and networking resources while monitoring device location and other relevant context. To handle resource constraints, we create a non-POSIX implementation of the resource manager's agent, adopt lightweight networking protocols and communication patterns, and support WASM tasks in addition to Docker containers.

We implement EdgeRM so that it's central resource manager runs on a server-class machine, but agent nodes can range from servers down to embedded platforms. To demonstrate the capabilities of EdgeRM, we create a Sensor MapReduce framework for collecting and processing sensor data, we adapt an existing edge computing framework, Edge Dataflow, and we implement a generic sensor sampling framework to allows users to specify sensor sampling rates, filters, and the destination of the sensor data. Additionally, because we expect the creation of new programming frameworks to be frequent, we provide a library to facilitate their creation.

We evaluate EdgeRM by measuring its computational, memory, and energy overhead for both embedded platforms and Linux-class edge devices. We find for Linux-class machines that utilization and memory are small, even for sub-second task distribution latency. On energy constrained embedded platforms, EdgeRM uses less than 0.1 % of CPU time and 34 μW of power for a 100 s task update latency, even less power for longer update times, and can operate with only 128 KB of RAM. While this overhead does not yield individual tasks that are as efficient as those implemented by a skilled embedded programmer, we believe this is a small price to pay for the features necessary to make this class of computing accessible to a broader audience.

In summary, this work:

1. Introduces EdgeRM, the first resource manager capable of managing resources for, and executing tasks across, both Linux-class and embedded-class devices.

2. Proposes active scheduling to more efficiently schedule tasks among the quickly changing context and heterogeneous device types present at the edge.

3. Demonstrates several application frameworks, including a Sensor MapReduce framework, which use EdgeRM and its scheduling libraries to interactively collect and process

data from a deployment of sensor agents.

4. Provides EdgeRM, along with its scheduling and application frameworks, as open source artifacts which can be used and expanded upon by the community.

4.1 Motivation

The EdgeRM motivation breaks down into three high level arguments: (1) executing code near the data source yields improvements in efficiency, reliability, and privacy, (2) to truly take advantage of executing code near the edge this code must be unconstrained and general, and (3) the ability to share access to sensors will increase the value they yield relative to the difficulty of deploying the devices.

4.1.1 The Edge Needs More Compute

The advantages of in-network compute have been well recognized in prior work [BHS07, MFH05, HKS05]. Due to the relatively high cost of networking compared to compute, placing computation closer to the data source can lower latency, total energy, cost, and enable applications that would otherwise not be feasible [LZH17]. Running applications locally limits the number of wide-area network links, potentially improving application reliability, and computing near the data source can prevent private data from needing to be stored and analyzed on remote servers. Technology trends and the rise of smart objects in the home are amplifying these drivers for in-network compute rather than alleviating them.

Specifically, Figure 4.1 indicates that microcontrollers' ability to perform local computation is continuing to out-pace their ability transmit that data to the cloud. We see similar trends in sensing, leading to data that can be more efficiently collected but not transmitted without encountering bandwidth or energy constraints. Finally, as applications are more broadly deployed, they are going to gain access to more private information. These prob-

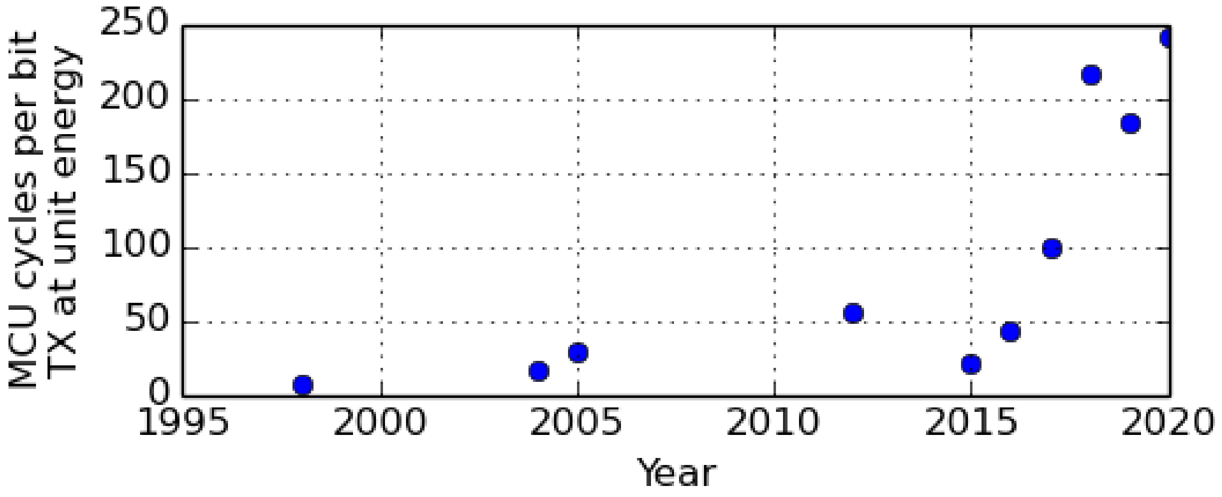


Figure 4.1: Microprocessor advancements outpace radio technology. Each year the most energy-efficient commercially available MCU is compared to the commercial radio with the lowest energy per bit transmitted at 10 m. Since the start of the wireless embedded systems field in the 90's, the number of cycles per bit transmitted have increased, and currently appears to be trending exponentially.

lems could be addressed, with the opportunity of lowering processor power, by pushing more compute toward the edges of our sensing networks.

4.1.2 Diversity Calls for Flexibility

To truly realize the benefits of computing at the edge, abstractions must be general enough to efficiently explore new application domains, thereby easing the transition of applications from exploratory data analysis to long-running processing workloads. Evidence of this point can be found in the fate of sensor programming models that have been proposed in the past. Systems like TinyDB enabled the querying of sensor data through an interactive SQL query interface, and Regiment facilitated in-network stream aggregation, however these systems have not gained traction as de facto methods for programming embedded sensor deployments [MFH05, NMW07].

We hypothesize that this is because no single programming framework is general enough to meet all the needs of a sensor deployment. Even a single programmer may want to access their sensors with multiple programming models throughout the lifetime of a deployment, and in the future they may access the same sensors with new programming frameworks. We need a layer of abstraction flexible enough to facilitate all existing and future programming models.

4.1.3 Infrastructure Should Be Shared

One key advantage of resource-constrained sensor networks is their ability to to collect data from previously uninstrumented locations, however deploying sensors in these locations is often physically difficult [DL20]. Many of the most compelling use cases for these sensors, especially in densely populated environments, call for their data to be used by multiple parties for disparate applications [WRB18]. The need for data sharing is so common that it is often achieved by deploying sensors which collect and stream the data believed to be the common denominator for applications to the cloud through a publish/subscribe system where it can then be shared in an iterative and interactive manner [AMP16, RBB11, AKC17].

Unfortunately, IoT vendors often package devices, gateways, and cloud services into isolated silos that inhibit resource sharing [Ama20, FJP16]. These systems are in direct contrast with in-network and on-sensor compute, often resulting in sensors which send more data than necessary for most applications and an insufficient amount for some applications. To achieve the efficiency promised by pushing compute to the edge while still enabling simultaneous access to sensor data by multiple consumers, we need a system to facilitate the multi-tenant management of multiple programs.

4.2 Background & Related Work

There exists significant prior work in scheduling and resource management for server class machines and no shortage of prior work which applies these techniques to the single board cloudlets and gateways that are commonly present in edge computing deployments. Separately, a body of work explores improved programming models for resource constrained sensors. EdgeRM sits firmly at the intersection of these domains, adopting key ideas from prior work to enable the programmability of the cloud extended to clusters of resource-constrained sensors. We explore each of these domains, noting our inspiration and where the EdgeRM architecture enables that which was previously untenable.

4.2.1 Better Sensor Programming

Since the rise of networked embedded systems the field has been plagued by the difficulty of programming these devices. Their resource constraints demand highly efficient programs, their hardware heterogeneity prevents standardization, and, because code is often deployed as a single binary, updates are high-risk operations, slowing iterative development.

4.2.1.1 Incremental configuration and updates

The sensor networks community began by introducing methods by which fixed-function programs can be dynamically reconfigured without performing a full code update, such as value dissemination in TinyOS [LMP05]. Then to improve generality works such as SoS dynamically loaded more expressive code modules [HKS05]. Most recently Tock enabled generality and isolated multi-tenancy on embedded processors by taking advantage of recently-introduced hardware memory protection capabilities to allow multiple applications to be executed on top of a shared kernel [LCG17]. Tock lowers the barrier to iterative single-board embedded development, but does not on its own solve the cluster programming problem.

4.2.1.2 Virtual Machines and Runtimes for Embedded Devices

Starting with Maté [LC02], and iterated upon by Impala and SensorWare [BHS07, LM03], sensor-node-specific virtual machine runtimes improved portability, enabled isolation between applications, and reduced code update size compared to full binaries. Later more common languages like Java, Javascript, and Python had stripped down to run on sensor nodes [The16, Win20, BCL08, Geo20]. These options lacked the tooling and standardization required to make them easily usable. Recently, Web Assembly (WASM) has gained traction as an instruction format to safely and efficiently execute code in the browser and the Web Assembly Micro Runtime (WAMR) has successfully enabled its execution on embedded nodes [HRS17a, Byt20a]. The existence of a portable, safe, efficient, and broadly-supported execution format is crucial to the success of a resource manager, and EdgeRM adopts WASM as an execution environment.

4.2.1.3 Macroprogramming

Early macroprogramming systems began to enable the programming of sensors en masse through unified programming models like SQL queries or domain specific languages for stream aggregation [MFH05, NMW07, GGG05, KWA03, BHS07]. Later Ravel provided a system for programming cloud, gateway, and sensor devices through a unified programming model [RHL15]. However, the lack of an underlying resource manager meant that all of these solutions tied sensors into a specific programming model, which is not general enough to meet diverse applications. We hope the the existence of EdgeRM will enable multiple of these programming models to run simultaneously and ultimately see their resurgence.

4.2.2 Resource Managers

Resource managers are standard components for hosting multiple applications or programming frameworks over clusters of server-class machines. Most programming frameworks have paired resource managers like Hadoop's YARN or Kuberenetes' combined scheduler/resource manager [VMD13, BGO16]. Some resource managers, like Apache Mesos, support multiple frameworks each with their own scheduler [HKZ11].

We long-considered extending an existing resource manager to support resource-constrained devices. Unfortunately, no existing solution can operate in a non-POSIX, limited-memory environment, and the networking protocols used to communicate from devices to the central manager were too complex to be ported. While it may be possible to create a resource-proxy which acts on the sensor nodes' behalf to register nodes and subsequently distribute tasks from that resource manager to the node, the changes in necessary executors and resource types to support general computing on the edge devices themselves further motivated EdgeRM.

4.2.2.1 Non-standard Device Management in the Cloud

While the architecture of EdgeRM is similar to Apache Mesos, the one key difference is that EdgeRM intends to support aggregation of sensor and actuator resources. In recent years, the aggregation of device resources has become more necessary in the cloud as well, with the potential for nodes that have local FPGAs or GPUs that could accelerate a computing task. Kuberenetes supports a device plugin which allows for each node to register devices by vendor and type with the kuberenetes node manager (kubelet) [Kub], and Mesos's resource types theoretically allow the registration of any resource, and only limits the types of resources by convention. We find both of these to be insufficient to handle the heterogeneity of devices and drivers that may be found on sensor nodes, and EdgeRM implements a specific device resource type to better capture this heterogeneity.

4.2.3 Edge Computing Frameworks

Many projects have aggregated distributed resources near the edge of the network to put their computational power to use. Volunteer Edge Computing systems addressed challenges in resource monitoring, management, and task scheduling over wide-area clusters with limited availability [TTL05, ROC14]. As mobile devices rose in ubiquity, so did efforts in designing localized clustering and efficient task offloading specific to smartphones [HAH15, SLA12, CBC10]. Application specific approaches have applied similar ideas to complex event processing, neural network acceleration, federated machine learning, and industry-specific (e.g. medical cyber-physical systems) deployments [RJK05, TMK17, GZG15, YLC19].

More recently, fog and edge computing frameworks have further fueled the desire to bring compute closer to the edge of the network. Numerous architectures, domain-specific languages, and frameworks have been built, many of which are inspired by or extensions of cloud programming frameworks, resource management, and virtualization abstractions [HV19, WVM17, LWB16, BZ17, MCD18, GND17, Hyp20]. Modern enterprise solutions take similar

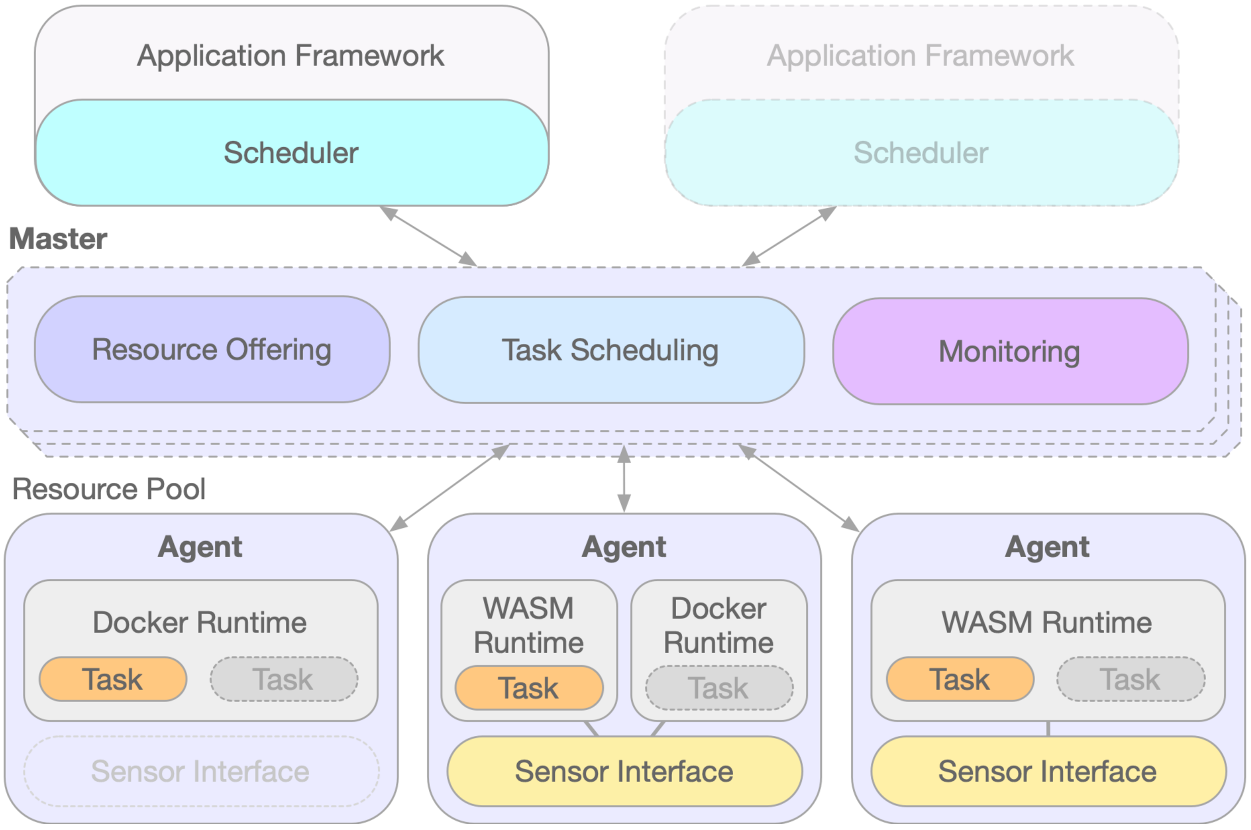
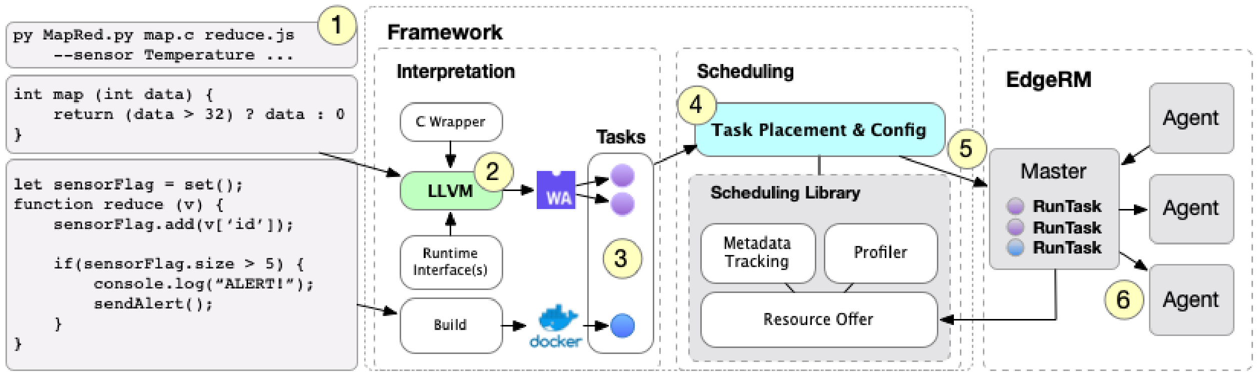


Figure 4.2: EdgeRM Architecture. Inspired by Apache Mesos, EdgeRM agents send available resources to a master to be offered to multiple frameworks with their own independent schedulers. Key differences include support for small WASM runtimes with communication protocols designed for resource-constrained agents with attached sensors.

approaches, extending their cloud container orchestration and function-as-a-service offerings to local linux-class gateways and cloudlets [Ama20, Azu20].

All of these systems consider the sensor and on-board microcontroller as a data-forwarder. The true "edge" has been left relatively ignored, mostly due to prior technical limitations in appropriately leveraging these devices within a resource cluster.



```
py MapRed.py map.c reduce.js
    --sensor Temperature ...
```

```
int map (int data) {
    return (data > 32) ? data : 0
}
```

```
let sensorFlag = set();
function reduce (v) {
    sensorFlag.add(v['id']);

    if(sensorFlag.size > 5) {
        console.log("ALERT!");
        sendAlert();
    }
}
```

Figure 4.3: A step-by-step workflow of using EdgeRM through the Sensor MapReduce framework (§4.4.1.3). (1) A user submits map and reduce jobs to the application framework; (2) The framework's interpreter wraps user code in boilerplate communication code and compiles it into WASM modules and docker containers. (3) These tasks are sent to framework's scheduler, (4) which uses the EdgeRM scheduling library to fetch available resources, plan task placement, and configure tasks (i.e. with source and destination addresses). Active scheduling techniques such as agent profiling are used to assist placement (§4.3.5). (5) Tasks are issued to the EdgeRM Master, (6) and forwarded to EdgeRM Agents to execute.

4.3 Design

EdgeRM is heavily inspired by the architecture of Mesos, however the details of communication patterns and resource types are modified to fit into a new application domain.

The high-level architecture is presented in Figure 4.2. The system is orchestrated through an EdgeRM *master*, the central entity responsible for resource aggregation, resource offering, task distribution, and system monitoring. Each device in the computing cluster advertises its available resources through an EdgeRM *agent* process. The agent is responsible for interfacing with the master to expose and maintain an updated list of available resources, as well as accept and launch tasks. Docker containers and WebAssembly tasks are both supported, with each of these runtimes incorporating the ability to access on-board sensor resources. Application *frameworks* rely on a scheduling library to request resource offers from the master and issue task requests.

4.3.1 End-to-End Workflow

Figure 4.3 presents a high-level overview of the Sensor MapReduce framework deploying a job (§4.4.1.3) in EdgeRM. Because EdgeRM supports multiple frameworks, the exactly details of each framework may differ, but Figure 4.3 can provide one example of how one framework may function.

A user begins by writing basic `map` and `reduce` functions and issuing the MapReduce job via command line. The job is accepted by the MapReduce framework, which independently constructs corresponding map and reduce tasks. The map function is wrapped in boilerplate communication code and the WASI-SN sensor interface (§4.3.4) and compiled into a WebAssembly module via clang/LLVM. The reduce function is built into a Docker reduce container. These tasks are then sent to the MapReduce scheduler. The scheduler leverages the EdgeRM *Active Scheduling* library (§4.3.5) to request a resource offer (§4.3.3) that meets both the necessary resources of the tasks, and user defined parameters (such as the sensor being sampled) (§4.3.2). The library fetches and provides relevant profiling statistics and other pertinent metadata including location information to enable the framework's scheduler to make intelligent placement decisions (§4.3.6). Tasks are then issued to the cluster through the EdgeRM master, which forwards those tasks sensor and server agents. The MapReduce framework continues to monitor the cluster (§4.3.5.2) to schedule new map and/or reduce jobs onto newly entering devices. Results are presented to the user's console in real-time, with support for immediate job termination and re-issuance.

4.3.2 Definition of a Resource

In addition to providing an isolated execution environment, a resource manager is primarily used to facilitate scheduling decisions by one or more schedulers. In the cloud, the information used to make scheduling decisions is the same information tracked by the resource manager. For example, scheduling decisions are primarily made based on a machine's sufficiency in CPU, memory, and data locality, all of which is inherently tracked by the resource manager.

As we extend to sensors placed throughout a physical environment, this no longer holds true. While schedulers still need to make decisions based on CPU and memory sufficiency, they also need to make scheduling decisions based on location, network topologies, and events that occur in the physical world. This begs the question—is this type of meta information a resource, and is it the resource manager's role to aggregate this information for a scheduler?

We take the stance that such metadata should not be handled by the resource manager because collecting this information often itself takes resources, and we cannot know a priori which types of metadata a given scheduler may need or the rate at which that metadata should be collected.

Therefore, EdgeRM supports: (1) Resources, which are physical devices that are isolated by and accessed through the resource manager's execution environment, and (2) Attributes,

RPi with Camera	Server	Embedded Sensor
Resources:	**Resources:**	**Resources:**
Scalar: CPU - 1.0	Scalar: CPU - 4.0	Scalar: CPU - 1.0
Scalar: Memory - 4 GB	Scalar: Memory - 8 GB	Scalar: Memory - 50 KB
Scalar: Disk - 8 GB	Scalar: Disk - 100 GB	Scalar: Disk - 100 KB
Range: Ports-3000-3005	Range: Ports-3000-4000	Scalar: Power - 1 mW
Device: Camera /dev/vchi		Device: Humidity - hum
shareable raspistill		shareable - WASIv1
		Device: Pressure - press
		shareable - WASIv1
Attributes:	**Attributes:**	**Attributes:**
Text: OS - debian-armv7l	Text: OS - debian-amd64	Text: OS - zephyr
Set: Executors - [Docker]	Set: Executors - [Docker]	Set: Executors - [WASM]
Text: ID - picam01	Text: Domain: pub.com	Text: ID - sensor01
	Text: ID - server1	

Table 4.1: Resources and attributes in an EdgeRM deployment. All devices list common resource types such as CPU and memory, however resources such as devices, domain names, and the available power are unique to a wide area sensor deployment. Device resource types have properties such as shareability among tasks and the API through which a device is accessed to facilitate their management.

which are static device properties that do not change through the deployment of a device. While these are the same categories used by Mesos, their definitions require clarification in the edge computing environment.

4.3.2.1 System resource types

EdgeRM initially used Mesos' resource types: scalar, text, range, and set, and we find these to be sufficient for both traditional resource types like CPU, memory, and networking ports, as well as previously unconsidered resource types that are important for embedded systems, such as energy consumption. These resource types, however, assume that a task understands the methods by which these resource may be accessed, and also assume that these resources must be strictly isolated between tasks. Neither of these assumptions hold for sensor devices which must be accessed through EdgeRM.

To facilitate sensor devices, we introduce a new resource type "device." Similar to other resource types, device types have a name, but they also have a handle by which they can be uniquely referenced on the sensor, a flag to indicate whether or not they can be shared, and a field to indicate an API by which the device can be accessed. This was necessary to handle device heterogeneity and enable schedulers to ship tasks that align with the supported sensor access API.

4.3.2.2 Attributes

Attributes are system constants which do not change throughout a device deployment. We commonly implement attributes specifying a unique device ID, the architecture and OS of a device, the executors supported by the device, and the public IP or domain name of a device if one is present. These attributes are critical to making scheduling decisions. For instance, many tasks can only be scheduled on machines that are publicly reachable because they need to collect sensor readings from a wide range of devices behind NATs. Example platforms and their EdgeRM resources are shown in Table 4.1.

4.3.3 EdgeRM Messaging Protocol

Resources and attributes are coordinated through the EdgeRM messaging protocol. Table 4.2 provides an overview of the messaging API. Unlike other resource managers that establish and maintain bi-directional communication between entities, EdgeRM opts for client-server communication model that is initiated by the agent or framework. The reasons for this are two-fold: first, the overhead involved in establishing and maintaining long-lived persistent connections on resource-constrained agents is impractical. Second, these clusters are often comprised of devices spanning wide-area networks and devices behind NATs. As a result, only the master is assumed to expose a public network address; thus the master serves as

Message Type	From → To	Fields	Actions Taken
Ping	Agent → Master	AgentID*, PingRate*, Resources, Attributes, TaskStatus	Register Agent, Update Agent State
Pong	Master → Agent	Ack*, TaskInfo	Run or Kill Task if Requested
RequestOffers	Framework → Master	—	Collect Offers
ResourceOffers	Master → Framework	OfferID*, Array{AgentID*, Resources*}	—
RunTask	Framework → Master	OfferID*, AgentID*, TaskInfo*	Queue Task for Agent
TaskStatus	Master → Framework	Ack*, TaskStatus*	—
SubMessage Types	—	Fields	—
Resources	—	Array{ResourceName*,ResourceType*,ResourceValue*}	—
Attributes	—	Array{AttributeName*,AttributeType*,AttributeValue*}	—
TaskInfo	—	TaskID*, TaskContainer*, TaskEnvironment, TaskResources*	—
TaskStatus	—	Array{TaskID*, TaskState*, ErrorMessage}	—

Table 4.2: EdgeRM messaging protocol. An overview of the messages between different components in EdgeRM and their fields, with submessages separated for clarity. Required fields are marked with *. All messages are client-initiated, where the Agent and Framework act as clients, and the master is the server. The master then responds, piggybacking information onto the response. This allows agents to control their energy usage at the cost of higher latency for task execution, and it allows for agents and frameworks to communicate with the Master from behind a NAT. Many fields are left optional so that agents can further limit communication to strictly what is necessary to keep their resources and task states up to date. Currently COAP is used as the communication protocol, however any client-server protocol could be used.

the centralized endpoint connecting the cluster components. Agents that are also publicly accessible can include their endpoint information as an attribute within the resource manager.

Resource Aggregation. Each agent connects to the cluster via a *Ping* issued to the master. The ping contains agent details, resource information, device attributes, current availability, and task statuses. The master aggregates the information from agents within the cluster to maintain an updated view of the resource pool. If an agent does not ping the master within its specified window the master does not consider its resources available, but the master imposes no requirements on the rate at which an agent pings to accommodate resource-constrained devices.

Resource Offering. Frameworks seeking to deploy tasks over the resource pool issue a *RequestOffers* message to the master. The master replies with *ResourceOffers* containing a subset of the current available resources based on the resource offering policy, which is configurable by the system administrator. Each offer is associated with an expiration, at which point unclaimed resources are made available for subsequent framework offer requests.

Task Scheduling. A framework can claim an offered resource by issuing one or more *RunTask* messages specifying a Docker or WebAssembly task to deploy on resources contained within the offer. Included in this message are configurations and environment variables necessary to launch the task on the specified agent. The master forwards the task request to the chosen agents on their next ping by attaching a RunTask message to the pong response. Frameworks can monitor task status via a ping request that collects *TaskStatus* updates, and can issue requests to kill running tasks.

4.3.3.1 Fault Tolerance

A fault tolerant master is critical to system reliability, as both frameworks and agents rely on the master coordinate resources offering and task execution. We designed the master to be able to reconstruct its complete state from the pings received by agents and framework schedulers. As such, recovery from a failed master simply requires that connected frameworks and agents redirect requests to a backup or standby master. Fault tolerance is achieved by running backup masters in standby mode. A standby master registered with the current master is added to a configuration disseminated to frameworks and agents with a total master ordering. Upon failure, the first standby master is promoted. Requests issued to any standby master are redirected to the current master.

Any agent or task failures are reported to framework schedulers on subsequent ping, allowing for framework-specific handling of these failures.

4.3.4 WebAssembly Execution Environment

Interface	Parameters	Return Value	Functionality
getSensors	*None*	String array	Fetch list of available sensor IDs
turnOn	Sensor ID	Boolean	Initialize the sensor if needed
turnOff	Sensor ID	Boolean	Deinitialize the sensor
config	Sensor ID, configuration, value	Boolean	Update a sensor's configuration (e.g. sampling rate)
read	Sensor ID, (capability — configuration), buffer, length	Boolean	Update buffer with current sensor or configuration value

Table 4.3: WebAssembly Sensor Interface. API primitives offered by the WebAssembly System Interface extension in support of generic and portable sensor access to deployed WebAssembly tasks on the WAMR runtime. The runtime mediates sensor accesses to the underlying platform SDK to ensure valid accesses from sandboxed WebAssembly tasks.

A key requirement of EdgeRM is a general execution environment and runtime suitable to the resource-constrained and low-power processors driving sensors and actuators. Linux-class machines in an edge environment may also benefit from low latency and low overhead task execution compared to continaers. To this end, EdgeRM adopts WebAssembly [HRS17b], a portable instruction format initially designed for secure and sandboxed computation in browser environments with near-native performance, that has also recently received attention as a suitable intermediary representation for embedded device applications [PPW20]. Specifically, we include the WebAssembly Micro Runtime (WAMR) in the embedded agent [Byt20a]. Embedded agents running WAMR can receive and launch arbitrary tasks compiled to WebAssembly; when augmented with a suitable sensor access interface, WebAssembly enables general and isolated computing.

4.3.4.1 Sensor Interface

Unlike Docker containers, where access to system resources is available through a well-defined POSIX interface, WebAssembly enforces a sandbox that constrains applications to structured control flow within a pre-allocated linear memory region. Any sort of external resource, such as filesystem access, is provided by explicitly granting functions to a WebAssembly module. While this design is well-suited to a resource management abstraction, the actively developed and in-progress WebAssembly System Interface currently lacks the notion of a sensor interface.

To support portable sensor access for WebAssembly tasks across the resource cluster, we propose a standard system interface for sensor access in WASM. When loading a module that requires sensor resources, the runtime explicitly grants the sensor API to the executing task, and each access to that API is validated by the runtime before a platform-specific implementation of the API function is called.

The sensor access API is summarized in Table 4.3. Sensors are defined by a set of capabilities, which refer to the raw underlying sensors (e.g. temperature), and configuration, which denote configuration parameters (e.g. sampling rate). Every sensor exposes two universal read-only configurations that allow a task to query for the current availability of *capabilities* and *configuration.* This generic interface allows tasks to update, and read from the set of on-board sensors.

We note that this sensor access API is not enforced by EdgeRM, and each EdgeRM agent specifies the API by which a device is accessed, however standardization of sensor access will improve task portability.

4.3.5 Active Scheduling for Dynamic Devices

As mentioned in §4.3.2 scheduling decisions in edge environments are often not strictly made due to resource availability, but also due to a devices' physical location, the network topology, events sensed by other devices, and many other possible optimizations. Additionally, in a cluster of heterogeneous devices, the capabilities of a unit amount of resources are necessarily also heterogeneous (i.e. 1 microcontroller CPU is not equivalent to 1 server CPU, and network links have very different throughput and latency).

To address this problem we propose *Active Schedulers* which schedule meta-tasks that are not designed directly to serve any one application, but which can be used to inform the scheduling decisions for all of the applications. There are several high-level classes for such meta-tasks that we have identified and included within the EdgeRM base scheduling library, including profiling and monitoring.

4.3.5.1 Profiling.

The *Profiler* meta-tasks periodically benchmarks the CPU and network link performance of a device to assist the scheduler in determining task placement. Both Docker and WebAssembly profiling meta-tasks are issued based on device supported executors, and results are made available within the scheduling library. Task scheduling can then consult the profiling data to determine, for example, the closest gateway available to a sensor.

4.3.5.2 Monitoring.

The *Monitoring* meta-tasks run on devices to alert a scheduler of device state change. The scheduler can then respond by scheduling tasks in response to this change. For instance, a scheduler may want to schedule a large task on a mobile device in a specific location, and could use a monitoring meta-task to avoid using the resources required by the large task until the device enters the specified area.

The writing and deploying of meta-tasks may be cumbersome for developers, especially if they are required to effectively use the underlying resources of the cluster. We envision an expanding library of services which schedulers can call for common meta-tasks. Associating this library with the scheduler rather than the resource manager allows for resources used by meta-tasks to be fairly attributed to a specific framework, and should be more efficient as meta-tasks will only be scheduled in response to the needs of a specific application being deployed by that scheduler.

4.3.6 Location, Context & Other Metadata

Schedulers rely on the ability to obtain necessary device context, including location, supported executors, and device architecture. Most of this metadata is naturally associated via static attribute properties that are presented to the master during device registration (§4.3.3). However, when considering dynamic device attributes, such as location, such information may not be statically pre-configured, or even known, to a device. Instead, this context may be associated with a device *after* deployment. For example, a homogeneous group of sensors may be collectively instrumented throughout an environment, with specific location information unknown until a particular sensor is placed. In these circumstances, we propose using EdgeRM attributes to provide a layer of indirection to a metadata store that is externally managed.

All attributes that are known to a device can be specified directly through the device's registration ping. Unknown or externally managed device attributes can instead be provided via a URI. For example, all EdgeRM devices can provide a location attribute whose value is a JDBC connection [FEB03]. The EdgeRM scheduling library unwraps these attributes by establishing a connection the the database and issuing a query for the specified attribute given the device ID provided during registration. The interface supporting attribute layer-of-indirection is easily extensible to support custom connections that are not yet implemented within the scheduling library.

4.4 Implementation & Frameworks

A POSIX-based agent implementation is written in Python which uses Docker as its container runtime and WAMR as its WebAssembly runtime. Embedded agents are implemented for the Zephyr OS [The20] and the Particle [Par20] embedded platform. Both of these systems use WAMR to execute WebAssembly modules. The embedded agent is currently capable of executing seven simultaneous WebAssembly tasks. This number is limited by both memory and the minimum footprint of a WAMR WebAssembly runtime, which uses excess memory due to memory alignment requirements that could be mitigated through additional implementation effort. The central EdgeRM master is implemented in Python along with a small local database to track updated system state.

Porting the edge agent to a new platform requires the implementation of a timer, malloc, free, thread creation, UDP send and receive, and functions to access the sensors which can be called from WebAssembly modules. Implementations are encouraged but not required to implement part of a standard system interface for WebAssembly modules. This standardizes sensor access and other common functionality such as timing and networking in WebAssembly [Byt20b]. While this functionality is not always present on embedded platforms, it is supported by most embedded operating systems.

In addition to the resource manager and agent implementations, we also implement a

Python library to simplify the process of creating a new framework scheduler. This library assists with requesting resource offers, filtering based on task requirements, and subsequently issuing one or more task execution requests on the provided resource offer. We expect subsequent development of additional libraries to perform other common frameworks tasks, such as sensor node data forwarders or device performance profiling.

4.4.1 Application Frameworks

We implement several applications frameworks which use EdgeRM to program an edge cluster. Specifically we implement a sensor sampling and filtering framework, a MapReduce framework, and we port an existing edge computing framework to use EdgeRM.

4.4.1.1 Sensor Sampling and Filtering Framework

The sensor sampling framework allows users to specify the sensor they wish to sample, the sampling rate, and several optional filters, then issues tasks to sample the sensor for several minutes before exiting. It consists of a container which runs a COAP server to receive and host client-specific sensor results and a WebAssembly task which samples and forwards data to the COAP server. If the selected sensor is a camera, an optional argument allows for image classification via a Docker container implementation using YOLOv3 [RF18].

A visual web interface provides users with a high-level view of all devices and tasks within the EdgeRM cluster, allowing them to select a sensor and issue a request. The framework is written statelessly; upon user request, the framework searches for an existing COAP server container, and, if one is not already executing, starts one on a publicly-accessible node. Once these infrastructure tasks are running, the framework searches for the device-of-interest within the resource offer. The sensor fetch task is then issued using EdgeRM with environment variables to specify the sensor, sample rate, filters, and COAP server location. This generic framework has been used by dozens of users simultaneously, with EdgeRM providing support for the scheduling or queuing of tasks depending on resource availability.

4.4.1.2 Porting an Edge Computing Framework

Retargeting existing edge computing systems to EdgeRM enable immediate benefits including isolation, multi-framework tenancy, and the inclusion of resource-constrained embedded devices into the computing cluster. Whereas these systems were classically deployed adjacent to the sensor network, they can now be incorporated into the same unified resource pool, enabling future iterations of edge computing frameworks to take full advantage of the resources nearest to the sensors.

To this end, we ported the *Edge Dataflow*[1] system to EdgeRM to validate its practicality and ease-of-use. Edge Dataflow allow developers to declaratively construct edge computing applications as directed dataflow graphs of microservice tasks to be distributed across the resource cluster. As is commonly the case in this setting, the intended target devices are gateways, wireless access points, and other near-edge devices. Porting the (1) client interface and (2) device webserver to a Docker container allows deployment using the EdgeRM abstraction over the same suite of devices.

Given the natural decomposition of distributed applications into containers [WVM17, LWB16, BZ17], porting edge computing systems to EdgeRM is a straightforward process. These systems can now also include previously ignored microcontrollers and IoT devices into their programming framework.

4.4.1.3 Sensor MapReduce

As initially described in §4.3.1 and illustrated in Figure 4.3, we have developed a Sensor MapReduce framework that operates on a cluster comprised of resource-constrained embedded devices and server-class machines. Users can develop `map` functions which can be

[1]Citation and exact name omitted to preserve author anonymity.

deployed on all sensors or specific sensors based on metadata filters. The output of these map functions is forwarded to a reduce function which can aggregate and publish the resulting data. A utility also displays the results of MapReduce dataflow on the user's terminal.

Map tasks are compiled and issued as WASM modules, while reducers are issued via Docker containers. The MapReduce framework continuously monitors the cluster to adjust deployed tasks based on the available resources, such as a newly registered sensor device, with support for killing and/or re-issuing MapReduce jobs. The framework leverages Active Scheduling principles incorporated into the EdgeRM scheduling library, including network profiling information, to schedule Reduce tasks to the nearest available and accessible server endpoint.

4.5 Evaluation

Our evaluation begins by demonstrating the multi-tenancy enable by EdgeRM and showing a snapshot the cluster's utilization (§4.5.1) and iteration time (§4.5.1.1) when being programmed by multiple users. We then perform an overhead analysis of the EdgeRM agent implementations (§4.5.2) and the WebAssembly execution environment (§4.5.3).

4.5.1 EdgeRM Cluster Utilization

To demonstrate the multi-tenancy enabled by EdgeRM we collect the share of the cluster CPU used by three frameworks as multiple users use the cluster. This 10 minute snapshot is shown in Figure 4.4. We see that multiple frameworks and users are able to share the cluster and its resources, with short-running tasks such as those generated by the Sensor Sample and Filter framework creating spikes in utilization. We also see the CPU utilization of a single sensor, seeing that tasks from both the Sensor Sample and Filter framework and the MapReduce framework are executing simultaneously. When resources are not available for more tasks, the frameworks may direct tasks to other sensor nodes as appropriate. The

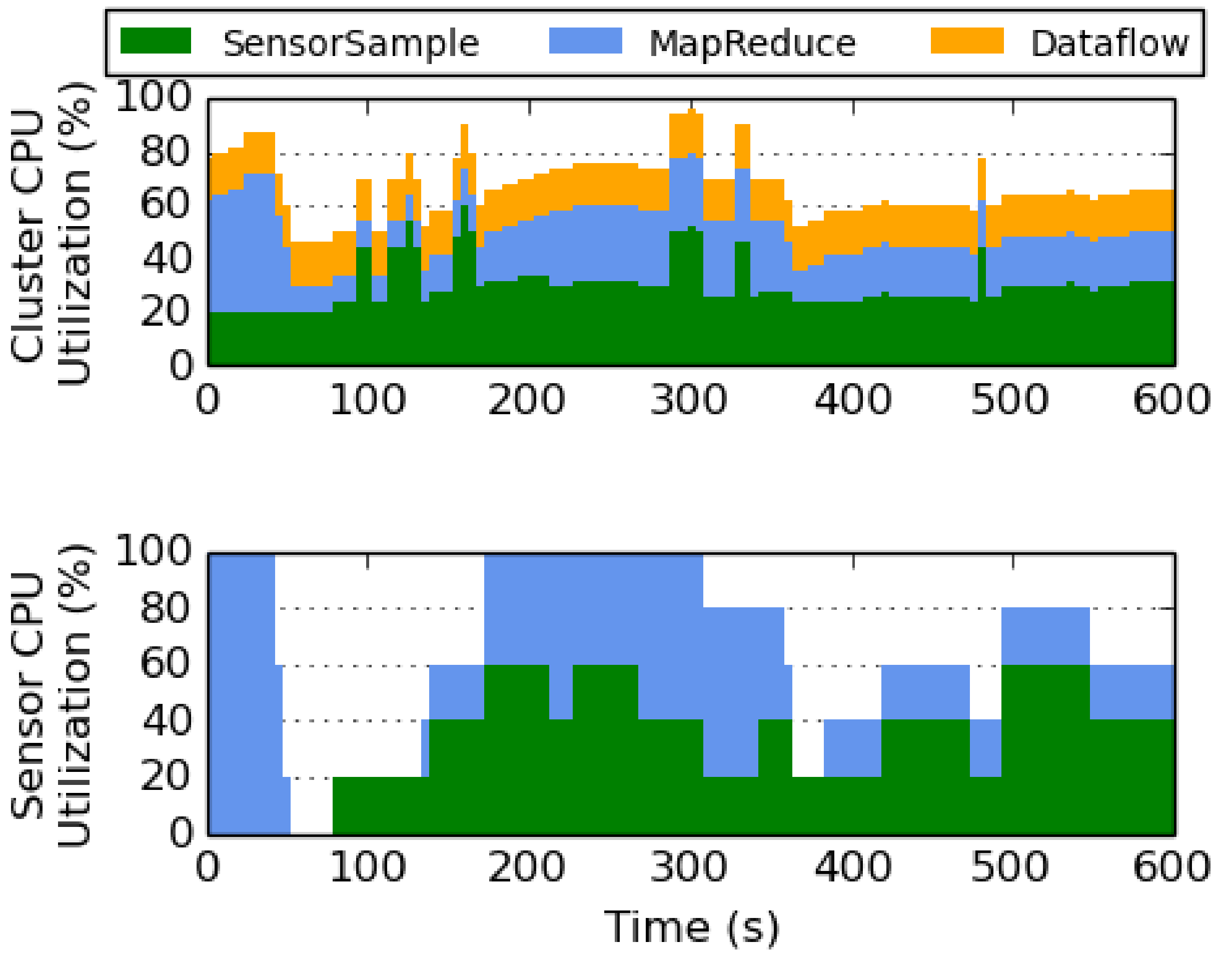


Figure 4.4: Utilization of the edge cluster (top) and a single sensor (bottom) by three programming frameworks over a ten minute period. Multiple users deploy jobs to the edge cluster through three programming frameworks using EdgeRM. These three frameworks are capable of multiplexing the cluster and can deploy tasks on both sensor and server nodes simultaneously. The mediation of resources through EdgeRM enables multi-tenancy on constrained, embedded devices that are traditionally singe-purpose.

running of multiple simultaneous tasks, and specifically the dynamic deployment of simultaneous tasks from multiple programming frameworks onto a resource-constrained embedded node would not be possible without EdgeRM.

4.5.1.1 Development Iteration Time.

A key disadvantage of most embedded development is the high time between code iterations, especially if testing must occur across multiple embedded nodes. Manually flashing each node is difficult and time-consuming, and cloud services may also have to change with each code iteration if data formats between the embedded device and the cloud change. EdgeRM addresses these problems by allowing frameworks to manage the deployment of new applications across multiple embedded and server-class nodes.

A snapshot of a user performing iterative development using the MapReduce framework and EdgeRM is shown in Figure 4.5. We see that a user is able to deploy 10 iterations of the MapReduce code in under 6 minutes, retrieving, collecting, and analyzing results between each iteration.

4.5.2 EdgeRM Agent Overhead

4.5.2.1 Memory and Code Size.

The memory footprint and code size of the EdgeRM agent implementations are presented in Table 4.4. The Python agent implementation was profiled on a Raspberry Pi 3B+, corresponding to a 2.4% memory footprint overhead [Ras20]. The embedded agent was profiled on an NRF52840 MCU. 65 kB of the embedded agent code size and 2.6 kB of the embedded agent SRAM are from the resource manager implementation and WAMR runtime. An additional 22.3 kB of SRAM is needed for every task executed on the embedded agent no matter the task size (and more is required of the task itself uses more memory). This relatively high per-task overhead is primarily due to the minimum memory region of 16 kB

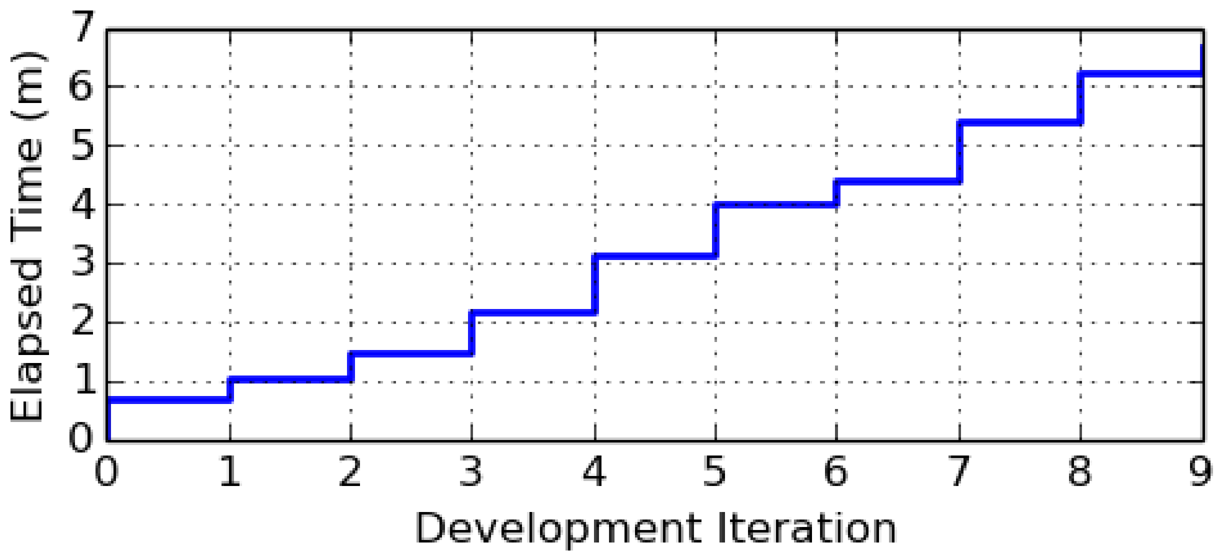


Figure 4.5: Time of interactive development cycles. Sensor MapReduce applications are iteratively deployed on devices with a ping rate of 20 s, and results are received and evaluated between each iteration. EdgeRM enables short interactive development cycles not achieved by many other ways of programming clusters of edge devices.

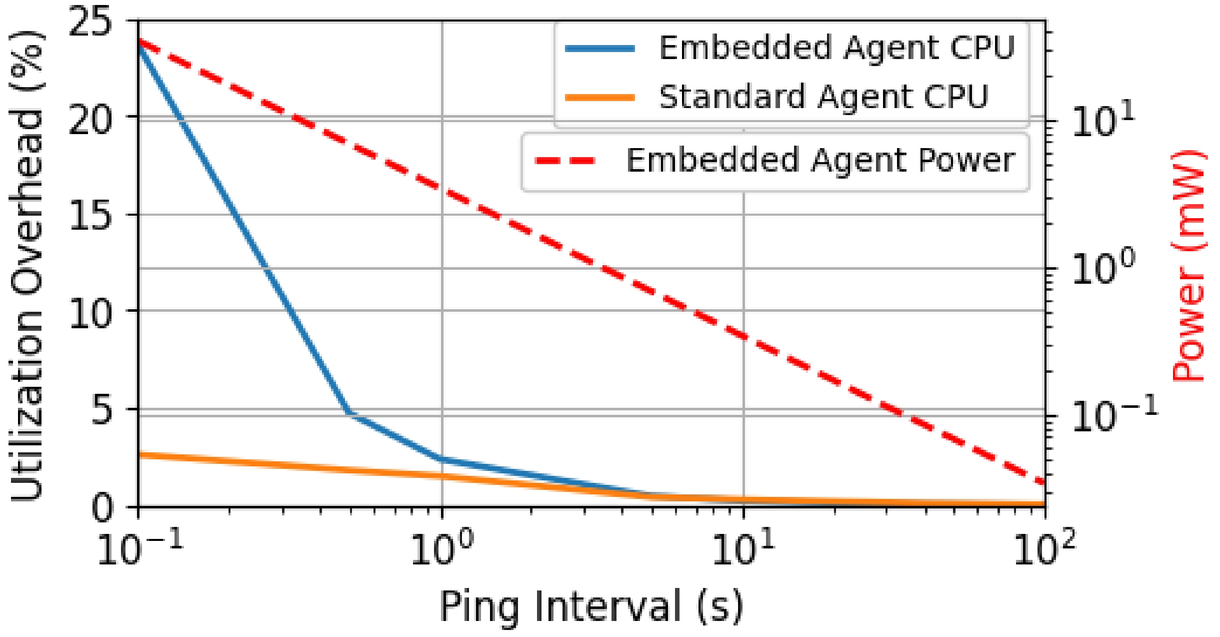


Figure 4.6: Compute and power overhead of the EdgeRM agent, plotted as a function of agent ping interval. As the ping interval is increased, overhead falls proportionately. On the embedded agent (evaluated on an NRF52840 MCU) ping intervals greater than 1 s have CPU utilization below 5 %, and ping intervals greater than 100 s have a power consumption of less than 34 µW. A bounded exponential back-off on ping interval maintain interactivity while decreasing power.

needed to execute a task in WAMR, however we expect this could be significantly decreased with a more optimized implementation. The remainder of the code and RAM are used by networking and OS libraries that the EdgeRM agent uses, but would also be required by most applications.

While the overhead introduced by EdgeRM on an embedded device is not insignificant, it nevertheless falls within the practical range of modern microcontrollers, especially when considering the computational benefits enabled by integrating an EdgeRM agent into the resource cluster. On the NRF52840, which has 256 kB of SRAM, we are able to execute seven simultaneous WASM tasks, and we expect this number to increase with more optimized WASM runtime implementations and ever-growing MCU memory sizes.

4.5.2.2 Compute and Power Overhead.

Compute and power overhead of the EdgeRM agent is directly proportional to the frequency at which an agent pings the master. Figure 4.6 presents the average compute utilization and power consumption as the interval between successive agent pings is increased. The standard agent compute utilization was profiled on a Raspberry Pi 3B+, while the embedded agent compute utilization and power consumption were profiled on an NRF52840 MCU connected to an OpenThread 802.15.4 network. Most of the CPU utilization and power consumption are used by the networking operations required to ping the master. For powered embedded devices, a ping rate of greater than 1 s keep CPU utilization below 5 %; for energy-constrained devices a ping rate of 10 s uses 340 µW and a ping rate of 100 s uses 34 µW. To conserve energy while still enabling fast iteration during interactive periods embedded agents can exponentially back-off their ping rate.

Code Segment	Text (B)	Data (B)	BSS (B)
Total	317,918	3,748	90,460
Openthread (Net)	149116	—	38,087
Agent Library	15,764	—	1,401
WAMR Runtime	51,564	—	1,250
WASM Task (min. ea.)	—	—	22,269

(a) embedded agent

	Memory Usage (MB)
Python Agent	22.8

(b) python agent

Table 4.4: Memory and code footprints of the EdgeRM agent implementations. The embedded agent flash and RAM utilization are decomposed into constituent components. A significant portion of Flash and RAM utilization is due to the networking stack and the underlying OS, which would also be required by a monolithic firmware. Remaining unused memory is available to store and execute WASM tasks. The minimum memory for each task is 22,269 Bytes, which includes all task state, thread stack and heap, and the minimum 16,384B required to execute a WASM module.

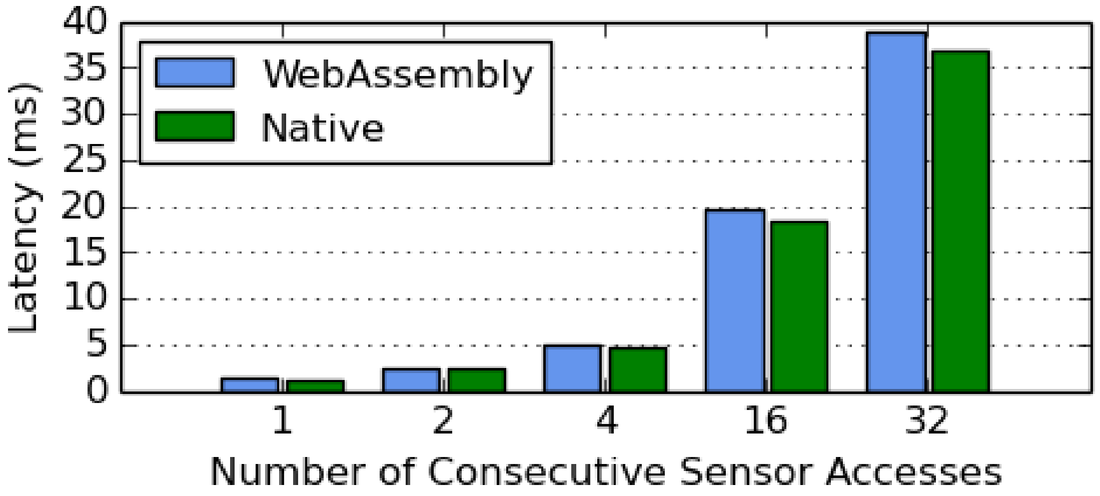


Figure 4.7: Latency overhead of accessing on-board sensors through WASM. Sensors are accessed a number of times using a WebAssembly task with the WASM sensor interface and access time is compared to directly accessing the sensor with the underlying platform SDK. WebAssembly introduces less than 5 % latency overhead.

4.5.3 WebAssembly Overhead

The overhead analysis of the WebAssembly execution environment is decomposed into a (1) sensor access latency overhead and a (2) compute overhead analysis. The former is encountered when tasks are issued to collect sensor data, while the latter indicates the overhead required to sandbox execution of pure compute tasks with the WebAssembly interpreter. Device memory consumed by the runtime is included in the memory footprint presented in Table 4.4.

4.5.3.1 Sensor access latency

Figure 4.7 presents a latency overhead analysis of the WebAssembly runtime and sensor interface with respect to native implementations. Latency was profiled by fetching temperature values from a BME280 sensor a varying number of times to indicate (1) the fixed startup cost of loading the WAMR runtime, and (2) the marginal overhead of individual sensor accesses, with respect to native.

The startup cost of booting the WebAssembly runtime is on the order of hundreds of microseconds, indicating a less than 10% overhead with respect to a sensor access. Once loaded, an individual WebAssembly sensor task access has a latency overhead of 5% with respect to the latency required by the underlying platform SDK.

4.5.3.2 WebAssembly compute overhead

The current implementation of the EdgeRM agent uses the WAMR runtime in interpreter mode, directly interpreting WASM bytecode. Currently WAMR reports that interpreted WebAssembly runs 11-16x slower than native code for common benchmarks such as Coremark and Fibonacci [Hua20, Shy20]. We soon hope to integrate ahead-of-time compilation and execution, which runs at 85-95 % of native speed, as a service so that EdgeRM can better support computationally intensive tasks. EdgeRM could transparently compile WebAssembly to a target architecture if the architecture attribute is present.

4.6 Discussion

As an early system we still envision significant future work will be done on EdgeRM, and we hope that as an open source system the community will help to define the direction of that work. We use this section to discuss several directions that we believe EdgeRM may need to take to better fulfill its role as the resource manager for the modern edge.

Programming in Accessible Languages. We hope that EdgeRM can facilitate the programming of sensor networks with higher level languages and specifically higher level languages that are commonly used by domain scientists who need sensor data. Many scientists use languages like R, and we believe enabling them to use an R API to program sensors directly would reduce the time and increase the precision of data processing compared to translating requirements to an embedded programmer.

Orchestrating Long Running Tasks. In most example frameworks built for EdgeRM we find that some of tasks are long-running (i.e. servers, databases, messaging brokers) and some tasks are intended to be more ephemeral (sensing, filtering, processing). As many other projects such as Kubernetes and the work leading to its creation have discovered, long running tasks have their own needs for monitoring, restarting upon failure, auto-scaling, and log aggregation. We do not intend nor do we think it necessary that framework builders recreate these features to support what has become a common application paradigm.

As future work we hope to explore how to layer in existing container orchestration solutions so that they can be easily used by application frameworks. This would allow container orchestration mechanisms to do what they were designed for, while still allowing framework developers the flexibility to create diverse schedulers for the data pipelines of more ephemeral jobs feeding into these infrastructure containers.

Networking and Failure Domains. One key piece of scheduling information that EdgeRM does not yet collect is the explicit network topology or a sense of failure domains. We have considered allowing users to specify failure domains as device attributes, where devices that have the same failure domain attribute have more reliable internet networking. We have also considered creating active scheduler jobs which scan the network at a framework's request, attempting to automatically map LANs. We plan to attempt these techniques in future deployments so that frameworks can better realize the reliability improvements that should be associated with placing computation in local area networks.

4.7 Conclusions

The ascension of IoT sensors into the computing cluster is a logical next-step over the common practice of maintaining two separate systems established in two separate domains. Our results indicate that the low-power processors driving sensors and actuators now possess the sufficient memory and computational capability to host a complete resource manager process and support isolated and generic application execution. We offer the EdgeRM abstraction and open-source implementation in hopes that its use can extend the edge computing cluster to the true edge of the network.

Further, EdgeRM aspires to bridge the distributed systems and sensor networks communities by serving as a shared framework spanning both domains. This common medium could offer a communication channel by which research innovations and technological advancements can be carried over from one to the other. The inclusion of isolated, interactive, and multi-tenant resource sharing into the sensor network enables security and usability innovations introduced into the systems community, while the incorporation of sensor and actuator nodes into the compute cluster enables more accessible, exploratory, and impactful application development.

CHAPTER 5

The ADAPT^2 Principles for Robust Adaptation

The essence of autonomic resource management in computer systems is to adapt resource allocation based on trusted load measurement data reported from the physical devices. We argue that one largely overlooked concern is the opportunity for an attacker to use information about adaptation triggers to launch an attack. These vulnerabilities in autonomic resource management enable complex attacks that leak cyber-physical aspects of the system, propagates malicious or exploitative code, and/or launch side-channel attacks against other co-located users. We enumerate cases where an attacker can report false utilization statistics to invoke adaptive network behaviors in a targeted fashion, e.g., over-reporting utilization to request additional resources from the resource manager.

Even without controlling a device in the system, a malicious entity can infer sensitive spatio-temporal information by merely observing the current state of network activation. The vast inter-connectivity of edge networks increase both the visibility and accessibility of network nodes. This inference of activity based on adaptation has significant implications in safety-critical applications, such as those in smart cities, industrial IoT, and the Internet-of-Battlefield-Things [SWW15]. Failing to obfuscate location information of these devices can be crippling in mission-critical contexts.

We use three attack scenarios derived from the aforementioned vulnerabilities to propose the ADAPT^2 framework for secure and robust resource management. Using techniques derived from Moving Target Defense [JGS11], we suggest state estimation of computational models to identify, isolate, and invalidate devices that are reporting false utilization statistics.

To obfuscate cyber-physical characteristics of the network, ADAPT^2's resource manager injects dummy workloads to attest suspicious nodes. These dummy workloads will also be used to activate idle regions for the purpose of obfuscating latent activation patterns of the network.

Our contributions can be summarized as follows. In Section 5.1 we present the system and adversary models considered. In Section 5.2, we discuss and exemplify how an attacker can manipulate and/or compromise the system using knowledge about vulnerabilities in the resource manager. In Section 5.3, we discuss ADAPT^2, a suite of autonomic resource manager system extensions designed to protect against the previously discussed attacks. We compare our work to the state-of-the-art in Section 5.4.

5.1 System and Adversary Models

We begin by presenting the system model considered followed by the proposed adversary model for each of the attacks on adaptation. We elaborate on the feasibility of each assumption of the adversary model by enumerating four possible attack vectors.

5.1.1 System Model

We consider a distributed autonomic edge cloud system that is comprised of a networked set of devices with a centralized entity responsible for resource management, referred to as the autonomic resource manager. The autonomic resource manager controls the amount of capacity allocated to an application, e.g., how many instances of an application software component and which devices they are mapped to. This allocation can be based on application load (e.g., increased load might require extra instances to maintain QoS), the type of the application (e.g., critical applications require replication for availability), and the workload dynamics (e.g. device movement may require migrating associated computations to minimize transmission latency). The resource manager has the ability to communicate with any

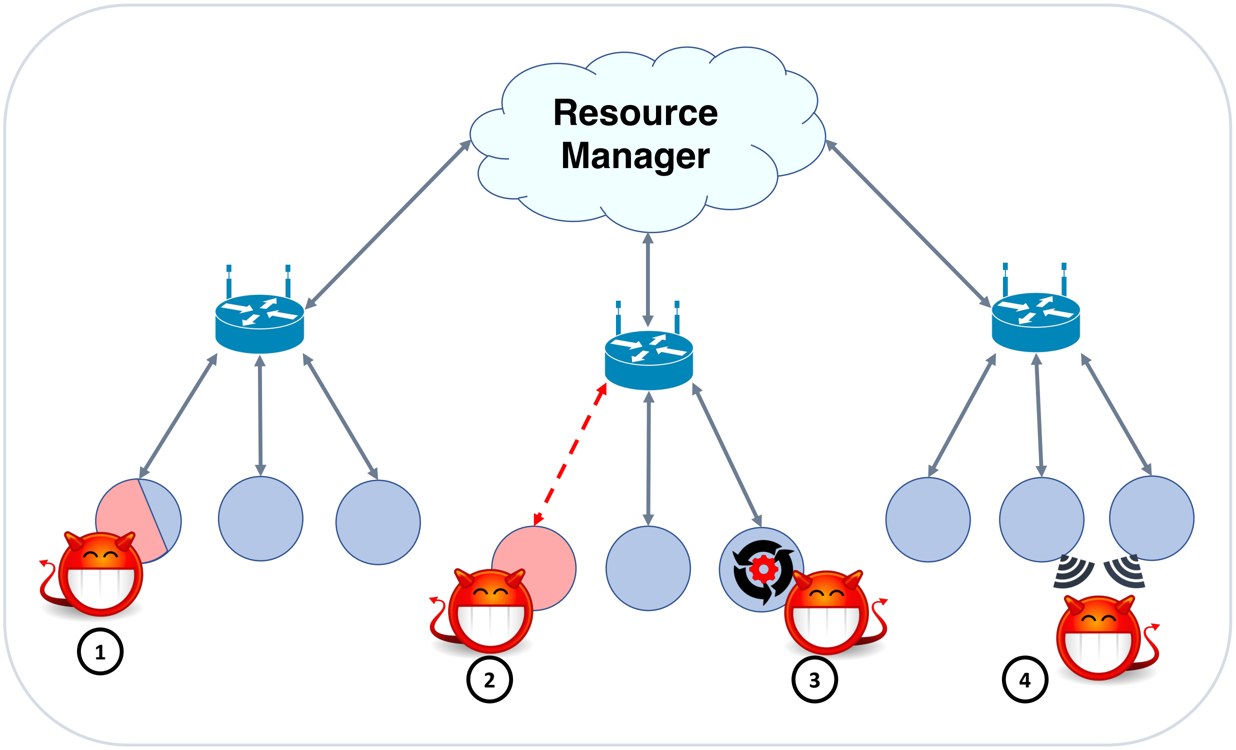


Figure 5.1: An example distributed network illustrating four attack vectors that can be used to compromise devices in an autonomic system. 1) An attacker has launched a physical attack. 2) An attacker has injected a device into the network. 3) An attacker has exploited a device's software vulnerability. 4) An attacker has obtained meta-information via side-channel attack.

networked device, thereby having a full view of the distributed computation of workloads. Finally, computing resources are shared between the software components using light weight virtualization such as Micro-VMs, thus providing multi-tenancy on the network's sensing, computation, and actuation resources [MBS17, YPK14].

5.1.2 Adversary Model

The adversary's goal is to manipulate resource allocation and infer sensitive application information. We assume that the adversary wants to maintain stealthiness, so as not to raise suspicion. Stealthy attacks such as Stuxnet [FMC11] have proven that stealthiness enables longer and more impactful attacks, especially in a cyber-physical context. We focus on attacks that specifically target the resource adaptation mechanism.

Attack vectors. In order to manipulate resource allocation, an attacker must compromise at least a single device. However, an attacker can still derive system information without compromising a device. In summary, we consider the following attack vectors:

1. An attacker has gained access to at least one physical device. Gaining physical access to a device is significantly easier in an edge network [SST16]. This allows an attacker to mount a physical attack by either attaching malicious hardware or manipulating the device platform [GRB12].

2. In IoT networks that allow ad-hoc admission of IoT devices, an attacker uses a malicious device to join the network, and is accepted by a Resource Manager via the legitimate admission interface.

3. The attacker has not gained access to a physical device, but has gained access to a software component. The attacker may exploit a software or network vulnerability to gain root or user access. While this type of attack is not specific to the IoT edge domain, IoT and mobile software are often less robust, thereby offering more exploits when compared to typical machines [SST16].

4. The attacker has not gained device access, but can obtain meta-information regarding device activation. This information can be obtained via side-channel attacks, such as remotely detecting changes in signal output, generated heat, or electromagnetic leakage [LDP15].

5.1.3 Assumptions

These four attack vectors, illustrated in Figure 5.1, lead to an assumption of an untrusted edge network. The attacker has managed to circumvent existing security mechanisms implemented on the system to gain access to devices. This model also assumes that we cannot attest the integrity of either the software or hardware for some devices[1]. Similarly, we assume that any trusted execution environments can be circumvented by attackers [Van18].

Number of compromised devices. We assume the attacker can only compromise a subset of the system, as a distributed attack spanning all heterogeneous devices located in different physical regions would be infeasible.

Trusted resource management. The resource manager is assumed to be a trusted entity that has much stronger security guarantees than other devices, particularly those residing at the edge. We assume this entity has more stringent cyber and physical security mechanisms.

Computing power. We assume an attacker has access to an adversarial pool of external resources that can be used for processing and offloading of computations. While not necessary, this effectively provides an attacker infinite compute resources by which to fool any source of compute validation checker.

Applications. We assume the applications of interest are deployed and managed by an autonomic resource manager over a distributed edge network. For the purposes of our discussion,

[1]Although remote attestation can be implemented, this typically requires a hardware root-of-trust. If a device doesn't support such hardware, software-based attestation can be used but typically requires very strong assumptions regarding the timing and authenticity guarantees of the communication channels between the device and the external verifier [SS04]

we will look to applications that span a smart city environment to aid law enforcement and first responders.

5.2 Attacks on Adaptation

In this section, we enumerate three attacks on autonomic resource managers in edge computing systems. The first two focus on using the utilization statistics reported by a device to the monitoring entity to attack the system. The latter attack focuses on making inferences about spatio-temporal network characteristics given a network's resource allocation and system meta-information. There are many more attacks that can be launched from a set of compromised devices in a autonomic computing system; we focus on those that target the resource management substrate.

5.2.1 Falsely reporting low utilization

Resource managers often implement a dynamic scheduling policy that considers current device utilization or computational capacity (e.g. CPU cores, memory, etc). Tasks are more likely to be assigned to devices experiencing low utilization as opposed to those experiencing high utilization. Reported utilization and/or capacity is often accepted at face-value without verification from the resource manager; however, in an ad-hoc network it may be important to treat device-reported statistics with scrutiny. A compromised device can fool a resource manager by providing false utilization information.

In this attack, a compromised device reports under-utilization and/or claims to have very large capacity. The resource manager may then choose to map more tasks to the underutilized device, allowing the compromised node to gain access to more application semantics. In order to maintain stealthiness, the compromised node can offload some assigned computations to an adversarial external cluster.

The attacker can use the placement of multiple application components by the resource

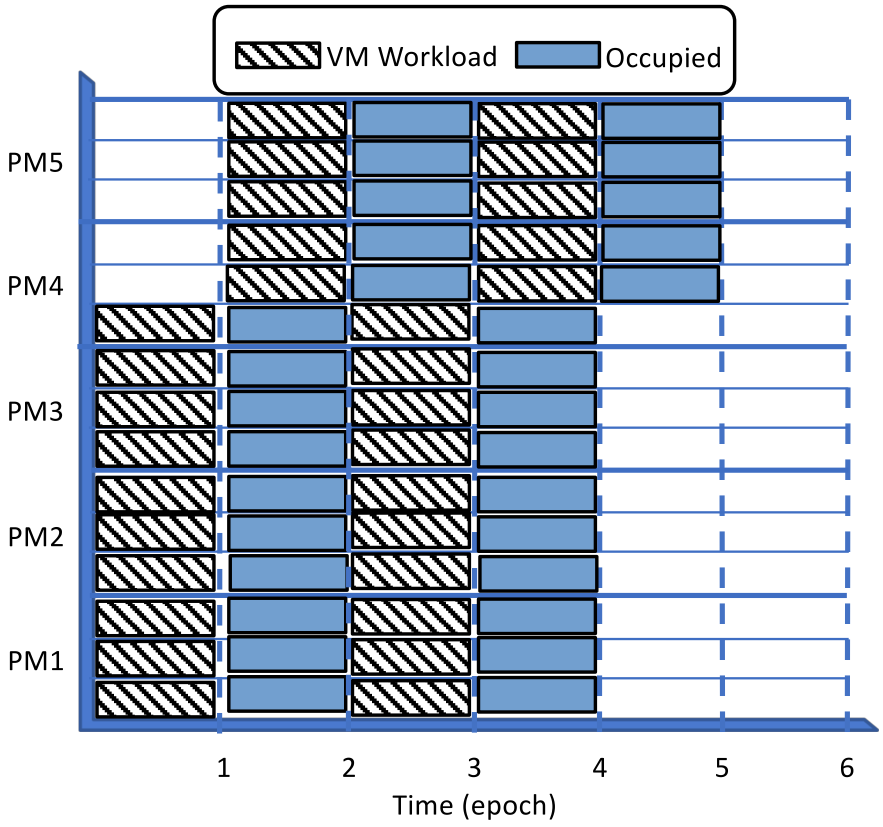


Figure 5.2: The expected behavior of workload placement for the first-fit algorithm (PM = Physical Machine, VM = Virtual Machine).

manager on the compromised node to stage other attacks such as data-poisoning. In the worst case, the attacker will have access to all applications' semantics, as the resource manager will blindly place software components onto the device with the lowest resource utilization in the network.

Example attack. An application deployed over a smart city is likely to span a wide range of devices. Exfiltrating semantic behavior to untrusted entities can ultimately cripple application objectives. We present this attack against the first-fit placement algorithm, a simple but efficient placement algorithm [CKM19]. Given a software component, e.g., a virtual machine, first-fit assigns to the first available physical machine with sufficient capacity to meet QoS requirements. For our example, we define a scenario with 5 homogeneous physical machines (PMs) capable of supporting 3 equivalent virtual machine (VM) workloads each. The placement algorithm fits these 10 VM workloads onto the PMs at each time epoch, with each workload requiring 2 time epochs to complete. Figure 5.2 presents this scenario under normal operation. Figure 5.3 illustrates an attacker exfiltrating workloads to an external entity via a compromised PM. At each time epoch, the compromised PM reports underutilization, thus gaining access to 3 VM workloads at every epoch.

5.2.2 Falsely reporting high utilization

One of the key tasks of a resource manager is to maintain Quality-of-Service (QoS). Variations in device and application performance are common, typically a result of workload burstiness, energy dynamics, and device mobility. A resource manager may initiate an adaptation to remap computation to resources and maintain QoS with respect to a pre-defined system metric (e.g. latency, energy, utilization). While important for all applications, maintaining strict QoS is essential for mission-critical and real-time applications, such as those in search-and-rescue and military applications [CHM18]. When faced with an over-utilized device or unacceptable QoS, a resource manager will typically assign more resources to the application, either by horizontally scaling, i.e., adding replicas of a software component to additional

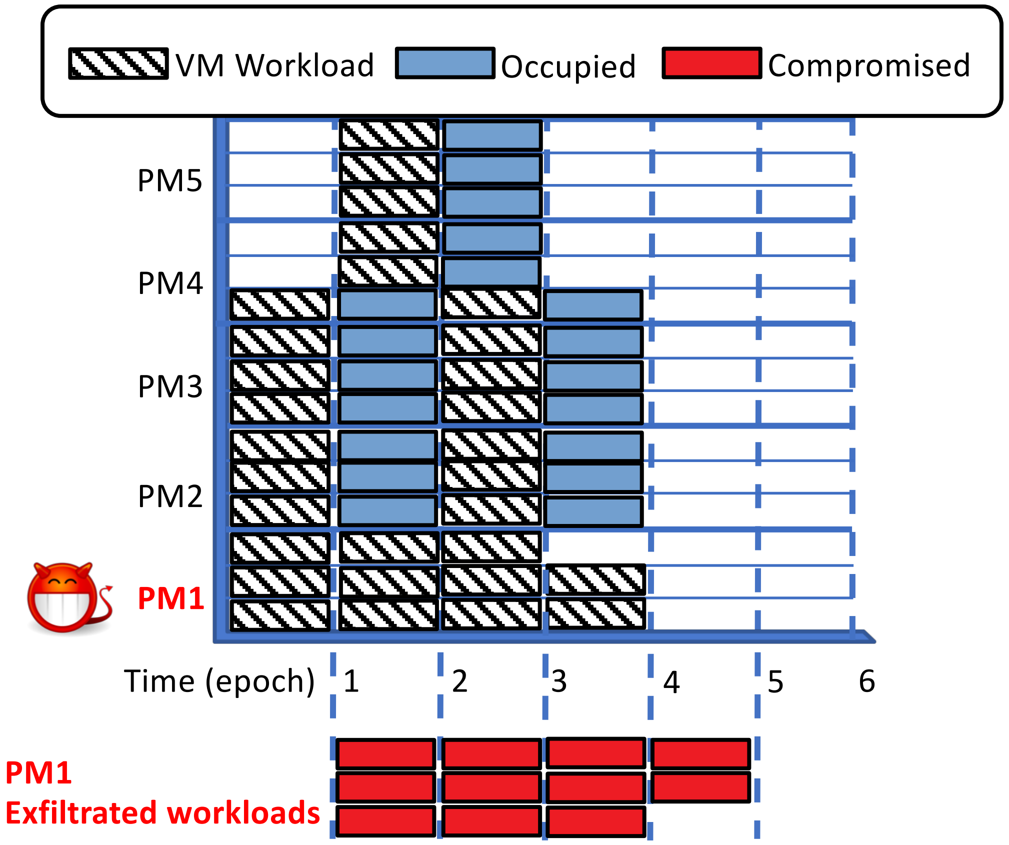


Figure 5.3: The behavior of workload placement when an adversary is reporting low utilization for a physical machine.Workloads are exfiltrated to an adversarial entity (PM = Physical Machine, VM = Virtual Machine).

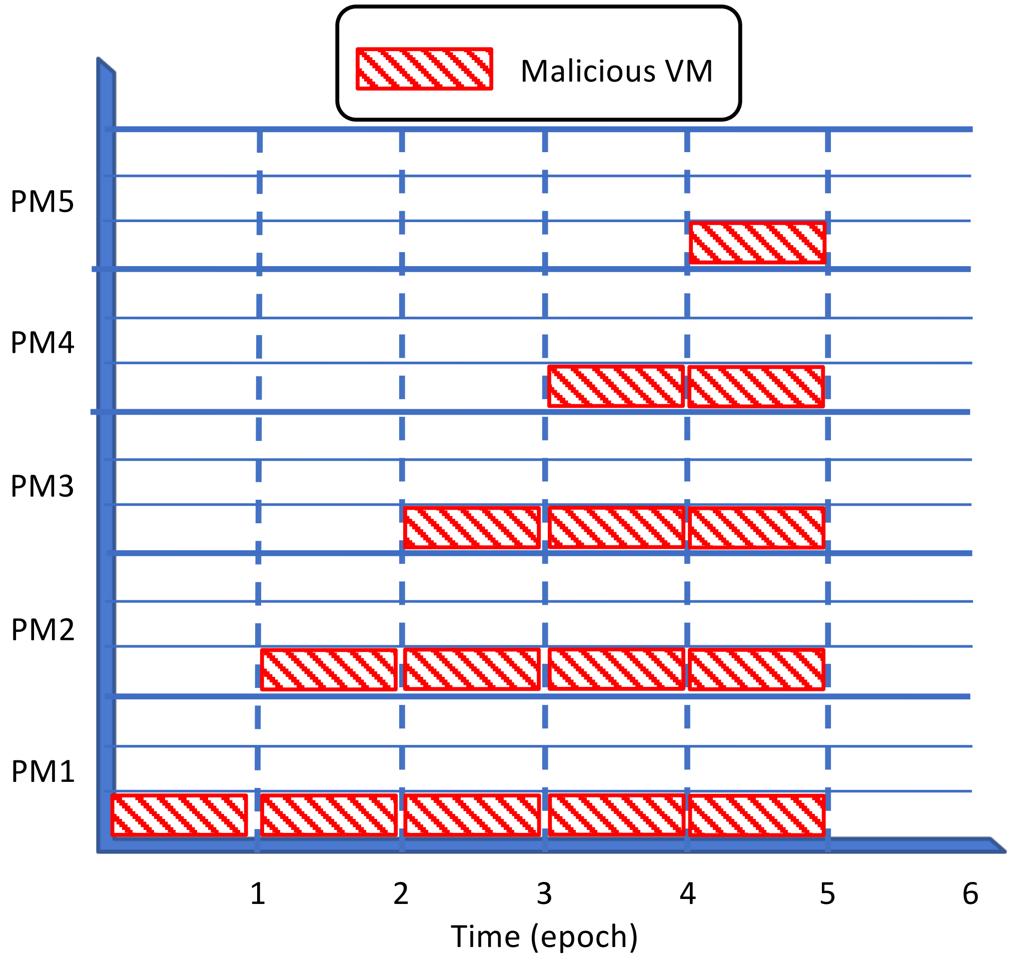


Figure 5.4: The behavior of workload placement when a malicious VM reports high utilization in order to replicate malicious code onto other physical machines (PM = Physical Machine, VM = Virtual Machine).

devices, or by vertically scaling, i.e., increasing the memory or CPU allocation to a particular software component.

In this attack, the compromised software component reports that it has experienced an increase in utilization and violated QoS. If vertically scaled, the resource manager will allocate more resources to the compromised software component, allowing the malicious component to control more resources that now can not be used by legitimate applications.

Figure 5.4 illustrates how a compromised VM workload is horizontally replicated by the resource manager at each time epoch if it is consistently reporting that it requires an additional PM. The resource manager scales the malicious component creating more replicas of the malicious software, thereby allowing an attacker to infect more devices and mount serious attacks, such as cross-VM side channel attacks [ZJR12]. By replicating malicious software, the total available resources will be effectively reduced, leading to a Denial-of-Service launched by the resource manager itself. This style of DoS attack is quite common in the IoT and edge computing domains, including those incorporating robust autonomic resource management systems [KKS17].

5.2.3 Inferring action from adaptation

Resource Managers often optimize computation placement based on a particular heuristic, for example end-to-end latency. Applications with a spatio-temporal aspect may leak sensitive location information as a direct result of a resource manager adapting computation placement or sensor/actuator activation.

In this attack, we consider an application that computes data over a dynamic region. For example, consider a smart city environment where a set of law enforcement vehicles are moving through an IoT and infrastructure-rich environment. An application is deployed to identify suspicious activity and potential threats surrounding the vehicles. As these vehicles traverse the territory, edge devices are active depending on whether a vehicle is located nearby. An attacker can observe the currently active set of devices to infer information on vehicle locations. In this way, adaptation can leak sensitive spatio-temporal information to an attacker, who can construct a location heat-map based on device activity.

5.3 Adapting Adaptation

To counter the aforementioned attacks on adaptation, we present how existing characteristics of adaptive models can be instrumented to detect false reports of device utilization statistics

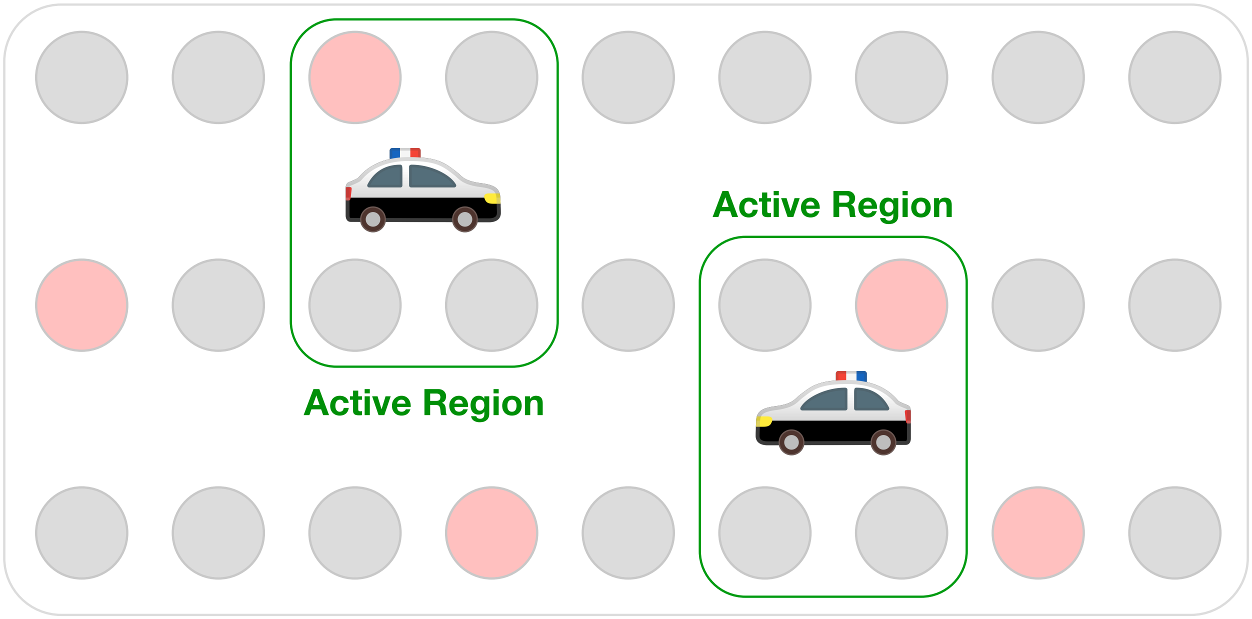


Figure 5.5: An example illustration of how adaptation can leak location information. As vehicles traverse the smart city, nearby devices are activated. An attacker can observe the active set of compromised devices and make inferences on the location of the vehicle.

and obfuscate leaked spatio-temporal information.

5.3.1 Detecting False Device Utilization Statistics

Due to potential attacks on a resource manager's placement algorithm, a robust system must scrutinize reported device utilization statistics. An obvious solution to detect if a device has been compromised is to verify the integrity of the device's software state via device-fingerprinting [FSL16] or standard trusted computing techniques [Mit05]. However, in dynamic distributed IoT systems where new devices are being recruited onto the network, it is difficult both to build deterministic fingerprints for devices as well as to ensure that these devices are enabled with trusted platform modules to perform remote attestation. As such, we opt for a solution that depends on the state of the network's resource consumption model.

5.3.1.1 State estimation for resource consumption

The first component is analogous to state estimation for detection of false data injection attacks [MGC10]. We propose generating probabilistic models of expected resource consumption based on device characteristics. For a given task and device, a predicted target is compared to reported statistics. This enables a challenge-response against each device to verify that a reported resource utilization is within certain bounds of the model. An attacker may have the ability to spoof responses given a particular workload challenge, but can no longer report drastically inaccurate utilization statistics, thus preventing a crippling attack leveraging reported resource utilization.

5.3.1.2 Moving target defense for more robust state estimation

If an attacker can learn a probabilistic model of resource consumption for a given application workload, then the attacker may be able to provide spoofed data to fool the challenger. To counter this spoof, a resource manager can rely on Moving Target Defense (MTD) techniques, where resource mappings are constantly changed to randomize computation assignment to devices [JGS11]. As a device's workload changes, accurately predicting expected resource consumption becomes more difficult, and an increased opportunity arises for identifying malicious devices. To spur workload changes, we can adapt MTD by randomly initiating an unprovoked "adaptation" to remap computation to devices and initiate a new round of challenge-response. The resource manager can improve initial state-estimation models by collecting utilization and resource consumption data from each round of MTD for all the components and devices. This presents a clear security-performance trade-off dependent on overhead in adaptation; furthermore, meeting QoS requirements under network constraints may restrict the space of possible MTD configurations.

5.3.1.3 Isolating suspicious devices

Suspicious devices that are marked as deviating significantly from an expected model can be isolated and further evaluated. This group can be tested with either latency-tolerant applications or dummy workloads. Dummy workloads can be explicitly designed to intentionally generate resource consumption outside a given model's expectations. In doing so, it becomes easier to determine whether a device is intentionally misreporting utilization statistics.

5.3.2 Obfuscation of Leaked Spatio-temporal Characteristics

Although the state estimation model with induced randomness can provide a means of verifying a device's resource utilization characteristics, it does not obfuscate the side channels that are exposed by adaptation, e.g., identifying movement by detecting computational workload patterns in different locations. As such, we propose instrumenting the aforementioned redundant workloads in a way that randomizes the activation of different nodes. Dummy workloads, initially spanning the set of suspicious devices, can additionally span devices that would otherwise be relatively inactive. In this way, application-specific location data can be obscured such that attackers can no longer generate a valid activity heatmap. Furthermore, for applications that are not latency sensitive, a resource manager can move these applications to idle regions in a manner that is cognizant of the resource allocation's entropy. In this way, we can trade-off latency for security.

5.3.3 Adapt^2 Principles

To enhance resource managers with the capabilities required to shield against the above attacks, we propose ADAPT^2, a model framework for resource manager extensions that include a state estimator, MTD, and spatio-temporal obfuscation.

Figure 5.6 illustrates the ADAPT^2 extension to resource management, which is comprised of three main system components. The first is a state estimator that is used to construct

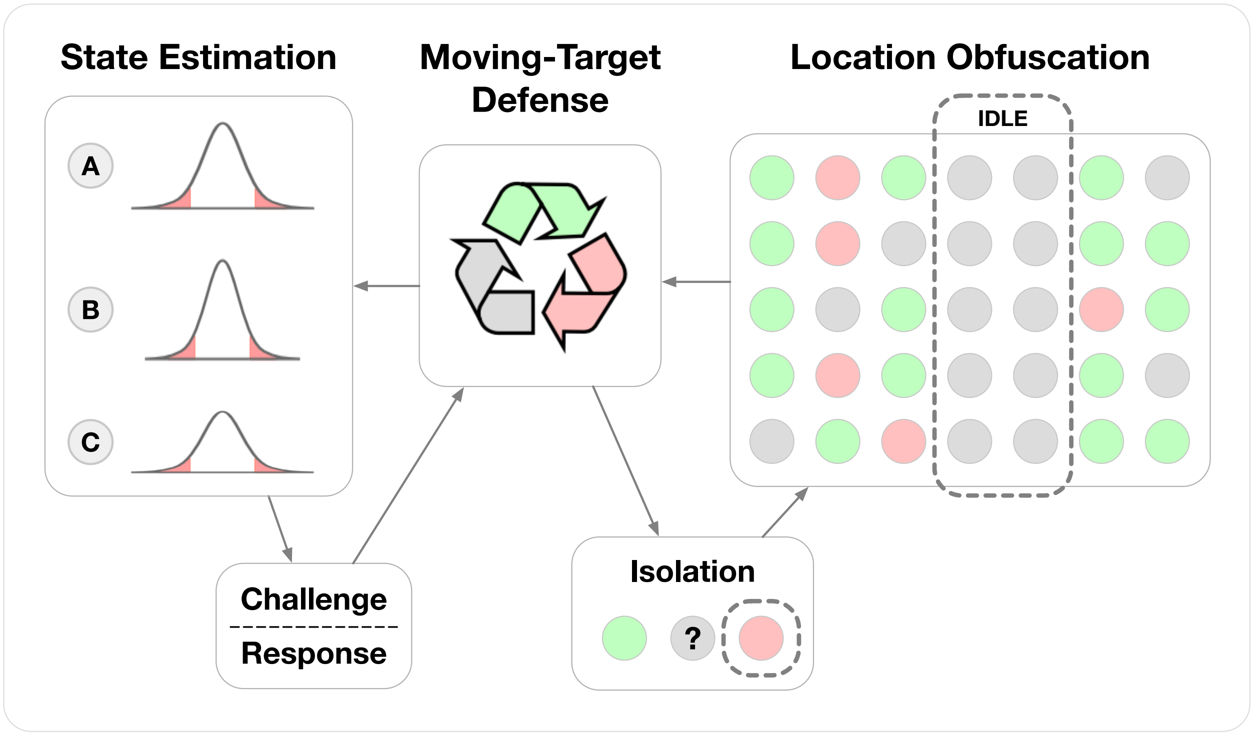


Figure 5.6: ADAPT^2 system components, serving as an extension to existing resource management. The three main components are a state estimator, a moving target defense generator, and a spatio-temporal location obfuscator. In tandem, state estimation generates challenge-responses to identify suspicious devices, which may be isolated and tested with dummy workloads.

a probabilistic model that maps device characteristics and application tasks to a predicting range of resource consumption. The second incorporates MTD by monitoring adaptation history and application mapping in order to determine appropriate times to initiate a randomization. It tracks suspicious devices based on challenge-response information generated from the state estimator to determine appropriate workloads to deploy onto suspicious devices. Suspicious devices can additionally be isolated to enable in-depth testing. Finally, the location obfuscation component uses current resource mappings to identify latent idle regions and have (potentially dummy) workloads scheduled in order to minimize spatio-temporal location leakage as a result of localized activity.

Implementing the ADAPT^2 model framework as an extension of an existing resource manager prevents the attacks on resource usage discussed in Section 5.2. In doing so, one can reduce the attack surface exposed by the Resource Manager in distributed edge-cloud networks.

5.4 Related Work

IoT devices have been the target of a number of attacks. DDoS-as-a-service attacks such as the Mirai botnet have shown how the vast quantity of these deployed devices can be exploited as a cyber weapon [KKS17]. As the Internet-of-Things has encompassed more safety critical applications, we have also seen cyber-physical attacks on autonomous vehicles [MV15].

Securing edge-cloud systems and their applications has received considerable attention [HCT18, RLM18, SWW15]. Similarly, the problem of securing IoT networks has also received significant attention in the research community [CHM18, DK18, MJ17]. Fundamentally, the work on IoT and edge network security can be broadly divided into two main subcategories: work illustrating novel attacks, and work introducing mitigation techniques. To the best of our knowledge, we are unaware of any other work that focuses on exploiting or securing the resource management substrate.

Among the plethora of work considering side channel attacks exploiting IoT and mobile devices, Chen et al. discuss a number of attacks on real-time IoT systems [CHM18] focused on four main attack classes, namely, integrity violations due to malicious code injection, side-channel attacks, attacks on the communication channels, and Denial-of-Service attacks. Chen et al. [CBI18] discuss how IoT devices at home can reveal sensitive information about the users. Mitigation techniques include systems like CellPot [LLB14], a novel honeypot for cellular networks to detect threats and provide defence against malicious mobile devices in the network. Naveed et al. [ACS17] introduce the idea of using Physical Unclonable Functions (PUFs) to provide secure authentication protocols for IoT devices in an ad-hoc network. Other techniques such as Post-Quantum public cryptography systems to secure edge devices [LCG18], Chen and Xu design a collaborative based edge cloud system where social trust networks are built for managing security risks among the cloud edges due to collaboration [CX17].

5.5 Conclusions

In this work, we enumerated and characterized the side-channel vulnerabilities in distributed IoT systems that arise as a result of network adaptation. We showed how an attacker can falsify utilization statistics for the purpose of manipulating a network's resource allocation. Furthermore, we illustrated how adaptation can leak a system's valuable meta-information about an IoT node. We then describe ADAPT^2, a framework for distributed IoT systems that can attest device utilization statistics and obfuscate system meta-information. To detect false utilization statistics, ADAPT^2 uses a state estimation model of the computation workloads for each device to determine if there is a discrepancy in the device's reported statistics versus the device's expected statistics. ADAPT^2 can use dummy workloads to isolate suspicious devices and perform the attestation procedure. To obfuscate meta-information, ADAPT^2 can either issue dummy workloads or workloads that are not latency-sensitive onto idle nodes to obfuscate the node-activation characteristics of the distributed IoT system.

CHAPTER 6

Conclusion

My research explores the notion of enabling an adaptive system runtime spanning the cloud, fog, and edge. At its core, the focus is on coalescing the inherent heterogeneity of cloud-edge resource pool composition to support the future of IoT. I seek to lift embedded systems into the general purpose resource cluster to bridge the gap between sensor networks and distributed systems. This ambitious goal has yet to be attained; the efforts I have taken aim to bring our community one step closer, building upon those that have paved the way in an attempt to address the three unsolved problems that constitute the essential elements of a robust autonomic cloud-edge system as described in Section 1.1. DDFlow ameliorates the complexity of coalescing hardware specific implementations for unified management of distributed applications. Portkey aims to address the unique geo-distribution of high-volume data producers and consumers at the edge. EdgeRM builds upon these notions with a layer of abstraction offering a central entity supporting dynamic resource aggregation and task management. Finally, ADAPT^2 defines autonomic computing principles that ensure adaption itself does not create system vulnerability. The precepts of these compositional elements realize aspects of the overarching vision for a self-aware system. By observing system state, predicting the impact of proposed adaptations, and efficiently executing reconfiguration, the self-aware adaptive system can tune itself to the specifics of a deployment environment, thereby providing the performance of a custom solution while maintaining the flexibility of a generic framework.

6.0.1 Future Directions and Limitations

There are a number of limitations and directions for future research that have been revealed from my efforts.

General-Purpose Machine Learning for the Edge. Machine learning is undeniably one of the most important technological advancements in recent years, and its nascent application to IoT systems continues to require research innovation. WebAssembly, and by extension EdgeRM, have yet to consider a unified approach to machine learning applications. Its practical implementation requires many important considerations. First, a number of competing ML frameworks are vying for the spotlight, and compiling these systems to WebAssembly, particularly such that it can fit on an embedded device, is in-and-of-itself an arduous challenge. Second, each device offers certain resource limits, and balancing model selection across the set of available devices provides key tradeoffs in performance and latency. Third, even if a highly resource-efficient model is developed and selected, how should it be deployed over an embedded device? Perhaps this requires the exploration of a WebAssembly System Interface extension for ML. How should these systems take advantage of potential hardware accelerators, such as an Edge TPU [Cas19]? Ultimately, can a unified ML model representation in WebAssembly be deployed over varying hardware and varying ML frameworks? Is the same applicable to model training, such as in federated learning?

Energy as a First-Class Resource. Energy and power are resources currently missing from EdgeRM, yet are key factors in determining whether an application should run on a particular resource constrained device or be scheduled elsewhere in the system. Extending the scheduling profiler to infer energy and/or power consumption of an application, and incorporating that information into an energy-aware scheduler is a difficult challenge. While trends continue to indicate that on-device compute is more advantageous than incurring wireless radio, identifying those dynamic thresholds to optimize for energy is a current limitation and necessary direction of future research.

Extending to the Most Resource Constrained. No matter how efficient a resource management agent implementation is, there will seemingly always be a class of embedded device that falls below the minimum threshold. This is particularly revealed when considering the extension of EdgeRM to battery-less and energy-harvesting devices. As researchers continue to design system tooling to execute over minimum-energy devices, EdgeRM agent implementations and scheduling decisions will likely require revisiting to target these systems.

Intelligent RL Scheduling over Dynamic Systems. While my initial prospectus proposed a complete approach to scheduling, developments in EdgeRM revealed that a one-size-fits-all scheduler is neither practical nor possible in the cloud-edge computing domain. This led to the introduction of a lower level of abstraction in an attempt to capture all possible deployment scenarios, thereby seeking to enable EdgeRM as a foundational platform for future scheduling solutions. As such, a logical next-step in research is the exploration of various means to intelligently schedule subsets of cloud-edge applications in a self-aware fashion. Similar to how the Sensor MapReduce framework continuously monitors the set of available sensing resources to potentially reschedule Map tasks, a reinforcement learning-based approach to task scheduling can potentially allow developers to specify optimization metrics that dictate how distributed applications are managed under particular circumstances.

A Novel WebAssembly Runtime for IoT. Embedded systems are often developed for specific platforms or devices; for example, our initial implementation of the embedded resource management agent sits atop Zephyr. While the messaging protocol and architecture may remain the same, expanding the compatibility of EdgeRM across embedded platforms will undoubtedly aid in its adoption. Similarly, the WAMR runtime serves as an initial, but potentially suboptimal solution to executing WebAssembly modules on resource constrained devices. Recent attention has been placed on developing a runtime custom tailored to the embedded device setting. While retargeting our distributed resource manager to a new WebAssembly runtime is not a complex task, it certainly requires engineering effort and serves as a valuable complement to this research. Finally, I personally ascribe to the notion that the best research is driven by real-world problems, as opposed to developing ideas from within an isolated bubble. As frameworks begin to build atop EdgeRM, they are likely to expose unsolved mysteries that will be certain to inspire the best innovations.

A Serverless Execution Model. Finally, I'd like to touch upon a logical next-step in exploring a serverless runtime and system architecture for cloud-edge applications. As the execution model of containerized applications continues to develop, it is undoubtedly leading toward finer-grain modular tasks that can quickly bind and unbind to resources. With the introduction of Portkey as an adaptive state management solution, serverless functions-at-the-edge developed in a stateless fashion can truly be realized. The key factor preceding serverless IoT is a stable and production-ready runtime for WebAssembly on embedded devices. I am confident that this solution is imminently approaching, and truly believe that serverless is the future of distributed application design. Our Sensor MapReduce framework is one such implementation.

6.0.2 The Breadth of My Research

While my thesis is focused on exploring system self-awareness, the full scope of my research covered tangentially related efforts that serve to compliment this notion. Although these projects may not be best served within the core thesis, I am compelled to briefly touch upon their design and utility:

Self-awareness without sufficient explanation leads to opaque decision-making. Despite the ubiquity and unrivaled performance of deep neural networks in certain tasks, its black box nature often inhibits its application in practice. Moreover, DNN explanation methods have typically focused on tasks that operate over image, audio, or textual data, leaving sensing applications relatively ignored. To this end, we developed ExMatchina [JNC20], an explanation-by-example framework that leverages training data to provide high-quality explanations for DNN inferences. Our user study indicated that explanation-by-example was not only applicable to sensing tasks, but was preferred over superimposition explanation methods for image, audio, and sensing data. To support user or admin-facing deployments that require explainable decisions yet desire the incorporation of deep neural network models, we provide ExMatchina as an open-source explanation-by-example framework.

Time awareness is too often overlooked when designing distributed systems. One-way online profiling and shared synchronization depend upon a shared notion of time, however relying on a device's system clock can unexpectedly degrade application performance. As part of designing reliable profiling, we observed notable deviations in the smartphone system clock, leading to an observation study and analysis of the discrepancy in "synchronized" system clocks on smartphones. At the current observable scale of timing discrepancies, on the order of seconds, applications fusing data across smartphones can expect dramatic performance drops with up to an order-of-magnitude increase in error rates. To this end, we designed GoodClock, a system clock replacement for smartphones, and introduced data augmentation techniques that increase the resilience of machine learning models to timing discrepancies [SNA19, SNA20].

In conjunction with the availability of low-cost instrumentation of sensors and actuators, there has been an increase in adversarial sensing, where nefarious entities invade upon the privacy of individuals occupying a third-party space. Self-awareness in this context takes a different approach, revealing the existence of unknown wireless sensors to ensure a more trusted physical environment. We introduced SnoopDog [SGN21], a generalized framework to detect clandestine wireless sensors monitoring a user in a private space. SnoopDog uses Granger Causality to determine if the values of a trusted ground-truth sensor cause patterns in Wi-Fi traffic of other nearby devices. Once detected, a localization approach systematically reduces the search space to find the hidden sensor using the notion of sensor coverage, thus allowing users to identify and localize any bugs spying on them.

6.0.3 Final Thoughts

I have truly been blessed to be given the opportunity to spend the last decade of my life studying, exploring, and creating knowledge at UCLA. My research culminates in this vision for an adaptive, self-aware system, able to comprehend its state to work towards one that is superior. My proposal for edge computing is not unlike the aspirations I set for my own life, navigating the path to fulfill the most complete expression of my creation.

REFERENCES

[ABC15] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al. "The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing." *Proceedings of the VLDB Endowment*, **8**(12):1792–1803, 2015.

[ACS17] Muhammad Naveed Aman, Kee Chaing Chua, and Biplab Sikdar. "Physically secure mutual authentication for IoT." In *IEEE Conference on Dependable and Secure Computing*, pp. 310–317. IEEE, 2017.

[ADJ10] Sharad Agarwal, John Dunagan, Navendu Jain, Stefan Saroiu, Alec Wolman, and Harbinder Bhogan. "Volley: Automated Data Placement for Geo-distributed Cloud Services." In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, Berkeley, CA, USA, 2010. USENIX Association.

[AKC17] Michael P Andersen, John Kolb, Kaifei Chen, David E. Culler, and Randy Katz. "Democratizing Authority in the Built Environment." In *Proceedings of the 4th ACM International Conference on Systems for Energy-Efficient Built Environments*, BuildSys '17, New York, NY, USA, 2017. Association for Computing Machinery.

[Ama20] Amazon Web Services, Inc. "AWS IoT Greengrass - Amazon Web Services." https://aws.amazon.com/greengrass/, 2020.

[AMO15] Ramia Babiker Mohammed Abdelrahman, Amin Babiker A Mustafa, and Ashraf A Osman. "A Comparison between IEEE 802.11 a, b, g, n and ac Standards." *IOSR Journal of Computer Engineering (IOSR-JEC)*, **17**:26–29, 2015.

[AMP16] Aleksandar Antonić, Martina Marjanović, Krešimir Pripužić, and Ivana Podnar Žarko. "A mobile crowd sensing ecosystem enabled by CUPUS: Cloud-based publish/subscribe middleware for the Internet of Things." *Future Generation Computer Systems*, **56**:607 – 622, 2016.

[AQE20] Fawad Ahmad, Hang Qiu, Ray Eells, Fan Bai, and Ramesh Govindan. "CarMap: Fast 3D Feature Map Updates for Automobiles." In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pp. 1063–1081, 2020.

[ARS18] Muthukaruppan Annamalai, Kaushik Ravichandran, Harish Srinivas, Igor Zinkovsky, Luning Pan, Tony Savor, David Nagle, and Michael Stumm. "Sharding the shards: managing datastore locality at scale with Akkio." In *13th {USENIX}*

Symposium on Operating Systems Design and Implementation ({OSDI} 18), pp. 445–460, 2018.

[AS18] Irshad Ahmed Abbasi and Adnan Shahid Khan. “A review of vehicle to vehicle communication protocols for VANETs in the urban environment.” *future internet*, **10**(2):14, 2018.

[AT14] Masoud Saeida Ardekani and Douglas B Terry. “A self-configurable geo-replicated cloud storage system.” In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pp. 367–381, 2014.

[AXF12] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. “Workload analysis of a large-scale key-value store.” In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, pp. 53–64, 2012.

[Azu20] Microsoft Azure. “Azure IoT – Internet of Things Platform: Microsoft Azure.” https://azure.microsoft.com/en-us/overview/iot/, 2020.

[BBC17] Tristan Braud, Farshid Hassani Bijarbooneh, Dimitris Chatzopoulos, and Pan Hui. “Future networking challenges: The case of mobile augmented reality.” In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pp. 1796–1807. IEEE, 2017.

[BBL14] Lorenzo Bracciale, Marco Bonola, Pierpaolo Loreti, Giuseppe Bianchi, Raul Amici, and Antonello Rabuffi. “CRAWDAD dataset roma/taxi (v. 2014-07-17).” Downloaded from https://crawdad.org/roma/taxi/20140717, July 2014.

[BCL08] Niels Brouwers, Peter Corke, and Koen Langendoen. “Darjeeling, a Java Compatible Virtual Machine for Microcontrollers.” In *Proceedings of the ACM/IFIP/USENIX Middleware 08 Conference Companion*, Companion ’08, p. 18–23, New York, NY, USA, 2008. Association for Computing Machinery.

[BCP20] Marc Brooker, Tao Chen, and Fan Ping. “Millions of Tiny Databases.” In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pp. 463–478, 2020.

[BEG19] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloe Kiddon, Jakub Konečný, Stefano Mazzocchi, H Brendan McMahan, et al. “Towards federated learning at scale: System design.” *arXiv preprint arXiv:1902.01046*, 2019.

[BGO16] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. “Borg, Omega, and Kubernetes: Lessons Learned from Three Container-Management Systems over a Decade.” *Queue*, **14**(1):70–93, January 2016.

[BHS07] Athanassios Boulis, Chih-Chieh Han, Roy Shea, and Mani B Srivastava. "SensorWare: Programming sensor networks beyond code update and querying." *Pervasive and mobile computing*, **3**(4):386–412, 2007.

[BRS08] Ivan Baev, Rajmohan Rajaraman, and Chaitanya Swamy. "Approximation algorithms for data placement problems." *SIAM Journal on Computing*, **38**(4):1411–1429, 2008.

[BTD06] Subir Biswas, Raymond Tatchikou, and Francois Dion. "Vehicle-to-vehicle wireless communication protocols for enhancing highway traffic safety." *IEEE communications magazine*, **44**(1):74–82, 2006.

[Byt20a] Bytecode Alliance. "WASM Micro Runtime." https://github.com/bytecodealliance/wasm-micro-runtime, 2020.

[Byt20b] Bytecode Alliance. "WebAssembly System Interface." https://wasi.dev/, 2020.

[BZ17] Paolo Bellavista and Alessandro Zanni. "Feasibility of fog computing deployment based on docker containerization over raspberrypi." In *Proceedings of the 18th international conference on distributed computing and networking*, pp. 1–10, 2017.

[Cas19] Stephen Cass. "Taking AI to the edge: Google's TPU now comes in a maker-friendly package." *IEEE Spectrum*, **56**(5):16–17, 2019.

[CBC10] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. "MAUI: making smartphones last longer with code offload." In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pp. 49–62, 2010.

[CBI18] Dong Chen, Phuthipong Bovornkeeratiroj, David Irwin, and Prashant Shenoy. "Private Memoirs of IoT Devices: Safeguarding User Privacy in the IoT Era." In *Proceedings of the 38th IEEE International Conference on Distributed Computing Systems (ICDCS'18)*, 2018.

[CBM17] Sergio Correia, Azzedine Boukerche, and Rodolfo I Meneguette. "An architecture for hierarchical software-defined vehicular networks." *IEEE Communications Magazine*, **55**(7):80–86, 2017.

[CHM18] Chien-Ying Chen, Monowar Hasan, and Sibin Mohan. "Securing real-time internet-of-things." *Sensors*, **18**(12):4356, 2018.

[CKM19] Maxime C Cohen, Philipp W Keller, Vahab Mirrokni, and Morteza Zadimoghaddam. "Overcommitment in Cloud Services: Bin Packing with Chance Constraints." *Management Science*, 2019.

[CKY20] Eunhee Chang, Hyun Taek Kim, and Byounghyun Yoo. “Virtual reality sickness: a review of causes and measurements.” *International Journal of Human–Computer Interaction*, **36**(17):1658–1682, 2020.

[CLB18] Aakanksha Chowdhery, Marco Levorato, Igor Burago, and Sabur Baidya. “Urban iot edge analytics.” In *Fog computing in the internet of things*, pp. 101–120. Springer, 2018.

[Col18] Louis Columbus. “10 Charts That Will Challenge Your Perspective Of IoT’s Growth.”, 2018.

[Cro19] Steve Crowe. “How drones & robots helped extinguish Notre Dame fire.” https://www.therobotreport.com/how-drones-robots-helped-extinguish-notre-dame-fire/, 2019. Accessed: 2020-12-26.

[CST10] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. “Benchmarking cloud serving systems with YCSB.” In *Proceedings of the 1st ACM symposium on Cloud computing*, pp. 143–154. ACM, 2010.

[CX17] Lixing Chen and Jie Xu. “Socially trusted collaborative edge computing in ultra dense networks.” In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, p. 9. ACM, 2017.

[DCB19] Miguel Diogo, Bruno Cabral, and Jorge Bernardino. “Consistency Models of NoSQL Databases.” *Future Internet*, **11**(2):43, 2019.

[DK18] Daniel Dinu and Ilya Kizhvatov. “EM Analysis in the IoT Context: Lessons Learned from an Attack on Thread.” *IACR Transactions on Cryptographic Hardware and Embedded Systems*, **2018**(1):73–97, 2018.

[DL20] Bradley Denby and Brandon Lucia. “Orbital Edge Computing: Nanosatellite Constellations as a New Class of Computer System.” In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS ’20, p. 939–954, New York, NY, USA, 2020. Association for Computing Machinery.

[Dor20] Dormando. “memcached - a distributed memory object caching system.” https://memcached.org/, 2020. Accessed: 2020-12-26.

[EKN17] Milan Erdelj, Michał Król, and Enrico Natalizio. “Wireless sensor networks and multi-UAV systems for natural disaster management.” *Computer Networks*, **124**:72–86, 2017.

[EN16] Milan Erdelj and Enrico Natalizio. “UAV-assisted disaster management: Applications and open issues.” In *2016 international conference on computing, networking and communications (ICNC)*, pp. 1–5. IEEE, 2016.

[EPB18] Mohammed S Elbamby, Cristina Perfecto, Mehdi Bennis, and Klaus Doppler. "Toward low-latency and ultra-reliable virtual reality." *IEEE Network*, **32**(2):78–84, 2018.

[ESK05] Nasif Ekiz, Tara Salih, Sibel Kucukoner, and Kemal Fidanboylu. "An overview of handoff techniques in cellular networks." *International journal of information technology*, **2**(3):132–136, 2005.

[FAB15] Ramon R Fontes, Samira Afzal, Samuel HB Brito, Mateus AS Santos, and Christian Esteve Rothenberg. "Mininet-WiFi: Emulating software-defined wireless networks." In *Network and Service Management (CNSM), 2015 11th International Conference on*, pp. 384–389. IEEE, 2015.

[FEB03] Maydene Fisher, Jon Ellis, and Jonathan Bruce. *JDBC API tutorial and reference*. Addison-Wesley Professional, 2003.

[FH08] Marco Fiore and Jérôme Härri. "The networking shape of vehicular mobility." In *Proceedings of the 9th ACM international symposium on Mobile ad hoc networking and computing*, pp. 261–272, 2008.

[FJP16] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. "Security analysis of emerging smart home applications." In *2016 IEEE symposium on security and privacy (SP)*, pp. 636–654. IEEE, 2016.

[FKD15] Denzil Ferreira, Vassilis Kostakos, and Anind K Dey. "AWARE: mobile context instrumentation framework." *Frontiers in ICT*, **2**:6, 2015.

[FMC11] Nicolas Falliere, Liam O Murchu, and Eric Chien. "W32. stuxnet dossier." *White paper, Symantec Corp., Security Response*, **5**(6):29, 2011.

[Fou18] JS Foundation. "Node-RED.", 2018.

[FSL16] David Formby, Preethi Srinivasan, Andrew Leonard, Jonathan Rogers, and Raheem A Beyah. "Who's in Control of Your Control System? Device Fingerprinting for Cyber-Physical Systems." In *NDSS*, 2016.

[Fut19] Futuriom. "5G, IoT and Edge Compute Trends: Technology challenges, solution, and forecasts for the low-latency edge." ""http://wan.velocloud.com/rs/098-RBR-178/images/Analyst%20Report%205G%2C%20IoT%20and%20Edge%20Compute%20Trends%20with%20Futuriom.pdf"", 2019. Accessed: 2021-05-15.

[GBL15] Nam Ky Giang, Michael Blackstock, Rodger Lea, and Victor CM Leung. "Developing iot applications in the fog: A distributed dataflow approach." In *2015 5th International Conference on the Internet of Things (IOT)*, pp. 155–162. IEEE, 2015.

[GDT18] Edward Gan, Jialin Ding, Kai Sheng Tai, Vatsal Sharan, and Peter Bailis. "Moment-based quantile sketches for efficient high cardinality aggregation queries." *Proceedings of the VLDB Endowment*, **11**(11):1647–1660, 2018.

[Geo20] Damien George. "MicroPython." https://micropython.org/, 2020.

[GGG05] Ramakrishna Gummadi, Omprakash Gnawali, and Ramesh Govindan. "Macro-programming wireless sensor networks using Kairos." In *International Conference on Distributed Computing in Sensor Systems*, pp. 126–140. Springer, 2005.

[GND17] Alex Glikson, Stefan Nastic, and Schahram Dustdar. "Deviceless edge computing: extending serverless computing to the edge of the network." In *Proceedings of the 10th ACM International Systems and Storage Conference*, pp. 1–1, 2017.

[Goo18] Google. "Google Vision Kit.", 2018.

[GoS20] GoSquared. "gosquared/redis-clustr: Redis Cluster client for Node.js." https://github.com/gosquared/redis-clustr, 2020. Accessed: 2020-12-26.

[GR18] Harshit Gupta and Umakishore Ramachandran. "Fogstore: A geo-distributed key-value store guaranteeing low latency for strongly consistent access." In *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems*, pp. 148–159. ACM, 2018.

[GRB12] Nico Golde, Kevin Redon, and Ravishankar Borgaonkar. "Weaponizing Femtocells: The Effect of Rogue Devices on Mobile Telecommunications." In *NDSS*, 2012.

[GZG15] Lin Gu, Deze Zeng, Song Guo, Ahmed Barnawi, and Yong Xiang. "Cost efficient resource management in fog computing supported medical cyber-physical system." *IEEE Transactions on Emerging Topics in Computing*, **5**(1):108–119, 2015.

[HAH15] Karim Habak, Mostafa Ammar, Khaled A Harras, and Ellen Zegura. "Femto clouds: Leveraging mobile devices to provide cloud service at the edge." In *2015 IEEE 8th international conference on cloud computing*, pp. 9–16. IEEE, 2015.

[HCT18] Hamed Haddadi, Vassilis Christophides, Renata Teixeira, Kenjiro Cho, Shigeya Suzuki, and Adrian Perrig. "SIOTOME: An edge-ISP collaborative architecture for IoT security." *Proc. IoTSec*, 2018.

[HHS17] Syed Monowar Hossain, Timothy Hnat, Nazir Saleheen, Nusrat Jahan Nasrin, Joseph Noor, Bo-Jhang Ho, Tyson Condie, Mani Srivastava, and Santosh Kumar. "mCerebrum: A Mobile Sensing Software Platform for Development and Validation of Digital Biomarkers and Interventions." In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, p. 7. ACM, 2017.

[HKS05] Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava. "A dynamic operating system for sensor nodes." In *Proceedings of the 3rd international conference on Mobile systems, applications, and services*, pp. 163–176, 2005.

[HKZ11] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center." In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, p. 295–308, USA, 2011. USENIX Association.

[HLR13] Kirak Hong, David Lillethun, Umakishore Ramachandran, Beate Ottenwälder, and Boris Koldehofe. "Mobile fog: A programming model for large-scale applications on the internet of things." In *Proceedings of the second ACM SIGCOMM workshop on Mobile cloud computing*, pp. 15–20. ACM, 2013.

[HRS17a] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. "Bringing the Web up to Speed with WebAssembly." *SIGPLAN Not.*, **52**(6):185–200, June 2017.

[HRS17b] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. "Bringing the web up to speed with WebAssembly." In *ACM SIGPLAN Notices*, volume 52, pp. 185–200. ACM, 2017.

[Hua20] Wenyong Huang. "WASM Micro Runtime Performance." https://github.com/bytecodealliance/wasm-micro-runtime/wiki/Performance, 2020.

[HV19] Cheol-Ho Hong and Blesson Varghese. "Resource management in fog/edge computing: a survey on architectures, infrastructure, and algorithms." *ACM Computing Surveys (CSUR)*, **52**(5):1–37, 2019.

[Hyp20] Hypriot. "Docker Pirates ARMed with explosive stuff." https://blog.hypriot.com/, 2020.

[IBG16] Valérie Issarny, Georgios Bouloukakis, Nikolaos Georgantas, and Benjamin Billet. "Revisiting service-oriented architecture for the IoT: a middleware perspective." In *International conference on service-oriented computing*, pp. 3–17. Springer, 2016.

[JGS11] Sushil Jajodia, Anup K Ghosh, Vipin Swarup, Cliff Wang, and X Sean Wang. *Moving target defense: creating asymmetric uncertainty for cyber threats*, volume 54. Springer Science & Business Media, 2011.

[JNC20] Jeya Vikranth Jeyakumar, Joseph Noor, Yu-Hsi Cheng, Luis Garcia, and Mani Srivastava. "How can i explain this to you? an empirical study of deep neural network explanation methods." *Advances in Neural Information Processing Systems*, 2020.

[JS14] Rafał S Jurecki and Tomasz L Stańczyk. "Driver reaction time to lateral entering pedestrian in a simulated crash traffic situation." *Transportation research part F: traffic psychology and behaviour*, **27**:22–36, 2014.

[JSF18] Xiaolin Jiang, Hossein Shokri-Ghadikolaei, Gabor Fodor, Eytan Modiano, Zhibo Pang, Michele Zorzi, and Carlo Fischione. "Low-latency networking: Where latency lurks and how to tame it." *Proceedings of the IEEE*, **107**(2):280–306, 2018.

[KGM07] Nupur Kothari, Ramakrishna Gummadi, Todd Millstein, and Ramesh Govindan. "Reliable and efficient programming abstractions for wireless sensor networks." In *ACM SIGPLAN Notices*, volume 42, pp. 200–210. ACM, 2007.

[KK03] Srinivas Kashyap and Samir Khuller. "Algorithms for non-uniform size data placement on parallel disks." In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pp. 265–276. Springer, 2003.

[KKS17] Constantinos Kolias, Georgios Kambourakis, Angelos Stavrou, and Jeffrey Voas. "DDoS in the IoT: Mirai and other botnets." *Computer*, **50**(7):80–84, 2017.

[KLA19] Qingkai Kong, Qin Lv, and Richard M Allen. "Earthquake early warning and beyond: Systems challenges in smartphone-based seismic network." In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*, pp. 57–62, 2019.

[KMY16] Jakub Konečnỳ, H Brendan McMahan, Felix X Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. "Federated learning: Strategies for improving communication efficiency." *arXiv preprint arXiv:1610.05492*, 2016.

[Kub] Kubernetes. "Device Plugins." https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/device-plugins/.

[KWA03] Rajnish Kumar, Matthew Wolenetz, Bikash Agarwalla, JunSuk Shin, Phillip Hutto, Arnab Paul, and Umakishore Ramachandran. "DFuse: A framework for distributed data fusion." In *Proceedings of the 1st international conference on Embedded networked sensor systems*, pp. 114–125. ACM, 2003.

[LC02] Philip Levis and David Culler. "Maté: A Tiny Virtual Machine for Sensor Networks." In *Architectural Support for Programming Languages and Operating Systems*, 2002.

[LCG17] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. "Multiprogramming a 64kB Computer Safely and Efficiently." In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, p. 234–251, New York, NY, USA, 2017. Association for Computing Machinery.

[LCG18] Zhe Liu, Kim-Kwang Raymond Choo, and Johann Grossschadl. "Securing edge devices in the post-quantum Internet of Things using lattice-based cryptography." *IEEE Communications Magazine*, **56**(2):158–162, 2018.

[LDP15] Jake Longo, Elke De Mulder, Dan Page, and Michael Tunstall. "SoC it to EM: electromagnetic side-channel attacks on a complex system-on-chip." In *International Workshop on Cryptographic Hardware and Embedded Systems*, pp. 620–640. Springer, 2015.

[LJW07] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. "Multi-probe LSH: efficient indexing for high-dimensional similarity search." In *Proceedings of the 33rd international conference on Very large data bases*, pp. 950–961. VLDB Endowment, 2007.

[LLB14] Steffen Liebergeld, Matthias Lange, and Ravishankar Borgaonkar. "Cellpot: a concept for next generation cellular network honeypots." *Internet Society*, pp. 1–6, 2014.

[LM03] Ting Liu and Margaret Martonosi. "Impala: A Middleware System for Managing Autonomic, Parallel Sensor Systems." *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP*, **38**, 06 2003.

[LM10] Avinash Lakshman and Prashant Malik. "Cassandra: a decentralized structured storage system." *ACM SIGOPS Operating Systems Review*, **44**(2):35–40, 2010.

[LMP05] Philip Levis, Samuel Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, et al. "TinyOS: An operating system for sensor networks." In *Ambient intelligence*, pp. 115–148. Springer, 2005.

[LWB16] Peng Liu, Dale Willis, and Suman Banerjee. "Paradrop: Enabling lightweight multi-tenancy at the network's extreme edge." In *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, pp. 1–13. IEEE, 2016.

[LZH17] Gierad Laput, Yang Zhang, and Chris Harrison. "Synthetic sensors: Towards general-purpose sensing." In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pp. 3986–3999, 2017.

[MBG15] Sarah E Minson, Benjamin A Brooks, Craig L Glennie, Jessica R Murray, John O Langbein, Susan E Owen, Thomas H Heaton, Robert A Iannucci, and Darren L Hauser. “Crowdsourced earthquake early warning.” *Science advances*, **1**(3):e1500036, 2015.

[MBL18] Seyed Hossein Mortazavi, Bharath Balasubramanian, Eyal de Lara, and Shankaranarayanan Puzhavakath Narayanan. “Toward session consistency for the edge.” In *USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18)*, 2018.

[MBS17] Amardeep Mehta, Rami Baddour, Fredrik Svensson, Harald Gustafsson, and Erik Elmroth. “Calvin Constrained—A Framework for IoT Applications in Heterogeneous Environments.” In *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*, pp. 1063–1073. IEEE, 2017.

[MCD18] Roberto Morabito, Vittorio Cozzolino, Aaron Yi Ding, Nicklas Beijar, and Jorg Ott. “Consolidate IoT edge computing with lightweight virtualization.” *IEEE Network*, **32**(1):102–111, 2018.

[MFH05] Samuel R Madden, Michael J Franklin, Joseph M Hellerstein, and Wei Hong. “TinyDB: an acquisitional query processing system for sensor networks.” *ACM Transactions on database systems (TODS)*, **30**(1):122–173, 2005.

[MGC10] Yilin Mo, Emanuele Garone, Alessandro Casavola, and Bruno Sinopoli. “False data injection attacks against state estimation in wireless sensor networks.” In *Decision and Control (CDC), 2010 49th IEEE Conference on*, pp. 5967–5972. IEEE, 2010.

[MGS17] Ruben Mayer, Harshit Gupta, Enrique Saurez, and Umakishore Ramachandran. “Fogstore: Toward a distributed data store for fog computing.” In *2017 IEEE Fog World Congress (FWC)*, pp. 1–6. IEEE, 2017.

[min20] mininet. “mininet/mininet: Emulator for rapid prototyping of software-defined networks.” https://github.com/mininet/mininet, 2020. Accessed: 2020-12-26.

[Mit05] Chris Mitchell. *Trusted computing*, volume 6. Iet, 2005.

[MJ17] Arsalan Mosenia and Niraj K Jha. “A comprehensive study of security of internet-of-things.” *IEEE Transactions on Emerging Topics in Computing*, **5**(4):586–602, 2017.

[Mon20] MongoDB. “Sharding – MongoDB Manual.” https://docs.mongodb.com/manual/sharding/, 2020. Accessed: 2020-12-26.

[Mor16] Kief Morris. *Infrastructure as code: managing servers in the cloud.* ” O’Reilly Media, Inc.”, 2016.

[MP11] Luca Mottola and Gian Pietro Picco. "Programming wireless sensor networks: Fundamental concepts and state of the art." *ACM Computing Surveys (CSUR)*, **43**(3):19, 2011.

[MRL19] Charles Masson, Jee E Rim, and Homin K Lee. "DDSketch: A fast and fully-mergeable quantile sketch with relative-error guarantees." *Proceedings of the VLDB Endowment*, **12**(12):2195–2205, 2019.

[MRS18] Sumit Maheshwari, Dipankar Raychaudhuri, Ivan Seskar, and Francesco Bronzino. "Scalability and performance evaluation of edge cloud systems for latency constrained applications." In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pp. 286–299. IEEE, 2018.

[MV15] Charlie Miller and Chris Valasek. "Remote exploitation of an unaltered passenger vehicle." *Black Hat USA*, **2015**:91, 2015.

[MZP08] Ratul Mahajan, Ming Zhang, Lindsey Poole, and Vivek S Pai. "Uncovering Performance Differences Among Backbone ISPs with Netdiff." In *NSDI*, pp. 205–218, 2008.

[NAE18] Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. "Dpaxos: Managing data closer to users for low-latency and mobile applications." In *Proceedings of the 2018 International Conference on Management of Data*, pp. 1221–1236, 2018.

[NF13] Diala Naboulsi and Marco Fiore. "On the instantaneous topology of a large-scale urban vehicular network: the cologne case." In *Proceedings of the fourteenth ACM international symposium on Mobile ad hoc networking and computing*, pp. 167–176, 2013.

[NMW07] Ryan Newton, Greg Morrisett, and Matt Welsh. "The regiment macroprogramming system." In *2007 6th International Symposium on Information Processing in Sensor Networks*, pp. 489–498. IEEE, 2007.

[NSB18] Piotr Nowakowski, Krzysztof Szwarc, and Urszula Boryczka. "Vehicle route planning in e-waste mobile collection on demand supported by artificial intelligence algorithms." *Transportation Research Part D: Transport and Environment*, **63**:1–22, 2018.

[NSG19] Joseph Noor, Sandeep Singh Sandha, Luis Garcia, and Mani Srivastava. "DDF LOW visualized declarative programming for heterogeneous IoT networks on Heliot testbed platform: demo abstract." In *Proceedings of the International Conference on Internet of Things Design and Implementation*, pp. 287–288. ACM, 2019.

[NTG19] Joseph Noor, Hsiao-Yun Tseng, Luis Garcia, and Mani Srivastava. "DDFlow: visualized declarative programming for heterogeneous IoT networks." In *Proceedings of the International Conference on Internet of Things Design and Implementation*, pp. 172–177. ACM, 2019.

[NW04] Ryan Newton and Matt Welsh. "Region streams: Functional macroprogramming for sensor networks." In *Proceeedings of the 1st international workshop on Data management for sensor networks: in conjunction with VLDB 2004*, pp. 78–87. ACM, 2004.

[NW05] Ryan Newton, Matt Welsh, et al. "Building up to macroprogramming: an intermediate language for sensor networks." In *IPSN 2005. Fourth International Symposium on Information Processing in Sensor Networks, 2005.*, pp. 37–44. IEEE, 2005.

[OCW17] Kwangsung Oh, Abhishek Chandra, and Jon Weissman. "TripS: Automated multi-tiered data placement in a geo-distributed cloud environment." In *Proceedings of the 10th ACM International Systems and Storage Conference*, p. 12. ACM, 2017.

[OKL15] Gihwan Oh, Sangchul Kim, Sang-Won Lee, and Bongki Moon. "SQLite Optimization with Phase Change Memory for Mobile Applications." *Proc. VLDB Endow.*, **8**(12):1454–1465, August 2015.

[Par20] Particle Industries. "Particle Platform." https://www.particle.io/iot-platform/, 2020.

[Pay19] Ben Paynter. "Could a network of sensors give first responders more time to control wildfires?" https://www.fastcompany.com/90424260/could-a-network-of-sensors-give-first-responders-more-time-to-control-wildfires, 2019. Accessed: 2020-12-28.

[PNS18] Petar Popovski, Jimmy J Nielsen, Cedomir Stefanovic, Elisabeth De Carvalho, Erik Strom, Kasper F Trillingsgaard, Alexandru-Sabin Bana, Dong Min Kim, Radoslaw Kotaba, Jihong Park, et al. "Wireless access for ultra-reliable low-latency communication: Principles and building blocks." *Ieee Network*, **32**(2):16–23, 2018.

[PPR11] Swapnil Patil, Milo Polte, Kai Ren, Wittawat Tantisiriroj, Lin Xiao, Julio López, Garth Gibson, Adam Fuchs, and Billie Rinaldi. "YCSB++: benchmarking and performance debugging advanced features in scalable table stores." In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, p. 9. ACM, 2011.

[PPW20] Gregor Peach, Runyu Pan, Zhuoyi Wu, Gabriel Parmer, Christopher Haster, and Ludmila Cherkasova. "eWASM: Practical Software Fault Isolation for Reliable

Embedded Devices." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **39**(11):3492–3505, 2020.

[QAB18] Hang Qiu, Fawad Ahmad, Fan Bai, Marco Gruteser, and Ramesh Govindan. "Avr: Augmented vehicular reality." In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, pp. 81–95, 2018.

[Ras20] Raspberry Pi Foundation. "FAQs - Raspberry Pi Documentation." https://www.raspberrypi.org/documentation/faqs/, 2020.

[RBB11] A. Rowe, M. E. Berges, G. Bhatia, E. Goldman, R. Rajkumar, J. H. Garrett, J. M. F. Moura, and L. Soibelman. "Sensor Andrew: Large-scale campus-wide sensing and actuation." *IBM Journal of Research and Development*, **55**(1.2):6:1–6:14, 2011.

[Red20a] RedisLabs. "Pub/Sub - Redis." https://redis.io/topics/pubsub, 2020. Accessed: 2020-12-28.

[Red20b] RedisLabs. "Redis Cluster Specification." https://redis.io/topics/cluster-spec, 2020. Accessed: 2020-12-26.

[Red20c] RedisLabs. "Redis Commands - CLUSTER SETSLOT." https://redis.io/commands/cluster-setslot, 2020. Accessed: 2020-12-26.

[Red20d] RedisLabs. "Redis on ARM." https://redis.io/topics/ARM, 2020. Accessed: 2020-12-26.

[REF16] Ashish Rauniyar, Paal Engelstad, Boning Feng, et al. "Crowdsourcing-based disaster management using fog computing in internet of things paradigm." In *2016 IEEE 2nd international conference on collaboration and internet computing (CIC)*, pp. 490–494. IEEE, 2016.

[RF18] Joseph Redmon and Ali Farhadi. "YOLOv3: An Incremental Improvement." *arXiv*, 2018.

[RG18] Arun Ravindran and Anjus George. "An Edge Datastore Architecture For Latency-Critical Distributed Machine Vision Applications." In *USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18)*, 2018.

[RHL15] Laurynas Riliskis, James Hong, and Philip Levis. "Ravel: Programming IoT Applications as Distributed Models, Views, and Controllers." In *Proceedings of the 2015 International Workshop on Internet of Things towards Applications*, IoT-App '15, p. 1–6, New York, NY, USA, 2015. Association for Computing Machinery.

[RJK05] Shariq Rizvi, Shawn R Jeffery, Sailesh Krishnamurthy, Michael J Franklin, Nathan Burkhart, Anil Edakkunni, and Linus Liang. “Events on the edge.” In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pp. 885–887, 2005.

[RLM18] Rodrigo Roman, Javier Lopez, and Masahiro Mambo. “Mobile edge computing, fog et al.: A survey and analysis of security threats and challenges.” *Future Generation Computer Systems*, **78**:680–698, 2018.

[ROC14] Mathew Ryden, Kwangsung Oh, Abhishek Chandra, and Jon Weissman. “Nebula: Distributed edge cloud for data intensive computing.” In *2014 IEEE International Conference on Cloud Engineering*, pp. 57–66. IEEE, 2014.

[roc17] “RocksDB.” ”http://rocksdb.org/”, 2017.

[SBS17] Guni Sharon, Stephen D Boyles, and Peter Stone. “Intersection management protocol for mixed autonomous and human-operated vehicles.” *Transportation Research Part C: Emerging Technologies (Under submission TRC-D-17-00857)*, 2017.

[SCS15] Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. “C3: Cutting tail latency in cloud data stores via adaptive replica selection.” In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pp. 513–527, 2015.

[SCZ16] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. “Edge computing: Vision and challenges.” *IEEE Internet of Things Journal*, **3**(5):637–646, 2016.

[SDL18] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. “Airsim: High-fidelity visual and physical simulation for autonomous vehicles.” In *Field and service robotics*, pp. 621–635. Springer, 2018.

[SGN21] Akash Deep Singh, Luis Garcia, Joseph Noor, and Mani Srivastava. “I Always Feel Like Somebody’s Sensing Me! A Framework to Detect, Identify, and Localize Clandestine Wireless Sensors.” In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[Shy20] Volodymyr Shymanskyy. “WASM3 Performance.” https://github.com/wasm3/wasm3/blob/master/docs/Performance.md, 2020.

[SKK16] Alon Shalita, Brian Karrer, Igor Kabiljo, Arun Sharma, Alessandro Presta, Aaron Adcock, Herald Kllapi, and Michael Stumm. “Social hash: an assignment framework for optimizing distributed systems operations on social networks.” In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, pp. 455–468, 2016.

[SLA12] Cong Shi, Vasileios Lakafosis, Mostafa H Ammar, and Ellen W Zegura. "Serendipity: Enabling remote computing among intermittently connected mobile devices." In *Proceedings of the thirteenth ACM international symposium on Mobile Ad Hoc Networking and Computing*, pp. 145–154, 2012.

[SMA04] Minho Shin, Arunesh Mishra, and William A Arbaugh. "Improving the latency of 802.11 hand-offs using neighbor graphs." In *Proceedings of the 2nd international conference on Mobile systems, applications, and services*, pp. 70–83, 2004.

[SNA19] Sandeep Singh Sandha, Joseph Noor, Fatima M Anwar, and Mani Srivastava. "Exploiting smartphone peripherals for precise time synchronization." In *2019 IEEE International Symposium on Precision Clock Synchronization for Measurement, Control, and Communication (ISPCS)*, pp. 1–6. IEEE, 2019.

[SNA20] Sandeep Singh Sandha, Joseph Noor, Fatima M Anwar, and Mani Srivastava. "Time awareness in deep learning-based multimodal fusion across smartphone platforms." In *2020 IEEE/ACM Fifth International Conference on Internet-of-Things Design and Implementation (IoTDI)*, pp. 149–156. IEEE, 2020.

[Sol20] SolidIT. "DB-Engines Ranking - popularity ranking of database management systems." https://db-engines.com/en/ranking, 2020. Accessed: 2020-12-26.

[SS04] Ahmad-Reza Sadeghi and Christian Stüble. "Property-based Attestation for Computing Platforms: Caring About Properties, Not Mechanisms." In *Proceedings of the 2004 Workshop on New Security Paradigms*. ACM, 2004.

[SST16] Shachar Siboni, Asaf Shabtai, Nils O Tippenhauer, Jemin Lee, and Yuval Elovici. "Advanced security testbed framework for wearable IoT devices." *ACM Transactions on Internet Technology (TOIT)*, **16**(4):26, 2016.

[STA19] Mauro Salazar, Matthew Tsao, Izabel Aguiar, Maximilian Schiffer, and Marco Pavone. "A congestion-aware routing scheme for autonomous mobility-on-demand systems." In *2019 18th European Control Conference (ECC)*, pp. 3040–3046. IEEE, 2019.

[SWW15] Ahmad-Reza Sadeghi, Christian Wachsmann, and Michael Waidner. "Security and privacy challenges in industrial internet of things." In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2015.

[SZL16] Kaixin Sui, Mengyu Zhou, Dapeng Liu, Minghua Ma, Dan Pei, Youjian Zhao, Zimu Li, and Thomas Moscibroda. "Characterizing and improving wifi latency in large-scale operational networks." In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, pp. 347–360. ACM, 2016.

[The16] The Hybrid Group. "JavaScript Robotics, By Your Command." https://cylonjs.com/, 2016.

[The20] The Linux Foundation. "Zephyr OS." https://www.zephyrproject.org/, 2020.

[TMK17] Surat Teerapittayanon, Bradley McDanel, and Hsiang-Tsung Kung. "Distributed deep neural networks over the cloud, the edge and end devices." In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pp. 328–339. IEEE, 2017.

[TRV98] Nishith D Tripathi, Jeffrey H Reed, and Hugh F VanLandinoham. "Handoff in cellular systems." *IEEE personal communications*, **5**(6):26–37, 1998.

[TS18] Hsiao-Yun Tseng and Sandeep Singh Sandha. "nesl/Heliot.", Dec 2018.

[TTL05] Douglas Thain, Todd Tannenbaum, and Miron Livny. "Distributed computing in practice: the Condor experience." *Concurrency and computation: practice and experience*, **17**(2-4):323–356, 2005.

[TWB20] Animesh Trivedi, Lin Wang, Henri Bal, and Alexandru Iosup. "Sharing and Caring of Data at the Edge." In *3rd {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 20)*, 2020.

[UHG20] Muhammed Uluyol, Anthony Huang, Ayush Goel, Mosharaf Chowdhury, and Harsha V Madhyastha. "Near-Optimal Latency Versus Cost Tradeoffs in Geo-Distributed Storage." In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pp. 157–180, 2020.

[Van18] Jo Van Bulck et al. "Foreshadow: Extracting the Keys to the Intel {SGX} Kingdom with Transient Out-of-Order Execution." In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pp. 991–1008, 2018.

[Var96] Yehuda Vardi. "Network tomography: Estimating source-destination traffic intensities from link data." *Journal of the American statistical association*, **91**(433):365–377, 1996.

[VMD13] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. "Apache Hadoop YARN: Yet Another Resource Negotiator." In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[Vou14] Panagiotis Vouzis. "What is Network Tomography?" https://netbeez.net/blog/network-tomography/, 2014. Accessed: 2020-12-26.

[WGM20] Xiangpeng Wan, Hakim Ghazzai, and Yehia Massoud. "A Generic Data-Driven Recommendation System for Large-Scale Regular and Ride-Hailing Taxi Services." *Electronics*, **9**(4):648, 2020.

[Win20] Wind River Systems Inc. https://www.windriver.com/, 2020.

[WM04] Matt Welsh and Geoffrey Mainland. "Programming Sensor Networks Using Abstract Regions." In *NSDI*, volume 4, pp. 3–3, 2004.

[WRB18] Jonathan Woetzel, Jaana Remes, Brodie Boland, Katrina Lv, Suveer Sinha, Gernot Strube, John Means, Jonathan Law, Andrés Cadena, and Valerie von der Tann. "Smart Cities: Digital Solutions for a more Livable Future.", 2018.

[WVM17] Nan Wang, Blesson Varghese, Michail Matthaiou, and Dimitrios S Nikolopoulos. "ENORM: A framework for edge node resource management." *IEEE transactions on services computing*, 2017.

[WZZ17] Shuo Wang, Xing Zhang, Yan Zhang, Lin Wang, Juwo Yang, and Wenbo Wang. "A survey on mobile edge networks: Convergence of computing, caching and communications." *Ieee Access*, **5**:6757–6779, 2017.

[YLC19] Qiang Yang, Yang Liu, Tianjian Chen, and Yongxin Tong. "Federated machine learning: Concept and applications." *ACM Transactions on Intelligent Systems and Technology (TIST)*, **10**(2):1–19, 2019.

[YPK14] Jinhong Yang, Hyojin Park, Yongrok Kim, and Jun Kyun Choi. "IoT gadget control on wireless AP at home." In *Consumer Communications and Networking Conference (CCNC), 2014 IEEE 11th*, pp. 1148–1149. IEEE, 2014.

[ZJR12] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. "Cross-VM side channels and their use to extract private keys." In *Proceedings of the 2012 ACM conference on Computer and communications security*, pp. 305–316. ACM, 2012.

[ZWH19] Guangxu Zhu, Yong Wang, and Kaibin Huang. "Broadband analog aggregation for low-latency federated edge learning." *IEEE Transactions on Wireless Communications*, **19**(1):491–506, 2019.