University of California
Santa Barbara

Uniformly Programmable, Distributed, Reliable, Event-based Systems for Multi-Tier IoT Deployments

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Computer Science

by

Wei-Tsung Lin
UCSB Technical Report #2022-01

Committee in charge:

Professor Chandra Krintz, Co-Chair
Professor Rich Wolski, Co-Chair
Professor William Wang

June 2021

The Dissertation of Wei-Tsung Lin

UCSB Technical Report #2022-01 is approved.

Professor William Wang

Professor Rich Wolski, Committee Co-Chair

Professor Chandra Krintz, Committee Co-Chair

June 2021

Uniformly Programmable, Distributed, Reliable, Event-based Systems for Multi-Tier IoT Deployments

by

Wei-Tsung Lin

UCSB Technical Report #2022-01

Acknowledgements

My sincere gratitude goes to my advisors Chandra Krintz and Rich Wolski. They are the most amazing, supportive, and generous people I have ever met in my academic journey. Their contagious enthusiasm made me understand research is a thing of joy. When I needed it most, they provided enormous support to help me get through the challenging time. They are not only my academic advisors but also inspirational mentors. It is a great honor and privilege to work with them.

I thank William Wang for honoring me by being my dissertation committee member. His insightful feedback on my research made me able to better my research. I am also grateful to all the UCSB faculties who shared their knowledge and experiences with me and made me the researcher I am today. I also thank all the staffs who have helped me on my journey toward my doctoral degree.

I am thankful to my lab mates for being good friends and colleagues in the past years. I would also like to thank the friendships from the amazing people I met in Santa Barbara who always share the highs and lows in life with me.

I am truly grateful to my amazing wife. Her tremendous support helped me get through the difficult and uncertain times in the past years. She is the lighthouse of my life.

Finally, I thank my family for their unconditional love. I know it is difficult for my parents that their only son went abroad to pursue a career; nevertheless, they tolerate my absence and support me selflessly. They taught me to be a better person. Without them, I would not have been able to realize my dream.

Curriculum Vitæ

Wei-Tsung Lin
UCSB Technical Report #2022-01

Education

2021	Ph.D. in Computer Science (Expected), University of California, Santa Barbara.
2014	M.S. in Computer Science, National Tsing Hua University.
2012	B.S. in Computer Science, National Tsing Hua University.

Publications

A Programmable and Reliable Publish/Subscribe System for Multi-Tier IoT

W-T. Lin, R. Wolski, and C. Krintz

(in submission)

Data repair for Distributed, Event-based IoT Applications

W-T. Lin, F. Bakir, C. Krintz, R. Wolski, and M. Mock

ACM International Conference on Distributed and Event-Based Systems (DEBS), 2019.

Tracing Function Dependencies Across Clouds

W-T. Lin, C. Krintz, and R. Wolski

IEEE International Conference on Cloud Computing (CLOUD), 2018

Tracking Causal Order in AWS Lambda Applications

W-T. Lin, M. Zhang, C. Krintz, R. Wolski, Xiaogang Cai, Tongjun Li, and Weijin Xu

IEEE International Conference on Cloud Engineering (IC2E), 2018

CSPOT: Portable, Multi-scale Functions-as-a-Service for IoT

R. Wolski, C. Krintz, F. Bakir, G. George, and W-T. Lin

ACM Symposium on Edge Computing (SEC), 2019

CSPOT: A Serverless Platform of Things

R. Wolski, C. Krintz, and W-T. Lin

UCSB Technical Report 2018-01

Abstract

Uniformly Programmable, Distributed, Reliable, Event-based Systems for Multi-Tier IoT Deployments

by

Wei-Tsung Lin

UCSB Technical Report #2022-01

With the emergence of the Internet of Things (IoT), data-driven applications have become increasingly powerful. By leveraging a variety of IoT devices and computational resources, developers are able to implement applications to perform a wide range of tasks, from simple data analytics to complex machine learning. While resource-constrained devices and cloud computing have been widely studied in recent years, the rapidly evolving and unique requirements of IoT applications introduce a new set of challenges that inspire lots of discussions and research.

Modern IoT deployments often consist of "multi-tier" resources (combinations of distributed sensors, edge devices, and cloud systems). Developers must consider whether to execute computations in remote, resource-rich cloud instances or closer to the sensors and data (at the "edge" of the network) to achieve the desired application performance and operation cost. In addition, modern IoT devices are vastly diverse in hardware capability, network protocols, and reliability. It is increasingly challenging for applications to manage the huge number of components with different characteristics while ensuring scalability, availability, and reliability.

To address the above challenges of heterogeneity, programmability, and reliability, we investigate the design of an end-to-end application platform for multi-tier IoT deployment. Our system adopts a publish/subscribe pattern for data exchange and functions-

as-a-service programming model to enable event-driven computation. Moreover, it intrinsically integrates data replication using consensus protocol for availability and reliability as well as data and computation locality control. To facilitate analysis, debugging, and reasoning about the highly concurrent applications in the system, we embed a new method to capture the causality of events across distributed multi-tier deployments using a distributed log. Finally, we develop tools that use event causality to repair corrupted data and computation on the fly.

We empirically evaluate our system using real-world IoT applications. The results show that our approach significantly simplifies application deployment while improving latency and throughput performance. In addition, it provides a novel way for developers to reason about interdependencies within IoT applications and facilitate failure recovery. The result, we believe, enables us to approach application design with a new event-driven and locality-aware paradigm that is critical for multi-tier IoT environments.

Contents

Chapter 1

Introduction

The Internet of Things (IoT) is a rapidly emerging set of technologies in which ordinary physical objects in our environment are equipped with Internet connectivity, sensing, control, and computing capabilities. With its ever-growing popularity and increasing capability, IoT is fueling remarkable innovations in different fields. Combined with technology advances such as the reduced cost of computational power, hardware manufacturing, next-generation network connectivity, and different computing models, IoT has quickly transformed the way we approach and utilize data.

Because of their ubiquitous nature and increasing capability, IoT devices allow us to collect a wider range of data and leverage the data in new ways. Software engineers and data scientists use IoT deployments of different scales, from single-device applications to distributed applications that span a wide area. For example, consumer-grade wearable health monitors are used to collect health metrics; smart home appliances collect indoor temperature and humidity; RFID-equipped devices deployed for security systems. At the other end of the spectrum, factories use IoT to automate the manufacturing process; cameras deployed across a city are used to coordinate transportation; field sensors help with agriculture optimization by collecting environmental data. These applications show

that IoT is deeply embedded in our everyday lives and will continue to be.

While greatly enhancing our quality of life, these innovative IoT applications increasingly demand significant amounts of storage and computing power. Large-scale applications such as smart city and smart agriculture collect and generate a considerable volume of data at great velocity. Even smaller-scale applications like object recognition rely on computationally intensive machine learning. For developers to utilize data generated by IoT efficiently, the infrastructure must be able to store and process the data efficiently. However, because of IoT's heterogeneous and remote nature and design options, we are at a point where the needs of storage and computational capacities of applications have exceeded what IoT devices and sensors can offer. Thus, the industry and academia have been putting significant effort into incorporating extra resources required by IoT applications.

One solution to the IoT computation and data capacity problem is cloud computing. Cloud computing has become the de facto standard computing paradigm to build and deploy network-enabled applications. Since its birth, cloud computing has realized the vision of providing storage and computation as a utility, on-demand. Cloud vendors offer virtual machines, blob storage, and databases for consumers to use and purchase on a pay-per-use basis. Because consumers need not set up and manage their own infrastructures, cloud computing enables them to save considerable labor and upfront cost. The cloud computing paradigm not only makes computational resources more accessible but also greatly reduces the complexity of application development and deployment.

While migrating IoT applications to clouds seems promising, due to the unique characteristics of IoT devices and applications, it is a challenging task to implement IoT applications and facilitate high performance using existing cloud computing platforms. To achieve ubiquitous computing, IoT deployments span a wide range of environments that cover different geographic locations with different network qualities. For such en-

vironments, cloud datacenters are accessible only via long haul, high latency networks, which many times are only intermittently connected. As a result, IoT endpoints increasingly experience high latency or unstable connectivity to cloud datacenters. High latency translates into degraded user experiences and reduced scale. Moreover, for applications that rely on real-time data and decision-making, e.g., military, security, and autonomous vehicle systems, latency degradation can render the application unusable. Furthermore, because of the volume of data that IoT devices generate, offloading data and computation to cloud datacenters consumes considerable bandwidth. IoT environments often employ less reliable networks and rely on radio links for the last mile connection. As such, IoT bandwidth requirements for moving data and the potential for intermittent, unstable connectivity between IoT deployments and the cloud, make the cloud-direct approach impractical for many deployments.

In addition to network connectivity, the power-constrained nature of IoT devices also poses a challenge. The long-haul connection to clouds translates to extra power consumption and will significantly shorten the battery lives of IoT devices. Moreover, IoT radios consume significant power to move data, which increases as the square of the distance. While much research focuses on reducing the power consumption of IoT devices, additional methods are required since so many devices are battery-powered. Thus, the power consumption of IoT devices in cloud computing remains a critical issue.

In order to address these challenges, a new computing paradigm, "edge computing," has emerged. By moving operations and corresponding storage and computational resources "near" (in terms of network latency) where data is sensed, generated, or used (i.e., at the "edge" of the network), edge computing improves application response time and saves bandwidth and battery consumption. Since the conception of edge computing, IoT applications have come to employ and depend on edge devices even for small-scale deployments, to facilitate local storage and computation offloading.

Edge computing bridges the gap between traditional clouds and IoT devices; nevertheless, it is not a direct replacement for clouds. One of the disadvantages of edge clouds is that they are less capable than their cloud counterparts. While most edge clouds can be horizontally scaled, it is less cost-efficient to do so. The same problem goes for storage. Because of the enormous volume of data generated by the IoT applications, it is often infeasible to store all the data on an edge cloud without a significant amount of cost. Furthermore, due to the hardware, deployment environments, and maintenance difficulties associated with managing edge systems, edge clouds are generally more prone to failure and less reliable than datacenter-based clouds. Thus, developers currently bear the burden of implementing reliable protocols and systems services to ensure the reliability of data and computation, which can be complex and error-prone.

We refer to distributed systems that combine and couple IoT devices and sensors with edge and cloud computing as "multi-tier" computing. Multi-tier computing is an exciting new paradigm in which compute and data resources are vastly heterogeneous and distributed across geographic locations. This tiered hierarchy enhances the flexibility of application development and deployment and facilitates cost-effective and adaptive resource usage for IoT deployments.

However, the heterogeneity and complexity of these deployments imposes a significant challenge and burden on application implementation and deployment. IoT devices implement a wide range of hardware and software and employ a wide range of program interfaces, protocols, libraries, and services. Multi-tier deployment adds connectivity and synchrony complications. Moreover, many devices operate on duty cycles (i.e., periodic sleep cycles) to conserve battery power. As such, popular programming models such as client/server and synchronous model which maintain connections between participants may not be feasible in many IoT deployments. Thus, it is critical that we consider and design new programming models that account for and hide this heterogeneity and

that simplify application programming, deployment, and robustness management across multi-tier IoT systems.

Finally, because of the number of devices involved in multi-tier IoT applications, hardware failures and misconfigurations are inevitable. When failures or misconfiguration occur, applications will yield invalid results. In distributed multi-tier environments where application components are tightly coupled, Any invalid result generated by a simple error can potentially propagate and corrupt the downstream computations that use it. In real-world scenarios, it is seldom practical to repair such corruptions in production systems because it requires stopping the system, isolating the errors, repairing and replacing the incorrect data with the correct one. Doing so can render the system unavailable and cause data or even financial loss. Thus, we must consider the application reliability and failure recoverability as part of multi-tier IoT systems design.

Given these limitations and challenges surrounding the state-of-art in multi-tier IoT, the goal of our work is to develop and answer the following thesis question.

Can we build a uniformly programmable, distributed, reliable, event-based system for multi-tier IoT deployments?

Specifically, we investigate the following topics in the IoT domain:

- an end-to-end platform for IoT application that is uniformly programmable and portable to all IoT tiers, reliable and resilient to common failures in IoT environments, and capable of supporting large-scale IoT applications,
- a framework to help us track event interdependencies within applications and facilitate debugging, analyzing, and reasoning about IoT applications, and
- a methodology to repair and recover from data corruptions caused by inevitable hardware and network failures.

To develop the above systems and answer the thesis question, we take a multi-pronged approach in which we first advance the state-of-the-art in data delivery and computing models for IoT. Multiple IoT deployments employ a publish/subscribe (pub/sub) protocol to address data delivery. Pub/sub systems implement a messaging pattern that provides a loose coupling of content producers and consumers. Unlike the traditional client-service paradigm, content producers (publishers) do not send data directly to consumers (subscribers). Instead, pub/sub systems rely on data brokers to filter and route data based on subscribers' interests.

While decoupling data publishers and subscribers enables scaling data delivery, this feature of pub/sub systems poses a challenge to locality-aware multi-tier IoT applications. The loose coupling of data and clients makes it difficult for developers to leverage the network topology to tune and optimize the application performance. Moreover, programmability, as mentioned earlier, remains a problem in pub/sub systems. Pub/sub systems typically do not support any "in-stream" data processing. To use pub/sub systems, developers must deploy data brokers at multiple tiers of environments for efficient data delivery. Also, they must use different languages and runtimes to implement application logic to react to delivered data.

Toward this end, we investigate the design and implementation of a portable and programmable pub/sub system called Canal. Canal is a scalable pub/sub system designed explicitly for locality-aware multi-tier IoT applications. Canal implements a lightweight append-only data structure for efficient and durable data storage. Once published to a topic, data persists in the data structure that shares the same format and interface across tiers. For reliability, Canal uses a replica consensus protocol to replicate the data structure. In order to eliminate the bottlenecks in data brokers, we integrate distributed hash table (DHT) into Canal for efficient service discovery and topic lookup. The use of distributed hash table also decouples the replica placement

from DHT topology and allows developers to explore different replica placements and optimize applications for locality and different objects.

To provide a uniform programming interface across tiers and fully realize the potential of the data-driven nature of IoT applications, we integrate a "function-as-a-service" (FaaS) programming model into Canal. Programmers implement applications as a series of stateless and short-running functions that react to the event. Unlike public cloud FaaS offerings, Canal does not rely on external service as the trigger of function invocations. Instead, the distributed append-only data structure servers the purpose of a log comprising a series of publishing events. When new data is published, Canal runtime checks the subscription list of the topic and invokes the functions that subscribe to the topic.

While Canal addresses the programmability, reliability, and data distribution issues associated with multi-tier, the vast number and types of data and their associated events make applications difficult to reason about. Although cloud vendors provide debugging and performance monitoring tools, they are limited in that they only provide basic sampling-based dependency tracing and do not work in wide-area settings. Moreover, the vendor lock-in makes these services not portable and compatible with other clouds.

To facilitate application monitoring and debugging, we next focus on new ways of leveraging casual dependencies between events. We first specialize such dependency tracking for the de facto standard FaaS platform for cloud-based systems via a new system called GammaRay. GammaRay traces the event dependency among Lambda functions and other AWS cloud services without programmer intervention. It does so by dynamically injecting causal information into every cloud SDK call and recording them to a synchronized database. Once stored in the database, the causal information is consumed by an offline analysis tool to reconstruct the event dependency graph. GammaRay has three improvements of the vendor's own tracing service AWS X-Ray. First, it captures

both direct and indirect causal dependency among events from different cloud services. Second, it does not rely on sampling; hence it captures all events. Third, it works in a wide-area setting that spans multiple regions.

To further reduce the overhead and make our tracing framework more versatile and portable to other infrastructures, we next leverage the concepts that underly GammaRay to design a cloud-agnostic distributed event tracing system called Lowgo. Lowgo uses a geo-replicated distributed log to record all the events in their global causal order. Lowgo reduces the event recording latency by having multiple instances located in all tiers and regions of applications. When a cloud operation such as database read/write and function invocation occurs, a corresponding event is written to a Lowgo instance co-located in the network where the event takes place. Instances located distributedly cooperate in the background and use the causal information embedded in the events to reconstruct the event dependency, thus eliminating the overhead introduced to the application to maintain causal order locally.

Finally, to address the data corruption problem, we develop a data repair framework called Sans-Souci. Sans-Souci implements the dependency tracing technique we develop in Lowgo. We integrate Sans-Souci into our event-triggered FaaS platform to automatically record all the events and their causal dependencies using a distributed event log. When an anomaly or corruption occurs, Sans-Souci uses the incorrect data as anchors to identify and isolate all the causally related events. Then, developers can fix the misconfiguration, replace the malfunctioning device, or provide correct application input to "repair" the incorrect value. Once the root of corruption is fixed, Sans-Souci uses the event replay technique to replay all the dependent downstream computations using the corrected input. When the repair is concluded, the corrected application timeline replaces the corrupted history dynamically without disrupting the runtime system.

In summary, we answer the thesis question via the following new research innova-

tions. We present a uniformly programmable publish/subscribe system called Canal to address the programmability, data delivery, and reliability problems in multi-tier IoT applications. We decouple the replica placement from the network topology to allows developers to optimize locality-aware IoT applications. To facilitate application monitoring, debugging, and analysis, we develop event dependency tracing frameworks to reason about causalities in applications. Finally, we utilize the causality information we capture and design a data repair framework for dynamic IoT application environments.

The remaining sections of this dissertation are structured as followed. Chapter 2 overviews the background for the technology stacks currently used in modern IoT applications and their limitations. We examine the advantages and limitations of these technologies and discuss the current state-of-art systems and services that address these challenges. In Chapter 3, we introduce CSPOT, a portable FaaS computing platform that serves as the basis for our research. Chapter 4 details our design and implementation of a programmable pub/sub system for IoT, Canal, including how we integrate scalable lookup and locality-aware replication into the event-triggered computing architecture. Chapter 5 and Chapter 6 present our event tracing systems and how they compare to the available public service that provides similar functions. Chapter 7 illustrates how we retroactively and dynamically repair data corruption in distributed and multi-tier environments. Finally, we conclude our research and discuss the plans of future work in Chapter 8.

Attributions

- Parts of Chapter 3 include text and data from *CSPOT: Portable, Multi-scale Functions-as-a-Service for IoT* by R. Wolski, C. Krintz, F. Bakir, G. George, and W-T. Lin and it was published in ACM Symposium on Edge Computing (SEC),

2019.

- Parts of Chapter 5 include text and data from *Tracking Causal Order in AWS Lambda Applications* by W-T. Lin, M. Zhang, C. Krintz, R. Wolski, Xiaogang Cai, Tongjun Li, and Weijin Xu and it was published in IEEE International Conference on Cloud Engineering (IC2E), 2018.

- Parts of Chapter 6 include text and data from *Tracing Function Dependencies Across Clouds* by W-T. Lin, C. Krintz, and R. Wolski and it was published in IEEE International Conference on Cloud Computing (CLOUD), 2018.

- Parts of Chapter 7 include text and data from *Data repair for Distributed, Event-based IoT Applications* by W-T. Lin, F. Bakir, C. Krintz, R. Wolski, and M. Mock and it was published in ACM International Conference on Distributed and Event-Based Systems (DEBS), 2019.

Chapter 2

Background

IoT is used in a vast diversity of modern applications and has been one of the primary foci of multiple research fields in recent years. According to reports, the number of connected IoT devices worldwide is expected to reach 29.3 billion, more than three times the global population by 2023[1]. We can find the use of IoT in differnt applications in our everyday lives. Applications such as wearable devices for healthcare[2, 3, 4], traffic control[5, 6], warehouse management[7], smart city[8, 9, 10], and smart agriculture[11, 12, 13, 14] significantly improve our quality of life.

Due to their unique nature, IoT applications require disparate programming paradigms, message protocols, data distribution, and execution models to implement their functionality. The complexity of IoT also requires developers to employ a stack of technologies, including cloud computing, publish/subscribe systems, and data replication. While these disciplines have been widely studied, the fast growing popularity of IoT breathes interest into these established fields and as inspired many innovations. Moreover, new computing paradigms such as edge computing and functions-as-a-service have emerged and are proving critical to IoT. These area also have attracted much research interest. In this chapter, we provide background on the enabling technologies of IoT.

2.1 Cloud and Edge Computing

Although most IoT devices are equipped with storage and computing capabilities, IoT applications often rely on external resources to support the lacking computational resources. Cloud computing is the de-facto solution for large-scale distributed applications in the past decades. It realizes the vision of providing computational resources at large-scale on-demand. Botta et al. analyze and discuss the need and challenges to integrating IoT and cloud computing[15].

The major cloud vendors all provide numerous cloud computing services available to the public. Some notable examples are Amazon Web Service[16], Google Cloud Platform[17], and Microsoft Azure[18]. Enterprises and individuals can also opt to deploy their private cloud infrastructures using commercial or open-source solutions such as Eucalyptus[19], OpenStack[20], and Apache CloudStack[21]. Cuervo et al. and Chun et al. present MAUI[22] and CloneCloud[23] that demonstrate the possibility to offload mobile computation to the cloud.

While cloud computing provides diverse and virtually unlimited computational resources to customers, merging IoT applications to clouds is a complicated task that needs careful consideration. As Bonomi et al.[24] and Zhang et al.[25] point out, the unique requirements and natures of IoT raise compatibility issues to bring it to the cloud. While cloud providers have IoT cloud services available, such as Google Cloud IoT[26], AWS IoT[27], and Azure IoT[28], these integrated services are still cloud-centric at the core. Therefore, they do not fully address important issues such as high latency and power consumption caused by the long-haul connection from IoT endpoints to cloud data centers.

In an attempt to address the issues caused by the centralized approach of cloud computing, a new computing paradigm, "edge computing," has emerged to offer proximal

computing capabilities *near* (in terms of network latency) where data is produced and data driven actuation and control are needed [29, 30, 31]. Instead of relying on remote data centers for the extra computational resources, edge computing deploys servers at smaller scales at the "edge" of the network, to extend the capabilities of IoT devices and to provide better responsiveness. Among these paradigms, Satyanarayanan et al. pioneered one of the newer edge paradigm, "cloudlet"[32, 33]. Cloudlets are clusters based on virtual machines with the same functionality and capacity as clouds but at a smaller scale. Ismail et al. evaluate using Docker container[34] as the backbone technology of cloudlets. The concept of cloudlet took off as a "datacenter-in-a-box" solution that directly replaces clouds on edge. However, its ability to bridge the gap between IoT application requirements and the distant cloud data centers was quickly realized to be beneficial in wireless metropolitan area networks (WMAN). Jia et al. study the cloudlet placement and mobile user allocation to the cloudlets in a wireless metropolitan area network[35]. Zhang et al. present an optimal cloudlet offloading algorithm for mobile clients in environments with intermittent connectivity[36]. Cozzolino et al.[37], Ma et al.[38], and Wei et al.[39] present several different approaches to offload computation in edge environments.

Because of the rapid growth of interest in combining IoT and edge computing, cloud vendors have begun to offer integrated IoT edge solutions. Using AWS Greengrass[40], developers can write AWS Lambda[41] codes and execute them locally on edge devices. AWS Greengrass relies on AWS cloud services for storage, managing, and analytics. Azure IoT Edge[42] shares a similar functionality in that it allows workload deployed at the edge using Docker container[43]. These IoT devices and runtimes can be connected and managed using vendor-specific services such as AWS IoT Core[27] and Azure IoT Hub[44], and Google Cloud IoT Core[45]. Das et al. provide a detailed benchmark of AWS and Azure edge offerings[46]. However, these solutions are locked in the vendor

ecosystems and developers must use cloud-specific services to implement the applications.

The biggest challenge of IoT edge, however, is its capacity and reliability. Edge clouds provide better latency and response time at the proximities of IoT end-users, but compared to traditional clouds, they are less capable of storage and computational power. The current IoT edge offerings only provide a subset of cloud functionality at the edge, and the ability to schedule computation according to resource availability network conditions is very primitive. Moreover, the edge devices are more prone to hardware failure and have less ideal network connections.

Much prior work has attempted to address these issues and bridge the gap between traditional datacenter-based clouds and edge clouds. Chang et al.[47] demonstrate the advantages of using both edge cloud and traditional clouds to implement applications. Suryavansh et al.[48] evaluate the performance of different computing models (e.g., mobile-only, edge-only, cloud-only, and hybrid setting.) Vasconcelos et al.[49] propose an algorithm to decide the best infrastructure to use based on cost and network. Jain and Tata[50] propose an annotation-based approach to split applications into fragments that can be deployed to nodes in edge and clouds.

2.2 Functions-as-a-Service (FaaS)

The scale and complexities of cloud applications have grown exponentially since cloud computing was born. As the data volume and the number of involved components grow large, the traditional client-server programming model is too inefficient to support such applications at scale. Firstly, the architecture does not scale well because of the high volume of requests and data being delivered. Secondly, supporting large numbers of concurrent servers imposes significant overhead. Lastly, it is challenging to load balance, provision, and make these systems reliable.

Functions-as-a-Service (FaaS), also known as serverless computing, is a popular cloud programming paradigm that attempts to address these issues. FaaS abstracts away the hardware resources and operating systems, providing developers a simple runtime to execute functions [51, 52, 53]. FaaS applications comprise several stateless and short-running functions written in platform-specific languages. In addition, FaaS uses an event-triggered programming model such that developers can configure functions to be triggered by certain events, such as HTTP requests, database updates, data storage uploads, or new messages in message queues. FaaS platform typically use stateless container[54, 55, 43] for function executions, therefore enables a lightweight and faster function startup.

Compared to the traditional programming models, Faas provides several benefits. First, since FaaS platforms take care of resource provisioning and managing, developers need not provision application-specific resources, making the development more straightforward and effortless. For the same reason, it eliminates the huge upfront cost to set up infrastructure. Moreover, because the functions are stateless and invoked dynamically, it saves the massive overhead of maintaining a centralized configuration and synchronizing all the servers.

Amazon Web Services (AWS) released the first commercially available FaaS called AWS Lambda[56]. AWS Lambda is tightly integrated into the AWS ecosystem. Since the Lambda functions are stateless and cannot persist any state beyond their lifecycles to store computation results and any data, applications must rely on AWS services such as S3 and DynamoDB. Following the success of AWS Lambda, other cloud vendors also provide FaaS additional to their cloud offerings, Few examples are Google Cloud Function[57], Azure Functions[58], and IBM Cloud Functions[59]. In addition to public cloud offering, other companies and open-source communities also have FaaS frameworks that allow enterprises and individuals to host their own FaaS platforms. Such frameworks include OpenFaas[60], OpenWhisk[61], Kubeless[62], Knative[63], and Fission[64].

Kritikos and Skrzypek[65], Mohanty et al.[66], and Palade et al.[67] review and evaluate the popular FaaS platforms. The fast-growing popularity of FaaS leads to cloud vendors' attempts to bring it to the edge. Currently, most commercial edge computing platforms integrate FaaS functionality. For instance, AWS Greengrass uses AWS Lambda as the function runtime.

2.3 Publish/Subscribe Systems

Because of the volume and velocity of data generated by IoT, point-to-point and multicasting messaging approaches have become insufficient to support such large-scale applications. One solution to facilitate distributed data delivery is the publish/subscribe (pub/sub) pattern. In pub/sub systems, content producers (publishers) are decoupled from data consumers (subscribers). Interested parties express their interest in data by subscribing to specific properties (content-based) or categories (topic-based) of data. Instead of directly sending messages to receivers, publishers tag data with properties or topics and publish it to message channels (often implemented as queues.) A data broker retrieves the data from channels and routes it to the subscribers based on the subscriptions when new data is published.

While the use of pub/sub systems in IoT environments has been the focus of recent research, the systems have been established and studied extensively. Among all the network protocols, MQTT[68] is the standard choice for IoT. MQTT is a topic-based, pub/sub protocol designed for message exchange between devices. MQTT employs a centralized design with which publishers and subscribers connect via a data broker using TCP/IP. A single broker server routes data from publishers to subscribers based on topic interest. MQTT supports multiple quality of service (QoS) levels, including at most once (level 0), at least once (level 1), and exactly once (2) delivery guarantee. Because of its

lightweight design, MQTT has been the de-facto protocol of choice for many IoT systems. In fact, all major cloud providers (e.g., AWS, Azure, IBM Watson) expose their services through MQTT. MQTT-SN[69] is an extension of MQTT specifically for use by sensor networks. In order to reduce power consumption for messaging, MQTT-SN decreases message payload and topic name size and uses UDP to transfer messages.

Although the pub/sub pattern used in MQTT is scalable, MQTT relies on data brokers for message routing, and it can become a bottleneck. There are data broker systems such as Apache ActiveMQ[70] and RabbitMQ[71] that support clustering of MQTT brokers for better scalability. MQTT-ST an extension to MQTT to interconnect a group of workers to implement a more available broker service.[72]

At the alternative end of the spectrum, Apache Kafka[73] provides scalable, cloud-based data stream processing in an attempt to maximize throughput. Kafka partitions and replicates topics to tolerate faults and balance load. In-memory data is periodically flushed to disk for persistence. Kafka is unique in that it uses a pull model to stream data. In this model, data brokers are stateless while consumers maintain their own read progress and pull data from topics. Facebook Scribe[74] and Apache Flume[75] are similar systems that manage log data. Unlike Kafka, they rely on external data storage, e.g., HDFS[76], for persistence. Hermes[77] and Scribe[78] are peer-to-peer pub/sub systems. They use a distributed hash table to cluster brokers and handle more data streams. P2S[79] is similar but uses Paxos[80] to replicate broker data for fault tolerance and availability.

While decoupling of data publishers and subscribers makes it easier to tackle the implementation and scaling of IoT applications, it also introduces challenges to application deployment and optimization. Because IoT applications are locality-aware, developers often leverage the network topology to adjust the data plan and control plan of applications for desired performance properties. However, the pub/sub pattern abstracts away

the network address from messaging process. This abstraction, while enables scalability, also prohibits developers from optimizing applications with data and computation locality. Happ and Wolisz[81] and Mishra and Kertesz[82] discuss the current limitations of the pub/sub pattern for IoT.

2.4 Fault Tolerance and Replication

In a vastly large distributed system where there can be more than hundreds of connected components, failures will inevitably happen. In order to achieve fault tolerance and better availability, distributed systems often use replication to make redundant copies of unreliable components and compose a reliable service. When a few replicas fail, data can still be retrieved as long as there is still a working replica that has valid data.

It is difficult to maintain the strong consistency of replicas in a distributed system because of netowkr delay[83]. State machine replication (SMR) is a classic strategy to ensure multiple distributed nodes observe the same application progression in the same order. In state machine replication, each replica is treated as a state machine that applies some operations and performs state transition based on its current state and operation input. Since network communication is unreliable in distributed systems, it is possible for replicas to receive different sets of operations because of data loss and out-of-order delivery. Therefore, consensus protocols are used to ensure that all replicas receive the same set of operations in the same globally consistent order.

While there are many consensus protocols proposed, there are only two that are dominantly used in production systems: Paxos and Raft. Proposed by Lamport[80] and published in 1998, Paxos is synonymous with the distributed consensus protocol. However, it is notorious for being hard to understand, implement, and reason about despite its popularity. Therefore, numerous attempts have been made to explain Paxos in

simpler forms, including Boichat et al.[84], Chandra et al.[85], Lampson[86], van Renesse and Altinbuken[87], and Lamport[88] himself. Paxos is widely used in distributed cloud systems, including Google Megastore[89], Spanner[90], Chubby[91], F1[92] and Borg[93].

Raft[94] is another attempt for distributed systems to reach consensus. The author keeps simplicity as one of the main goals when presenting the algorithm. While it is a relative newcomer in the game, it gains popularity quickly because of its ease of understanding and implementation. Many systems opt for Raft for the distributed primitivity, including Kubernetes[95], etcd[96], M3[97], Trillian[98], CockroachDB[99], and Redis[100]. Our work relies on Raft for data reliability. We detail the Raft algorithm and how we integrate it into our platform in Section 4.2.2.

In the research area of data replication, Cohen and Shenker[101], Ye and Chiu[102], and Brinkmann and Effert[103] explore the replication in distributed peer-to-peer systems. Gao et al.[104, 105] present different approaches that use replication to improve edge service performance and availability. Lin et al.[106] and Hao et al.[107] present how to effectively replicate database tables in resource-constrained edge environments. Finally, Saxena and Salem[108] and Xia et al.[109] propose novel algorithms for dynamic replica placement in two-tier, edge-cloud deployments.

Chapter 3

CSPOT: Portable Functions-as-a-Service Runtime for IoT

As the IoT technologies emerge and become more popular and easily accessible, application designers increasingly combine the IoT devices and the scale, power, and cost-effectiveness of cloud and edge computing to implement applications. However, at present, the heterogeneity of hardware, software, and APIs, asynchronous, highly scalable, dynamically changing, and geographically distributed nature of IoT-cloud applications, make their infrastructure complex and difficult to provision, program, and optimize for high performance, energy efficiency, and scale.

Functions-as-a-service (FaaS, also known as serverless programming) is a programming paradigm that aims to tackle such complexity. FaaS implements an event-triggered execution model. In FaaS, developers structure applications as several transient and stateless functions that react to specified events, such as database update, message delivery, HTTP request, API call, etc. They upload the functions to a FaaS platform

without setting up a cloud infrastructure, leaving the resources provisioning and managing runtime to the FaaS platform. FaaS enables a fine-grained pricing model, allowing developers to be charged with the services and runtime resources they use on-demand, saving a potentially huge upfront and operation cost. In terms of programmability, it also lets developers focus on application logic rather than IT operation.

The event-driven programming style, fine-grained costing, easy integration of cloud services make FaaS attractive to IoT developers. More and more developers use FaaS to host IoT applications because of the event-driven characteristics (devices trigger computation, communication, and storage events) of IoT applications. Due to the increasing interest in FaaS for IoT and to reduce the response latency associated with using the public cloud, public cloud providers have begun to offer restricted versions of their FaaS platforms for "edge" servers and devices, i.e., remote compute and storage resources located near and directly connected to IoT devices and sensors. Notable examples are AWS Greengrass[40] and Azure IoT edge[42].

However, while being a huge success in cloud computing, FaaS faces several challenges in edge environments because of the unique nature and requirements of IoT applications. Firstly, because of the FaaS execution model and the resource-constrained nature of IoT edge devices, IoT FaaS applications rely on external services for storage, data distribution, and complex computations such as machine learning. For instance, Amazon's IoT platform AWS Greengrass implements FaaS functionality at the edge, but it does not support data storage on edge devices. Therefore, applications need to rely on cloud services such as Amazon S3[110] for data storage. Moreover, while IoT edge platforms, including AWS Greengrass and Azure IoT Edge and alike, allows FaaS functions to execute on edge devices, they still connect to the cloud for device deployment, configuration, and management. Furthermore, while edge devices can work on their own when losing connection to the cloud, there is no transition policy to utilize the multiple tiers of edge-cloud

environments. Lastly, the public edge FaaS platforms have the disadvantage of vendor lock-in. To build applications on a specific platform, developers must use other services from the same vendor. Hence, they have less flexibility for application development and deployment.

In this chapter, we present CSPOT, a portable function-as-a-service runtime to facilitate robust and low-latency application development and deployment for multi-tier IoT environment – from sensors and edge devices to multi-scale clouds. In particular, CSPOT enables code portability across all scales of IoT, integrates a number of features that enable efficient, low-latency function execution across IoT tiers, application robustness, event logging for application debugging and analysis.

Specifically, we make the following contributions with CSPOT:

- We explore the feasibility of a single FaaS for IoT programming that makes multi-language program components portable across all platform scales – from microcontroller devices through the edge to the public clouds – in an IoT setting. Thus, in a CSPOT deployment, it is possible to move the computation to the data, or the data to the computation (whichever is most efficient) without recoding the application or its constituent software components.

- We extend FaaS with support for geo-distributed execution and dependency tracking. CSPOT storage is append-only (i.e., versioned) for data durability and robustness in a distributed execution setting. No CSPOT function can be triggered without the generation of some concomitant datum in persistent storage. In addition, CSPOT runtime system is log-based[111], and all log records carry an identifier specifying the event that triggered the function. These features together make it possible to determine the causal ordering of events efficiently and in a way that does not rely on statistical sampling. For scalable FaaS programs, particu-

larly in an IoT setting where there may be many hundreds or thousands of sensors and actuators operating asynchronously and, thereby triggering a wide variety of analytical and control computations, determining the root cause of an error or unexpected program state is critical. The CSPOT abstractions are designed with this capability in mind.

- We define new FaaS abstractions for messaging and locality. Automatically garbage-collected, append-only storage objects (called WooFs) are addressed by Universal Resource Identifiers (URIs) and any access to a WooF across locality regions (called namespaces) is via a message. The combination of distributed, append-only storage and global causal event tracking makes it possible to implement useful debugging and data repair capabilities via targeted execution replay.

- We leverage emerging cloud and operating system technologies to enable isolated portability and low latency execution (key for near real-time, data-driven actuation, control, and automation at the edge). CSPOT couples Linux containers[54, 55] for isolation (where available) with memory-mapped storage to support application data structures. The result is that CSPOT can dispatch isolated FaaS functions with latencies that are two orders of magnitude lower than current commercial cloud offerings.

In the sections that follow, we first present the related works in Section **??**. The design and implementation of CSPOT are presented in Section 3.2 and 3.3. Then, in Section 3.4.2, we evaluate the performance of CSPOT with benchmarks and provide a comparison to popular IoT edge offerings AWS Greengrass and Azure IoT Edge. Finally, we give our conclusions in Section 3.5.

3.1 Related Work

Among the commercial FaaS platforms, Amazon's AWS Greengrass[40] and Microsoft's Azure IoT Edge[42] combined with IoT Hub[44] are most similar to our work. These commercial services are the first successful attempts to bring FaaS functionality to the edge. They are similar to CSPOT in that they allow FaaS functions to run on edge devices. However, unique to CSPOT is that it intrinsically integrates an append-only versioned data structure to support robust data storage and event-triggered execution model. Moreover, the most significant difference of CSPOT to these commercial services is that from the ground up, CSPOT is designed for dynamic multi-tier environments. As a result, CSPOT developers use the same abstraction, application programming interface (API), and communication protocol to implement functions that run on devices of every scale without modification.

Open-source platforms such as OpenWhisk[61], OpenFaaS[60] and OpenLambda[112] are similar to CSPOT in terms of execution model. They all use Linux containers for namespace isolation and function execution. What distinguishes CSPOT from these systems is that these open-source platforms do not emphasize and specialize in the edge environments. Also, CSPOT defines low-level abstractions that are portable to many devices and allows developers to implement higher-level of data structure and applications, while these systems aim to provide a language-level runtime framework.

Besides commercial products, there are many interests to bring FaaS to the IoT-edge environment. For example, Mohammad et al.[113] and Gadepalli et al.[114] evaluate and analyze the challenges of bringing serverless computing to the edge. Palade et al. review the use of current open-source FaaS frameworks in edge computing[67]. Hall and Ramachandran[115] and Gadepalli et al.[116] present WebAssembly-based FaaS execution models at the edge. Paraskevoulakou and Kyriazis[117] and Wang et al.[118] leverage

existing open-source FaaS frameworks at the edge. Nastic et al.[119] and E. Al-Masri et al.[120] demonstrate using serverless computing platforms at the edge for data analysis. Baresi et al. use OpenFaaS to bring computation to multi-tier environments in the form of microservices[121]. The studies of usage, performance, and limitations of FaaS have been conducted in many works [46, 122, 123, 124, 125, 126, 127, 128, 129].

3.2 CSPOT Abstractions

Our CSPOT design is motivated by several observations that we have made while building and deploying IoT applications "in the wild". These applications are long-lived, operate in remote, unattended locations, and perform sensing, data production, processing, and analysis, and data-driven operations for surrounding systems. Many of the underlying devices are battery-powered or rely on intermittent energy sources, have restricted or intermittent access to computational and network infrastructure, and experience hardware and software failures that result in data loss and corruption. To extend their capability, longevity, and robustness, the applications increasingly rely on more capable, co-located edge systems for computational, communications, storage, and analytics offloading as well as for other proxy services. Low-latency access to these services is critical to enable the near real-time, data-driven alerting, actuation, control, and automation of physical, mechanical, and digital systems that these applications perform.

We find that most of these applications are modular and event-driven (i.e., operations execute in response to state changes and data arrival), making them suitable to the FaaS programming and execution model. However, FaaS systems today are not distributed, restrict both the type and scale of applications, and provide no automated, end-to-end error tracking. Furthermore, they incorporate heterogeneous devices using disparate technologies (different operating systems, protocols, software development kits

(SDKs), and FaaS services) – making them challenging to program, deploy, debug, and maintain. Extant FaaS systems for IoT also require that functions communicate global state through remote, location-concealed data services. This limitation prevents intelligent code placement and the exploitation of data locality (i.e., moving the code to the data and the data to the code) and becomes a bottleneck that limits performance and scale[122, 123].

CSPOT addresses the above limitations via a portable, high-performance, robust runtime system capable of running on all IoT tiers – from sensors to public clouds – so that code and data can move between these infrastructures without the need for transformation or recoding. To enable this, CSPOT defines, combines, and exports three abstractions:

- **Wide Area Objects of Functions** (WooFs), which are append-only memory objects with which developers persist state,
- **namespaces**, which root separate, hierarchical, declarative regions that provide a scope for WooFs, and
- **event handlers**, which developers use to define functions that are triggered when a data item is appended to a WooF.

CSPOT adopts an event-driven programming model; events are triggered by state updates to WooFs, and the only way to persist data beyond handler execution is via WooF updates. Thus a CSPOT application consists of event-triggered computations that may generate volatile local state, but that result in updates to application "variables", which are global to the application, have append-only semantics (i.e., multiple versions), and are persistent.

This "insistence on persistence" with versioning is motivated by the observation that at the device and edge tier computation, communication, and storage capabilities are

far less reliable than in less hostile environments. Thus, CSPOT abstractions mandate that data persist after all meaningful computations so that it can be processed when temporary outages have been resolved. Our work focuses on making FaaS persistence and communication "fast" and ensuring low latency response, application scalability, and effective management of scarce resources. CSPOT uses append-only semantics for its persistent data structures and distributed log to handle eventual consistency robustly and efficiently as is done in other distributed systems (e.g., HDFS[76], SafeStore[130], Chariots[131], Corfu[132], etc.), by prioritizing high availability and partition tolerance over consistency[133].

Also unique to CSPOT, each namespace has a location (e.g., a host machine), which gives application developers control over code and data locality. Developers currently define namespaces via Universal Resource Identifiers (URIs), which map to IP addresses. Each WooF and handler identifies uniquely with a CSPOT namespace, and namespaces cannot overlap.

Each handler operates directly on WooFs within its namespace only. Handlers are triggered when a WooF is appended to by a handler. That is, CSPOT couples FaaS function invocation with persistent storage in a 1-to-1 correspondence to aid debugging, profiling, and management of highly concurrent IoT applications.

Communication between namespaces is performed via point-to-point messages (direct socket connections) sent by a handler in one namespace performing an update on a WooF encapsulated in another namespace. This mechanism also makes it possible for WooFs to be accessed (and thus to trigger computations) from external processes, including non-CSPOT programs and services.

Within each namespace, CSPOT maintains an append-only, event log of WooF updates. CSPOT uses this event log to trigger handlers and to track causal dependencies within a CSPOT application. Causal dependencies can be used to facilitate data replica-

tion, synchronization, root cause analysis, and replay by CSPOT. The size of the event log is a tunable parameter.

Each WooF is logically also a log of append operations where each element appended is an untyped memory region of fixed size. The element size is set when a WooF is created. Each WooF append returns a unique sequence number associated with the appended element. All elements between the most recently appended element and the "earliest" element in history are accessible (i.e., there are no missing elements between elements present in the WooF history.) Thus CSPOT maintains a version history of fixed-size for each persistent data structure, and each version is identified via a unique ID. CSPOT's append-only semantics make data structures logically immutable[134, 135, 111], facilitating data durability, access/update efficiency, version control, debugging, and analysis. CSPOT garbage collects WooFs and logs by overwriting the oldest elements (i.e., using a circular buffer.)

The CSPOT API

The CSPOT API consists of a create operation that defines the name, element size, and history length for each WooF, a put operation that appends an element to a WooF (returning its sequence number), and a get operation that returns the element corresponding to a sequence number. The API also includes operations that allow a programmer to get the attributes associated with a WooF, to delete a WooF, and to create and destroy namespaces and handler resources.

WooFCreate() creates a WooF in the local namespace. **WooFPut()** causes the untyped memory pointed to by one of its parameters to be appended to the specified WooF. When **WooFPut()** succeeds, the unique sequence number assigned to the appended element is returned to the caller. **WooFPut()** also takes the name of a handler as an

optional parameter that is the name of a handler binary located in the same namespace as the WooF that CSPOT will invoke once the append has been successfully completed. Thus data is always appended to a WooF, but the programmer can determine when a handler should be triggered as a result of the append. However, there is no facility for invoking a handler without appending to a WooF. In this way CSPOT maintains a 1-to-1 mapping of data persistence to computation, i.e., every function invocation is unambiguously caused by a particular WooF update (identified by sequence number).

A call to **WooFGet()** returns the element corresponding to the sequence number that is optionally passed in as an argument (the default is the latest version, i.e., the WooF tail). Note, again, that the size of the memory region that **WooFGet()** writes is determined by the element size specified when the WooF is created.

Handlers and Clients

The CSPOT API can be invoked either within handlers or from "client" programs that are interacting with one or more CSPOT applications. Each handler must have the following C-language prototype. In addition, we also support handlers written in other languages (e.g., Python) via C bindings. Other high-level programming languages (e.g., those with managed runtimes) can be supported in a similar way, and it is considered as part of future work.

*int handler_function_name(WOOF *wf, ulong seq_no, void *ptr);*

`handler_function_name` is a legal C-language function name (i.e., the handler that will be compiled as a C-language function having these three arguments). The first parameter is a C-language pointer to a structure defined by CSPOT for manipulating WooFs. The second parameter is the sequence number that CSPOT assigned to the WooF append. Note that the append occurs prior to handler invocation. The third

parameter is an "in" pointer that points to a copy of untyped memory that has been appended. This memory is logically immutable, so a change by the handler to the memory pointed to by the ptr function does not persist beyond the handler's execution lifetime. By convention, handlers return zero on success and a negative value on failure.

Clients are programs written in any programming language that use the CSPOT API. Logically, a call to a CSPOT API function from a program that is not a handler results in a request to the WooF's namespace to perform the operation on behalf of the caller and to return the results.

3.3 CSPOT Implementation

We implement CSPOT using Linux memory-mapped files[136] as the operating system storage abstraction for WooFs. We isolate function handlers as Linux processes executing within a Docker container[43] associated with each namespace (each serviced by one or more containers). Handler execution constitutes "events" within the system. CSPOT triggers handler execution via its event log. Autoscaling is controlled by throttling invocations using a maximum concurrency level per namespace. Developers can query the log to extract the causal order of all events within a namespace for use as a debugging aid. For cross namespace invocation, we use ZeroMQ[137] as the messaging substrate and threads within the container to proxy namespace-external operations.

WooF

Each WooF is implemented as a separate memory-mapped Linux file containing a typed header structure and space to contain some number of fixed-sized elements. The header includes the local file name of the WooF, element size, the number of elements that are retained in the append history, and the current sequence number. Each WooF keeps

a circular buffer of appended elements; the head and tail indices are stored in the header. The space for the buffer is located immediately after the header in the memory-mapped file.

Therefore, WooFs are self-describing in that all of the information necessary to manipulate a WooF is contained in the WooF itself. When a WooF is "opened", its contents are mapped into the memory space of the process opening the WooF as shared memory. Thus multiple threads and, indeed, multiple processes can access a WooF concurrently using the information contained in the WooF header.

To implement synchronization for internal operations, the WooF header includes two Linux semaphores. The first implements mutual exclusion for operations like buffer head and tail index update, sequence number assignments, etc. The second allows threads to synchronize on the "tail" of the WooF so that when a new append occurs, they can be activated. These semaphores are not exposed through the CSPOT API, however, and are used strictly by the runtime.

Handlers, Containers, and the Event Log

When CSPOT starts a namespace, it launches a Docker container for the namespace, which shares the namespace directory in which all WooFs and handlers for the namespace are located. Docker includes an option to specify where, in the container's directory structure, a directory shared with the host must be located. By using the same location within the container (e.g., "/CSPOT"), the API can locate the WooF and handlers within the container.

Each handler is compiled as a separate Linux executable program. When an invocation of **WooFPut()** includes a handler name, the API code appends the element specified in the call to **WooFPut()** to the WooF and then appends an event record

specifying the WooF, the sequence number of the element that has been appended, and the handler name to the CSPOT event log for the namespace.

The main process within the container spawns several threads that synchronize on the tail of the event log in the namespace using a semaphore in the event log header. These threads "claim" events from the log by atomically appending a claim record for an unclaimed event. They then call Linux fork() and exec() on the handler binary. When **WooFPut()** is called from within a handler, the sequence number of the caller is included in the event record, indicating that it is the "cause" of the handler firing. Thus the event log that serves as the dispatch mechanism for handlers, also includes the dependency information necessary to determine causal order.

To determine a global causal ordering, CSPOT includes a log-merge toolchain that combines event logs from multiple namespaces. It creates a single total order of events in which the causal dependencies, both within and across namespaces, are correctly represented. Multiple log merges produce the same causal order but may produce different total orders depending on the namespace merge order.

3.4 Evaluation

We empirically evaluate the performance of CSPOT using IoT devices, an edge cloud, a private cloud, and a public cloud. We also design a suite of multi-tier applications to compare CSPOT's performance to public cloud offerings AWS Greengrass and Azure IoT Edge.

3.4.1 Function Invocation

To evaluate CSPOT's performance as a FaaS runtime, we run a simple benchmark on multiple hosts that are commonly used in IoT deployments, including microcontroller,

single-board computer, edge cloud, and traditional datacenter-based private and public clouds. Table 3.1 shows the hosts that we use to conduct the benchmark. Espressif esp8266[138] is a microcontroller with the lowest CPU and memory capacity and represents the sensor tier. Raspberry Pi Zero W[139] is a single board computer that has more computational power than a microcontroller. While it is a low-cost device, it runs a complete operating system and is capable of performing complicated computations. Intel NUCs[140] are x86 machines that we use to form an edge cloud. Functionality-wise, these NUC machines are as capable as the cloud instances. Note that CSPOT is a portable runtime. While these hosts have different computational capacities and run different operating systems, we only write the benchmark applications once and use the same code to compile the binaries that can run on all of them without modification.

Host	**Make and model**	**CPU**	**Memory**
Microcontroller	Espressif esp8266	80 MHz L106 RISC	112 KB
RPi 0	Raspberry Pi Zero W	1 GHz BCM2835	512 MB
NUC	Intel NUC 6i7KYK	2.6 GHz Intel i7-6770HQ	32 GB
Campus private cloud	cg1.4xlarge	2.1 GHz Xeon, 4 vCPU	8 GB
AWS EC2	m5.xlarge	2.5 GHz Xeon, 4 vCPU	16 GB

Table 3.1: Testbed for CSPOT performance benchmarking

On the microcontroller, CSPOT runs natively as an operating system. Raspberry Pi Zero W runs Raspbian Stretch Lite (build 2018-06-27) as the operating system, while the Intel NUCs and the campus cloud instances run CentOS 7.2. All of the microcontroller, Raspberry Pi Zero W, and Intel NUCs are connected via WiFi. We use Eucalyptus 4.3[19] to run our edge cloud and private cloud. The private cloud is a campus-wide cloud located at UCSB. It is connected to a layer-3 IP network with a 10 Gb connection. All of the hosts use Network Time Protocol (NTP)[141] to synchronize the clocks.

In the first benchmark, we evaluate the function invocation latency. On each host, we first use **WooFCreate()** to create a WooF for later function invocation. Then,

we implement and install a simple client on the same host where the WooF is created. When executed, the client uses *gettimeofday()* system call to get a timestamp. It then calls **WooFPut()** to put a data object with the timestamp to the created WooF. Once the data object with the timestamp persists, CSPOT runtime triggers a test handler. The handler reads the timestamp in the data object and calls another *gettimeofday()* to get a new timestamp and subtracts the first timestamp from the second timestamp to get the invocation latency.

In the benchmark, we use Intel NUCs to test two edge cloud deployments. In the first deployment, the NUC machine runs CSPOT natively without virtualization, while in the second deployment, NUCs are running Eucalyptus (with KVM as the hypervisor) to host a private edge cloud.

To compare CSPOT to the popular FaaS platform AWS Lambda, we also implement the same benchmark in Python. (at the time of the experiment, Lambda doesn't support C.) Because Lambda does not have any built-in persistence mechanism and relies on external storage, we use DynamoDB[142] to store the data and trigger Lambda functions. To avoid the cold start problem[143], we first "warm-up" AWS Lambda by running the benchmark once before taking measurement.

Host	**Mean (ms)**	**std-dev(ms)**	**95% (ms)**
esp8266	38	1.2	40
RPi 0	37	6.8	48
NUC (native)	4.0	0.6	4.9
NUC (VM)	6.5	3.3	15
Campus private cloud	5.0	1.6	7.0
AWS EC2	5.0	1.0	6.6
AWS Lambda	253	90	584

Table 3.2: CSPOT invocation time across different hosts.

We repeat the benchmark 100 times on each host and record the mean, standard deviation, and 95th percentile. Table 3.2 shows the benchmark result. Both esp8266

Host	Mean (ms)	std-dev(ms)	95% (ms)
NUC (native)	16	0.6	17
NUC (VM)	18	1.2	19
Campus private cloud	22	0.6	23
AWS EC2	18	3.1	23

Table 3.3: CSPOT invocation time using Python binding across different hosts.

and RPi 0 are able to achieve invocation time one order of magnitude faster than AWS Lambda. Compared to Lambda, all the other cloud deployments are able to achieve even faster invocation time by two orders of magnitude.

CSPOT is developed natively in C. since C is considered significantly faster than Python in general cases (AWS Lambda uses Python), we implement a Python binding for CSPOT and use the binding to implement a Python benchmark to have a fair comparison. Table 3.3 shows the result. Using Python binding makes the function invocation roughly 3 to 4 times slower than native C implementation. However, it is still one order of magnitude faster than AWS Lambda. The result shows that CSPOT not only is portable but also performs better than AWS Lambda in terms of function invocation.

3.4.2 IoT Deployment

The above results show that CSPOT can achieve fast function invocation as a FaaS runtime. To further evaluate how CSPOT performs in a multi-tier IoT deployment, we extend the first benchmark and run the benchmark on an IoT deployment that spans across multiple hosts.

Similar to the first benchmark used to evaluate local function invocation performance, the second benchmark consists of a client and WooFs. The benchmark first calls **WooFCreate()** to create a WooF on each host (RPi 0, edge cloud, campus private cloud, and AWS EC2.) The client is installed on the microcontroller esp8266 and starts

the benchmark by taking a timestamp and calls **WooFPut()** to put a data object with the timestamp to a remote WooF. This remote **WooFPut()** triggers a handler on the remote host. The handler forwards the data object to the next host by making another remote **WooFPut()** call on the next host. This remote **WooFPut** and handler invocation repeat until the data object reaches the last host. The handler on the last host takes a final timestamp and subtracts the first timestamp to compute the total latency.

Note that the "chain" of remote **WooFPut** described above implements the "weak-chain replication" described by van Renesse and Schneider[144]. That is, when a data object is appended to a WooF, it persists a replica on the WooF's host. We implement the same chain replication on AWS Greengrass using AWS IoT SDK. Greengrass runtime, once installed, enables the edge device's capability to execute Lambda functions. In the Greengrass benchmark, a Lambda function is registered on our NUC-based edge cloud. The function subscribes to an MQTT event stream. The microcontroller uses AWS IoT SDK to publish timestamp data via MQTT and trigger the Lambda function. When triggered, the Lambda function persists the data on AWS public cloud (Greengrass doesn't support persistence at the edge).

Deployment	**Replicas**	**Mean (ms)**	**std-dev(ms)**	**95% (ms)**
RPi0→edge→campus→AWS	4	513	48	607
edge→campus→AWS	3	323	17	457
AWS Greengrass (esp8266→edge→AWS)	$\geqslant 3$	4136	632	4288

Table 3.4: End-to-end latency comparison of multi-tier IoT deployments.

Table 3.4 shows the number of replicas and end-to-end latency of each deployment. The number of replicas in our CSPOT deployments represents the number of WooFs involved in the path of the replication chain. For example, the deployment *RPi0→edge→campus→AWS* stores four replicas of the data object in RPi0, edge cloud, campus cloud, and AWS cloud. The microcontroller esp8266 takes the first timestamp and calls **WooF-**

Put() to put the data object to the WooF on RPi0, and so on, until it makes a replica on AWS cloud.

Across all deployments, CSPOT achieves end-to-end latency one order of magnitude lower than the AWS Greengrass with fewer hops. This result shows that CSPOT performs significantly faster than AWS in a wide-area setting. In addition, note that AWS Greengrass doesn't support persistence at the edge, and the data must persist on the AWS public cloud (which has a replication factor of at least 3). Thus, developers cannot exploit locality. On the other hand, CSPOT provides the flexibility to develop replication strategies that exploit locality. Moreover, we use the same abstraction and source code on all tiers of our CSPOT deployments, while we need to use different protocols and SDK on AWS Greengrass.

3.5 Conclusion

We present CSPOT, a portable FaaS platform for IoT application development and deployment. CSPOT implements a set of lightweight abstractions specifically to support FaaS across a spectrum of device scales. Thus, it is possible to run the same CSPOT application code, without modification, on microcontrollers, edge devices and edge clouds, private clouds, and public clouds.

CSPOT implements an intrinsic, low-level append-only data structure for data durability and robustness in a distributed execution setting. In addition, CSPOT uses the append-only data structure to implement a log-based runtime. The runtime log carries the system-wide events that enable fast function invocation, causality extracting, and failure recovery. CSPOT exposes a simple and portable API that uses URIs to identify remote resources. The use of URI allows CSPOT developers to leverage data locality and implement messaging, persistence, and replication easier when designing multi-tier

IoT applications.

To empirically evaluate CSPOT, we implement several operation benchmarks using multiple heterogeneous systems. Our results show that CSPOT is able to achieve lower function execution latency on all scales of IoT deployments – from microcontroller, IoT devices to edge and public clouds. Furthermore, compared to extant public IoT FaaS systems, not only CSPOT allows more flexible deployment strategies (functions can be deployed on any tier of deployment at developers' wish) CSPOT is able to achieve better end-to-end performance.

CSPOT lays the groundwork for our multi-tier IoT researches. The portable abstraction and FaaS programming model greatly reduce the implementation complexity and enables our work on an event-driven programming system Canal described in Chapter 4. Further, CSPOT is able to capture the causal dependencies of events in wide-area distributed settings, which inspires our research on event dependency and causality tracking in Chapter 5 and 6. In addition, CSPOT's intrinsically integrated distributed log enables us the work of data repair and replay in Chapter 7.

Chapter 4

Canal: Event-driven Programming for Mul-tier IoT

In the previous chapter, we successfully tame the implementation complexity of IoT applications with CSPOT using the FaaS programming model. However, one of the biggest challenges of large-scale IoT – efficient data messaging and service discovery – still remains. Even though FaaS alleviate the heterogeneity and programmability issues, The huge number of device names, network addresses, and function names makes it difficult to manage when there are hundreds or even thousands of components involved in IoT applications.

To simplify and expedite data processing in such large-scale settings, publish/subscribe (pub/sub) systems play an important role. Designers of IoT deployments often envision a scalable data management infrastructure in the form of a publish-subscribe (pub/sub) framework that locates and matches data publishers with interested data subscribers. These systems provide discovery services and a higher-level, more abstract messaging pattern than direct point-to-point messaging or network multicast. Consumers express their interest by subscribing to data carrying certain attributes (content-based)

or explicitly labeled by category (topic-based). Producers label data and forward it to data brokers, which implement data discovery by subscribers and route data to them.

Most pub/sub systems, however, do not implement a computational model that supports processing "in stream." Developers must implement their own functionality in separate frameworks, manage failures, load balancing, and scaling themselves, and port and optimize their IoT applications and deployments manually. This latter limitation is particularly challenging because pub/sub abstracts away the details about publishers and subscribers, and there is no way for developers to intelligently "place" their application functionality in ways that exploit locality or resource availability – which are critical for IoT applications that consume battery power or require a low-latency response at the edge.

To address these issues, we present CANAL. CANAL is a portably programmable, reliable, distributed pub/sub system for multi-scale IoT deployments. CANAL implements the FaaS computing programming model and provides reliable data persistence via replication. Using the former, developers implement applications as a set of independent functions that subscribe to data topics from producers. CANAL invokes the functions on-demand when data arrives. To enable persistence and scale, CANAL couples a distributed hash table (DHT) for fast, robust lookup with consensus-based, strongly consistent replication. Moreover, CANAL replicates topic data, application functions, and DHT state to facilitate fault tolerance efficiently. Developers use a uniform and portable programming interface (API) to implement functions, manage their placement, and control the performance/reliability trade-off as required by their applications.

We empirically evaluate CANAL using multi-tier cloud environments consisting of edge clouds and a large-scale private cloud. Our evaluation shows that applications can be easily deployed in CANAL and tuned to meet different performance, availability, and reliability requirements.

In the sections that follow, We first discuss other work that is related or similar to CANAL. We then describe the design of CANAL and its implementation in Section 4.2.4. In Section 4.3, we show the evaluation results of CANAL's end-to-end performance. Finally, and we give our conclusion in Section 4.4.

4.1 Related Work

Today, many protocols and systems are available that provide cluster management, storage, event-driven functions, messaging, and data brokering. These systems, however, target particular deployment tiers (e.g., sensors, edge, or cloud). To our knowledge, CANAL is the first system that integrates these features into a single, lightweight, distributed system that can be deployed across multi-scale settings. Here we overview the key protocols and systems that inspire our research. The details of the following protocols are described in Section 2.3.

MQTT[68] is the de-facto message protocol used in the IoT landscape to enable lightweight messaging using pub/sub pattern. MQTT-SN[69] is an even more lightweight extension to MQTT for sensor networks. MQTT-ST[72] is another extension to enable MQTT broker clustering. RabbitMQ[71] is an MQTT-compatible messaging middleware to support scalable pub/sub systems.

Apache Kafka[73], Flume[75], Facebook Scribe[74] are data streaming platforms. Different from MQTT, these systems focus on the different performance attributes (e.g., throughput and latency). In addition, these systems process (and in some cases store) the data as logs.

The most differentiating feature of CANAL, however, is its built-in multi-tier programmability. To our knowledge, no extant systems intrinsically integrate data processing and intelligent data-driven application deployment. Instead, they use 3rd-party sys-

tems that provide these services (i.e., connect computation to messaging/data streams). Apache Storm[145], Spark[146], and Flink[147] are examples of popular data processing frameworks. Being data processing frameworks, these systems consume data from external systems, e.g., Kafka, that must be deployed, managed, and configured separately. CANAL is unique in that we co-design messaging, portable data-driven computation, and data persistence within a single system.

4.2 Design

Our goal with CANAL is to facilitate and simplify the development and deployment of data-driven applications in distributed settings. Toward this end, CANAL combines publish/subscribe messaging, event-triggered computation, distributed discovery, and strongly consistent data replication to achieve the following properties.

- **Programability:** Application are programmed against data *topics* instead of network addresses and hardware resources.
- **Reliability:** Published data are resilient to hardware and network failure.
- **Availability:** The system remains available and serves client requests even when partial failures occur.
- **Flexibility:** Deployments can be customized for different application requirements (e.g., reliability, availability, and performance).
- **Portability:** Applications span heterogeneous resources of varying quality efficiently and without modification.

Fig. 4.1 illustrates the CANAL system. A CANAL deployment consists of a cluster of geo-distributed, virtual nodes organized to facilitate distributed lookup. Nodes service

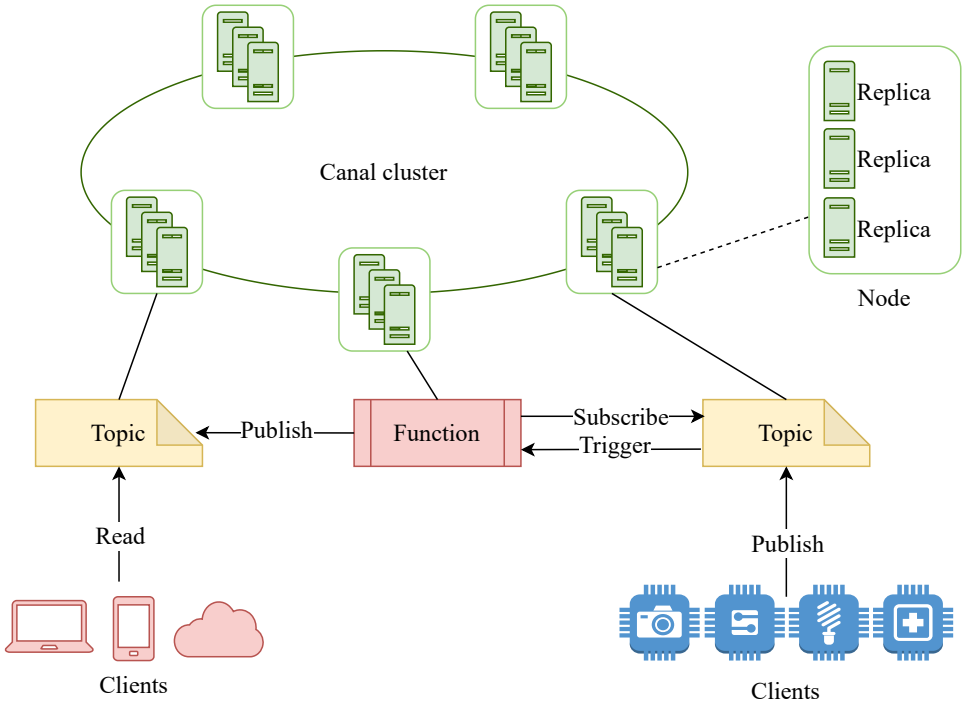


Figure 4.1: The system overview of CANAL.

client requests and hosts topics that are published and subscribed to by applications and clients. CANAL routes published data to subscribers and exports an API that clients use to create, access, and publish topics data.

CANAL is unique in that it also embeds distributed application execution to automate data-driven response, analysis, and computation (e.g., actuation, automation, and control). Specifically, CANAL implements the serverless programming and deployment model (also known as functions-as-a-service) [51, 52, 53, 148]. The serverless model is well suited to our setting in that it is event-based, scalable, and tolerant to intermittent connectivity. Using this model, programmers write simple event handlers (i.e., application functions), which are invoked (i.e., triggered) by the platform in response to an event. Functions are short running and functional (they only persist state via runtime APIs/services) which makes them robust to failures and amenable to autoscaling.

In CANAL, events are topic publish operations. Developers use CANAL function tem-

plates to construct their applications and the CANAL client API to register and deploy their applications (which consist of topics, functions, subscriptions, and configuration preferences, if any). Functions can access topic data, process and analyze data from multiple topics, and publish new data to geo-distributed topics when they execute. Developers register functions as compiled binaries. To enable this, we design CANAL to be a programmable publish/subscribe system that can be deployed across devices with different resource capacities (e.g., physical and virtual machines, single board computers, etc.). A CANAL system is a cluster consisting of multiple nodes that together provide data broker services and serverless application execution. Developers use the CANAL client API and function template to write application functions to be deployed to a CANAL cluster. To enable this, CANAL defines the following system abstractions.

- **Topic:** A topic is an object that encapsulates data elements of the same type. Each topic is unique in a CANAL cluster. A strictly increasing index identifies each element in a topic.

- **Function:** A function is an event handler implemented by an application that CANAL executes in response to a publish event on a topic to which it has subscribed.

- **Node:** A node consists of one or more physical or virtual machines, which service client requests and host topics that are published and subscribed to by applications and clients.

- **Replica:** A replica is a copy of a topic and its functions. For redundancy and parallelism, each node can implement multiple replicas.

The CANAL client API exports topic operations (create, publish, subscribe). User applications define functions that use these operations to process, analyze, and generate data. Each topic in CANAL is an append-only data structure, i.e., past versions of topics

are maintained for some configurable, fixed-size history that is automatically garbage collected. This way, past data can be retrieved for processing and analysis; append-only storage also reduces coordination overhead and enhances data durability since data elements within a topic are immutable. CANAL maintains the list of functions subscribed to each topic and triggers upon a topic append.

Functions can retrieve the most recent version of the topic or a previous version, if available, via its index. This index is returned upon a publish (an append), and the history can be scanned using it via the client API. We describe the CANAL storage and replication model in more detail in Section 4.2.2.

Functions are compiled binaries that implement the CANAL function template format. Functions subscribe to topics, consume data in response to publish operations, and can perform data processing, analysis, and computation and publish to topics across the system as required by an application. When there is new data published to the topic, all the functions on the subscription list are triggered by the platform in parallel.

Topics and functions can be placed on any node in a CANAL cluster. This flexibility enables developers to optimize their applications in different ways, e.g., for locality (co-locating functions and topics) or for resilience (geo-distributing replicas). For example, a machine learning application that subscribes to IoT devices for its input data may wish to place the IoT topics near where the data are generated, and place the functions that perform more heavyweight computation on more powerful nodes at the edge or in the cloud.

Fig. 4.1 illustrates the CANAL system. A CANAL cluster consists of multiple nodes. A node usually hosts some user-defined topics and functions.

A client request includes the target topic, precluding the need to specify the physical address of the topic's host. The CANAL platform constructs the mapping of topic to the node for the client library, using the nodes in the cluster as a *distributed lookup table*.

Each node stores the addresses of some topics and their subscription lists. In practice, a publisher uses CANAL client API does not need to know the physical addresses of the topics, as routing is done by the CANAL

To enhance availability and reliability, a node can also implement multiple replicas. All the topics and functions, as well as the topic lookup entries stored in a node, are copied across replicas. A group of replicas in a node uses consensus protocol to elect a leader. The leader is responsible for all the queries and requests from other nodes. If the leader fails, a new leader is elected among the remaining working replicas and assumes the role of leader. We detail how CANAL manages and uses data replicas in Section 4.2.2.

Nodes can join and leave the cluster dynamically, and data/function lifetime is that of their hosting nodes. Besides hosting data and functions, nodes also handle client queries and publish requests. Client requests specify topics, which CANAL uses to identify hosting nodes using a distributed lookup.

4.2.1 The Canal Runtime and Lookup System

CANAL is unique in that it uses the functions-as-a-service programming paradigm to facilitate data-driven computation automatically. The FaaS model is well suited to our setting in that it is event-based and robust to failure and intermittent connectivity (functions are stateless and isolated). Moreover, functions are typically small and short running and thus easy to scale, port, place, move, replay, and execute across a wide variety of devices and scales (from microcontrollers to public cloud systems) [149].

To support FaaS applications, CANAL builds upon and extends *CSPOT*. CANAL extends FaaS execution model with support for publish/subscribe semantics to abstract away the details of this placement and complex deployments. To enable this, CANAL functions persist state and share data via topics distributed across the system using the

append-only, distributed storage abstraction exported by CSPOT.

CANAL achieves scalable, distributed lookup (for clusters in which nodes can join and leave dynamically) by integrating protocols from Chord[150] into CSPOT. Chord is a scalable peer-to-peer lookup protocol that works as a distributed hash table (DHT). It provides one simple operation: mapping a key (and its value) to a node. The topology of the DHT cluster is structured as a ring, as shown in Fig. 4.2a.

We equate a node in Chord with a node in CANAL. When joining a cluster, CANAL assigns each node an m-bit identifier by hashing the unique name or IP address of the node using SHA-1 hashing [151]. CANAL places the node, ordered by its identifier, on a ring with a circular key space of 2^m. When inserting or querying about a key, the key is hashed again with SHA-1 to compute its identifier k. CANAL uses the key's identifier to decide which node is responsible for storing the key-value pair. Once the identifier is acquired, CANAL forwards the query to the node whose identifier is equal to or immediately after k, i.e., the successor of k (denoted by $successor(k)$).

For scalability, a node maintains only a subset of successor addresses in an address table. A node with identifier n maintains the addresses of its immediate successors (we use three herein). Each node also maintains m pointers (called fingers) to other nodes in the ring. The ith finger in an address table is denoted as $finger(i)$ and it is defined by $sucessor(n + 2^{i-1})$. Upon receiving a query of target key k, a node n checks if its immediate successor is the successor of k by testing if $k \in (n, successor]$. If so, it answers the query with the successor's address, where the value of key k can be found.

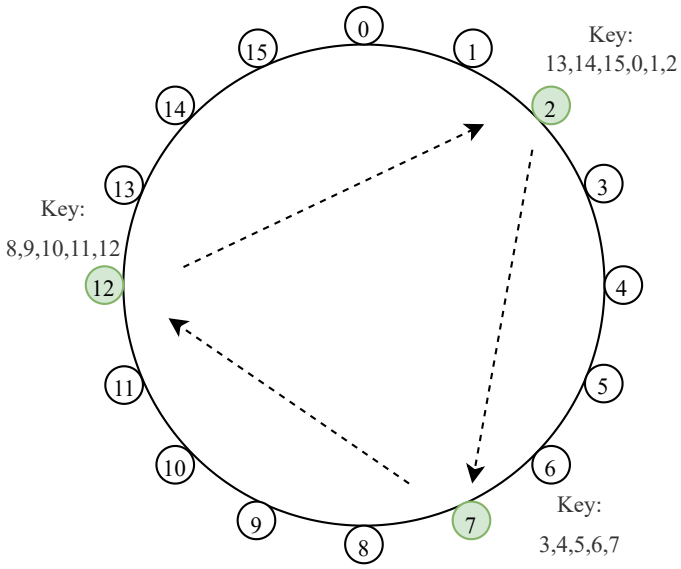
If the node's immediate successor is not the successor of k, the node passes the query to the closest preceding node of k in the finger list. The query is passed along the ring in this way until it reaches a node that can answer the query. Fig. 4.2b shows the process of querying a key on a 6-bit ring with 10 nodes. This protocol is proven to be scalable such that in a Chord cluster with N nodes, each node need only store $O(log(N))$ addresses,

and a query can be answered in $O(log(N))$ communications.

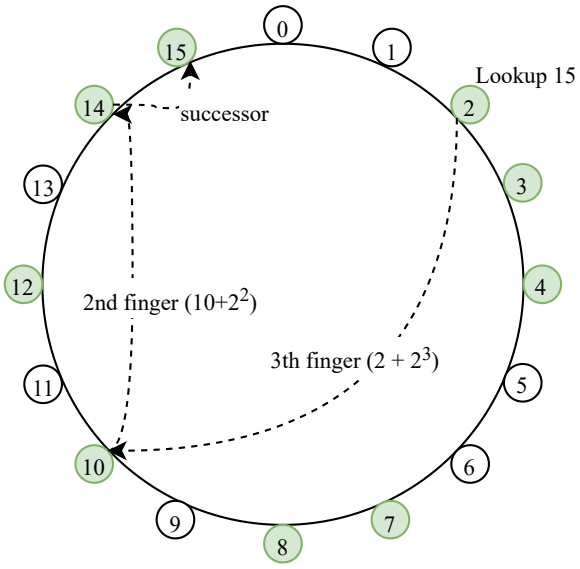
When node n joins the cluster, it queries the cluster about $successor(n)$. Once the query is answered, CANAL inserts node n into the ring ahead of $successor(n)$ by notifying $successor(n)$ about it. After being notified, $successor(n)$ adds node n as its predecessor. The next time $successor(n)$'s old predecessor sends a heartbeat to $successor(n)$, it learns that a new node was inserted and updates its successor from $successor(n)$ to node n, which completes the "stabilization" process. When node n leaves the cluster, its predecessor $predecessor(n)$ learns this fact via a lack of response to its heartbeat request. In this case, $predecessor(n)$ uses the next successor as its immediate successor.

When a topic is created, CANAL hashes the topic name for use as a lookup key. The address of the node that hosts the topic is stored in the successor of the topic name hash. This design allows us to decouple data placement from address lookup, making application deployment strategies more flexible while achieving a fast, logarithmic querying time.

One challenge with this integration (serverless and DHT) is that function invocation is not synchronized. Unlike traditional programming models that use RPC (remote procedure call), serverless functions are non-blocking, and functions are only invoked in response to an event (an append to an object). The mechanism for determining when a function completes is for a function to append to an object immediately prior to terminating and for another function to check the tail of the object. Such "spinlocks" on the tail of objects are costly in a serverless setting. To invoke a function in CSPOT (e.g., a Chord query or stabilizing protocol), the function caller needs to append the input of function as a data object to a function-specific WooF. CSPOT runtime detects the appended object and triggers the target function. When the function finishes, it can optionally append the result to a result WooF. To know the execution result of the function, the function caller needs to check the tail of the result WooF. It is inefficient



(a) A 5-bit Chord DHT ring consisting of 3 nodes: 2, 7, 11. The arrows shows the successor/predecessor relationship.



(b) A path of a query of key 15. The query begins at node 2.

Figure 4.2: An example of Chord DHT cluster with 5-bit key space.

for the function caller to invoke a remote function and spinlocks on the tail of result WooF to know the result since each read on a remote WooF involves network traffic and delay. Our solution to this problem is for each function call in Chord, we implement a separate function that consumes the result as input. The function caller's address and the separate function's name are included in the input data object when making the call. When the invoked function finishes and appends the result to the result WooF, it will trigger the separate function that handles the result.

Another challenge is locking on the internal state. Serverless functions are short-lived and stateless. Hence, node information such as the node identifier and address table of successors and fingers are stored in append-only objects. Because of the way functions are implemented and invoked, there will be multiple function instances trying to read or write to the same object simultaneously. This is not a problem in a traditional programming model as it can be easily solved via locks in shared memory. Since there is no shared memory between functions, we must solve this problem another way.

To do so, we implement a simple yet efficient locking library for CANAL called *monitor*. Monitor serializes functions that compete for the same storage object using the serverless model. To enable this, the monitor records the competing functions in a pending list (using a second object). When a function is passed in with an append, the monitor records the function in this list. When execution on the monitored object finishes, a monitor invoker function invokes the next function in the list. The invoker function is triggered the first time when the monitor is created to kickstart the process.

4.2.2 Replication and the Canal Data Model

Chord's DHT design provides a simple way to replicate data for fault tolerance. When a key-value pair is stored in a node, it is replicated to immediate successors of

the node. Because nodes are connected and ordered as a ring, when a node leaves the cluster unwillingly due to failure, its successor will assume the node's position on the ring and take over the responsibility of the departing node. When the departing node's successor does so, it has a data replica, and thus, it can serve queries seamlessly without performing any data migration. CANAL uses this mechanism to replicate the address information and subscription lists for topics.

However, this mechanism cannot support the data in topics or the application functions. On a DHT ring, nodes are ordered by their identifier hash, which is random by nature. In essence, replicating data to a node's successors is replicating data to random nodes in the cluster geographically. This method works well for replicating lightweight key-value pairs such as topic addresses and subscription lists, but it is inefficient for large data objects. In addition, random replication is unable to exploit data locality. When deploying a cluster, if we can place replicas according to network topology and resource availability, we can explore deployments with different replication performance and resiliency characteristics.

To enable this, CANAL uses Raft[94] to replicate topic data. Raft uses a consensus algorithm to manage a replicated log among the members of a Raft cluster. To distinguish this from a CANAL cluster, we consider each CANAL node a Raft cluster.

Raft provides a simple leader election algorithm to guarantee that there is only one leader at any time. Only the leader responds to client requests and appends entries to its log. Raft implements a strong leader approach in that the leader actively replicates its log to the other members (followers). If a follower receives conflicting entries from the leader, it overwrites its own log with the entries from the leader. Followers never ask the leader to resolve the conflict, and the leader never rewrites its own log. Once the leader confirms a log entry is replicated to a majority of replicas, it commits the entry and notifies its followers.

We then divide time into Raft *terms* across the cluster and define a random timeout for each. If a follower does not hear from the leader by the timeout, the follower promotes itself as a leader candidate and starts a new term. The candidate starts an election by requesting votes from all the members in its cluster. Each member can only vote for one candidate in each term, and it can only vote for a candidate if the candidate has more up-to-date entries than its own. If a candidate's vote requests are accepted by the majority of members (itself included), it is promoted and becomes the leader. This algorithm guarantees that only members with the most up-to-date log entry will be elected as the new leader so that the committed log entries are never rewritten.

To integrate data replication, we configure each node in CANAL as a Raft cluster and replicate topic data across the node. Each node has a leader representing the replica with the most up-to-date data. Each replica in a node knows the leader of the current term and the addresses of all other replicas. When a node joins a CANAL cluster, CANAL gives it a unique name (identifier) using the Chord protocol. (joined replica addresses can also be used, but since replica membership can change by adding or removing replicas, an arbitrary unique name keeps things simple).

A node is represented by its leader. All the DHT queries and client requests to publish to a topic or read data are answered by the leader. If a follower receives a client request, it redirects the request to the leader. If a leader fails, the node becomes unavailable temporarily until a new leader is elected among working replicas. Once a new leader is elected, the node becomes available again and the leader serves requests.

In CANAL, we also replicate a node's successor list. Therefore, during DHT stabilization, if a node updates its successor address, the update will be reflected on all replicas. This prevents DHT topology disruption when a node's leadership changes. As an optimization, we do not replicate the finger table and predecessor list. We can avoid doing so because losing this information does not break node connectivity, and it is automatically

regenerated during stabilization.

Moreover, we modify Chord stabilization to take advantage of replication. When a node detects that its successor has stopped responding to heartbeats, it first tries other replicas before deeming the node as failed and removing it from the successor list. Since each replica knows the address of the current leader, the node can update it to use the new leader of the successor node once elected. When the node with a new leader sends a heartbeat to its successor the next time, it notifies its successor of the leadership change.

We also define a timeout value to limit the time a node can try other replicas because of a nonresponsive leader. When the successor's known leader becomes nonresponsive, the node keeps trying until one of the replicas reaches a working leader. It moves on to use the next successor when a timeout occurs.

The timeout also ensures that the node has not lost the majority of replicas. If such a loss occurs, the election process may not terminate since no candidate can get a majority of votes. In this case, no replica identifies a working leader, and the timeout ends the process. We determine the timeout value empirically (i.e., in practice, we find that a single election is sufficient to select a new leader). We use a value double that of the maximum Raft election timeout in this study.

In the case when the majority of replicas fail, the node leaves the CANAL cluster. When/If the node recovers from failure and becomes available again, it will try to send a heartbeat to its last known successor. This will cause CANAL to initiate stabilization, which will enable the node to rejoin the cluster.

Since a node can host multiple topics, CANAL uses a single data object to store the data from all topics the node hosts. However, as described previously, the persistent objects in our serverless runtime are append-only (logs). To support this and multi-topic objects (that are replicated), CANAL employs a mapping object per topic. This object records the index of the data element when published, in the multi-topic data object.

Fig. 4.3 exemplifies this node mapping process. When a data element is published to a topic, CANAL encapsulates it with its topic name and appends it to the multi-topic data object. When the entry is replicated and committed, each committing replica checks which topic the entry belongs to and writes the index of committed entry to the topic's mapping object. When a client reads data in the topic by index, CANAL first checks the topic's mapping object to get the index of the data element in the multi-topic data object. It then reads and returns the data element from the data object. Only the multi-topic data object is replicated (i.e., managed) by the Raft algorithm. Mapping objects are indirectly (and automatically) replicated since each replica updates its mapping object when a data element is committed.

We choose to map multiple topics onto a single data object for implementation and performance reasons. The Raft algorithm maintains a single commit index in a managed cluster to indicate the latest committed entry. CANAL treats each data object as a replicated log managed by Raft; therefore, a node can only have one data object managed by Raft. Another option would be running multiple Raft algorithm instances, one per topic. This approach, however, introduces a significant computation and network overhead and is more complicated to implement. Thus we use a single, replicated, multi-topic object per node.

4.2.3 Canal API and Data Publishing

CANAL exports a simple client API for publish/subscribe operations. The client API provides the following functions:

- ***create_topic(topic_name):***

  Create a topic *topic_name* on the calling node.

- ***subscribe(topic_name, function_name):***

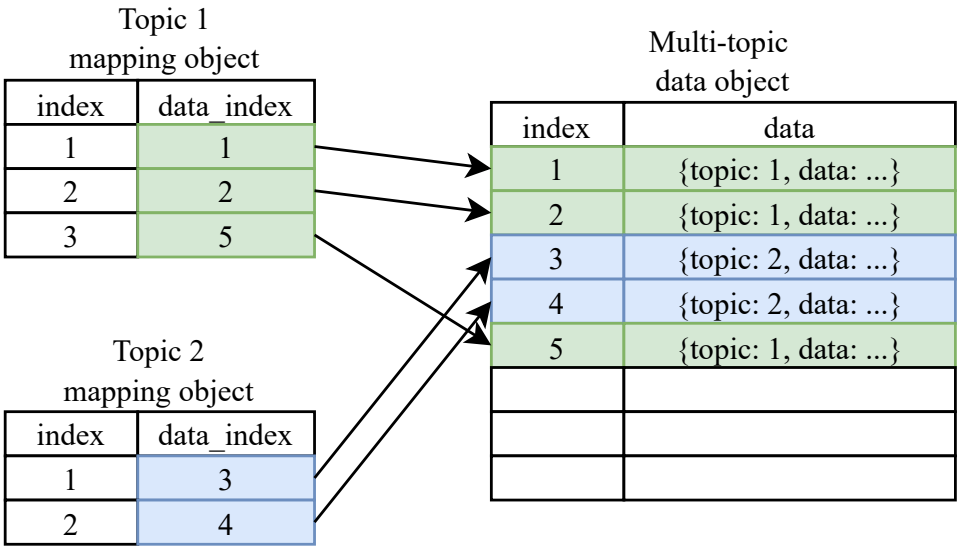


Figure 4.3: An example of mapping two topics onto a data object.

Subscribe the function *function_name* on the calling node to the topic *topic_name*.

- ***publish(node, topic_name, data):***

  Publish data to *topic_name*.

- ***latest_index(node, topic_name):***

  Get the latext index of published data in *topic_name*.

- ***get(node, topic_name, index):***

  Get the data in *topic_name* by its index.

A node in a CANAL cluster calls *create_topic()* to create a topic and store its address in the DHT. *create_topic()* also creates the topic mapping object as described in the previous section. A node can also call *subscribe()* to subscribe a function to a topic. There must be a binary file named *function_name* on the calling node for the call to succeed. The binary function can be written and compiled using the provided toolchain and it should follow the CANAL function template: $function\_name(topic\_name, index, data)$. *function_name*

can be any UNIX-style filename. *topic_name*, *index*, and *data* will be filled in by the CANAL runtime when the function is triggered.

Any client can use *publish()*, *latest_index()*, and *get()* to publish or read data. All of these operations take an optional *node* parameter to specify the address of the node that should receive the request. The node address can be the address of a leader or follower replica of any node in the CANAL cluster. It does not have to be the node hosting the topic, as the topic name will be used to query the actual address of the topic. The *node* parameter can be omitted if the calling client is a node itself in the CANAL cluster.

Fig. 4.4 illustrates the process of publishing data to a topic and triggering the functions subscribing to the topic. When a node (referred to as the publishing node) receives a publish request, it queries the cluster by topic name to determine where the topic data are hosted; it uses this information to acquire the subscription list for the topic. The query is answered with the addresses of all the replicas of the node hosting the topic (the topic node). The publishing node then appends the data element to the multi-topic data object of the topic node, as described in the previous section. Once the data element is committed using the Raft algorithm, the topic node notifies the publishing node, and the later triggers the functions on the subscription list to consume the published data.

4.2.4 Client Caching

For each client API call, the CANAL cluster performs a lookup of the topic's node address. Although it only takes logarithmic network messages to answer this query, in a large cluster with poor connectivity, this process can introduce significant overhead.

To save unnecessary network communication, we implement a client-side cache for topic node address lookup. When a node queries about a topic the first time to serve a client request, it writes the node address of the topic into its cache. Later, when it tries

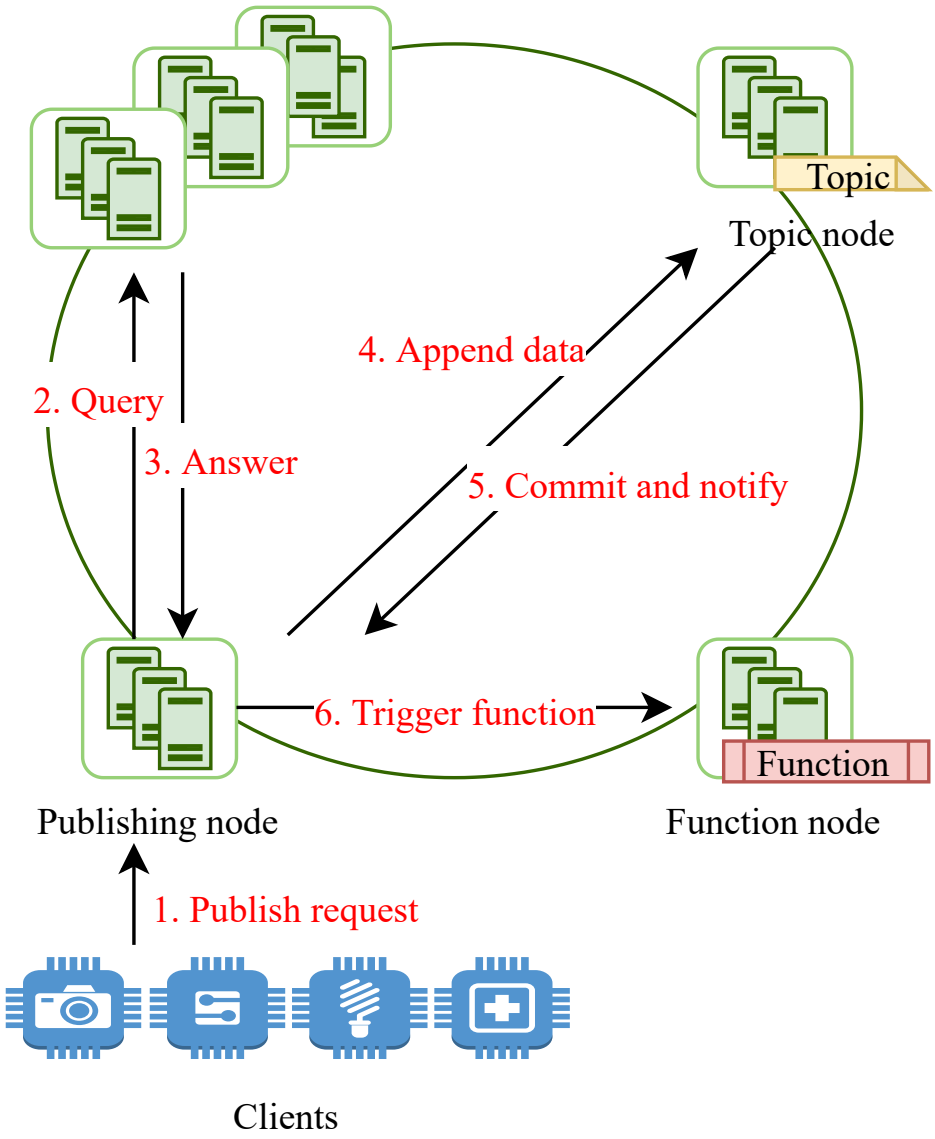


Figure 4.4: The process of publishing data and trigger function. 1) A client sends the publish request to any node (the publishing node). 2) The publishing node sends a query to know the topic node's address. 3) The query is answered with the address and the topic's subscription list. 4) The publishing node appends the data to the topic node's data object. 5) The topic node commits the data and notifies the publishing node. 6) The publishing node triggers the function on the subscription list.

to read or publish data to the topic, it first checks if the topic exists in the cache. Since the address of a topic does not change during the lifetime of the topic, the cache can be invalidated when the client cannot find the topic data in the cached node address. Because the subscription list for a topic changes over time, it is challenging to invalidate it efficiently. Thus, we do not cache subscription lists in CANAL.

We only implement the caching mechanism above on node addresses of topics. Although subscription lists are stored in the same way and can be cached using the same mechanism, there is no simple and effective way for cache invalidation. Throughout the life of a topic, its subscription list can change at any time. In order to invalidate subscription list cache, cache expiration or versioning has to be added to the mechanism. We decide against it because the extra layer of complexity does not warrant better client performance.

4.3 Evaluation

In this section, we evaluate the performance of CANAL using different replica placement strategies. Specifically, we instrument its processing pipeline (which include Chord and Raft extensions) and employ a series of benchmarks and a wide range of testing configurations. In particular, we attempt to answer the following questions:

- How fast can CANAL publish data and trigger functions?
- How does CANAL scale under load?
- How resilient is CANAL to failure?
- How can CANAL be used to facilitate the deployment of data-driven distributed applications?

To answer these questions, we deploy CANAL clusters with different configurations using virtual machine (VM) instances on three private and edge clouds, called *Campus*, *Lab*, and *Farm*. Each cloud runs Eucalyptus v4.2.2, which is API-compatible with AWS EC2 and S3. *Campus* is a shared, private cloud located on the UCSB campus connected via 10Gb Ethernet. It is part multi-campus cloud federation [152] that consists of 1400 cores and 30TBs of storage. *Lab* is an edge cloud located in the authors' research lab that consists of 9 Intel Next Units of Computing (NUC) devices connected via a gigabit switch. *Farm* is also a NUC-based edge cloud sited on a remote research reserve roughly 50 miles away from the campus. Each VM instance has 2 cores; *Campus* instances have 1GB memory while *Lab* and *Farm* instances have 4GB memory. During our experiments, we observe that CANAL uses very little memory and, as such, instance memory size does not significantly impact performance or our results.

The key difference between these clouds that does impact performance is the network. While *Campus* and *Lab* have a robust, high-performance network, the bandwidth of *Farm* (a microwave link to campus/Internet) is significantly lower to the instances outside the reserve cloud, which we find to be the case for edge clouds in general. Figure 4.5 shows the average bandwidth between clouds in this study, as measured by iperf[153].

For our experiments, we configure CANAL with eight nodes. Each node implements a Raft cluster with 3 replicas. Each Raft replica is hosted by a single cloud instance (each of which hosts a single Raft replica) for a total of 24 Raft instances. Although Raft is designed to enhance reliability and availability, we can further increase robustness via geo-distributed placement. To evaluate the performance impact of different replica placement strategies, we consider two different CANAL configurations: *distributed* and *colocated*.

In the *distributed* configuration, we choose one instance from each cloud to host a replica, and 3 replicas from different clouds to form a Raft cluster. Since the leader

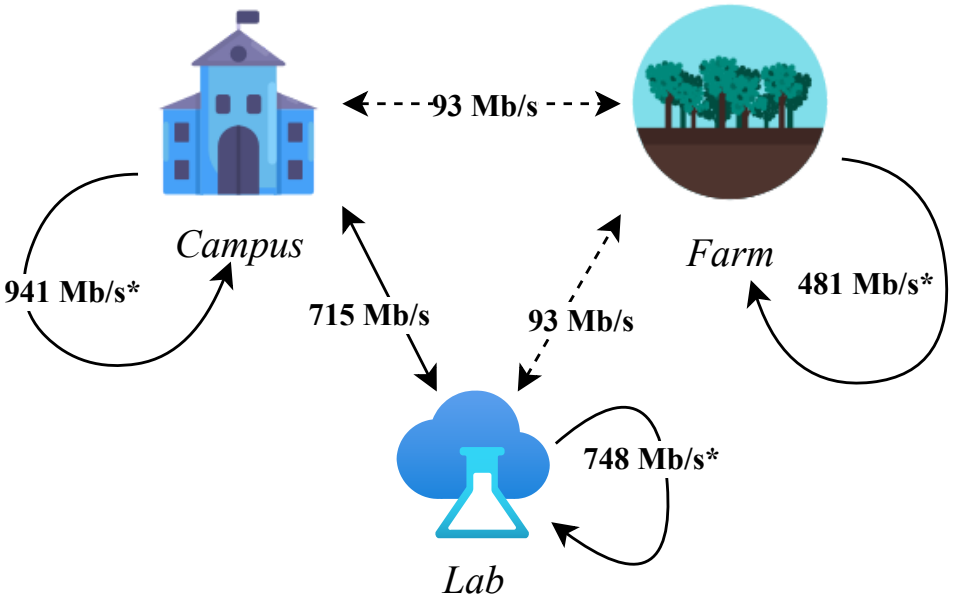


Figure 4.5: Bandwidth across our multi-cloud, experimental deployment as measured by iperf. *'d measurements show the bandwidth between instances within the same cloud.

of a Raft cluster is in charge of replicating data, handling Chord queries, and function subscription, it is more sensitive to resource availability than its followers. Hence, we also evaluate performance when the elected leader is hosted on different clouds. We do so by adding a whitelist to the Raft leader election mechanism. Only vote requests issued by the instances in the whitelist can be granted. In the following experiments, the configuration where all the leaders are from *Lab* is denoted as $distributed_{Lab}$, and the configuration where all the leaders are from *Farm* is denoted as $distributed_{Farm}$. We omit the other configuration combinations due to space constraints.

In addition to *distributed* configuration, we also experiment with configurations in which all the Raft replicas are all hosted in the same cloud. The intuition is that if replicas are colocated, Raft replication may be more efficient and hence provide better overall performance. In the *colocated* configuration, all Raft replicas hosting the test topic are hosted by instances within either *Lab* or *Farm*, respectively denoted as $colocated_{Lab}$

and $colocated_{Farm}$. The nodes not hosting the topic are still distributed across clouds.

All instances are synchronized using NTP. We use the instance clock to measure the latency of each CANAL client request. We instrument the client API to take a timestamp when a request is created, and another when the subscribing function is triggered. In addition to end-to-end latency, we measure Raft performance, which we record using Raft log. A log entry has three timestamps filled when processed by Raft: when the entry is written to the leader, when the entry is replicated to the followers, and when the entry is committed.

4.3.1 Publish Benchmark

To evaluate the performance of CANAL, we implement a simple benchmark using the CANAL client API. We start a CANAL cluster and create a test topic of size 8KB on one node (referred to as the topic node). The data published to the test topic is replicated among the topic node's three replicas, either distributed or colocated depending on the configuration. We create a test function that records a timestamp and returns and subscribe it to the test topic. The function is triggered (recording a timestamp) each time new data is published to the test topic.

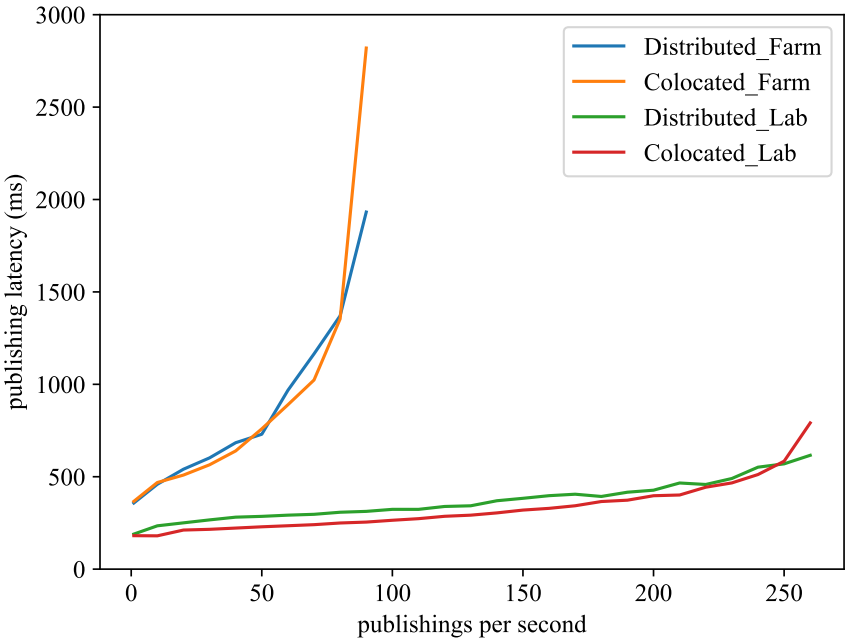
We use a simple client, which calls *publish*(), to publish data to the topic at a fixed rate. We experiment with different publishing rates and run each test for 10 minutes. During each test, we monitor the Raft commit lag to see if CANAL can process publishing requests without buffering them. The Raft commit lag is the number of log entries that are appended to the Raft leader but have not been committed. This metric identifies when CANAL is not able to process requests fast enough, causing the Raft commit lag to start to grow. We compute the publish latency as the average difference between the time at each *publish*() is called and the time at which the test function is triggered

by the published data (without buffering). Only the publishing latencies at publishing rates when CANAL can process publishing requests without buffering are recorded. In the following experiments, we place the client, topic, and function on the same node.

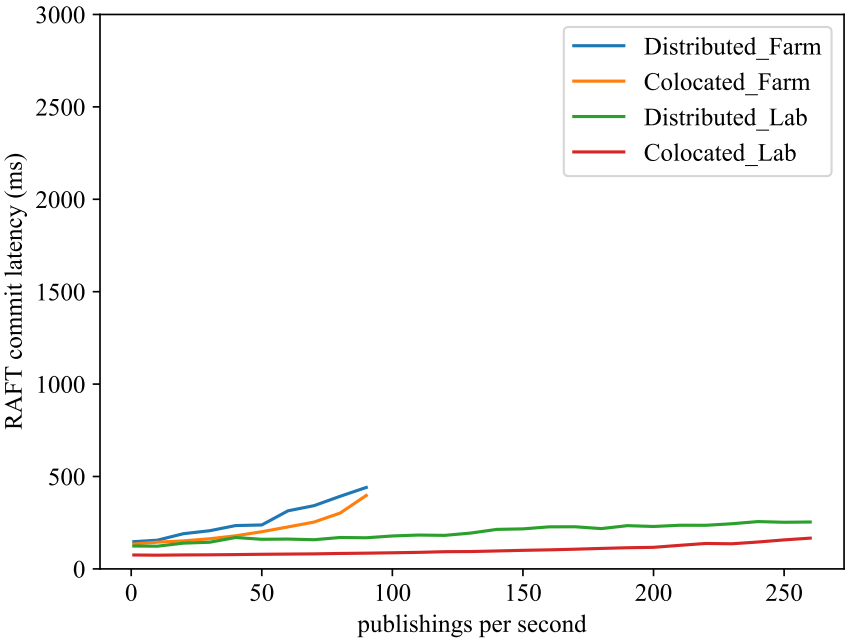
Figure 4.6a shows the average publishing latency of each data publishing for different publishing rates. Across the different test configurations, we find that $colocated_{Lab}$ has the lowest publishing latency and can process requests at the highest publishing rate, while $distributed_{Lab}$ comes in second, followed by $colocated_{Farm}$. Both $distributed_{Lab}$ and $colocated_{Lab}$ configurations perform well compared to their *Farm* counterparts. Both configurations are able to process 260 publish requests per second, while each individual publish and commit takes 791 and 616 milliseconds on average, respectively. In both configurations, publish latency grows linearly and is able to handle more than 260 publish requests per second, which is the fastest rate at which the host can spawn test functions.

However, publish latency of the other configurations begins to grow with the publishing rate. This occurs because the host spends more CPU time to commit log entries and cannot keep up with the publish request and function trigger rates. Both $distributed_{Farm}$ and $colocated_{Farm}$ handle up to 90 publish requests per second, with 1.93 and 2.8 seconds average publish request latency, respectively.

Figure 4.6b shows the average latency for Raft to commit a log entry. The $colocated_{Lab}$ configuration exhibits a clear advantage over $distributed_{Lab}$. That is because in the $colocated_{Lab}$ configuration, Raft only needs to replicate log entries within the network, while in $distributed_{Lab}$ configuration, the leader of the topic node must communicate to followers outside its local cluster. Although higher performance, $distributed_{Lab}$ sacrifices the data reliability when there is a local failure. On the other hand, although Raft only needs to perform replication locally in $colocated_{Farm}$ configuration as well, because of its poor network performance, it does not benefit the overall publishing performance in this deployment configuration.



(a) Average publishing latency per data publishing



(b) Average Raft commit latency per log entry

Figure 4.6: Publishing latency

From these experiments, we also observe that under heavy workloads, CANAL spends the majority of CPU time in Raft functions. As it starts to work harder to commit log entries, the overall performance degrades.

Since in the benchmark, the data are published in parallel, the maximum publishing rate translates to the average time CANAL needs to process each publish request in the pipeline. We next present the publish throughput using CANAL in Table 4.1. In each 10 minutes test, we record the timestamp when the first *publish*() is called and when the last test function is triggered. The duration divided by the total number of data published is how long it takes for each publishing to be processed in throughput test, as shown in Since the maximum publishing rate is the number of publishing requests CANAL can handle without buffering, if we run the throughput test continuously, the throughput is bounded by $1/maximum\ publishing\ rate$.

Table 4.1: Benchmarks of different cluster configuration. Individual latency is the time it takes for CANAL to commit published data and trigger the subscribing function. Average latency is how fast CANAL can process each publishing request at maximum workload.

	Maximum publishing rate	Individual latency	Average latency
$distributed_{Lab}$	260	791 ms	3.85 ms
$colocated_{Lab}$	260	616 ms	3.85 ms
$distributed_{Farm}$	90	1932 ms	11.13 ms
$colocated_{Farm}$	90	2820 ms	11.20 ms

We next evaluate the latency of extracting a data element from a topic. With the same setup, this experiment sequentially calls *get*() on the test topic. We compute average *get*() latency as the total time of benchmark divided by the number of *get*()s issued, which we present in Table 4.2. The results indicate that the performance of *get*() is affected only by the network performance and not by the data publishing rate. In the best case where the client and the topic leader reside on the same instance, it takes 10 milliseconds on average to get data. When a slow network involved (e.g., *Farm* cloud),

it takes more than 50 milliseconds on average for a get to complete.

Table 4.2: *get*() latency

	Client location		
Leader location	*Campus*	*Lab*	*Farm*
Campus	23*/36 ms	45 ms	55 ms
Lab		10*/28 ms	50 ms
Farm			25*/36 ms

*Client and topic leader is on the same instance.

4.3.2 Configuration Comparison

Raft provides reliability and redundancy for data topics in CANAL. As long as the majority of Raft replicas are working, a leader can be elected among the working replicas, and the Raft cluster is available. Even though the stale data is available in follower replicas, since the only leader can respond to CANAL queries and client requests, we consider a Raft cluster (and the topic it holds) as available, only when the majority of replicas are responsive.

In real-world scenarios, distributed applications that utilize pub/sub pattern collect data from multiple sources. These applications are often deployed in multiple locations with different characteristics. For instance, a distributed IoT application can benefit from placing data collection and computation at an edge cloud and storing the static analysis results and archival data on a remote public cloud. Placing computation near the source at the edge of a network often provides better response time and energy efficiency by precluding the need to move large amounts of data across long-haul networks. However, edge clouds are typically more failure-prone due to their smaller scale, low cost, and potentially harsh deployment environment. Therefore, when making a decision where to place the replicas, we must consider the tradeoff between reliability and performance in

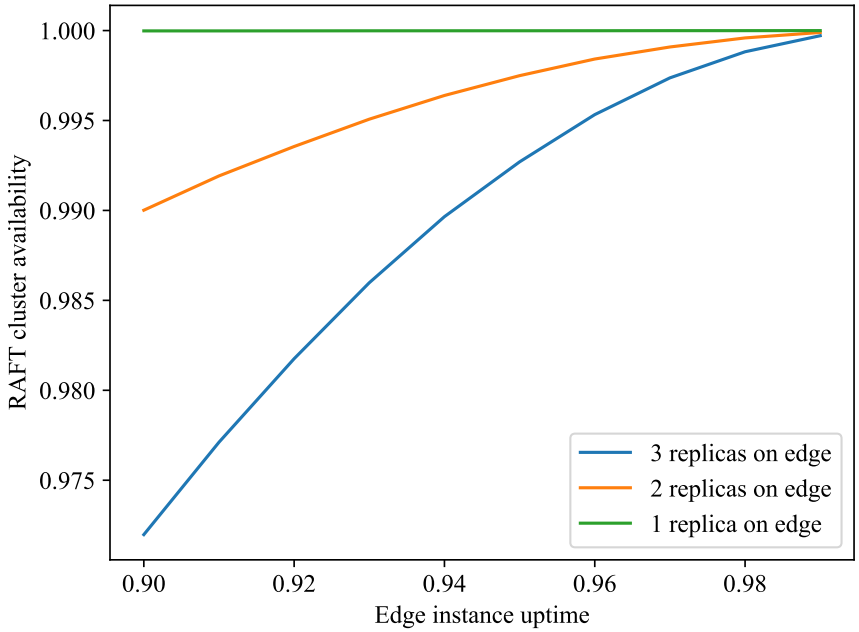


Figure 4.7: Availability of a Raft cluster with 3 replicas. A Raft cluster is considered available when the majority of its replicas are available.

such settings.

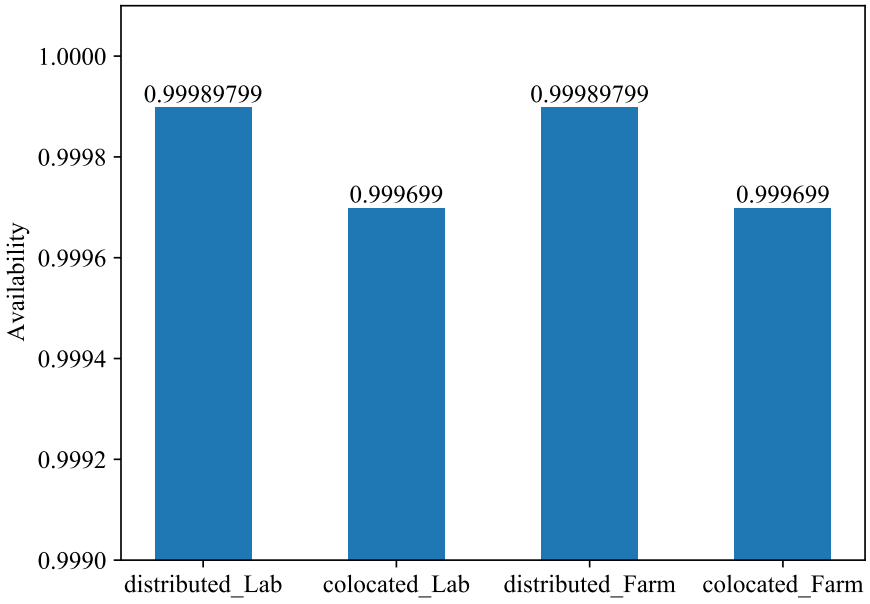
Consider a Raft cluster with 3 replicas placed in both a traditional cloud and an edge cloud. Assuming the traditional cloud has 99.99% uptime (a typical service level agreement of public commercial clouds), Figure 4.7 shows the probability of the cluster being available. Mixing traditional cloud instances to edge cloud instances makes the edge data availability higher in all cases. The more replicas placed on the traditional cloud, the more available the data is likely to be.

From the historical data, we perform an experiment using a real-world IoT application (as described in Section 4.3.5). We assume that our edge clouds (*Lab* and *Farm*) have an uptime of 99%. We also assume that the *Campus* cloud has a similar uptime as public clouds (99.99% uptime). Figure 4.8 shows the tradeoff between availability and performance for our cloud and edge configurations. In the $distributed_{Lab}$ and $distributed_{Farm}$

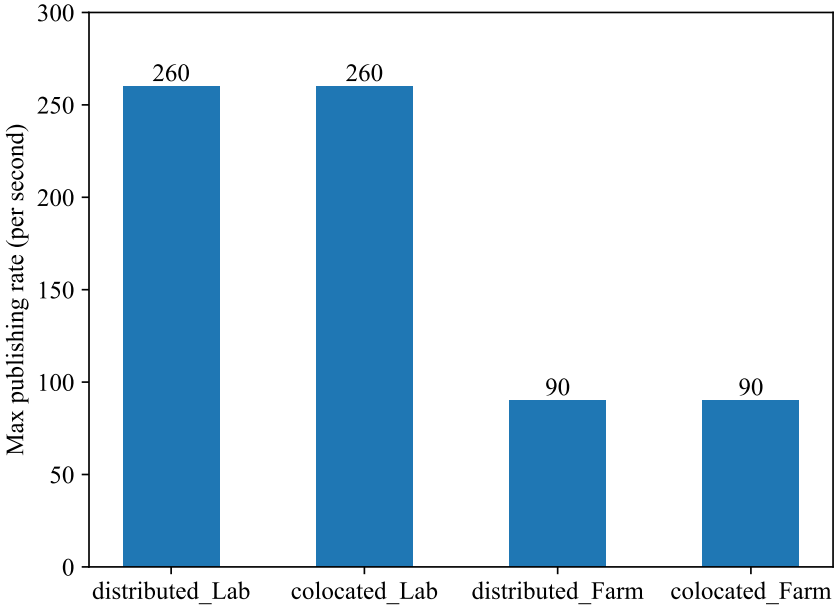
configurations, since we use one instance from each cloud, a Raft cluster has 2 replicas (*Lab* and *Farm*) on the edge. According to Figure 4.8a, both achieve availability of 99.9898%. In the $colocated_{Lab}$ and $colocated_{Farm}$ configurations, all replicas are on the edge. Both configurations have availability of 99.9699%.

Figure 4.8b shows the maximum publishing rate that the clusters can handle without buffering publish requests. Since the maximum publishing rate is limited by Raft leader's computation and network capacity, colocated configurations do not have an advantage over distributed configurations.

Figure 4.8c shows the latency a client must wait when calling *publish*() until when the data is replicated and the subscribing function is triggered. The latencies when the workload is light (API functions are called and measured sequentially without overlapping) and when the benchmark is publishing data at the cluster's maximum publishing rate are both shown. This experiment shows that when the Raft leader is in a cloud with high availability and connected with a high-performance network (*Lab*), the colocated configuration has slightly lower latency than its distributed counterpart. At the maximum publishing rate, the publishing latencies of $colocated_{Lab}$ is 616 milliseconds, 175 milliseconds lower than the $distributed_{Lab}$ configuration (791 ms). However, when the Raft leader is sited in a less capable edge cloud (*Farm*), $colocated_{Farm}$ does not achieve the same advantage. In the $colocated_{Farm}$ configuration, because all replicas are under stress from internal CANAL function invocations, replication, and request responses, the latency is higher than in $distributed_{Farm}$ configuration, where followers do not slow down the overall system.

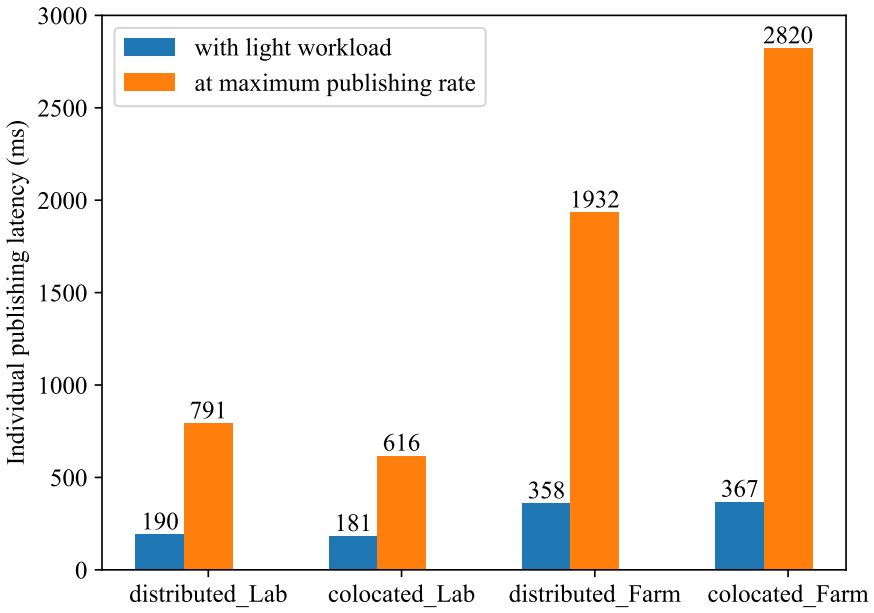


(a) Availability of different cluster configurations



(b) Maximum publishing rate each different cluster can handler without buffering requests

Figure 4.8: Comparison of 4 different CANAL cluster configurations



(c) Lowest publishing latency for each individual publishing in different cluster configurations

Figure 4.8: Comparison of 4 different CANAL cluster configurations

4.3.3 Throughput Comparasion

We compare the performance of CANAL to MQTT, the most commonly used topic-based publish/subscribe protocol in IoT. In order to set up a wide area distributed environment, we use RabbitMQ with MQTT client to deploy a broker network. RabbitMQ enhances the availability and reliability of MQTT brokers by supporting *clustering* and *federation*. A RabbitMQ cluster is a group of brokers whose status is replicated by the Raft algorithm. When the leading broker fails, another broker takes over its role. Federation is formed by multiple clusters in a wide area. Once registered, clusters can share and access the topics in other clusters in the federation.

RabbitMQ recommends forming a cluster with a group of brokers connected with a strong network. In the following test, we use six *Campus* instances to create two clusters (three instances per cluster) and form a federation of two clusters. We use Paho

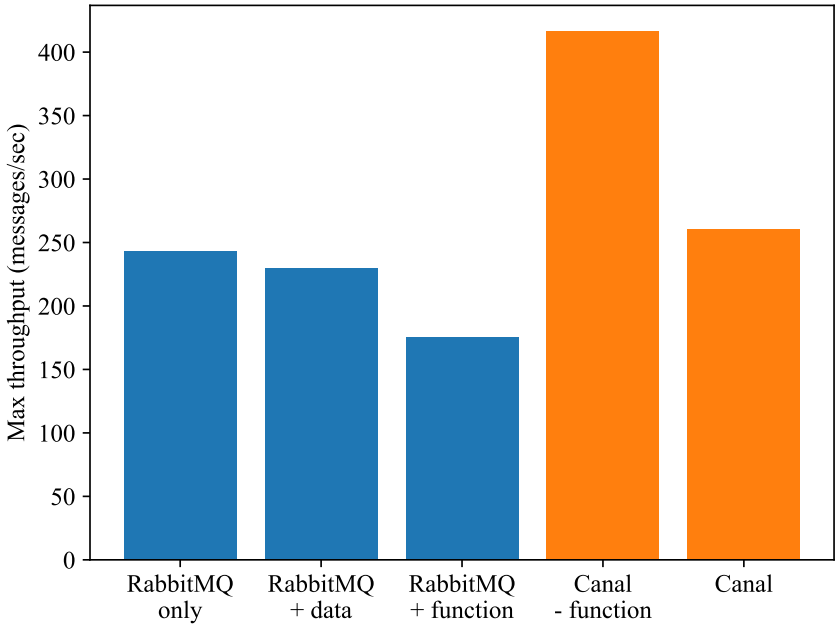


Figure 4.9: The throughput comparasion of RabbitMQ with MQTT and CANAL.

asynchronous MQTT library to implement consumer and producer clients in C. The consumer client is placed in one cluster, while the producer client is in the other. QoS level is set to 1 (at least once delivery) as RabbitMQ does not support QoS level 2. The consumer keeps consuming messages sent by the producer, and the producer keeps publishing messages with a fixed size of 8KB. To determine the maximum throughput of the setup, we keep increasing the number of messages the producer publishes per second until RabbitMQ message queue begins to buffer.

We compare the throughput of RabbitMQ 3.8.14 with MQTT clients to that of our system. Since RabbitMQ does not persist the data once the message is delivered, we use CSPOT's append-only data object to persist messages once the consumer client receives them. Once the consumer client receives the messages, it writes the message to all brokers' CSPOT objects. Since RabbitMQ only supports QoS level 1, this method does

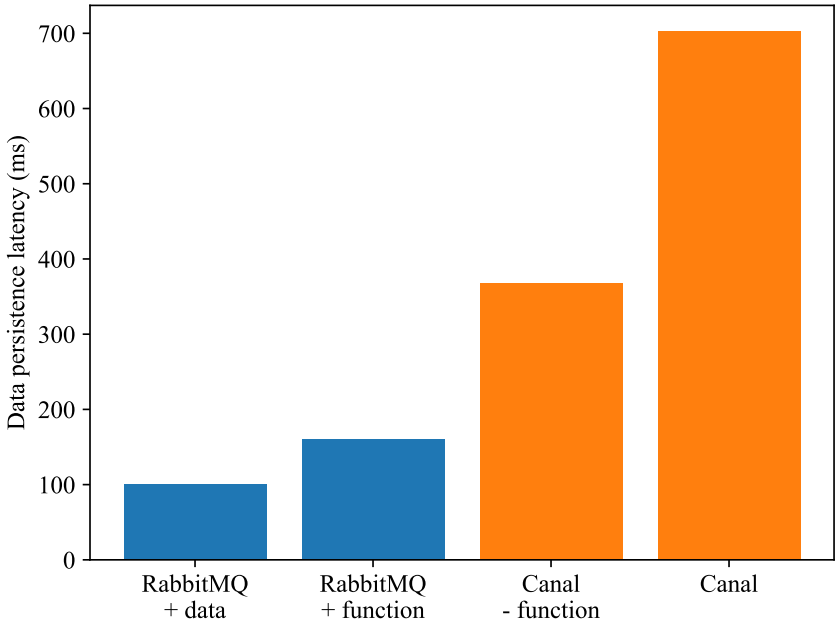


Figure 4.10: The data persistence latency comparasion of RabbitMQ with MQTT and CANAL.

not actually replicate the data, but it provides an illustration of how data persistence affects performance. In addition, even though one can implement application logic in consumer client, we trigger a separate function using CSPOT for each received message to measure the performance impact of event-based function invocation.

Figure 4.9 shows the comparasion result. Without persisting data and triggering functions, RabbitMQ can route 243 messages per second. When storing data to CSPOT, it can route 230 messages per second. If triggering function, it can handle 175 messages per second. Compared to RabbitMQ federation, CANAL can process 260 messages per second using the same instances and network. If we remove the function triggering from CANAL, it can route 416 messages per second to the topic. Since data persistence is a core design of CANAL, we are not able to strip it from the comparison test. Figure 4.10 shows the latency it takes for a message to persist in CSPOT's data object.

4.3.4 Failure Recovery

CANAL uses Raft to replicate data. As long as the majority of Raft replicas are available, the data in the cluster is available. However, since all client requests are directed to the leader and only the leader can respond to the requests, how fast a Raft cluster can elect a new leader when the old one is down plays a key role in CANAL's availability.

To evaluate the impact of leader election time in CANAL, we launch a Raft cluster with 3 replicas from different clouds and repeatedly shutdown the leader and measure the time it takes to elect a new one. The timeout value is set to 1 second at the minimum and 2 seconds at the maximum. The test is repeated 1000 times. The results (omitted due to space constraints) show that, on average, it takes 1820 milliseconds for a new leader to be elected (with a standard deviation of 537 milliseconds). During the 1820 milliseconds, the topic is unavailable until a new leader is elected. On average, it takes 1.5 seconds for replicas to discover a leader failure, and the election process requires 320 milliseconds.

Because the Raft timeout value is a configurable parameter, it can and should be configured according to the network condition of the deployment. In the tests, we find that failures do not happen frequently, and our applications can tolerate the topic being temporarily unavailable. If faster recovery and better availability are required, the timeout value can be adjusted accordingly.

4.3.5 End-to-end Application Performance

To evaluate CANAL in a real-world scenario, we implement and deploy a temperature prediction application that uses CPU temperature of IoT devices to predict the ambient temperature of the surrounding environment. Recent studies[11, 12] show that it is possi-

ble to accurately predict temperature using multiple regression across CPU temperatures of low cost, single board computers (SBCs). We use this IoT application to demonstrate CANAL and use it to evaluate its performance end-to-end.

During the implementation and deployment, we assume no knowledge of the underlying hardware, data storage, network configuration and condition. The data and functions are programmed against the abstraction of topic and data index, illustrating the programmability of CANAL.

We deploy 4 Raspberry Pi Zeros in *Farm*, the same research reserve we deploy our edge cloud. Each device has onboard sensors that collect and report CPU temperature readings. An external sensor attached to one of the devices capture the environment temperature readings (ground truth), which we use to train the regression model. Only CPU temperatures and this model are used to make predictions; ground truth is used to compute the accuracy of the predictions.

Figure 4.11 shows the application architecture. There are three functions in the application: *smooth*, *train*, and *predict*. Each Pi Zero publishes its CPU temperature reading to the corresponding *cpu_raw* topic every 5 minutes. Raw readings trigger the *smooth* function. *Smooth* reads the readings from the past hour and publishes the average to the CANAL *cpu_smooth* topic. When publishing the average, *smooth* also aligns the timestamp of reading to the closest 5 minutes mark since each device may publish its reading a few seconds apart. Besides CPU reading, the Pi Zero with an external sensor attached also publishes the ground truth temperature every 5 minutes to *env_raw* and publishes the average to *env_smooth*. *Train* subscribes to the *env_smooth* topic and is triggered every time a new smoothed environment temperature reading is published. *Train* checks if the latest model in the *model* topic was trained recently (within 3 days). If not, it deems the latest model expired and collects the data in *cpu_smooth* and *env_smooth* in the last 3 days and applies multi-regression. The resulting

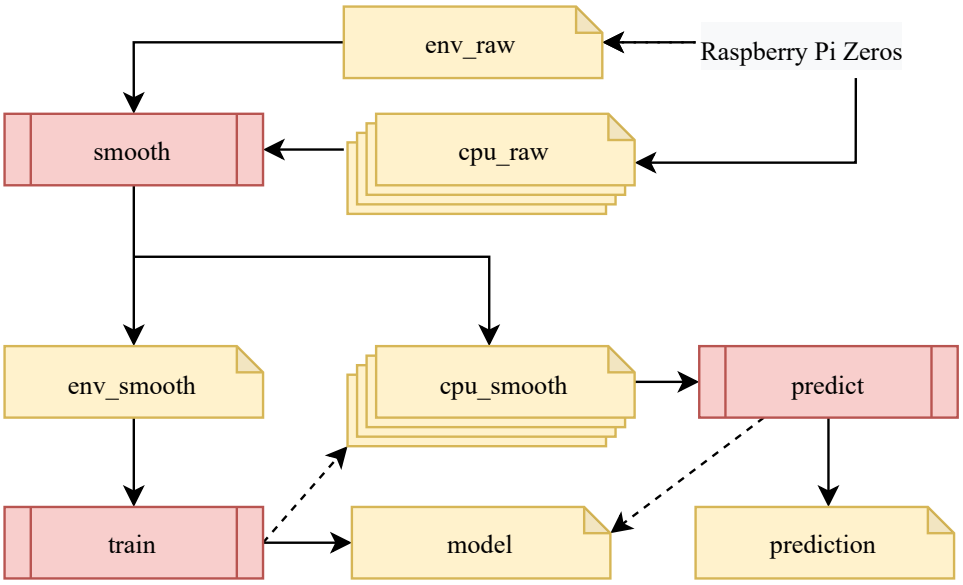


Figure 4.11: Temperature prediction application architecture. A topic having a solid arrow pointing to a function means the function subscribes to the topic. Dotted line indicates the function reads from the topic for extra input.

model is published to *model*. When there is new data published to *cpu_smooth*, the *predict* function is triggered. *Predict* checks if CPU readings of the same timestamp (within a 5 minute window) are presented in all 4 *cpu_smooth* topic. If so, it reads the latest regression model and makes a prediction. It then publishes its result to the *prediction* topic.

We collect one month of data (about 8880 samples per topic) in January 2021 and use the data to drive the application. Figure 4.12 shows the data plotted in time-series format. With CPU temperatures from four Pi Zeros as input and environment temperature as the ground truth, we are able to make temperature predictions with an average error of 0.21 degree and 95 percent of time less than 0.54 degrees. The error is within the range that is useful for real world agriculture applications. The average function invocation latency in the application run is 311 milliseconds with a standard deviation of 76 milliseconds. Figure 4.13 shows the CDF of the function invocation latency.

To evaluate how quickly the system can recover from leader failure, we simulate

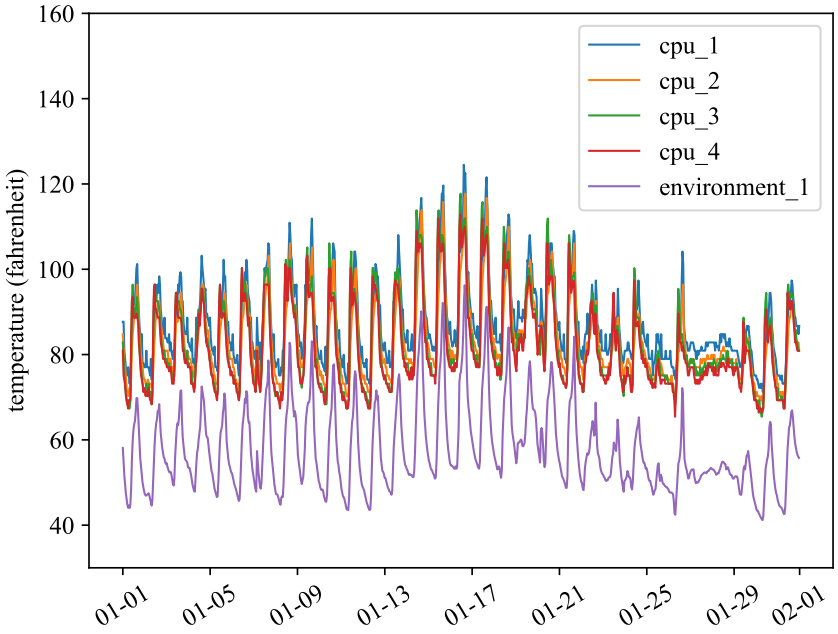


Figure 4.12: The temperature data reported by Raspberry Pi Zeros in January 2021.

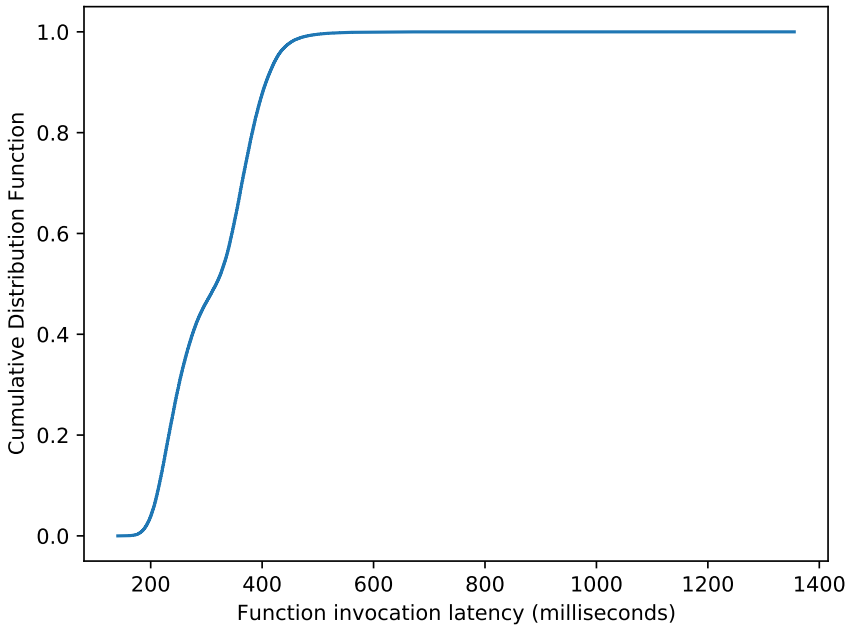
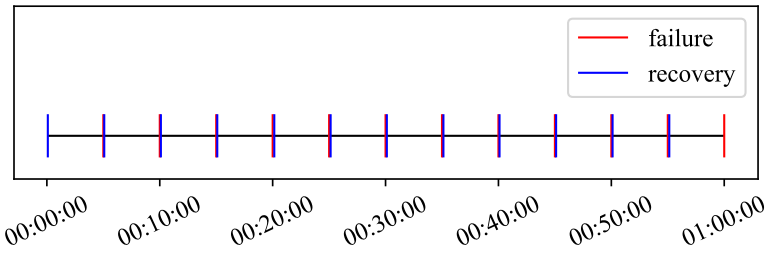
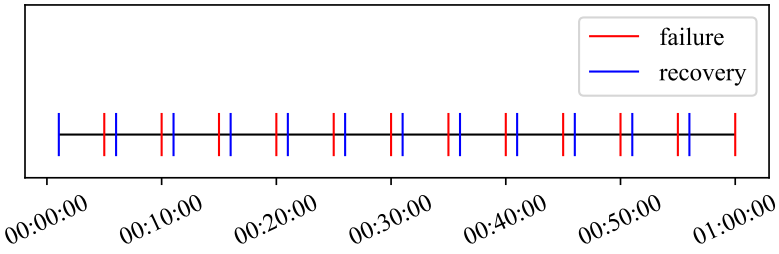


Figure 4.13: The CDF of function invocation latency of the application.



(a) Timeout value set to 5 to 10 seconds. The average time a node is unavailable is 6.63 seconds.



(b) Timeout value set to 60 to 65 seconds. The average time a node is unavailable is 61.02 seconds.

Figure 4.14: An hour long availability of the application with failure injected every 5 minutes.

failure by disabling the network of the current leader instance every 5 minutes. Then, we measure the time it takes for the Raft cluster to elect a new leader and become available again. Once it becomes available again, we enable the network of the previous disabled instance again to allow it to rejoin the cluster. We configure the Raft timeout range to 5 to 10 seconds and 60 to 65 seconds and run the test for 3 hours for each setting. Figure 4.14 shows an hour of the result's time series.

4.4 Conclusion

In this chapter, we present a programmable and reliable pub/sub system called CANAL for IoT. CANAL implements function-as-a-service execution model to incoporate

the event-driven programming into application implementation. Combining with pub/-sub messaging, CANAL allows IoT developers to write function against data instead of hardware and network address. Hence, CANAL greatly reduces the implementation complexity of event-driven IoT applications. CANAL implements DHT for fast and scalable topic lookup. For reliability, CANAL uses Raft to replicate data.

CANAL extends the widely adopted topic-based pub/sub messaging pattern in IoT for data delivery and service discovery. However, unlike the traditional pub/sub systems, CANAL allows developers to exploit data and function locality to optimize multi-tier IoT applications for reliability or performance. CANAL achieves the above by decoupling the DHT topology and the replica placement.

To evaluate the performance of CANAL, We use benchmarks and a real-world IoT application and different replica placement strategies. We use two edge clouds and a large-scale private cloud that have different capacity and network characteristics for the experiements. We deploy CANAL cluster on these clouds using different strategies: co-locating and geo-distributing replicas. In addition, we also test the performance impact caused by the leader replica. The results show that CANAL is able to achieve publish data and trigger subscribing functions with low latency. Compared to MQTT, CANAL has slight advantages of latency and throughput while being more reliable and available. We also discuss and demonstrate the trade-off between reliability and performance of co-locating and geo-distributing strategies. We found that in most cases, co-locating replicas leads to better performance in sacrifice of reliability.

Chapter 5

GammaRay: FaaS Application Monitoring

In the previous chapter, we have shown that FaaS programming paradigm greatly simplifies parallel and concurrent programming. Combined with efficient data delivery and reliable data replication, FaaS reduces the implementation complexity and allows developers to design and deploy IoT applications with greater ease.

However, the challenges of IoT applications does not only reside in implementation and deployment. the complexity of the asynchronous programming model of FaaS requires tools that developers can use to reason about, debug, and optimize their applications. Today, such tooling for these new applications is nascent, with only simple logging services available. Logging forces developers to write complex secondary applications that download, aggregate, analyze, and provide effective anomaly alerts or feedback. Such effort is error-prone, takes focus away from innovation, and must be repeated for every application.

While some cloud vendors provide tools for event logging and monitoring on their FaaS platforms, these tools are limited in that they provide only basic performance metrics

and do not trace interdependency across services and regions. Moreover, these tools are cloud-specific and only work with the FaaS platforms they are designed for. Therefore, in the modern landscape where applications can be deployed to multiple clouds, these tools cannot provide the debugging and analyzing capabilities required by IoT applications.

To understand the current limitations of available post-deployment FaaS toolings, we study and investigate AWS X-Ray[154], a sampling-based application monitoring service designed for AWS Lambda. As a result, we present GammaRay as an alternative to X-Ray that provides better debugging and monitoring functionality. GammaRay is a cloud service for Lambda applications that provides a holistic view of causal application behavior and performance end-to-end. It requires no developer intervention and works across AWS regions and AWS cloud services. GammaRay intercepts Lambda function entry points and calls to AWS services made by the application. It records these events synchronously using transactional database streams (to guarantee causal consistency) and processes them offline, in near real-time, to provide developers with service graphs and analysis data at both the function aggregate and instance level. As such, GammaRay provides a causal ordering for concurrent, multi-function Lambda applications.

We investigate the overhead introduced by GammaRay using micro-benchmarks and multi-function FaaS applications. In terms of monetary cost, GammaRay is able to track the causal order of events with an additional $0.00000004 per Lambda function and $0.03315 per hour for maintaining the transactional log. For the Lambda applications that we consider, this translates to less than 4 cents per hour.

As our first step into the journey of post-deployment operation research of IoT applications, GammaRay provides significant and useful insights that help us better understand the interaction and relation between application components. While GammaRay's usage is confined in AWS Lambda applications, the results we gathered from GammaRay serve as guidelines for us to build our own event tracking and data repair systems in later

chapters.

In the sections that follow, we first discuss the current status of FaaS toolings in Section 5.1. Then, we illustrate the design and limitations of X-Ray in Section 5.2. We present our design and implementation of GammaRay that aims to improve and address the limitations in Section 5.3. In Section 5.4, we show the evaluation result of the performance overhead introduced by GammaRay. Finally, we summarize our contribution in Section 5.5.

5.1 Related Work

Functions-as-a-service is an event-driven programming paradigm. Functions in FaaS are triggered by events such as writes to database and blob storage, HTTP requests, or messages. The causalities among these events compose a partial order on the events representing their "causal order".

Causality is an important tool employed in concurrent and distributed systems that facilitates reasoning about, analyzing, and drawing inferences from a computation.[155, 156] In particular, causal order is required for function design (to enable mutual exclusion, consistency, deadlock detection), for distributed debugging, failure recovery, and inconsistency detection, for reasoning about progress (termination detection, collection of obsolete data and state), and for measuring and optimizing concurrency.

There are multiple tools for tracking the performance of FaaS applications and tracing their interdependencies. AWS X-Ray[154] is a tracing tool for AWS that samples the entry and exit of Lambda function instances using unique trace identifiers. It records function duration and times SDK calls and HTTP accesses that a function makes. This data is sent to an X-Ray logging service via UDP. The X-Ray logging service visualizes and presents data to developers as logs and dependency trees, called service graphs.

Google employs a similar production tracing infrastructure for its cloud-wide services called Dapper[157]. The paper describes how they achieve low overhead, application-level transparency, and ubiquitous deployment at a large scale. Google has made Dapper[157] available for public use. Both X-Ray and Dapper can infer relationships between serverless functions but cannot capture causal order of events because (i) they only sample events, missing uncommon and rare events, and (ii) they do not trace through the services that functions access (which trigger other functions).

This lack of support limits the degree to which developers can identify the root cause of errors, performance bottlenecks, cost anomalies, and optimization opportunities for Lambda applications.

There are many attempts to provide different lacking aspects of the current FaaS toolings. The Serverless framework[158] simplifies the process of developing Lambda applications. With an offline plugin, users debug Lambda functions locally. Docker-lambda[159] is a reverse-engineered sandbox that replicates AWS Lambda. It supports all Lambda runtimes and guarantees the same behavior of on AWS Lambda. Josef Spillner studied FaaS and implemented Snafu[160], a modular system compatible with AWS Lambda, which is useful for debugging Lambda applications. New Relic[161] and Dashbird[162] provide AWS Lambda monitoring by summarizing data from Amazon Cloudwatch[163]. Zipkin[164] is a distributed tracing system based on Google Dapper[157].

Capturing causal ordering in support of debugging and performance analysis is well understood and has been extensively researched. Schwarz et al.[165] shows that characterizing the causal relationship is key to understanding distributed applications. Bailis et al. revisits causality in [166] in the context of real-world applications and proposes a number of interesting extensions to the model.

Other research contributes new approaches to achieving causal consistency in distributed and scalable datastore systems. Lloyd et al. proposed COPS[167], a key-value

store that delivers causal consistency across a wide area. They identify and define a new consistency model, causal consistency, with convergent conflict handling. Bolt-on[168] is a system proposed by Bailis et al., which provides a shim layer that provides causal consistency on top of general-purpose and widely deployed datastores. Saturn[169] is a metadata service for geo-replicated data management systems. It can be used to ensure that remote operations are in a visible order that respects causality. Saturn has been evaluated in Amazon EC2, and the work demonstrates that weakly consistent datastores can provide an improvement (via causal consistency) over eventually consistent models.

5.2 Monitoring AWS Lambda Applications

There are two performance monitoring services available to AWS application developers: CloudWatch and X-ray. CloudWatch is a service that collects information about AWS service and resource use. It also includes the ability for applications to write their own performance records and an API for filtering, downloading, and reading CloudWatch logs. Accessing CloudWatch via the API, however, is limited (e.g., 5 transactions per second per region) with commonly long delays (on the order of seconds) between event execution and the availability of its log record (imposed for scaling and system stability purposes).

CloudWatch logging is available for AWS Lambda functions in all AWS regions. However, log streams are local to a region and may or may not be distinct for concurrent invocations of the same function. Developers must write complex applications (potentially as Lambda applications themselves) to extract actionable insights about the performance and behavior of their Lambda applications, which is tedious and error-prone given the use limits, region isolation, and eventual consistency of CloudWatch logging. Such efforts

are infeasible and costly for even medium-scale AWS Lambda applications, which can consist of hundreds to thousands of function instances.

To address some of these limitations for highly dynamic Lambda applications, web applications, and microservices, AWS provides X-Ray. X-Ray is a monitoring and display service that automatically samples the entry and exit of function instances, called segments, using unique trace identifiers (`trace_id`). When a sample is taken, X-Ray records function duration and container startup overhead, and the duration of dynamic function SDK calls and HTTP accesses, as *subsegments*. Users can define, annotate, and record their own subsegments. X-Ray data is sent to an X-Ray daemon running in the container with the function via UDP. The daemon buffers and sends monitoring data (when sampled) to the X-Ray logging service.

The X-Ray service presents data to developers as logs and dependency trees, called service graphs. X-Ray links the segment and its subsegments (per `trace_id`) into an application's service graph as leaf nodes with meta-information, such as the DynamoDB table name that the function updated and the region in which it is located. Service graphs visualize X-Ray log data for specified time durations (aggregating multiple invocations of the applications). X-Ray data is automatically deleted after 24 hours.

Figure 5.1 presents an X-Ray service graph for an example AWS Lambda application [170]. The application control flow is as follows (captured by the service graph). The application, called ImageProcPyF, is triggered by a user uploading a photo to an S3 bucket. The function invokes the AWS Rekognition image processing service (via the SDK) on the photo and writes the labels returned in a DynamoDB table (image-proc-F). The function then updates a web page (cf the requests object) and exits. The table write triggers a second function (DBSyncPyF), which copies the data across regions (reading table image-proc-F in the west region and writing to table eastSyncTable-F in the East region). This second write triggers a third function (UpdateWebsiteF) in the East region,

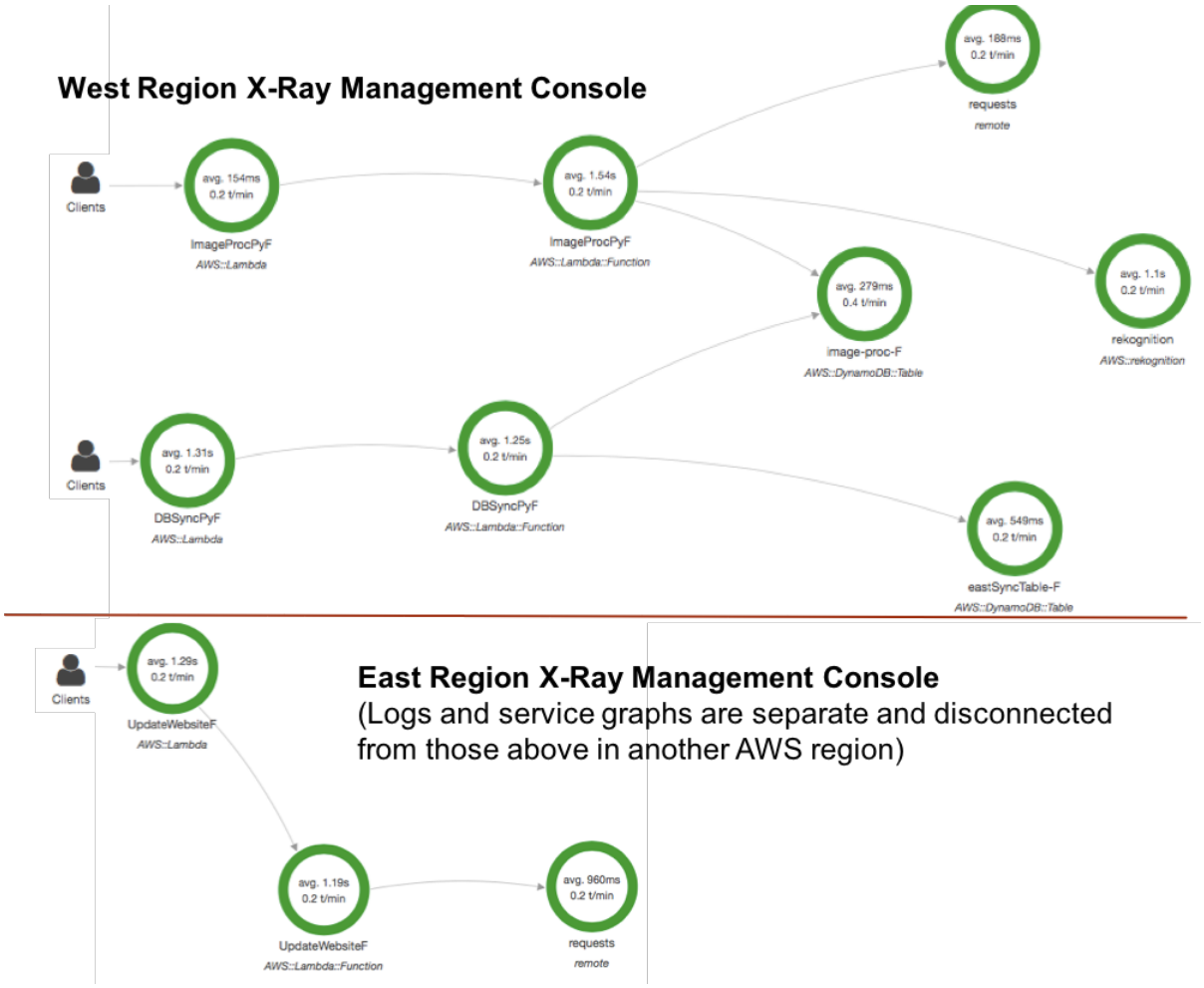


Figure 5.1: X-Ray service graph for ImgProc [170].

which writes to a web page in its region.

When a function is triggered by an unknown source, the service graph represents this via a `Clients` icon. X-Ray divides the function into two parts (subsegments): its startup overhead (type AWS::Lambda) and its execution time (AWS::Lambda::Function). Multiple instances of functions are combined into aggregate service graphs by X-Ray. However, within the raw data of the logs from which the service graphs are drawn, there is a segment with a unique `trace_id` for each service graph with an unknown source. The segment consists of metadata and subsegments (for SDK calls, HTTP requests, and

any user-defined operations). The metadata includes start and end times, `trace_id`, and details about the operation type and outcome (e.g., error status, if any). Segments and subsegments are linked via `Id`'s and `parent_id`'s for subsegments with the same `trace_id`. Thus it is possible to construct an ordering of events (subsegments) that originate from the same function (parent) but not across top-level segments.

Figure 5.1 reveals multiple limitations of X-Ray. First, even though ImageProcPyF triggers DBSyncPyF and DBSyncPyF triggers UpdateWebsiteF via DynamoDB table updates, X-Ray logs and service graphs do not capture these relationships. Similarly, across regions, functions are disconnected and part of independent X-Ray traces. Moreover, the only option for viewing service graph data is in aggregate; only raw log data contains per function instance data.

Other X-Ray limitations relate to record loss. X-Ray uses statistical sampling of performance information. Highly scalable applications, "rare" events that exercise code paths that are difficult to test can cause faults that are difficult to reproduce and diagnose. If the events are sufficiently rare, a statistical technique may miss them. In addition, X-Ray uses UDP messages to a separate process in the container to offload logging overhead. Because UDP is an unreliable network transport mechanism, it may be that messages are lost before their content is logged. Given this implementation, it is not possible to use X-Ray alone to construct the causal order of events across Lambda applications.

5.3 GammaRay Design

To facilitate causal order tracking for serverless applications in AWS that is *cloud-wide* – across all AWS services and regions – we have developed a cloud service for AWS called GammaRay. GammaRay extracts causal precedence for Lambda applications by monitoring both function-internal events (like X-Ray) and the message-passing performed by

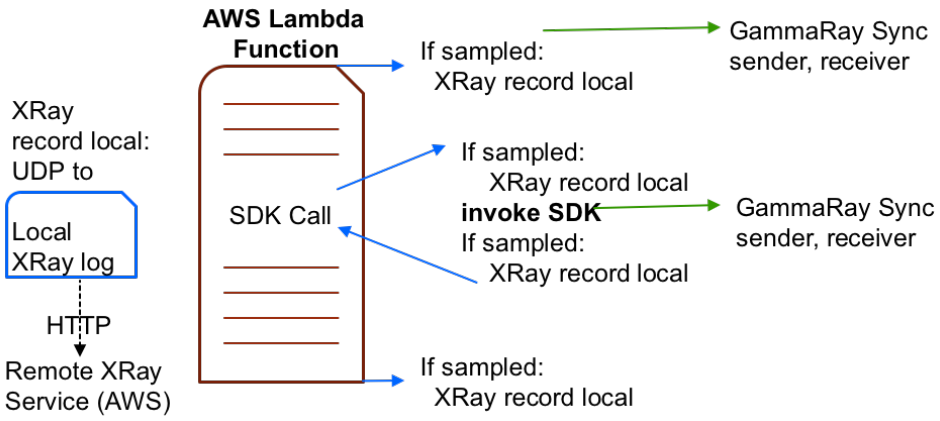


Figure 5.2: GammaRay Overview. GammaRay automatically injects support that captures the entry/exit and SDK (cloud service access) of AWS Lambda applications. Each SDK invocation carries a trace identifier through the call. GammaRay extracts the identifier upon function entry for any triggered functions.

functions *through* AWS cloud services (which X-Ray does not). Moreover, GammaRay augments causal relations with both aggregate and instance-level performance data. To enable this, GammaRay automatically injects instrumentation into Lambda functions and into the SDK with which they invoke cloud services (event sources that trigger other functions) upon function deployment to AWS Lambda. GammaRay considers messages in this setting to be SDK calls between functions and services.

Figure 5.2 depicts a Lambda function with X-Ray support. On the left is the local X-Ray logging progress, which receives UDP messages from the function for sampled events. X-Ray records performance information before and after Lambda function execution and invocation of SDK calls (calls to other AWS and web services). GammaRay augments this support with synchronized recording (GammaRay Sync in the figure) of the sender and receiver for each function invocation and immediately prior to any SDK call to a potential event source (function-triggering service invocation). GammaRay consumes these records offline to compute causal relations and performance statistics for each event and to construct a service graph that can be easily interrogated by developers and analysis

tools.

The GammaRay design consists of three components: a Lambda function deployment tool, GammaRay runtime support, and GammaRay event processing engine. The function deployment tool takes a code directory and a list of libraries and builds a code package that it then uploads to Lambda using the developer's credentials. The tool filters out unused libraries to minimize package size. If GammaRay support is requested (a command-line option), the tool injects GammaRay instrumentation to perform synchronized record-keeping by replacing the function entry point with the GammaRay entry point and by "wrapping" AWS SDK and HTTP operations.

GammaRay records metadata about each event synchronously in a database that tracks update order (implements a transactional log). This metadata includes unique function ID and details about the SDK call such as table name and keys for DynamoDB updates, bucket name, prefix, and key for S3 updates, SNS topics, and HTTP URLs. GammaRay uses this metadata offline to map event sources to functions to form the causal order of events across AWS Lambda applications, services, and regions. GammaRay augments this information with performance data about each event. GammaRay can obtain this data from X-Ray samples, or it can collect the data itself (with or without sampling) via additional instrumentation (inserted at the same points as X-Ray instrumentation). We consider these various implementation options in the next section.

As mentioned previously, when a Lambda function invokes another (via the SDK or HTTP) there is no trigger-identifying information available in the callee. To overcome this limitation, GammaRay injects the caller's unique (request) ID into the payload of the SDK invocation as a hidden argument. This data is later used by the GammaRay event processing engine to map cloud service updates (event sources) to function invocations and to produce causal relations across the application.

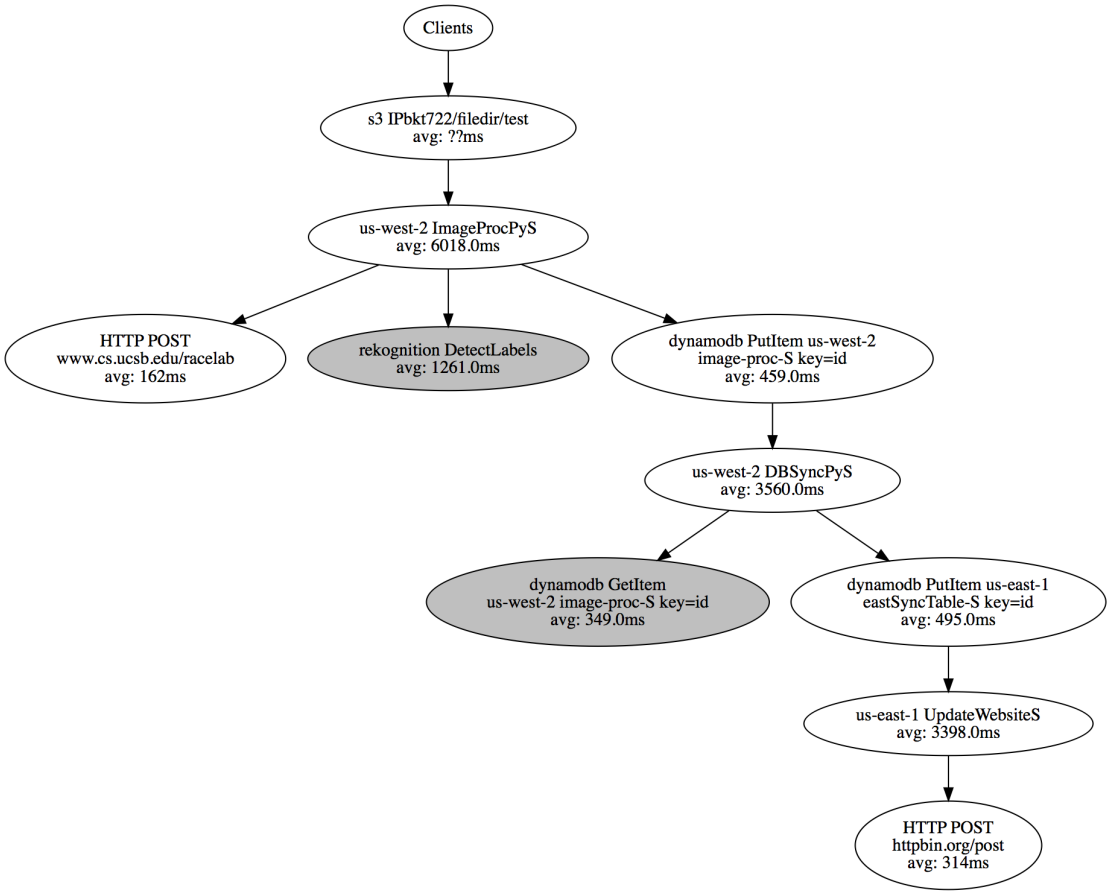


Figure 5.3: GammaRay service graph for ImgProc [170]. GammaRay captures causal dependencies and performance through AWS services and across regions.

5.3.1 Implementations

We next investigate three alternative implementations of GammaRay: `G-Ray-D`, `G-Ray-S`, and `G-Ray-H`. In all three, GammaRay automatically inserts "hidden" arguments into function invocations as needed, and processes all function arguments upon function entry. Additionally, all configurations implement the GammaRay log via a shared DynamoDB table and stream. DynamoDB Streams record the sequence of record-level DynamoDB table modifications and thus enable GammaRay to extract the causal relationships across events that it records (in the order they occur).

`G-Ray-D` injects the necessary GammaRay instrumentation dynamically using a library that "monkey-patches" [171] AWS Lambda SDK calls made by the function to invoke the GammaRay runtime before and after the call. It represents the most flexible, portable, and application-transparent implementation strategy. Alternatively, `G-Ray-S` implements the same functionality by adding the instrumentation code *statically* to the AWS Lambda Python SDK. It increases the size of an instrumented Lambda program (both in terms of memory and package size) but avoids dynamic runtime instrumentation overhead.

`G-Ray-D` and `G-Ray-S` are full replacements for AWS X-Ray (i.e., they also collect event performance data). They improve upon X-Ray in two ways. First, they track causal order across AWS regions and across service invocations. Secondly, they track *all* events (rather than a statistical sample,) so they can be better used for performance debugging activities such as diagnosis of faults due to rare events (X-Ray uses statistical sampling). Moreover, because they only depend on AWS's scalable database (DynamoDB), and this functionality is relatively common among public cloud providers, in theory, these implementations could be ported to other public clouds.

Alternatively, `G-Ray-H` is an AWS-specific implementation of GammaRay that makes maximal use of extant AWS services, including X-Ray and CloudWatch. It implements the same causal-ordering tracking as *G-Ray-D* and *G-Ray-S*, but because `G-Ray-H` relies on AWS for function timings, its performance data is sampled.

All implementations use the AWS SDK (boto [172]) and `G-Ray-H` and `G-Ray-D` rely on the Fleece library for X-Ray daemon support [173] for Python. The GammaRay deployment tool also uses the SDK to upload the compressed Lambda package to AWS Lambda, set up the necessary policy and permissions for the functions, and configure any event sources that trigger function invocation.

Our goal with the GammaRay implementation is to leverage as much of it as possible

via AWS cloud services for each of the three GammaRay components. The deployment tool, implemented in Python, uses the AWS SDK (boto [172]) to construct a compressed package containing the function and the libraries it uses, uploads it to AWS Lambda, and sets up the necessary policy and permissions for the function, configuring any event sources that trigger function invocation. For the runtime, GammaRay currently supports Lambda functions written in Python.

5.3.2 GammaRay Event Processing Engine

The GammaRay event processing engine runs offline – in the background and so does not introduce overhead on serverless applications. The engine processes the table data in append-order via the DynamoDB Stream. From this information, it constructs a service graph containing causal order dependencies for each application across AWS services and regions. It presents this data to users as graph aggregates (as X-Ray does) or for individual function instances (which X-Ray does not) and annotates the graph with performance data. The amount and type of data with which GammaRay annotates its graphs are configurable.

Figure 5.3 shows the service graph for the ImgProc application for one run of the `G-Ray-S` configuration. GammaRay leverages graphviz [174] for its service graph implementation. In this configuration, the engine displays SDK operation names, key names, and average performance across event instances. Because the S3 write is performed by a user (Clients) directly, the average time is not available (denoted ??ms in the figure). GammaRay displays non-event-source operations (e.g. DB reads) in gray and errors in red.

App	Description
empty	Micro: Returns immediately
DDB read	Micro: 100 random reads of DynamoDB table
DDB write	Micro: 100 random writes to DynamoDB table
S3 read	Micro: 100 random reads of random S3 object
S3 write	Micro: 100 creates of a new S3 object
SNS	Micro: 100 postings to SNS
Map-Reduce	A Big-Data-Benchmark [175] app implemented in Lambda by AWS Engineers [176]
ImgProc	Image Processing app [170]. Images uploaded to S3 trigger a function which extracts labels using AWS Rekognition service, and reads and writes DynamoDB tables within and across regions (performing geo-replication), and triggering a cross-region function

Table 5.1: Micro-benchmarks (demarked Micro) and Multi-Function Lambda Apps used to evaluate GammaRay. All are available from our project repository.

5.4 Evaluation

To evaluate GammaRay, as well as to illuminate the source of the overhead it introduces, we employ both multi-function Lambda applications and micro-benchmarks. We first overview these applications and our empirical methodology and then present our empirical results. We used only the AWS Free Tier for implementation and evaluation of this study (i.e., no costs were incurred for function invocation).

5.4.1 Methodology

The applications and micro-benchmarks that we use in this study are listed in Table 5.1. We present the baseline timings in milliseconds (ms) and memory used in megabytes (MB) for each in Table 5.2 and 5.3. For the micro-benchmarks, the DB payload is 4 bytes; the S3 operations are on empty files. We execute both sets of Lambda applications multiple times and compute the average and standard deviation. We execute the micro-benchmarks 200 times and the Lambda applications 50 times unless otherwise

App	Clean	X-RayND	X-Ray
empty	7 ms	10 ms	13 ms
DDB read	2,435 ms	2,761 ms	4,745 ms
DDB write	2,392 ms	2,927 ms	4,755 ms
S3 read	2,841 ms	3,134 ms	5,215 ms
S3 write	5,354 ms	6,073 ms	8,727 ms
SNS	4,217 ms	4,327 ms	6,433 ms
Map-Reduce	124,582 ms	122,156 ms	114,007 ms
ImgProc	3,418 ms	3,047 ms	3,067 ms

Table 5.2: Baseline performance data of total time for the micro-benchmarks and Lambda apps.

App	Clean	X-RayND	X-Ray
empty	21 MB	25 MB	41 MB
DDB read	25 MB	41 MB	44 MB
DDB write	41 MB	44 MB	65 MB
S3 read	44 MB	65 MB	64 MB
S3 write	65 MB	64 MB	49 MB
SNS	31 MB	35 MB	47 MB
Map-Reduce	1,175 MB	1,204 MB	1,231 MB
ImgProc	108 MB	113 MB	114 MB

Table 5.3: Baseline performance data of total memory consumed for the micro-benchmarks and Lambda apps.

noted. All run sets occurred in sequence so cold-start overhead is only experienced for the first run.

The baseline configurations, which we use for comparison and which include no GammaRay functionality, are `Clean`, `X-RayND` and `X-Ray`. `Clean` captures the performance of the application with all tracing turned off for all functions. `X-RayND` shows the performance of the stock AWS X-Ray with tracing turned on, but the data capture mechanisms do not use a separate X-Ray "daemon". Without this daemon option, X-Ray logs function entry and exit calls in Python applications but not the Python SDK calls that the function makes. `X-Ray` is full AWS X-Ray support for Python applications using the X-Ray daemon implemented via the Fleece library [173].

The baseline measurements reveal interesting characteristics about AWS Lambda. First, the `empty` micro-benchmark results (in which the function simply returns) show no statistical difference in either the mean or the variance of their execution times, either with or without tracing. This result seems to indicate that the X-Ray logs are updated asynchronously (i.e., there is intermediate buffering for which users are not charged).

Full X-Ray introduces overhead for both the DDBread and DDBwrite benchmark. Each benchmark reads/writes DynamoDB 100 times. In the case of `X-RayND`, only the start and exit of the benchmark are logged. For `X-Ray`, each of the internal 100 SDK calls to DynamoDB are also logged. Since the mean execution time approximately doubles, we conclude that internal SDK logging for DynamoDB using the X-ray daemon requires approximately 1/100 the time required for entry and exit logging. The same seems to hold for S3 reads but not for S3 writes (which take more time than reads. For long-running X-Ray does not wait but instead posts records that the operation is "in-progress" [177] potentially incurring more overhead for multiple log records.

For the multi-function Lambda applications, Map-Reduce and ImgProc, `X-Ray` executes in less total time than `Clean` (last two rows of top table in Table 5.2). We ran

a Student's t-test [178] on the datasets and find that their means are different. We do not have a good explanation as to why `X-Ray` is faster but believe that it is related to the AWS implementation and deployment of X-Ray. In our evaluation, we compare GammaRay to X-Ray for these applications.

5.4.2 Application Performance

We first empirically evaluate GammaRay for our long-running Lambda application: Map-Reduce. This application was written by AWS engineers and is based on one of the Big Data Benchmark programs [175]. This application implements the map-reduce protocol but relies only on AWS Lambda and S3 for its implementation. We use the pavlo/text/1node/uservisits dataset which is 24MB in size and contains IP addresses that have visited particular websites. The application invokes 29 mappers, which read their portion of the input from S3. Mappers count the number of access per IP prefix for a range of IPs and store the results in S3. A coordinator monitors this progress (via triggers from S3 writes) and invokes a single reducer function when all mappers complete. The reducer downloads the intermediate results and performs a reduction across them to produce the final per-IP count, which it stores in S3 (which again triggers the coordinator one final time).

Recall from Table 5.2, the application without GammaRay or X-Ray completes in approximately 125 seconds and 114 seconds with full X-Ray enabled. Figure 5.4 shows the percentage overhead versus full X-Ray introduced by GammaRay on the map-reduce application. On total time, `G-Ray-D` introduces 25.1%, `G-Ray-S` introduces 15.3%, and `G-Ray-H` introduces 11.9% overhead versus *X-Ray*. On memory use, `G-Ray-D` introduces 3.7%, `G-Ray-S` introduces 7.5%, and `G-Ray-H` introduces 4.4% overhead.

This overhead is primarily due to the instrumentation performed by each variant.

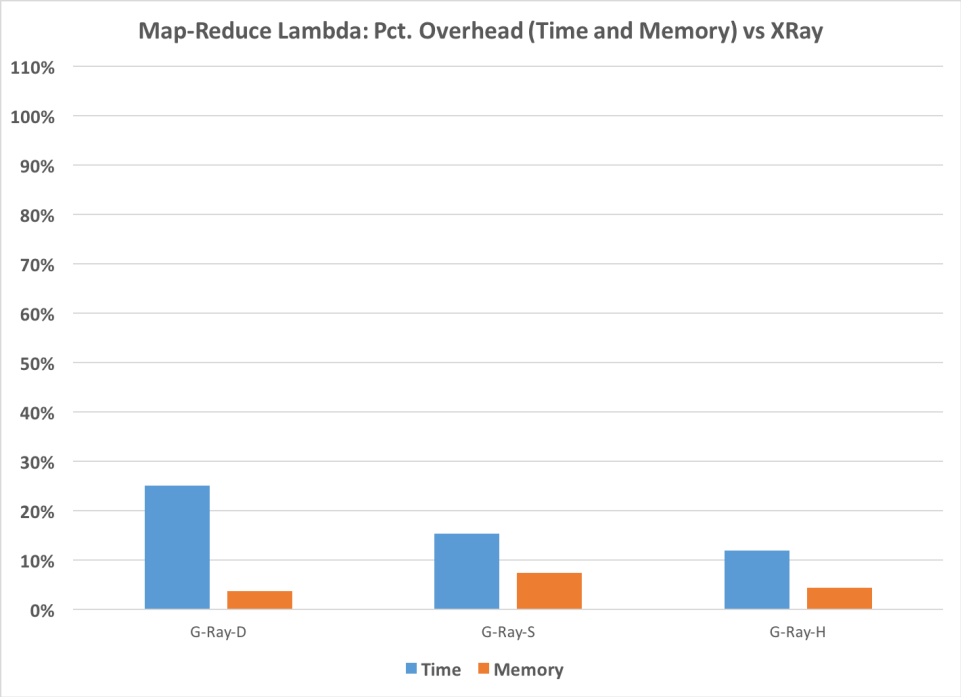


Figure 5.4: Percent overhead versus X-Ray for GammaRay for the Map-Reduce Lambda application. For each GammaRay variant (G-Ray-D=dynamic, G-Ray-S=static, and G-Ray-H=hybrid), we present the percent overhead on total time across functions and on memory used across functions, on average for 50 runs of the application.

On function entry, GammaRay parses and logs (in DynamoDB) function input data. `G-Ray-D` and `G-Ray-S` both also log exit events to DynamoDB (to record timings). `G-Ray-D` and `G-Ray-S` log before and after each SDK call to capture timings and causal dependencies; `G-Ray-H` records a log entry before each SDK call that might trigger other Lambda functions, to track causal dependencies. Moreover, X-Ray tracing is turned off for `G-Ray-D` and `G-Ray-S` (because it is not needed) and turned on for `G-Ray-H` which uses X-Ray data to annotate the causal service graph with performance data (offline).

The overhead of GammaRay is low for this application because the time spent not executing event-source-triggering calls is large relative (the application executes for over 124 seconds) to the number of calls that GammaRay instruments. For this app, `G-Ray-S` and `G-Ray-D` generate over 840 GammaRay tracing records; `G-Ray-H` generates

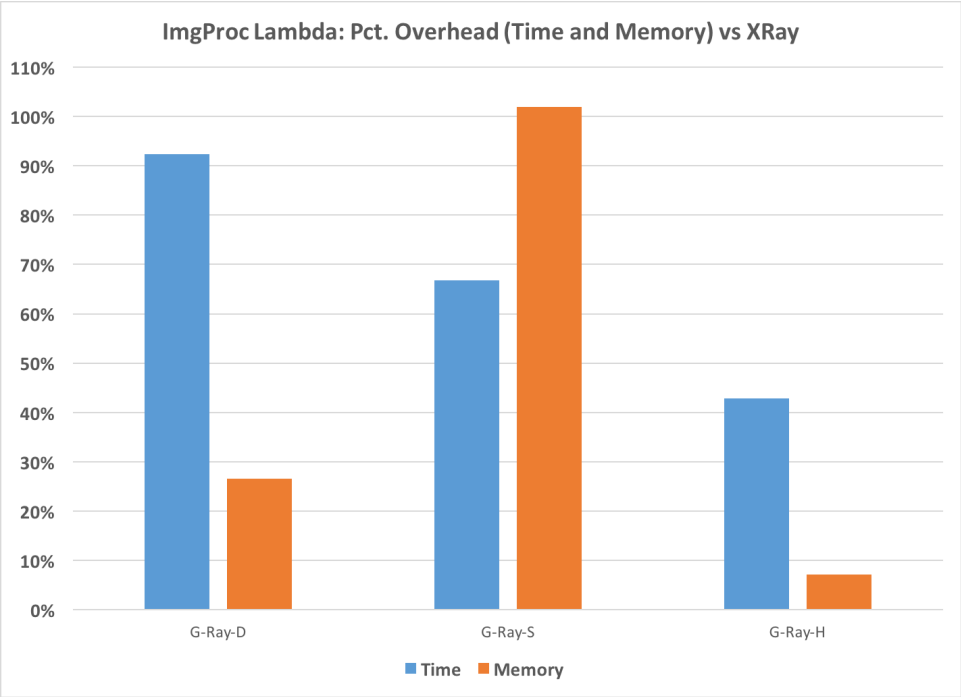


Figure 5.5: Percent overhead versus X-Ray for GammaRay for the Image Processing (ImgProc) Lambda application. For each GammaRay variant (G-Ray-D=dynamic, G-Ray-S=static, and G-Ray-H=hybrid), we present the percent overhead on total time across functions and on memory used across functions, on average for 50 runs of the application.

125 records. As a result, much of the time is spent in mapper and reducer functions for data processing.

We next evaluate the overhead of GammaRay for the short-running ImgProc application. ImgProc performs image processing and geo-replication of database tables. The application consists of three dependent functions (two in the East region and one in the west) that trigger each other via DynamoDB table updates (in both regions). Application execution is initiated by a file being placed in an S3 bucket.

Figure 5.5 shows the percentage overhead of GammaRay versus X-Ray for the ImgProc application. As shown in the baseline data, one instance of the app completes in 3.1 seconds and uses 114MB of memory for `X-Ray`. Because a single instance of this application is very short running, GammaRay consumes a significantly larger overall percentage

of total time than it did for Map-Reduce. For this application, `G-Ray-D` introduces 92.3%, `G-Ray-S` introduces 66.8%, and `G-Ray-H` introduces 42.9% execution overhead. In terms of memory use, `G-Ray-D` introduces 26.5%, `G-Ray-S` introduces 101.1%, and `G-Ray-H` introduces 7.2% overhead.

The implementation of `G-Ray-S` adds more memory overhead than the other variants. We believe this is because of the additional code footprint that we require for the GammaRay library extensions (we measure and discuss disk space usage further below). Moreover, a single invocation of the ImgProc application comprises 18 events that consume most of the execution time. GammaRay writes database records for all 18 events including entry/exit for configurations `G-Ray-D` and `G-Ray-S`. Configuration `G-Ray-H` posts only 5 records at during execution (those sufficient to capture the causal ordering). `Clean`, `X-RayND`, and `X-Ray` post no records during execution – all performance data is recorded via unreliable communication and eventually consistent, non-order preserving logs, asynchronously.

From the results of these two Lambda applications, we conclude that the execution overhead associated with tracking causal ordering across regions and AWS service invocations is lowest for the GammaRay–X-Ray hybrid (configuration `G-Ray-H`). This configuration enables GammaRay to use less memory and record the minimal set of events (required to identify causal relations across events) synchronously, and all other events asynchronously via the X-Ray daemon. We next investigate the overhead that GammaRay introduces at a finer grain using micro-benchmarks (the first six programs in Table 5.1).

Storage (MB)	Local project directory		/tmp directory	
	compressed	uncompressed	compressed	uncompressed
Clean	0.0002	0.0001		
X-RayND	0.0002	0.0001		
X-Ray	0.7311	2.2188		
G-Ray-D	0.7325	2.2266		
G-Ray-S	0.0021	0.0117	3.4353	23.8320
G-Ray-H	0.7330	2.9609		

Table 5.4: Container disk space usage for GammaRay wrapper and library support.

5.4.3 Container Disk Space Usage

As discussed previously, the empty micro-benchmark returns immediately when invoked. We use this micro-benchmark to evaluate the storage overhead GammaRay imposes on container disk space given the minimal `Clean` code package of this benchmark. The table in Table 5.4 reports disk space usage in megabytes (MB) for the Lambda function package (the function code and its libraries) that is downloaded and decompressed upon container instantiation when a function is invoked. Columns 2 and 3 show the size in MB for the package compressed and uncompressed. On average GammaRay increases compressed package size by less than 1% for `G-Ray-H`.

The size of the package is limited by AWS to have a maximum of 50MB compressed and 250MB uncompressed. Large package sizes also slow down function deployment times (including version replacement and code update). To keep deployment times low, libraries in the package can be placed in the `tmp` file system in the container. To use this option, developers package this code separately and upload it to S3. Upon invocation, the developer adds code to the start of the function that downloads, extracts, and links the code into the application. The GammaRay tool performs these operations automatically. AWS limits the maximum size of the `tmp` file system to 500MB.

We use this option for the `G-Ray-S` configuration. We do so because this configuration rewrites a small portion of the AWS SDK (botocore). AWS provides the SDK in the

Total Time Overhead (ms)	SDK (Overhead per operation)					
	DDB Read	DDB Write	S3Read	S3 Write	SNS	Avg
X-Ray (over Clean)	47	47	52	87	64	60
G-Ray-H (over X-Ray)	1	29	3	19	34	17
G-Ray-H (over Clean)	48	77	55	106	98	77

Table 5.5: Micro-benchmark total time result summary. At startup, the GammaRay wrapper introduces 125ms for obtaining a handle to the GammaRay database table from AWS and just under 300ms for processing function inputs and storing them in the table.

container for free. Because of the rewrite, GammaRay must include botocore in the deployment (to replace the default container version). By doing so, `G-Ray-S` has a very small project package and a large (compressed and uncompressed) `tmp` file system component as shown in the table using columns 3 and 4. We include the time required to download from S3 and uncompress the package in all `G-Ray-S` experiments. We find that if the function is executed repeatedly and AWS reuses the container, we can avoid this overhead. To do so, GammaRay first checks whether the downloaded package exists and if so, performs only library loading and linking.

We also believe that the additional `G-Ray-S` library code increases the overall memory footprint at runtime (cf `G-Ray-S` Memory in Figures 5.4 and 5.5). On average, however, GammaRay introduces a small overhead on container storage for its wrappers and additional libraries for both `G-Ray-D` and `G-Ray-H` versus X-Ray because it is able to leverage the same libraries as X-Ray for their implementation.

5.4.4 Micro-Benchmark Performance

We next breakdown the overhead of tracing on the remaining micro-benchmarks. We present results only for `G-Ray-H` (the best performing GammaRay configuration) due to space constraints. `G-Ray-H` keeps its overhead low by relying on X-Ray to collect performance statistics. Thus X-Ray must be turned on in this configuration (introducing

Memory Overhead (MB)	SDK (Total per benchmark)					
	DDB Read	DDB Write	S3Read	S3 Write	SNS	Avg
X-Ray (over Clean)	17	17	16	14	14	16
G-Ray-H (over X-Ray)	2	5	8	9	4	5
G-Ray-H (over Clean)	19	22	25	22	19	21

Table 5.6: Micro-benchmark memory consumption summary.

some overhead itself). Moreover, since X-Ray only performs sampling and its logs are eventually consistent (with delays of seconds in many cases), the performance information on the GammaRay service graphs is also subject to these disadvantages. However, GammaRay guarantees causal order for service graph connectivity through AWS services.

The two DDB micro-benchmarks execute random 100 reads and 100 writes to different AWS DynamoDB tables, respectively. The two S3 micro-benchmarks execute random 100 reads and 100 writes to AWS S3 buckets, respectively. And the SNS micro-benchmark posts 100 notifications to AWS SNS. The performance data for the `Clean` and `X-Ray` configurations to which we compare is shown in the baseline data (Table 5.2 and 5.3). The performance results for the micro-benchmarks is shown in Table 5.5 and 5.6. The top table presents data for total time overhead in milliseconds (ms) and the bottom table shows memory overhead in megabytes (MB) for `X-Ray` and `G-Ray-H`.

In Table 5.5, The first row of data in the top table shows the number of milliseconds added to `Clean` by `X-Ray`. `X-Ray` tracing overhead is lowest on DynamoDB reads and writes and highest on S3 writes and SNS notifications. We believe that this latter overhead is due to the multiple "in-progress" records that X-Ray posts for longer running operations such as these. We observe many such records for S3 write and SNS operations for these benchmarks.

The second row in Table 5.5 shows the additional overhead (over `X-Ray`) that `G-Ray-H` introduces. Since GammaRay relies on X-Ray for performance data, the total overhead

of GammaRay versus `Clean` is a combination of both X-Ray and GammaRay (last row in both tables). `G-Ray-H` overhead consists of obtaining a handle to the GammaRay DynamoDB table, parsing the function arguments to extract trigger information (either inserted by GammaRay or by AWS automatically), and synchronously writing a record containing this payload to the DynamoDB table. The payload contains a timestamp, the request ID, the function ID, and 4-8 additional strings describing the event source (triggering operation). Depending on the event source, this payload can vary in size from 16 bytes to an arbitrary length (e.g., a DynamoDB key, S3 bucket name, or SNS subject contains application-specific data, which can be large).

In our study, the largest payload we have observed is 4 kilobytes. Thus, table write time by the GammaRay wrapper can vary, but we observe it to be 293ms on average with a standard deviation of 78ms. We measured the time to obtain the DynamoDB handle from AWS via repeated executions in a Lambda function (i.e., as another micro-benchmark). We find that for `Clean`, this operation takes 126ms on average (with a standard deviation of 59ms). Because this startup overhead has a significant impact on short-running applications (as shown previously for the ImgProc application), we are investigating ways of minimizing payload size in particular and optimizing the GammaRay startup process (wrapper), as part of future work.

As shown in columns 2-6 (row 2 of data) in Table 5.5, GammaRay introduces 1-34ms of additional overhead (over X-Ray) on individual SDK operations. Because `G-Ray-H` writes to the DynamoDB table once per operation (immediately prior to the operation), only for events that can potentially trigger other Lambda functions, its overhead is small for DDB Read and S3 Read. Only DDB Write, S3 Write, and SNS include operations that are potentially triggering (i.e., they can be event sources).

The final column of the table shows the average overhead per operation. `X-Ray` introduces 60ms per operation for tracing and `G-Ray-H` introduces an additional 28% (17ms).

The sum of these overheads is what `G-Ray-H` requires to produce causal service graphs, through AWS services, annotated with performance data. This data is shown in the bottom row of both tables. In terms of memory (bottom table), `X-Ray` introduces 16MB of memory, with `G-Ray-H` adding another 31% (5MB), across the micro-benchmarks on average.

5.5 Conclusion

We take an initial step to investigate the limitations of the popular FaaS monitoring service X-Ray. We found that while providing basic performance monitoring and service dependency, X-Ray (and similar systems) does not provide causality tracking, does not work in wide-area settings, uses sampling-based approach and is vendor locked-in. As a result, we design and develop GammaRay to address the above limitations.

GammaRay is the first tool to track dependencies *through* cloud services. GammaRay utilizes DynamoDB for synchronization of event records. By injecting the causal information into every cloud SDK call and persist them as event records, GammaRay is able to infer the causal dependencies between Lambda functions and other cloud services. We provide a toolchain for deploying Lambda applications, allowing developers to set up the GammaRay monitoring and tracing service without modifying any of the source codes.

We investigate three different ways of engineering GammaRay and evaluate the overhead of each using serverless micro-benchmarks and applications. We implement GammaRay for AWS Lambda Python applications and show that it is possible to leverage existing cloud services for much of its implementation. For the applications, GammaRay introduces 12-43% execution overhead and 1-7% memory overhead on average (once free tier use is exhausted). This translates to $0.00000004 per Lambda and $0.03315 per

hour for maintaining the DynamoDB stream, which is less than 4 cents per hour for repeated execution of the Lambda applications that we consider. GammaRay also incurs a $0.2/hour EC2 charge to perform the data analysis concurrently in the background. As part of future work, we are converting the event processing engine that implements this analysis to a Lambda application to significantly lower this cost.

Chapter 6

Lowgo: Distributed Logging for Causal Ordering

In the previous chapter, we design and develop GammaRay to capture the causal dependency within AWS Lambda applications. While GammaRay achieves our goals and is able to perform more detailed monitoring than X-Ray in a broader range of settings, it relies on synchronous database writes to maintain the causal order. The synchronous writes to database introduce a significant overhead (12-43%) on Lambda function performance. In addition, because GammaRay is tightly tied into AWS ecosystems, its use outside AWS services is limited.

To further reduce the overhead and make our tracing framework more versatile and work in multi-tier IoT environments, we develop a cloud-agnostic causal order tracing system called Lowgo (an acronym for **Lo**gging for the **W**ide-area in **Go**) based on the experiences we gathered during developing in GammaRay. Like GammaRay and other previous distributed event logging systems, Lowgo produces a distributed, eventually consistent, partially ordered log of events. However, it also tracks and captures event dependencies, both explicitly between functions and through cloud service "triggers" that

invoke functions as a result of a native cloud service request.

In order to track causal dependencies efficiently across multiple clouds, a LOWGO instance operates a multi-stage pipeline co-located in each cloud with FaaS functions. Multiple LOWGO instances interoperate transparently across clouds to share a uniform view of the causal dependencies within an application. Functions report events (e.g., read, append, and invocation) to their local instance, which propagates event records and causal relationships to instances in other clouds. LOWGO reorders event records based on causal dependencies specified by developers as part of application configuration and deployment using this per-instance pipeline to maintain a geo-replicated, causally consistent log across clouds.

We empirically evaluate LOWGO using FaaS microbenchmarks and multi-function, multi-cloud FaaS applications. In our tests, LOWGO is able to capture causal dependencies with minimal overhead. The overhead that LOWGO introduces ranges from 2-12%, which is less than half that of GAMMARAY. We find that the overhead is proportional to the number of events; short-running applications tend to have a higher overhead, while computation-heavy applications typically have a lower overhead. The throughput of LOWGO ranges from 109K to 30K records per second, depending on dependency depth.

In the sections that follow, We first discuss the related work in Section 6.1. Then, we present the design and implementation of Lowgo in Section 6.2. In Section 6.3, we show the evaluation of LOWGO using microbenchmarks and FaaS applications. Finally, we give our conclusion in Section 6.4.

6.1 Related Work

While the study of causal ordering and logging is relatively new in the field of FaaS, it is a popular subject of research in non-FaaS settings. Chariots is a distributed logging

service [131] designed for multi-datacenter settings. It supports multiple datacenters by providing a global shared log that consists of all records generated at all datacenters. The log records are maintained by a group of *log maintainers*, which span multiple datacenters and collectively persist to a single shared log. Clients send records to a local Chariots instance, which records and replicates them to other datacenters. In the process, geo-replication preserves the causal order of records. When persisted in a log maintainer, a record is assigned a total order ID. All replicas of the record share the same total order ID. Chariots uses this ID to order records from the same datacenter.

Different from other geo-replicated shared logs such as Google Megastore [179], Chariots does not require all clients to write to the head of the log, thus eliminating the single point of contention. In addition, Chariots' pipeline design allows each stage to be scaled seamlessly and independently. This design enables the system to adapt to performance bottlenecks automatically.

Similar logging systems include X-trace[180], Kronos[181], LogBase[182], and Corfu[132]. X-trace reconstructs Internet services dependency trees. It appends metadata to network operations and propagates them across layers and applications. X-trace requires a specialized TCP/IP stack and application instrumentation for use. Kronos tracks dependencies and provides time-ordering for distributed applications. Kronos builds and maintains a service dependency graph internally. It relies on a centralized implementation and is not designed for cross-cloud use. LogBase is a multi-version log-structured database. It adopts an append-only approach to eliminate write bottlenecks. It reduces the write overhead by appending all write operations to the head of the log. Corfu is a similar append-only shared log that is built on distributed flash devices. Corfu maintains a static mapping between log position and flash page. Clients ask a sequencer to determine the next available position of the log.

6.2 Lowgo Design

LOWGO is a cloud-agnostic *casual event tracing* system for multi-function, multi-cloud FaaS applications. LOWGO captures causal dependencies across functions and through cloud services. It automatically infers dependencies within applications by LOWGO capturing calls that functions make via cloud software development kits (SDKs), which access cloud services that are potential event sources or to invoke other functions. Alternatively, developers can specify a subset of such dependencies to trace as part of application deployment using LOWGO tools.

Key to the LOWGO design is that it avoids synchronized writes, does not block application progress for record sequence assignment, and minimizes cross-cloud communication. To enable this, LOWGO comprises multiple instances co-located with FaaS functions in different clouds. Functions in one cloud only communicate with their co-located LOWGO instance. Records generated in different clouds are replicated across instances by LOWGO to compose a consistent, distributed shared log.

LOWGO avoids application delays by determining event position *after* events are reported to LOWGO. Functions generate event records that contain event details and dependency information and send them via the LOWGO interface. Upon receipt, LOWGO reorders records using the dependency information prior to persisting them to the log so that log order reflects causal order.

To enable both causal event tracing and causal log order, LOWGO implements a three-stage pipeline similar to that proposed by Chariots. However, LOWGO is significantly simpler than Chariots, since it needs only capture a partial order of events (those with detected or with specified dependencies). LOWGO also uses a simplified record structure that replaces Chariot's total order ID (which does not capture event dependency) with a record ID and parent ID pair representing the event dependency. We detail both the

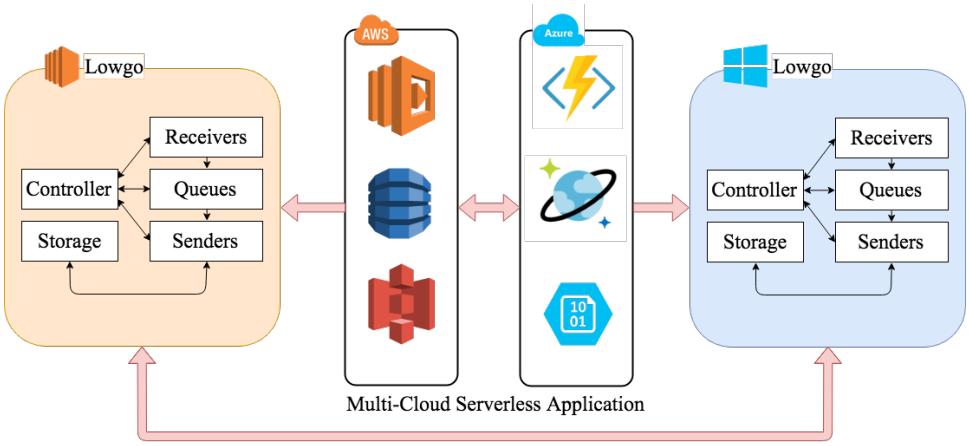


Figure 6.1: LOWGO overview. The figure shows a distributed application spanning two clouds. In each cloud, there is a LOWGO instance deployed. When an event is generated by cloud service, a corresponding record with causality information is sent to local LOWGO instance. Records are replicated across remote LOWGO instances with their causal dependencies preserved maintain a consistent record history.

pipeline and record structure in the next section. Events pass through the pipeline and are stored in a local data storage in causal order as depicted in Figure 6.1.

The figure also depicts an example FaaS application that executes using both AWS and Azure. Cloud functions access cloud services via cloud SDKs. A subset of service operations also serves as event sources that trigger functions (defined by each FaaS platform). Upon invocation of this application, a function in Microsoft Azure writes to an AWS S3 bucket via the AWS SDK. This write triggers a function invocation in AWS Lambda.

In order to trace events through cloud services and across clouds, LOWGO records function entry and exit and each SDK call that a function makes. These records contain information about the event source (for functions), the SDK operation, and the SDK arguments. LOWGO records SDK calls (only those that are potential event sources) via minor modifications to each cloud SDK. Function entry/exit instrumentation (function wrapping) occurs during FaaS application deployment to each cloud platform via a LOWGO tool.

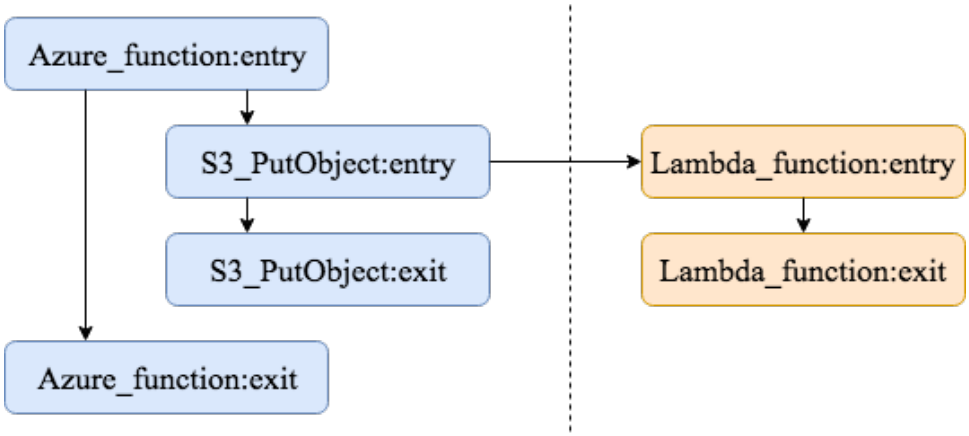


Figure 6.2: Multi-cloud FaaS application example using Microsoft Azure Functions and AWS Lambda.

Figure 6.2 shows the service graph that LOWGO records for this application. The arrows indicate causal relationships, i.e., A→B means event A causes event B. The records on the left of the graph are generated in Azure, while the records on the right are generated in AWS. Records generated locally are propagated to other LOWGO instance. All instances observe the same causal relationship graph.

6.2.1 Implementation

Events generated by the application are represented as records in LOWGO. Records contain application context and causal information. Context is used for debugging and analysis; causal information identifies details about the event that caused the invocation of the function. Each LOWGO record has the following fields:

- **Record ID**: A 128-bit statistically unique ID (UUID v4) that serves as an identifier of the record.
- **Host**: An integer that identifies the LOWGO instance for which this record is generated. For example, in Figure 6.2, records generated in Azure have the same host, while records generated in AWS have a different host.

- **Trace ID**: A 128-bit ID shared by a series of records that are part of the same causal event chain. A Trace ID is generated when a function without a parent (i.e., not triggered directly or indirectly by another function) is invoked. A Trace ID is transmitted downstream to all records generated by this *root* function and downstream functions via LOWGO SDK use.

- **Parent ID**: The record ID of the parent (i.e., the function that directly or indirectly triggered this function). LOWGO ensures that parent records are persisted prior to their child records in the log. This field is empty for a root function.

- **Payload**: Application context information. LOWGO allows developers to customize what information is recorded.

- **Log ID**: A unique and monotonically increasing number that represents the record's position in the log. Log ID provides an alternative way to infer causal dependency other than checking each record's parent ID. The order of multiple records which have the same trace ID can be obtained by comparing their log IDs.

Each of these fields, except for Log ID, is filled in as part of LOWGO SDK use. Since Log ID assignment (i.e., determining record position in the log) is a complex task, LOWGO separates assignment from SDK to keep overhead low. The overhead that LOWGO imposes on FaaS applications thus includes record construction and round trip communication of records and acknowledgments between functions and the co-located LOWGO instance.

Figure 6.3 illustrates the architecture of a LOWGO instance. The solid arrows represent the flow of records and the dotted arrows represent information exchange. A LOWGO pipeline isolates intra-datacenter communication, record ordering, and geo-replication using three stages: **receiver**, **queue**, and **sender**. Each stage can be scaled independently based on network congestion and workload.

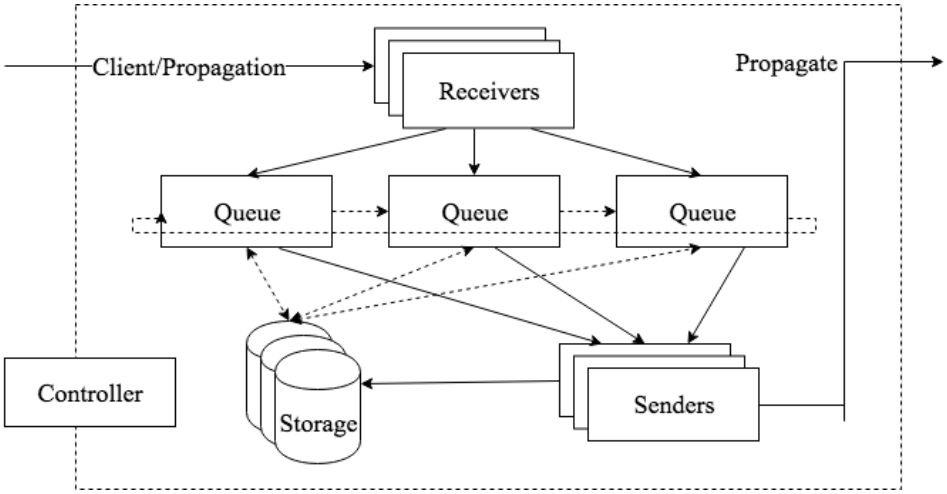


Figure 6.3: The architecture of a LOWGO instance. The solid arrows show the forwarding path of records. Queues form a ring. A token is passed around the queue ring to update the latest Log ID. Queues also send queries to the storage to see if a buffered records' parent is persistent in storage.

A **receiver** receives records sent by a local SDK or propagated by other LOWGO instances. Application clients can send records to any receiver using either gRPC[183] or HTTP POST. Co-located receiver instances are load-balanced via HAProxy. Also, since records are sent individually by each SDK, receivers buffer incoming records to reduce communication overhead. Receivers send batched records to a queue when the buffer fills or is flushed.

The **queue** orders records and assigns the record position (log ID) to each record. Queues buffer records until they can be sent (in batches) to the next stage. A record is batched when either (i) the record does not have a parent ID or (ii) the record's parent ID record is found in storage. The queue periodically queries LOWGO storage for parent IDs of buffered records. If there is a match, the queue assigns a log ID to the buffered record and forwards it to a sender. By doing so, queues ensure that log IDs of records coming from the same root function (trace ID) are in order. Thus, the log ID of a dependent record is larger than that of its parent.

For concurrency, multiple queue instances process records simultaneously, and re-

ceivers can send records to any queue instance. Queues avoid duplicating log IDs using token passing (a database can be used as a sequencer but doing so introduces a significant bottleneck and single point of failure). The token carries the current maximum log ID for the LOWGO instance and passes between queue instances in a ring. Only the queue holding the token assigns and increments the log ID. It assigns log IDs for all of its batched records. The queue then passes the token to its neighbor and communicates the batch to a sender.

The **sender** persists and geo-replicates records. Instances write records to the log in any order (avoiding synchronization) and can employ log ID sharding for additional scale and performance. Senders send batched copies of records (replicas) to other LOWGO instances after removing their log ID. When a record is received by a LOWGO instance (receiver stage), the receiver determines whether it is a replica by checking its host ID. Replicated records proceed through the instance's pipeline and receive a new log ID.

Log IDs of the same record across clouds can differ, and records from different event chains will be interleaved in the log of each LOWGO instance. This approach enables LOWGO to maintain causal order via parent IDs across instances (clouds) with very low overhead by enabling LOWGO instances to operate independently and concurrently at scale.

The final two components of each LOWGO instance are **storage** and the **controller**. Storage is where each LOWGO physically stores its log records; storage in different LOWGO instances are independent and do not interoperate. LOWGO storage responds to queries from queues and persists records sent from senders and so can be implemented using any database. Replication is not necessary since it is inherently supported across clouds by LOWGO. As mentioned previously, sharding and multi-index support can reduce the overhead of database use if available. The **controller** maintains LOWGO's system-wide configuration. Communication between cross-cloud controllers occurs only at LOWGO

startup or when resources/instances are added or removed.

6.2.2 Lowgo SDK

We provide a LOWGO SDK for developers to deploy FaaS function with LOWGO causal tracing enabled. The LOWGO SDK consists of three components: a LOWGO instance deployment tool, LOWGO client API, and FaaS function deployment tool for each cloud. Using these tools in combination enables developers to use LOWGO without modifying their FaaS applications.

Developers use LOWGO instance deployment tool to generate configuration files used to deploy LOWGO instance with Docker Swarm. The configuration consists of the number of instances per pipeline stage, debugging flags, and port numbers to use. The tool generates Docker Swarm stack files and configuration files for instances in each cloud. The deployed LOWGO instance uses the configuration file to set up the pipeline and load balancing.

The LOWGO client API is a simple library that sends records to a LOWGO instance. It uses gRPC framework and HTTP POST to send records to LOWGO receivers. We choose gRPC for its performance and portability. If a programming language used to implement a FaaS function does not have gRPC support, LOWGO uses HTTP POSTs. Currently, the LOWGO client API is available for Python, Node.js, and Java. Listing 6.1 shows an example using the LOWGO Python SDK. Although application modification is not required to use LOWGO, developers can add custom tracing via the LOWGO client API. The LOWGO client module exposes two variables: *traceid* and *parentid*. The FaaS application deployed using the LOWGO function deployment tool automatically updates these variables so developers can use them to tailor record information.

The LOWGO function deployment tool uploads functions to FaaS cloud platforms.

Listing 6.1: Example of use Python SDK

```
import lowgo_sdk

# setup Lowgo host
cli = lowgo_sdk.RPCClient('10.0.0.1')

# build record
record = lowgo_sdk.Record(traceid)

# add parent ID and extra information
record.setParent(parentid)
record.addTag('key', 'value')

# send record to Lowgo
result = cli.postRecord(record)
```

The tool takes the function source code and libraries, intercepts the function entry point with a LOWGO wrapper (to record entry/exit of functions and trace IDs), and constructs a deployment package. LOWGO currently supports AWS Lambda and Azure. A LOWGO SDK is also integrated within the package and loaded at runtime by the wrapper, for each cloud SDK that the application uses. A LOWGO SDK for a particular cloud is the cloud SDK modified to insert dependency information, to build name records, and to send records to a local LOWGO instance. The records sent by the LOWGO wrapper and SDKs provide the minimal amount of information necessary to enable LOWGO to track causal order between functions, through cloud services, and across clouds.

6.3 Evaluation

We evaluate LOWGO performance using FaaS microbenchmarks and applications (multi-function and multi-cloud). We first overview these applications and our experimental methodology and then present our empirical results.

6.3.1 Experimental Methodology

In our evaluation, we co-locate a LOWGO instance in each cloud and region in which the FaaS functions in an application are deployed. A LOWGO instance consists of a Docker Swarm (v17.12.0-ce) with five nodes. Nodes include a LOWGO receiver, controller, sender, and two queues. Each LOWGO instance also integrates a MongoDB (v2.6.10) database. In our experiments, we deploy Docker Swarm and MongoDB using instance type c4.large in AWS EC2 and our Eucalyptus private cloud, and instance type Standard.D2_v2 in Azure.

We evaluate the overhead of LOWGO using three microbenchmarks and three multi-function FaaS applications. Functions that run in AWS are written in Python 2.7 and use the boto3 AWS SDK to access AWS services. Azure functions are written in node.js. All other LOWGO tools are written in Python 2.7. We execute each experiment 100 times with and without LOWGO. We attempt to isolate cold start overhead by running the applications 10 times prior to performing our measurements. FaaS cold starts occur when the system must load both the function and container in which it executes prior to function invocations. Cloud providers maintain the container for a short (unspecified) duration following function invocation (optimizing for temporal locality), to avoid this overhead for potential repeat invocations.

The microbenchmarks are single-function applications and include a function that simply returns (EMPTY), a function that performs a gRPC request with a payload of 1125 bytes (gRPC_SDK), and a function that performs an HTTP POST with a payload of 1125 bytes (REST_SDK). We use these applications to isolate the overhead of LOWGO SDK use.

The three multi-function applications are called `Map-Reduce`, `Rekognition`, and `Thumbnail`. Map-Reduce is the same benchmark application we use to evaluate GAMMARAY.

The detail of Map-Reduce is described in Section 5.4.

Rekognition uses the AWS image processing service called Rekognition, to label images. A function is triggered when a file with jpg suffix is uploaded to S3 bucket in its region. The function calls AWS Rekognition API, which returns labels and probabilities that describe the image. The function uploads the result to an AWS DynamoDB table. We use a 152KB 640*427 jpeg file to trigger the application. We evaluate this application with and without LOWGO support using a single cloud (AWS).

Thumbnail is a multi-cloud application. We deploy a Python script in a private cloud instance that uploads images to AWS S3. The S3 upload event triggers a function in AWS, which creates a thumbnail of the uploaded image. The function uploads the thumbnail to Azure Blob Storage. This Blob update triggers an Azure function, which loads the image from Blob Storage module and measures the time for doing so. We use a 2.6MB (6000x4000) jpeg file as input. The resulting jpeg thumbnail size is 12.5KB (300x200).

6.3.2 Performance With and Without Lowgo

We first evaluate the overhead of using the LOWGO SDK by measuring the time to send records to LOWGO. For this test, we employ the FaaS microbenchmarks and AWS. We present the average execution times in milliseconds (ms) and memory used in megabytes (MB) in Table 6.1 for each microbenchmark. To measure time, we insert timers at function entry and exit; the function logs the duration prior to exiting. The LOWGO tools collect, aggregate, and summary these log entries, as well as the timings and memory usage recorded by the service (which is CloudWatch in AWS) and used for billing.

Since EMPTY simply returns upon being invoked, its duration is 0ms, the durations

	AWS Time	AWS Memory
EMPTY	0ms	19MB
gRPC_SDK	2085ms	36MB
REST_SDK	2391ms	27MB

Table 6.1: LOWGO microbenchmark average duration and memory use in AWS.

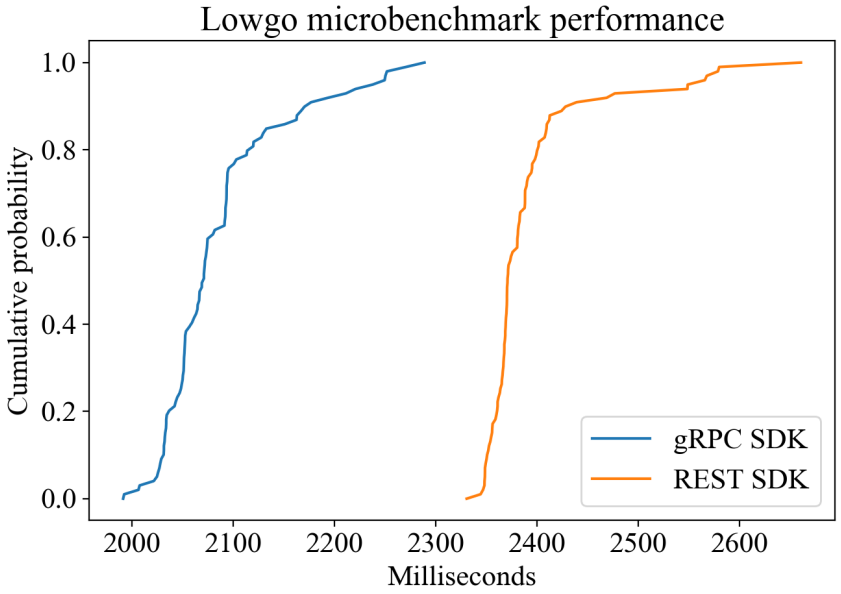


Figure 6.4: LOWGO microbenchmark performance CDFs (gRPC_SDK and REST_SDK). The means are 2085ms and 2391ms, respectively.

of the other two applications translate to the cost of sending 100 records using different protocols. It takes an average of 2085ms to send 100 records using gRPC protocol, and 2391ms to send 100 records using HTTP POST, which translates to approximately 21ms and 24ms to send a record to LOWGO using gRPC and HTTP POST, respectively. Figure 6.4 shows the full distribution of times as empirical cumulative distribution functions (CDFs) for both tests. The gRPC SDK test has a standard deviation of 63ms with 1991ms minimum and 2289ms maximum. The REST SDK test has a standard deviation of 58ms with 2331ms minimum and 2661ms maximum. In terms of memory use, gRPC uses 17MB, and HTTP uses 8MB, respectively, over EMPTY. The difference

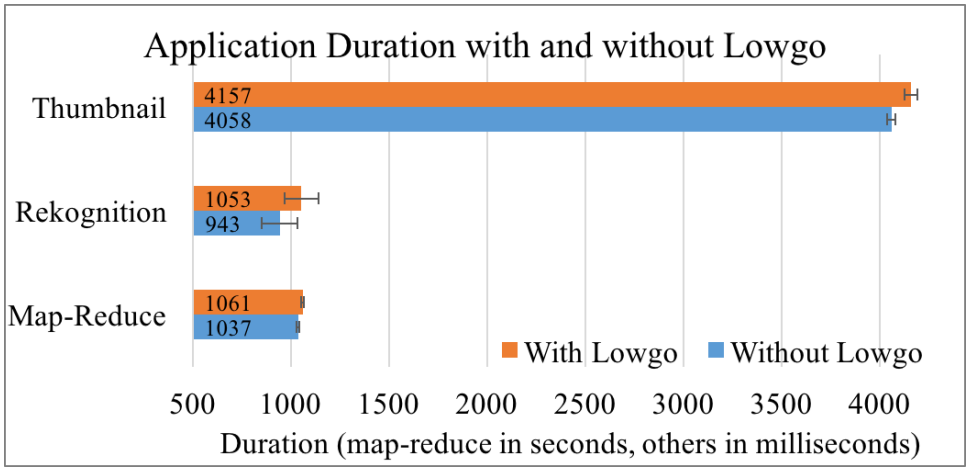


Figure 6.5: Average application performance with and without LOWGO. Error bars show the 95% confidence intervals.

between LOWGO and CloudWatch log measurements represents the function setup time. This difference is 6.3ms (stdev=8.7ms) for gRPC_SDK and 0.48ms (stdev=1.70ms) for REST_SDK.

We next investigate the dollar cost of importing the LOWGO SDK. We find that AWS Lambda only imports modules during cold starts (when a container is not reused). In such cases, we also find that Amazon does not charge for codes executed outside Lambda handler when under 10 seconds. Since the time Lambda takes to import LOWGO SDK is well under 10 seconds threshold, importing the LOWGO SDK does not introduce extra cost. The average time to import LOWGO SDK is 128ms for gRPC and 61ms for REST in AWS Python 2.7, and 13ms for REST in Node.JS 6.10.

6.3.3 Application Performance

We next evaluate the overhead imposed by LOWGO for the multi-function applications. Figure 6.5 shows the average execution time for each with and without LOWGO. For Map-Reduce, LOWGO introduces an average overhead of 24 seconds which is 2.3%. This difference is small but statistically significant according to a t-test with $\alpha = 0.05$.

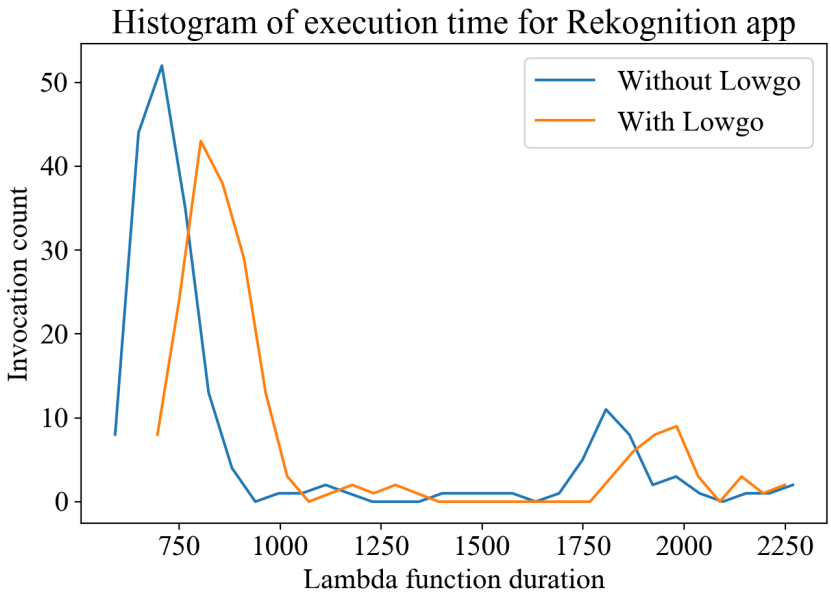


Figure 6.6: A histogram of execution time for Rekognition with and without LOWGO.

For Rekognition, LOWGO adds an average of 110ms (11.7%). Because this application is very short running and invokes multiple services that are potential event triggers instead of computing, LOWGO SDK overhead plays a larger role in overall execution time. There are 6 records sent to LOWGO by each application instance. Using our microbenchmark results, we expect an average overhead of 126ms, which is in line with what we observe for this application. Specifically, Figure 6.6 shows the histogram of execution times with and without LOWGO for Rekognition. Note the shift of approximately 100ms with and without LOWGO.

Next, we evaluate how LOWGO performs in a multi-cloud setting using the Thumbnail application. Figure 6.7 shows the results. The average total overhead introduced by LOWGO is 99ms or 2.4% over the uninstrumented version. According to a t-test with $\alpha = 0.05$, the difference is statistically significant.

Summarizing, LOWGO introduces an average of between 2 and 12% overhead for the applications we study. This is significantly lower than GAMMARAY which introduces

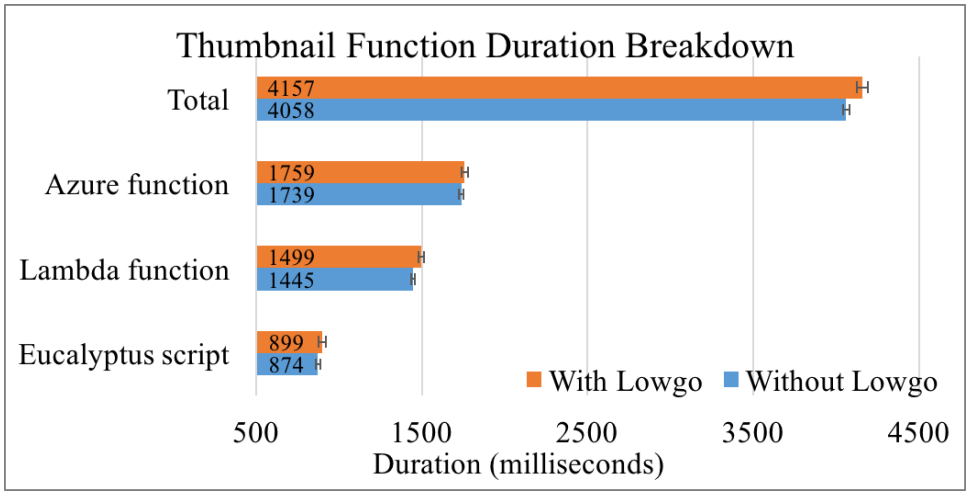


Figure 6.7: Time spent in each component on average for the Thumbnail application with and without LOWGO. Error bars show the 95% confidence intervals.

12 to 43% overhead for similar applications. The benefits come from LOWGO's multi-stage pipeline for log ID assignment and record persistence instead of using GammaRay's synchronized, inlined call to DynamoDB with DynamoDB Streams support.

6.3.4 Lowgo Throughput

To understand the impact of causal dependencies between functions on LOWGO performance, we next evaluate system throughput. We perform this test using AWS EC2. We launch 6 c4.large EC2 instances in US West to form a Docker swarm, which consists of 1 manager node and 5 worker nodes. Each node is responsible for hosting a LOWGO stage: controller, receiver, sender, while there are 2 queues hosted on 2 nodes. The remaining node is responsible for hosting MongoDB. In addition to the Docker swarm nodes, we launch a c4.large instance in the same region to drive the throughput benchmark.

Figure 6.8 shows LOWGO throughput for different dependency depths. We define dependency depth as the number of event records with the same root cause. An event chain with dependency depth 1 means that an event is independent. If one event causes another event, the event chain has dependency depth 2, and so on. When there is no

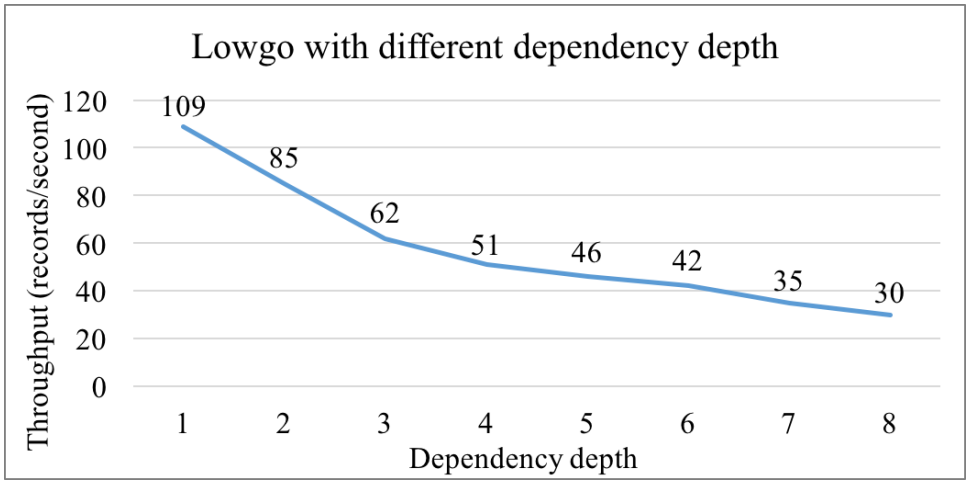


Figure 6.8: LOWGO throughput. The X-axis is dependency depth. The Y-axis is records per second.

dependency among records, LOWGO throughput is 106 kilo-records per second (Krps) compared to 142 Krps for a version of Chariots we implemented. Recall that Chariots does not capture dependencies, so this difference can be considered the "base" overhead cost of dependency recognition in LOWGO versus a top-performing logging system without dependency tracking.

As dependency depth grows, LOWGO throughput decreases slowly. This is caused by the dependency resolving mechanism in the queue stage. In this stage in Chariots, the system checks whether the maximum total order ID is greater than the buffered parent total order ID to decide if a buffered record can be appended to storage. In this stage in LOWGO, the system checks whether buffered records' parent record is in storage. The MongoDB query operation to perform this check results in the overhead introduced by LOWGO for dependency resolution.

6.4 Conclusion

In this chapter, we present LOWGO, an event tracking system for multi-cloud FaaS applications. LOWGO is able to trace service dependencies and causal ordering of FaaS applications that span a wide geographical area and multiple clouds. LOWGO utilizes a geo-distributed design to maintain a distributed shared log. A LOWGO instance that consists of a record processing pipeline is co-located within each site. When the application generates an event, it reports the event to the co-located LOWGO instance as a record. LOWGO instances communicate with each other in the background to reorder the records based on the causality information they carry, hence reduce the event record ordering overhead caused by the coordination between sites.

We implement LOWGO SDKs for developers to deploy their existing FaaS applications. Using the LOWGO SDK, the application dependencies through services and across clouds will be tracked automatically without modification. We also provide a simple way for developers to manually instrument applications to capture extra dependencies.

We evaluate the performance of using LOWGO to capture events using microbenchmarks. The results of microbenchmarks show that on average LOWGO takes 21 ms to capture and report an event in our setting. Then, we implement three multi-cloud applications to evaluate the end-to-end performance. We find that the overhead is proportional to the number of events; LOWGO introduces 11.7% runtime overhead to our short-running application, while to the long-running application, it only introduces 2.3% of overhead. This shows that in similar situations, LOWGO is able to outperform our previous work GAMMARAY.

With LOWGO, we are able to debug, analyze, monitor, and reason about IoT applications that typically span multiple scales and clouds. Furthermore, the lessons we learned from implementing dependencies tracking led to our work on distributed data repair and

replay. In the next chapter, we will show how to embed the causal information into all events in CSPOT and enable data repair and replay for multi-tier IoT applications.

Chapter 7

Sans-Souci: Data Repair and Replay

Because IoT devices are vastly heterogeneous and execute in a wide range of remote locations and operating conditions, they are subject to frequent hardware and software errors and failures, performance degradations, network partitions, etc., which in many cases cannot be easily or immediately remediated. Furthermore, many of these failure modes corrupt the data that these devices produce, as well as the "downstream" computations that depend on this data. Data corruption can mislead human and automated decision making, result in inaccurate predictions and inferences, and compromise the security, efficiency, and quality of service of data-driven IoT applications.

In this chapter, we explore a new approach for repairing corrupted data in distributed IoT settings. Replay is a technique used in single-host runtimes and distributed systems to fix errors in software and data structures, for trace-based simulation and prediction, to explore alternative application execution paths, and to perform post-mortem program analysis. We investigate a new approach to repair and replay, called Sans-Souci, which automatically tracks causal data dependencies and replays dependent computations across multi-tiered (device, edge, and cloud) IoT deployments. To enable this, we extend CSPOT to make portability and repair possible in a distributed IoT setting and

across heterogeneous devices.

SANS-SOUCI builds upon and extends the append-only and versioned storage abstraction in CSPOT to simplify and facilitate data repair and replay. In particular, SANS-SOUCI couples this persistent storage with distributed event logging to track events, function invocations, and causal dependencies among events in multi-function serverless applications. Using events records logged in the storage, SANS-SOUCI is able to maintain a history of transactions, repair persistent data structures, and re-initiate dependent computations. It uses this history to update corrupted data structures with corrected values at the historical point in time at which corruption occurred, and replays all causally dependent computation from that point forward. For example, if a network partition causes disruption in the production of sensor data, an application might instead forward interpolated, repeated, or error values (e.g., -1) to subscribers to mask the interruption. Subscribers downstream might use the values for analysis (e.g., prediction and classification). SANS-SOUCI can repair the historical data for the sensor when it comes back online and automatically replay any dependent analysis functions.

For robustness in the presence of partial failures, SANS-SOUCI is also unique in that it maintains append-only semantics as it implements repair. That is, it does not "update-in-place" corrupted data, but rather it generates a new set of uncorrupted appends that occur (logically) after the corrupted dependencies. After a repair is complete, however, an application considering the most recent appends to a set of persistent data structures will "see" only the repaired data.

We integrate SANS-SOUCI into CSPOT and evaluate it using multi-tier IoT deployments, applications, and benchmarks. We find that SANS-SOUCI is able to perform repair for corruption produced by either software (function) or sensor data errors with very low overhead.

In the sections that follow, we first present the related work that inspires our work

on causal dependency tracking and record/replay in Section 7.1. We next discuss the SANS-SOUCI design and use cases in Section 7.2.2 and describe its implementation in Section 7.3.4. We present our results in Section 7.4.4 and conclude in Section 7.5.

7.1 Related Work

Our work builds upon and extends a large body of related work on causal dependency tracking and record/replay. Causal dependency tracking is useful for a wide variety of applications, including debugging, provenance tracking, auditing, speculation, and accounting, among others [157, 184, 185]. We have shown how to track causal dependency for FaaS applications with GAMMARAY and LOWGO in the previous chapters. [184] combines causal tracing with dynamic instrumentation for user-guided, low-overhead application monitoring. We combine it with append-only, persistent data structures, and the FaaS programming model, to enable fast data repair and replay in distributed, heterogeneous settings.

Some record/replay systems leverage causal relationships to facilitate distributed debugging and exploration [186, 187, 188, 189], and deterministic replay and simulation [190, 191, 192]. The authors in [190] checkpoint applications and intercept system/API calls to facilitate simulated and deterministically reproduced runs of a program. Determinism is captured using a logical clock inserted into messages. The authors of [191] investigate retroactive programming – support for reprogramming application histories. They combine the use of FaaS and causal event capture but change the FaaS programming model by integrating Command Query Responsibility Segregation at the function level. Function types are partitioned into those that update state, view state, perform retrospection and retroaction. SANS-SOUCI in contrast, focuses only on distributed data structure repair (and dependency replay) and so is significantly simpler, does not change

the FaaS programming model, is fully distributed, and can be overlayed on any serverless system that supports causal event logging and state updates via versioned data service APIs.

7.2 Design

SANS-SOUCI is a new approach and system for repairing corrupted data in distributed IoT applications. It automatically tracks causal data dependencies and replays dependent computations across multi-tiered IoT deployments. We build develop SANS-SOUCI on top of our FaaS platform CSPOT for its event-driven execution model, durable append-only data structures, and capability to track distributed causal dependency. Although each of these features is well understood distributed systems concept, their combination reveals a rich set of design trade-offs that motivate this exploration.

SANS-SOUCI uses the histories of persistent storage updates and their causal relationships to update corrupted or approximated data structures with corrected values at the historical point in an applications state update sequence at which they occurred. It then replays all dependent computation from that point forward. To ensure robustness to partial failures, SANS-SOUCI performs this update and replay (of causally dependent functions) using append-only semantics (versus update-in-place).

7.2.1 Use Cases

We envision three primary use cases for such data repair capabilities in IoT deployments. The first is to correct downstream historical results when a faulty sensor or data source (that has been issuing "bad" data) is repaired, and some data correction for the data that has been produced is available. One such real-world example is a misconfigured microclimate monitoring system, in which a subset of temperature sensors are configured

to report Celsius rather than Fahrenheit as specified and required by the deployment. Such errors are commonly detected post-deployment after the data has been consumed by downstream analytics applications. SANS-SOUCI is able to correct the misconfigured readings and the downstream computations *in situ* – without stopping the deployment, gathering the data to a centralized location, cleaning it, and then redeploying it and the applications that use it.

The second use case is a debugging, exploration, and experimentation utility for deployed IoT applications. In a tiered IoT setting in which small sensor/actuator devices communicate with more powerful edge computing and storage devices, private clouds or public clouds, computation and data are often distributed throughout the deployment. To experiment with or debug new computational methods (e.g., improved analytics) it is often inconvenient (or impractical) to create a parallel deployment. SANS-SOUCI uses replay to propagate data repair throughout the deployment, making it possible to change specific computational components and then to observe the results and also to roll back such changes.

The third use case is to manage the arrival of late but correct data. As another real-world example, an IoT deployment might incorporate meteorological data from the CIMIS[193] network of weather stations. CIMIS publishes data on 5-minute intervals, but it does so retroactively, once every hour. The deployment itself expects data every 5 minutes. Existing applications generate an interpolation of the previous hour's CIMIS data every 5 minutes for the downstream components of the application to use immediately. SANS-SOUCI can "repair" the interpolations once the CIMIS data arrives at the top of the next hour.

Thus, the "corruption" that SANS-SOUCI is designed to repair covers several IoT use cases in which data gathered in the past can be replaced with better or more useful data in the future. Moreover, SANS-SOUCI can propagate the effects of those replacements

throughout a distributed deployment.

7.2.2 Sans-Souci Data Repair

To effect a repair, SANS-SOUCI depends on the following properties.

- It must have access to all the program state that is used as input by any function that is casually dependent on the target of the repair at the time the target was initially produced.
- It must be able to reproduce the order of execution (i.e., the causal execution order) of the functions that take this state as input.
- The functions must be side effect free so that their replay depends only on the program state visible to SANS-SOUCI.
- The repair itself cannot overwrite any of the previous program state that will be used as function input during replay before it is used.

An Illustrative Example

We overview the repair process via an example of the first use case above using a common prediction (or classification) streaming workflow. The example applies a trained model to each new datum that arrives from a sensor and produces a prediction. In the example, the programmer (or automatic service on her behalf) identifies an error in the model and produces a new version of it with the error corrected. They then initiate a repair to update the previous version and replay all previous predictions that depended upon the original "bad" value.

The example application has three data structures called SENS, MODEL, and PRED as depicted in Figure 7.1 (a). All data structure is persistent and append-only (i.e., each

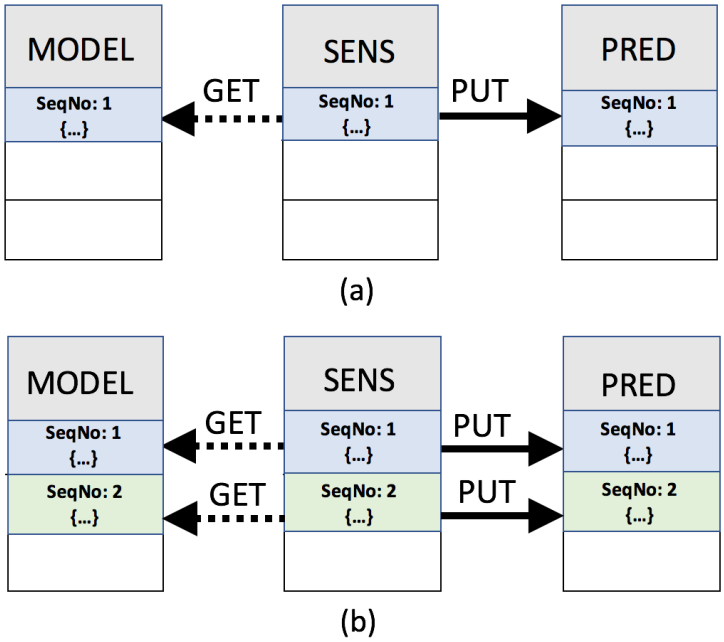


Figure 7.1: Data repair example. Application state is stored in persistent, append-only data structures (SENS, MODEL, and PRED) on 1+ hosts; each version has a sequence number (SeqNo). (a) shows the state after the first sensor element arrives; (b) shows the state after a second sensor element.

has multiple versions); the tail of each (i.e., the most recent version) holds the current state of the structure. Each version is identified via a sequence number (SeqNo). These data structures are persisted to disk and thus reside on a particular host. Data structures that make up an application can be on the same or different hosts. Functions access local data structures directly and remote data structures via messaging.

Periodically, the application receives sensor data, which it appends to SENS. The append triggers a function, fSENS, which reads from the tail of MODEL to retrieve the most recent prediction model. The function applies the model to the newly arrived data and produces a result (a prediction), which it writes to PRED. We refer to data structure reads as Gets and writes as Puts.

Dependent events are written to a local log as part of execution of the application. SANS-SOUCI records when data is appended, when the tail of the data structure is ac-

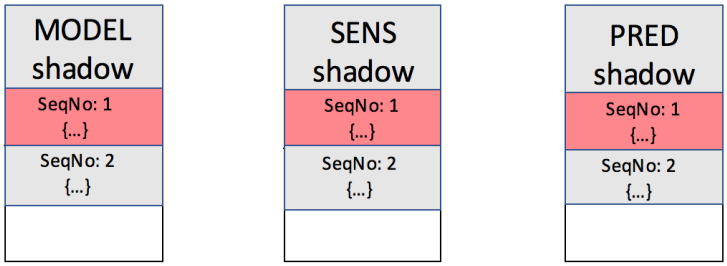


Figure 7.2: During repair SANS-SOUCI constructs shadows for dependent data structures to perform the repair. MODEL SeqNo 1 is repaired directly; its dependencies (marked in red) are repaired via replay.

cessed, and when functions are executed (fired). It also records the causal dependencies with sequence numbers. In this case, there are two such dependencies (sequence numbers are specified in parentheses): (i) *SENS(1)* **Get** *MODEL(1)*, and (ii) *SENS(1)* **Put** *PRED(1)*.

When the programmer realizes that the MODEL has a bug, she appends the new model to the MODEL data structure. The next time a SENS value arrives, it will use the new MODEL as shown in (b) in the figure. The updates in the system that result from this event are shown in green. The new dependencies appended to the log are (i) *SENS(2)* **Get** *MODEL(2)*, and (ii) *SENS(2)* **Put** *PRED(2)*.

The programmer then also initiates a repair to fix PRED(1) via the SANS-SOUCI API, passing in the new value and sequence number of the version in need of repair (SeqNo 1 in this case). To effect the repair, SANS-SOUCI first requests the logs from all hosts and merges them into a total order. SANS-SOUCI uses this merged log to identify

- the chains of data dependencies (i.e., Puts and Gets) rooted at the target that must be updated, and
- a correct execution order of functions that will be "replayed" to generate these updates.

We refer to the dependency tree rooted at the repair as the "repair graph." SANS-

SOUCI generates the repair graph via a scan of the merged log. SANS-SOUCI creates a shadow data structure for each data structure impacted by the repair as depicted in Figure 7.2. Versions in red are those marked as dependent in the repair graph. It then copies elements from the original structure to the shadow from the first element up through the element prior to the start of the repair. It then appends the repaired values that the user has passed in. As part of this append, SANS-SOUCI retriggers any functions (fSENS in this case) that implement dependent **Get**s. Note that SANS-SOUCI only re-fires functions that *also* perform writes (i.e., Puts) on persistent data structures (i.e., perform state updates) because those without writes have no impact on global, shared states. This process continues as SANS-SOUCI traverses the repair graph.

The Puts and Gets in replayed functions use the shadow versions of the data structures and the sequence numbers passed in by SANS-SOUCI. To enable this, SANS-SOUCI replaces API calls that read data structures with those that read specific sequence numbers, and those that read and write data structures with those that that target a shadow data structure during replay. The application functions execute concurrently with the repair without interruption using the original data structures.

SANS-SOUCI copies any remaining values (those independent of the repair), after (or interleaved with) the repair, from the original to the shadow. It then synchronizes the shadow and original (pausing the application briefly) and performs a rename so that the application uses the shadow (i.e., the shadow becomes the original, for use by applications and the next repair, if any) and the original is garbage collected. We next describe the details of this process and provide the intuition behind our design decisions.

7.3 Implementation

From a data dependency perspective, the requirements to implement SANS-SOUCI can be met by a number of different programming models (e.g. functional programming [194, 195, 196], single assignment languages [197, 198, 199], etc.) In this work, we choose CSPOT as the runtime platform for SANS-SOUCI to leverage its append-only storage abstractions to hold all program state that persists beyond function boundaries. That is, in a conformant CSPOT program, all functions are stateless and all program state occupies some position in a logically infinite log of state updates. SANS-SOUCI uses this feature to be able to access program state that is required by functions "downstream" from a repair. That is, CSPOT (up to the pre-configured history size in each storage abstraction) preserves all historical program state (individual state updates are versioned).

In this section, we first describe the modifications we make to extend CSPOT and then present an algorithm to merge distributed logs, identify causally related events, and repair (and replay) the events.

7.3.1 CSPOT Extension

The CSPOT runtime system maintains an internal append-only event log in each namespace. [1] The namespace log is used directly to record state updates (WooF appends) and to trigger handlers. When a call to WooFPut() creates a state update that specifies a handler to trigger, the caller appends the event to the end of the namespace log. Threads running within CSPOT containers associated with the namespace synchronize on the tail of the log and race to "claim" and then execute a newly added handler.

[1] Note that the namespace log is logically a WooF with elements that describe events, but because CSPOT uses the namespace log to implement handlers trigger for WooFs, the namespace log is implemented separately to avoid a circular dependence.

In the original design of CSPOT, only events describing handler triggers, and their eventual claims by container threads are logged. SANS-SOUCI modifies this CSPOT implementation to include additional log event types for its dependencies (e.g. WooFPut() and WooFGet()). We detailed the CSPOT API in Section 3.2.

All CSPOT namespace log events carry identifiers for the namespace, the object within the namespace, the "cause" namespace that originates the event, and the object within the cause namespace (implemented as hashes). Thus, within a namespace, the namespace log directly records causal order. This log-based runtime system organization is in contrast with commercial FaaS and serverless platforms, where event logging relies on a statistical sampling of CPU program counter values In CSPOT, the causal ordering is directly recorded and not reconstructed after the fact via sampling. Because it is not generated from samples, SANS-SOUCI can use the CSPOT runtime log to implement correct application replay.

When a WooFPut() is called with handler_name specified, the namespace logs a *TRIGGER* event. If the WooFPut() call does not have handler_name specified, a SANS-SOUCI *APPEND* event is logged instead. When WooFGet() is called, a SANS-SOUCI *READ* event is logged, and when a WooFGetLatestSeqno() is called, a SANS-SOUCI *LATEST_SEQNO* event is logged. The CSPOT API requires the programmer to implement a read of the current tail of a WooF as a call to WooFGetLatestSeqno() that returns a sequence number followed by a call to WooFGet() specifying the element from the WooF to retrieve. In this way, it is possible to implement applications that do not require strong consistency. Finally, when a thread claims a *TRIGGER* event, it appends a *TRIGGER_FIRING* event atomically. Thus, each thread within a namespace container can determine which handlers have yet to be claimed.

Listing 7.1: Log merging algorithm

```
merge_log(logs):
  pending = []
  events = []
  global_log = []
  for log in logs:
    for event in log:
      append(events, event)

  while !empty(events) and !empty(pending):
    for event in events:
      if cause_event(event) in global_log:
        append(global_log, event)
      else:
        append(pending, event)
      remove(events, event)
    for event in pending:
      if cause_event(event) in global_log:
        append(global_log, event)
        remove(pending, event)

cause_event(event):
    return event->cause_host,
      event->cause_seqno
```

7.3.2 Sans-Souci Implementation for CSPOT

An CSPOT log is only local to its namespace. SANS-SOUCI implements a system for gathering and merging the runtime logs from all namespaces used by an application, preserving the causal dependencies globally. Listing 7.1 shows the log merging algorithm. The algorithm is $O(n \times logn)$ in the total number of events; it keeps event lists in search trees to facilitate causal dependency lookup.

SANS-SOUCI's global log, once generated, contains a correct total order of all application events that have occurred and the storage locations (in WooFs) that are associated with the triggering of those events. The size of the CSPOT logs, which is a tunable

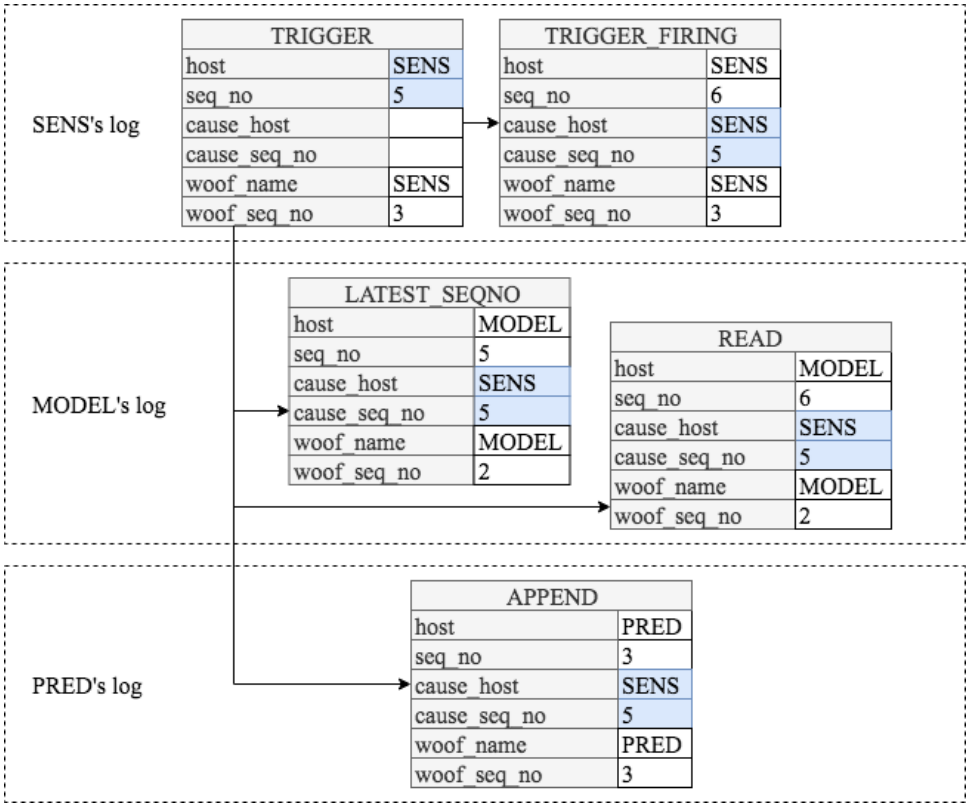


Figure 7.3: CSPOT namespace log events from the Sensor repair example.

parameter, determines the length of this global history. Consider the application shown in Figure 7.1, for example, and assume that *MODEL*, *SENS*, and *PRED* are implemented as three separate CSPOT WooFs hosted in three different namespaces. When a new element *SENS(2)* is put into WooF *SENS*, CSPOT logs a *TRIGGER* event and triggers the handler function (fSENS) to calculate the prediction. The handler first calls WooFGetLatestSeqno() to get the latest sequence number of WooF *MODEL* and then calls WooFGet() to get the latest model parameters with that sequence number. Finally, the handler uses these model parameters to calculate the prediction and puts *PRED(2)* to the *PRED* WooF without triggering a handler. This process generates five events, as shown in Figure 7.3.

Note that SANS-SOUCI only uses a global log to build a repair graph when applica-

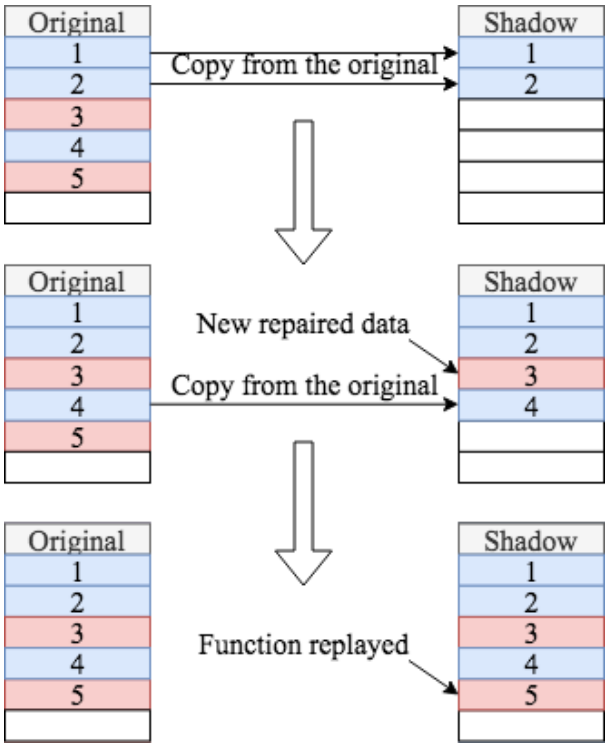


Figure 7.4: An example of a shadow WooF

tions are distributed (e.g., when the application comprises WooFs from more than one namespace). Otherwise, it uses the local namespace log for the namespace containing the application state. In either case, it uses the CSPOT *TRIGGER* and *APPEND* events to identify *Put* dependencies, it adds *READ* events to identify *Get* dependencies, and *LATEST_SEQNO* events to capture accesses to WooF tails. We refer to this latter event as a *Sync* dependency.

Using either the global log (or the local log in case of a single namespace), SANS-SOUCI creates *shadow* WooFs for all WooFs that contain data that is causally dependent on the data being repaired. SANS-SOUCI copies all the previous values from these original WooFs (up to the length of the preserved history) that occur before the target of a repair to the shadow. The repaired value is then inserted with the correct sequence number by replaying the event handler used in the original put with the shadow as the target.

Figure 7.4 shows a simple example of shadow construction. In the example, *Original(3)* is the target of repair and *Original(5)* is a downstream put caused by *Original(3)*. To repair the history, SANS-SOUCI first creates a shadow with the same capacity of the original WooF. All the elements from the earliest sequence number in the original WooF history to the last element before *Original(3)* are copied to the shadow, and then SANS-SOUCI waits for the arrival of the new value of *Original(3)*. After the new element to replace *Original(3)* is appended, SANS-SOUCI copies the intervening element *Original(4)* up to the next downstream element *Original(5)* from the original WooF. Once the intervening element is copied, the value of *Original(5)* which is produced by the function consuming *Original(3)* as input is appended at the correct place in the shadow history. Finally, after all elements are repaired and the remaining elements are copied from the original WooF, the shadow replaces the original WooF (via a rename), and the repair is complete.

7.3.3 Space Optimization

SANS-SOUCI replays *Put* and *Get* dependencies directly when constructing a shadow. However, when a *Sync* dependency is identified, SANS-SOUCI creates a separate mapping of the event's seq_no and the correct sequence number in the history that was returned when the application used the "current" latest sequence number in its original execution.

This contextualization is necessary to effect a space-saving optimization. SANS-SOUCI constructs the shadow in one pass without making a complete copy of *all* program state. Instead, it shadows only the state that is causally dependent on the data being repaired. Thus, it builds the shadow only by appending data that is dependent on either data occurring previously in the shadow *or* state that is unshadowed (i.e., state that contains no appends that are causally dependent on the data being repaired).

In either case, a *Get* dependency is correctly satisfied because it is identified by WooF and sequence number (either in the shadow or in the uncopied state). The "latest" sequence number is likewise correct if it refers to the shadow. However, when a put to the shadow depends on the latest sequence number in state that does not require repair (is unshadowed) this latest sequence number is current to the application and not with respect to the application's history. Thus, the *Sync* dependency represents a space-saving optimization opportunity because it allows SANS-SOUCI to avoid a complete copy of all previous program states into a shadow. However, it requires extra "bookkeeping" to ensure the correct contextualization.

7.3.4 Total versus Causal Ordering During Replay

Note that the current SANS-SOUCI implementation generates a correct *causal* ordering in the repaired WooFs even though each namespace log records a specific *total* order of events in its namespace. It is possible, within a single namespace, to reproduce the total order that occurred, but we elected to forego this additional level of replay accuracy for two reasons.

First, it is only possible to make a total order guarantee within the context of a single namespace (i.e., as recorded by a single log). For applications spanning namespaces, no such guarantee is possible because CSPOT does not use a centralized log in a distributed deployment. However, it may be possible to make such a guarantee in a future implementation that uses a system such as Chariots [131] to implement external consistency.

Secondly, even within the context of a single log (e.g., a single namespace), preserving the total order would not permit handler replay directly. That is, the current implementation of SANS-SOUCI literally refires CSPOT handlers during replay without further

synchronization. It is possible that during the original program's execution, events generated by a single handler may have interleaved with events generated by other unrelated (but concurrently executing) handlers in the namespace. The namespace log correctly captures this interleaving, but to reproduce it, each WooFPut() call in a handler would need to be synchronized with unrelated events in the log. The replay algorithm would need to pause (logically) after every put in every replayed handler and determine whether the non-dependent state (to reproduce the total order exactly) should be copied into the shadow prior to the next put. Because this additional synchronization would only be warranted for applications without cross-namespace dependencies, we felt it to be an unnecessary overhead in implementation of SANS-SOUCI for CSPOT.

Nonetheless, it is clear from our experience with the initial implementation of SANS-SOUCI that it may be possible to make stronger ordering guarantees when we consider implementing it for other FaaS platforms and runtime systems. The utility of such guarantees is the subject of our ongoing and future work.

7.4 Evaluation

We evaluate SANS-SOUCI using two serverless applications originally developed for CSPOT. We use these applications to show SANS-SOUCI's capability to repair the application history and to compare the performance with and without SANS-SOUCI integrated with CSPOT. These applications are designed to run on edge and cloud resources that relatively well-provisioned memory, disk, and operating system capabilities. In particular, they require numerical libraries that are not available or are too large to be hosted across a spectrum of device scales.

Thus, we also implemented a set of micro-benchmarks that are more portable than the IoT applications to further investigate the performance overhead introduced to each

CSPOT API call by SANS-SOUCI across device scales. We first overview our experimental methodology along with the applications and micro-benchmarks, and present our empirical results.

Our results focus on the overheads introduced by SANS-SOUCI with respect to application functions rather than end-to-end application performance to avoid an overly optimistic assessment. For example, the Temperature application (described below) executes on a 5-minute duty cycle when deployed for production use. In one deployment, it has been in continuous operation for almost 18 months. During that period, there have been several outages where SANS-SOUCI could have affected a repair. Had it been available, the total time required to repair these outages would have been approximately 50 seconds over the entire 18 month time period. The long-lived nature of the IoT applications implemented using CSPOT, characterized by possibly intensive computations executed on relatively long duty cycles, could obscure the true "costs" associated with SANS-SOUCI. Thus, we study the overheads on an application component basis rather than as a fraction of end-to-end execution performance.

7.4.1 Experimental Methodology

We use a cloud environment, a Raspberry Pi device (representing an edge computing device), and an esp8266 microcontroller to evaluate the SANS-SOUCI implementation. To evaluate the applications, CSPOT is installed on a campus-level private cloud (approximately 1500 cores) managed using Eucalyptus 4.2.2. We use an m3.2xlarge instance type having 4 CPUs (each 2.8 GHz) and 4GB of memory, and Eucalyptus is configured to use KVM and Virtio for VM hosting. The instances are located in the same availability zone, which interconnects physical hosts using switched 10Gb Ethernet. Each VM instance runs CentOS 7.6 and Docker 18.09 as the container engine. The portable

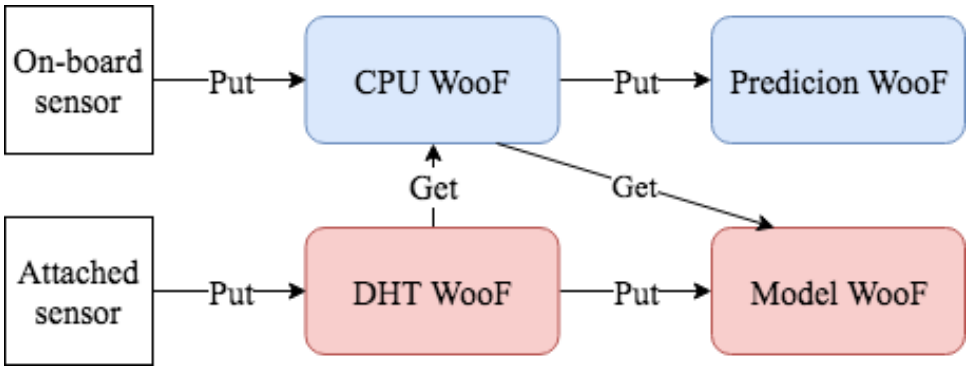


Figure 7.5: The Temperature prediction application structure

micro-benchmarks run on the same cloud environment and a Raspberry Pi 3 Model B+ device. The device has Raspbian 9 installed and has an ARM Cortex-A53 1.4GHz CPU and 1GB SRAM. We modified CSPOT to include necessary logging to implement SANS-SOUCI. We also run the benchmarks on the esp8266 microcontroller which has an 80 MHz RISC CPU with 80 KB of memory and 4 MB of flash storage. The microcontroller runs CSPOT as a native operating system.

The first sample application implements a "virtual" meteorological temperature prediction sensor using the onboard CPU thermometer. To avoid the need for an additional external thermometer (thereby freeing an I/O port on simple IoT devices) the application monitors internal CPU temperature and regresses it against temperature readings (employing a number of data conditioning techniques to improve the regression) taken from a weather station or remote thermometer (shared among all IoT devices in a deployment). In the evaluation, we use a Raspberry Pi as the IoT controller to aid in instrumentation and debugging and an externally connected DHT [2] humidity and temperature sensor as "ground truth." [3] The data conditioning and regression are numerically and memory-intensive computations. Thus, the typical application deployment sends CPU

[2]https://www.adafruit.com/product/393

[3]This sensor synthesis application is used in the field, but when deployed in a non-experimental setting, the sensor controllers are microcontrollers and not small Linux platforms.

temperature measurements either to an edge cloud (e.g., a small x86 cluster running private cloud software) sited in an out building, a private cloud, or a public cloud.

Figure 7.5 shows the structure of the application. To evaluate SANS-SOUCI, we use two VM instances to host the application in the private cloud. One instance stores the CPU temperature readings and generates and stores the "virtual" sensor values (i.e., uses the regression coefficients to produce a "predicted" outdoor temperature using CPU temperature as the explanatory variable). The other instance stores the DHT sensor readings and computes and stores the regression coefficients whenever a new "ground truth" reading is available.

When a new CPU temperature reading is put to the CPU WooF, it triggers a prediction handler. The handler gets the latest regression coefficients (erroring out if there are none yet), generates a predicted outdoor temperature, puts the result to the prediction WooF. Asynchronously, when the DHT sensor reports a temperature reading, a thread running on the sensor puts the reading to the DHT WooF, thereby triggering the regression handler. The regression handler uses the latest CPU temperature and DHT sensor readings to generate new regression coefficients (based on the latest data) and puts them to the model WooF.

We placed a Raspberry Pi in an outdoor environment and collected four days worth of data to run the application. It sends its CPU temperature to a private cloud triggering a CSPOT handler there. The collected data consists of 1152 CPU temperature readings and 1152 DHT sensor readings (one reading every 5 minutes over four consecutive days). Each run of the application generates 79,316 log events in total. To demonstrate SANS-SOUCI's ability to repair application history, we simulate data blackout (a frequent occurrence in the real deployments where the microcontroller uses Xbee [4] radios to communicate) by manually replacing one day of CPU temperature readings with

[4]https://en.wikipedia.org/wiki/XBee

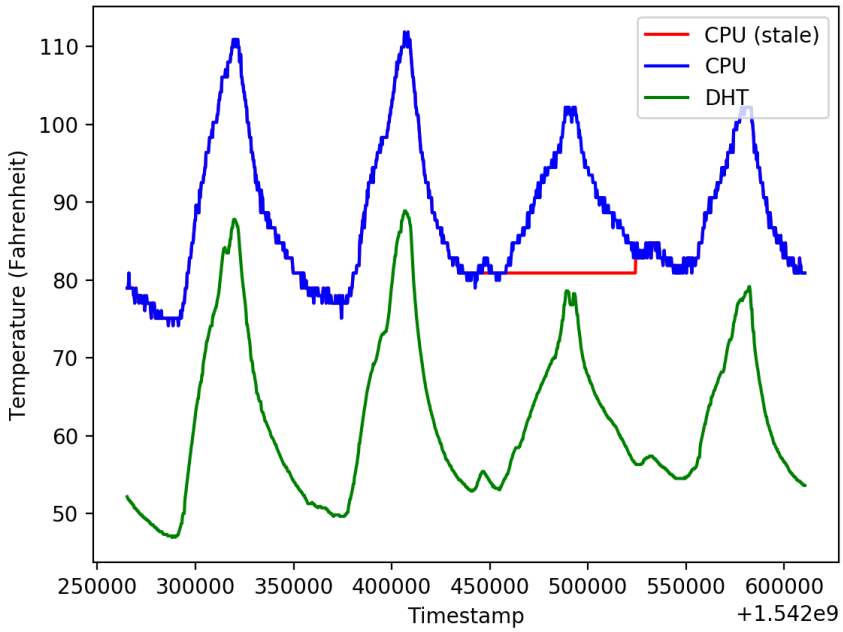


Figure 7.6: The CPU temperature and DHT sensor readings

stale data, as shown in Figure 7.6. After feeding the application with the data with the blackout period, we then use the correct CPU readings that were removed earlier to repair the application history. Figure 7.7 shows the prediction errors before and after the repair. Because of the stale CPU reading, the error before repair spikes to a maximum of 25 degrees Fahrenheit during the data blackout. However, after repair, the application manages to generate predictions with an error within 2 degrees Fahrenheit. While this test is contrived so that we could run it in a controlled environment, it reflects the types of outages that the application experiences in its various non-experimental deployments. Indeed, drop out caused by "late" data delivery in this application served as one of the motivations for this work.

The second sample application is a CSPOT programming example that implements

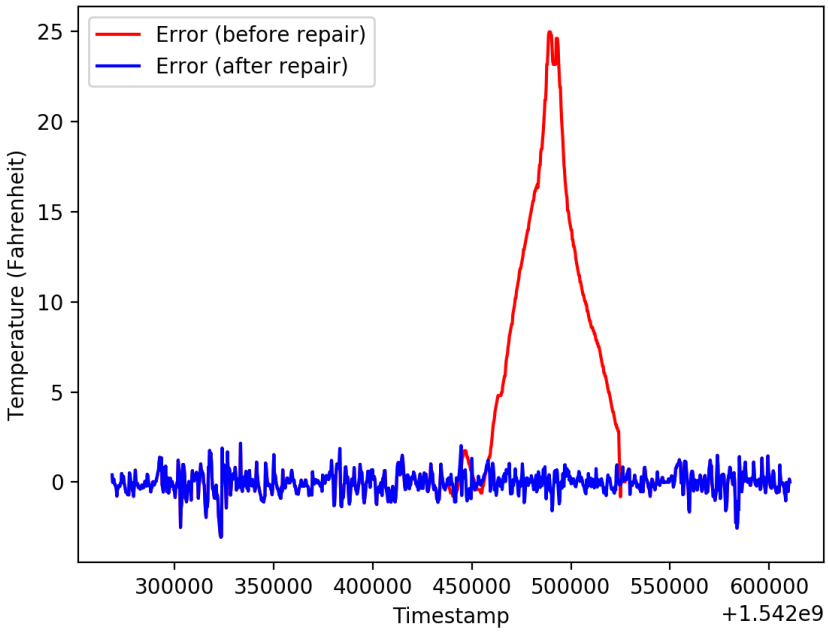


Figure 7.7: The prediction error before and after repair

the Wald-Wolfowitz Runs Test[5] for pseudo-random number generators. Its true function is as a CSPOT exemplar, illustrating a strategy for translating multi-threaded programs to the event-driven abstractions implemented by CSPOT, but it also correctly implements the Runs Test.

Figure 7.8 shows the structure of the application. The application consists of three handlers: a handler to generate a stream of random numbers (denoted R-handler in the figure), a handler to generate the Wald-Wolfowitz Runs Test statistic (denoted S-handler), and a handler to generate Kolmogorov-Smirnov[6]. This application uses only a single instance in our experiments, although the WooFs and handlers can be distributed. Further, the pseudo-random number generator we test in this application is the Mersenne

[5] `https://en.wikipedia.org/wiki/Wald-Wolfowitz_runs_test`
[6] `https://en.wikipedia.org/wiki/Kolmogorov-Smirnov_test`

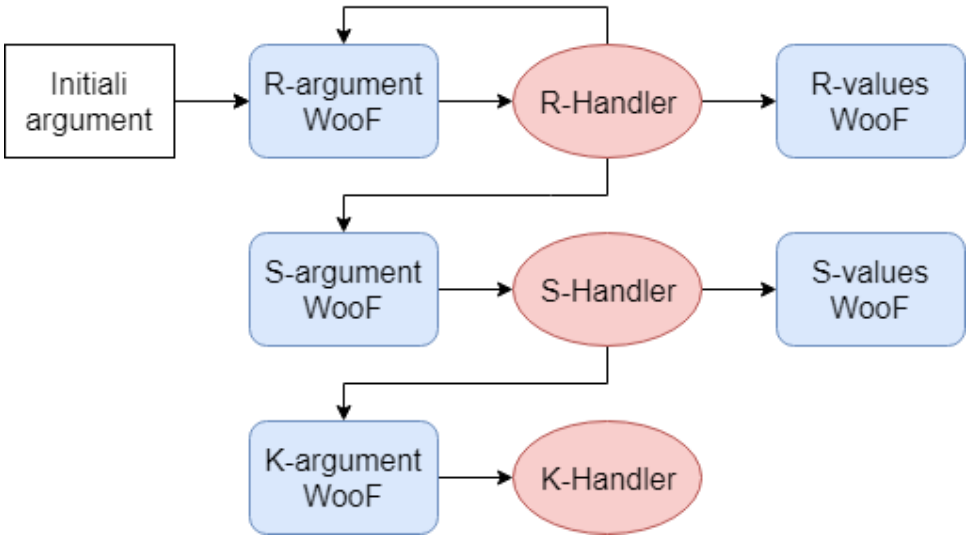


Figure 7.8: The Runs Test application structure

Twister [7] which is known to have good randomness properties. Thus, "ground truth" is a KS statistic that is less than a KS critical value comparing the distribution of Runs Test statistics (over a sample of runs) to a sample from a Normal distribution (having the same mean and standard deviation) for significance level $\alpha = 0.05$. That is, with a Mersenne Twister pseudo-random number generator, we'd expect a KS test comparing a sample of Runs test statistics to a Normal to fail to show a difference at $\alpha = 0.05$ if the test is working correctly.

The application is initialized with the sample size and number of samples to use. When initiated, it triggers the R-handler to generate a new random number which it puts to a sample WooF. It also triggers another R-handler, passing an iteration count, by putting the modified argument structure to the generator WooF.

The R-handler triggers a put to the S-handler when it has accumulated enough data in the sample WooF. The S-handler gets the values from the sample WooF and computes a Runs Test statistic which it puts to the Runs-test WooF. After all iterations (each one

[7] https://en.wikipedia.org/wiki/Mersenne_Twister

	KS stat	Critical value
Before repair	0.31	0.192065
After repair	0.15	0.192065

Table 7.1: Kolmogorov-Smirnov test result

producing a Runs Test statistic from a full sample) are finished, the S-handler triggers the K-handler to get the values from the Runs-test WooF, generate an empirical sample from a Normal distribution having the same sample mean and variance as that computed from the Runs-test WooF values, and to generate a KS test statistic comparing the sample from the Normal to the Runs-test WooF values.

For the purposes of evaluation, we use this application exemplar in two ways. As with the "virtual" temperature sensor application, we show that SANS-SOUCI is able to repair application history by replaying dependent events with new data. We also show how SANS-SOUCI can also be used as a development or debugging aid by replaying application execution after replacing a handler with a different version. To demonstrate this ability, we intentionally "broke" the random number generating function in the R-handler by having it replace every fourth value with zero in the stream of values it produces. Then, we fixed the R-handler, and use the same arguments to replay the application again. By comparing the output before and after the repair, we can see the different KS-test results of the numbers generated by two versions of random number generator.

Table 7.1 shows the KS-test before and after the repair. Before repair (i.e., with the broken generator), the KS statistics correctly shows that the sample of Runs test statistics differs from a Normal at significance level $\alpha = 0.05$. However, after the repair, the KS stat becomes 0.15, which is less than the critical value, i.e., the "fix" repaired the application.

	Temp. App.	Runs Test App.
Cloud w/o extra events	115.98s (1.63s)	2.84s (0.12s)
Cloud w/ extra events	118.56s (0.97s)	2.94s (0.09s)
Cloud w/ SANS-SOUCI	118.97s (1.46s)	3.12s (0.15s)
Total replay overhead (time)	2.99s	0.28s
Total replay overhead (%)	2%	10%

Table 7.2: Applications average elapsed time in seconds with and without SANS-SOUCI over 10 separate experiments. Standard deviations are shown in parentheses.

7.4.2 Replay Overhead

To evaluate replay overhead, we inserted timers at the entry and exit of all handlers that are triggered to time the application. Each application was executed 10 times on the campus private cloud with correct data input and working handler. After each run, the sum of all handler execution time was recorded.

Table 7.2 shows the average execution time with SANS-SOUCI and without SANS-SOUCI. The original CSPOT does not log *PUT* events that do not require a handler, *GET*, and *LATEST_SEQNO* events (these are needed by SANS-SOUCI for dependency tracking but not by CSPOT to implement handler activation). We separate the overhead introduced by SANS-SOUCI into the overhead associated with the necessary additional logging and the overheads associated with SANS-SOUCI processing during replay.

For the temperature prediction application, each run takes 115.98 seconds on average with the unmodified version of CSPOT. After adding the logging of handlerless *PUT*, *GET*, and *LATEST_SEQNO* events, the average time increases to 118.56 seconds. With SANS-SOUCI fully implemented, each run takes 118.97 seconds on average, that is 2.57% of execution overhead. This overhead mainly comes from the additional logging required by SANS-SOUCI. If compared to the CSPOT version, which logs these events (but does not implement other SANS-SOUCI functionality), the overhead is merely 0.35%. For the random number generator application, each run takes 2.84 seconds in average without

Task	Execution time
Dependency discovery	2632ms (76ms)
Merging global log	1426ms (61ms)

Table 7.3: Average execution time in milliseconds over 10 experiments for each task in repair request. Standard deviations are shown in parentheses.

extra logging and 2.94 seconds with extra logging. With SANS-SOUCI fully implemented, each run takes 3.12 seconds on average, which translates to 3.52% of logging overhead and 6.12% of SANS-SOUCI execution overhead.

For the Runs Test application, the overheads are lower in absolute terms but higher as a percentage (approximately 10% on average). Its computational intensity is significantly less than for the Temperature application meaning the overheads associated with the runtime environment are a larger fraction of overall execution time.

7.4.3 Log Processing Overhead

In this section, we describe the overhead associated with generating the repair graph necessary to enable replay. To generate the graph, SANS-SOUCI must "discover" the causal dependencies associated with the "root" of each repair. If the application uses only a single namespace, these dependencies are recorded in the log associated with the namespace. However, when the application spans namespaces, SANS-SOUCI first gathers the logs for all the namespaces and merges them into a global log to create a total order of application events that preserves causal order. It then uses the global log to identify the causal dependencies associated with each "root."

To understand the performance of dependency discovery and log merging, we run the Temperature application on the campus cloud described in the previous section and time the repair request 10 times. Table 7.3 shows the recorded time for each task.

Recall that with SANS-SOUCI enabled the Temperature application generates 79,316

Name	Description
Put	Put 1,000 elements to the WooF
Get	Randomly get 1,000 elements from the WooF
GetLatestSeqno	Get the latest sequence number 1,000 times
Replication	Replicate a 4k data object from edge device to cloud

Table 7.4: Micro-benchmarks

logged events (which CSPOT stores in approximately 55 megabytes) each run covering four days of measurement history. Parsing this event log to discover and build the repair graph requires approximately 2.6 seconds on average. If the application uses two separate namespaces (one containing CPU measurements and the predicted CPU values and the other containing all other WooFs) then average time to merge the logs from these namespaces is 1.4 seconds. These results are both application and deployment-specific. That is, the complexity of the repair graph and also the distribution of WooFs among namespaces will affect both the discovery and merge times. However, as an example, they indicate that the overhead associated with a repair is low.

Specifically, from Table 7.2, a SANS-SOUCI replay adds approximately 3 seconds to a 115 second execution time to the Temperature application when using unmodified CSPOT. Each repair will then impose an additional 2.6 seconds to parse the log and, if spanning namespaces, a further 1.4 seconds to gather and sequence the global log. Even when a repair is effected, the overall additional overhead introduced by SANS-SOUCI is under 10% for this application.

7.4.4 Micro-benchmarks

To better understand the overhead imposed by SANS-SOUCI on each CSPOT API call along with the SANS-SOUCI performance during the repair, we implemented a set of micro-benchmarks as listed in Table 7.4. We ran these micro-benchmarks 100 times

	Put	Get	GetLatestSeqno
Cloud w/o extra events	143.42us (2.57us)	136.50us (2.56us)	17.53us (1.72us)
Cloud w/ extra exents	-	142.94us (3.51us)	23.32us (1.66us)
Cloud w/ SANS-SOUCI	143.72us (3.55us)	142.76us (3.45us)	29.73us (2.34us)
Cloud during repair	185.40us (2.51us)	157.89us (1.10us)	38.42us (7.72us)
Rpi w/o extra events	504.62us (3.92us)	507.28us (4.49us)	103.18us (2.14us)
Rpi w/ extra events	-	521.20us (8.84us)	118.49us (2.71us)
Rpi w/ SANS-SOUCI	506.24us (10.44us)	523.14us (5.61us)	118.51us (2.61us)
Rpi during repair	681.91us (11.81us)	519.75us (4.61us)	116.19us (2.03us)
ucontroller w/o extra events	26.07us (0.17us)	7.64us (0.01us)	-
ucontroller w/ extra events	-	23.45us (0.15us)	-
ucontroller w/ SANS-SOUCI	26.45us (0.17us)	23.62us (0.13us)	-
ucontroller during repair	26.75us (0.19us)	29.9us (0.15us)	-

Table 7.5: Average Micro-benchmarks performance per 1,000 requests. The number shown represents the time for each CSPOT call. The units are microseconds and the standard deviations are shown in parentheses.

(each consisting of a batch of 1000 invocations) and recorded the average with three versions of CSPOT: the original version without extra dependency events logged, with the additional events needed by SANS-SOUCI logged, and with the SANS-SOUCI fully integrated. We also ran the micro-benchmarks, repaired the entire benchmark data history using the same input, and timed the operations during the full repair. In each micro-benchmark, we insert timers at the entry and exit of the each application function. We present the average execution times and standard deviations in microseconds (us) in Table 7.5 for each micro-benchmark. Note that no additional PUT events are required by SANS-SOUCI for the Put benchmark making the timings with and without these events the same.

On the cloud, adding SANS-SOUCI events introduces roughly 6us, on average. On the Raspberry Pi, adding extra logging adds roughly 15us, on average, while on the microcontroller the additional overhead due to logging is 16us, on average. SANS-SOUCI doesn't seem to introduce any overhead to WooFPut and WooFGet, mainly because

	Replication time
Normal run	39.32ms (3.16ms)
Replay	40.27ms (3.46ms)

Table 7.6: Average elapsed time in milliseconds (over 100 runs) to replicate a 4k data object from an edge device to a private cloud. Standard deviations are shown in parentheses.

SANS-SOUCI only needs to check whether the WooF is in repair mode (i.e., a boolean test). If so, the put and get request will be redirected to the shadow WooF. If the WooF is not being repaired, there's no additional performance overhead introduced. However, if the WooF is being repaired, CSPOT needs to open the shadow WooF and redirect the request to it, hence the overhead. For each WooFPut request, the overhead during repair is 42us on the cloud, 176us on Raspberry Pi, but we did not measure any noticeable overhead on the microcontroller. For each WooFGet request, the overhead is 16us on the cloud and 6us on the microcontroller, but we did not observe overhead on Raspberry Pi.

For WooFGetLatestSeqno request, since it needs to record the mapping between the caller WooF's sequence number and the callee WooF's latest sequence number, even if the callee WooF is not in repair mode, there is a slight overhead introduced. During repair, WooFGetLatestSeqno also needs to find the latest sequence number in the mapping corresponding to the caller WooF's sequence number, introducing more overhead. In the cloud environment, to implement SANS-SOUCI, each WooFGetLatestSeqno request requires an additional 10us. During repair, the overhead doubles to 20us. On the microcontroller, we have not yet implemented the sequence number mapping for the *Sync* dependency, so the overhead for WooFGetLatestSeqno is left out. That is, the current SANS-SOUCI implementation for the microcontroller stops the application during repair, making the *Sync* dependency superfluous. Again, we did not observe any overhead on Raspberry Pi and are still investigating the reason why SANS-SOUCI does not seem to introduce overhead to WooFGet and WooFGetLatestSeqno on this platform.

Finally, we also evaluate the end-to-end overhead from an edge device to a private cloud. In the "Replication" benchmark, we installed CSPOT on a Raspberry Pi located in a research laboratory located on the same university campus hosting the private cloud. The network interface attached to the Raspberry Pi is a 1 Gb/sec Ethernet, and all traffic between this edge device and an instance in the cloud running CSPOT traversed the shared campus network. Both edge and cloud systems use NTP [8] to synchronize their internal clocks using a campus NTP server.

The benchmark first puts an object with 4 kilobyte payload to the Raspberry Pi, triggering a handler that reads the local clock to generate a timestamp that it embeds in the 4K payload. It then forwards the object to the private cloud instance. Upon its arrival, a handler is triggered on the cloud instance, which takes another timestamp. We then record the difference of the timestamps as the end-to-end latency to replicate a 4K data object from the edge device to the private cloud.

We ran the benchmark, requested a repair, and then ran it again to evaluate the overhead SANS-SOUCI introduces to a simple cross-network replication. We repeated the process 100 times and showed the average in Table 7.6. It takes 39.32 milliseconds to replicate a data object from edge to cloud, on average. To replay the replication and repair the WooF requires 40.27 milliseconds. That is, for a simple replication task which is not computationally intensive, SANS-SOUCI introduces additional 0.95 milliseconds, which equates to 2.4% of overhead.

7.5 Conclusion

In this chapter, we explore a new methodology for implementing data repair in IoT applications that use the FaaS programming model. Our data repair framework leverages

[8] http://www.ntp.org

the knowledge we gained from previous work, including causal dependency tracking and event replay. We describe the process of repair *in situ*, taking advantage of FaaS function execution to "replay" an application from the point in its state update history where a faulty data item is to be replaced. To do so, our methodology relies on the availability of causal event tracking we develop and versioned state updates in the underlying infrastructure.

We evaluate a prototype implementation of SANS-SOUCI using CSPOT, a portable FaaS runtime platform for IoT, for its versioned data structure and FaaS execution model. With its append-only state update semantics, SANS-SOUCI is able to maintain a history of application state updates. Using the CSPOT's internal system log, we add several system events corresponding to different types of application state updates to implement distributed causal event tracking. Thus, we are able to facilitate debugging of highly concurrent distributed IoT applications with SANS-SOUCI.

We use microbenchmarks and two distributed IoT applications to evaluate SANS-SOUCI. We host the evaluation applications on a multi-tier environment that consists of devices from different tiers, including microcontrollers, single-board computers, and cloud virtual machines. Our results show that SANS-SOUCI can repair data corruption caused by misconfiguration and loss data while introducing runtime overheads typically less than 10%.

Chapter 8

Conclusions and Future Work

IoT realizes a new vision for software developers and data analysts and enables numerous application innovations by providing sensing, actuation and control, and computing capabilities that is embedded in the world around us. Combining the use of computational resources at different scales, we can implement remarkable applications in different fields, such as intelligent urban systems, healthcare, education, and warehouse and factory management. IoT defines the enabling technologies that allow us to perform a wide range of complex tasks and greatly improve our quality of life.

However, as a result of technological advances, these innovations are becoming more demanding on end-to-end response, network, and computational capabilities. In the modern IoT landscape, to implement an average IoT application, developers often need to utilize several different hardware, programming models, cloud services, and messaging systems. This approach, however, introduces new challenges surrounding heterogeneity, programmability, and reliability.

In this thesis, we study popular and emerging computing models and systems, including cloud and edge computing, Functions-as-a-Service, publish/subscribe messaging, and data replication. We investigate the limitations of the currently available services that

attempt to address the above problems. Then, based on our experiences and observations with real deployments and applications, we explore and develop several systems that answer the following question: **"Can we a uniformly programmable, distributed, reliable, event-based system for multi-tier IoT deployments?"** With careful design and after extensive empirical evaluation, our answer is a resounding yes.

We first design a portable FaaS runtime system (CSPOT) that makess distributed and multi-tier IoT applications portable. CSPOT implements FaaS execution model and lightweight abstractions uniformly across all tiers of IoT deployments. It allows developers to write short functions once for deployment across all tiers. In addition, it allows developers to exploit data locality that is often required by IoT applications to optimize for performance.

Built on top of CSPOT, we integrate a topic-based publish/subscribe (pub/sub) messaging pattern into the FaaS model. Toward this end, we develop Canal, an programmable pub/sub system for event-driven IoT applications. With the combination of FaaS and pub/sub, developers are able to implement FaaS functions against data topics instead of against hardware specifications and network addresses. Canal uses a distributed hashtable for topic lookup and the Raft consensus protocol for reliable data replication. To overcome the locality issue caused by the loose coupling design of traditional pub/sub systems, Canal decouples network address from topic lookup, allowing developers to exploit locality and explore different replica placement strategies to optimize applications for reliability and performance.

To tackle the challenges surrounding post-deployment management of FaaS applications, we study many popular services that are currently available for cloud computing and their limitations. As a result, we develop GammaRay and Lowgo to facilitate debugging, monitoring, and analysis in distributed FaaS applications. As the first step, we improve the design of AWS X-Ray and build GammaRay to capture the event depen-

dency within AWS Lambda applications across services and regions. With the experience we gathered from building this bolt-on service, we implement our own event tracking system Lowgo that works across different cloud systems. To minimize the event logging performance overhead, Lowgo uses a geo-distributed design to maintain a shared log. Multiple Lowgo instances communicate with each other asynchronously to reconstruct the causal order of events within applications. For the ease of deployment, we provide toolchains for both GammaRay and Lowgo to developers, allowing them to manage their applications and track causal dependencies without requiring source code modification.

Finally, to deal with inevitable hardware failures, misconfigurations, and data loss that are common in IoT environments, we develop a novel method to repair application history using a replay technique. The proposed repair framework, Sans-Souci, extends CSPOT's FaaS execution model and append-only abstraction to implement application history tracking and editing. We extend the event logging mechanism of CSPOT and add several types of events corresponding to different application operations (e.g., read, write, function trigger.) Sans-Souci uses the causality information embedded in events to construct a repair graph and identify related data and downstream computations. Finally, Sans-Souci repairs the history by replaying the causally related computations.

To evaluate the performance of our contributions in real-world scenarios, we use computational resources from all tiers of IoT environments, including microcontrollers, single-board computers, edge devices, private clouds, and public clouds to run a wide range of benchmarks and end-to-end applications across our systems. We find that our research artifacts simplify the design, implementation, and deployment of multi-tier IoT applications. Moreover, our contributions make IoT applications and their data more reliable, easier to reason about, and tolerant to corruption.

Our research contributions can be extended in multiple ways. First, we can add

additional language support to the FaaS platform. In the current state, CSPOT (and Canal) support C, C++, and Python binding for function implementation. We also would like to extend Canal with intelligent load balancing for function invocations and autotuning that optimizes replica placement and leader election according to deployments with different network characteristics. Finally, Canal can benefit from service discovery of topics and functions. With these features, it has the potential to be a fully automated application platform for FaaS execution and distributed data management that requires minimal manual intervention beyond application development.

Bibliography

[1] "Cisco Annual Internet Report (2018–2023) White Paper." `https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html`. [Online; accessed 6-May-2021].

[2] G. Yang, L. Xie, M. Mäntysalo, X. Zhou, Z. Pang, L. D. Xu, S. Kao-Walter, Q. Chen, and L.-R. Zheng, *A health-iot platform based on the integration of intelligent packaging, unobtrusive bio-sensor, and intelligent medicine box*, *IEEE Transactions on Industrial Informatics* **10** (2014), no. 4 2180–2191.

[3] M. Hassanalieragh, A. Page, T. Soyata, G. Sharma, M. Aktas, G. Mateos, B. Kantarci, and S. Andreescu, *Health monitoring and management using internet-of-things (iot) sensing with cloud-based processing: Opportunities and challenges*, in *2015 IEEE International Conference on Services Computing*, pp. 285–292, 2015.

[4] C. Doukas and I. Maglogiannis, *Bringing iot and cloud computing towards pervasive healthcare*, in *2012 Sixth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, pp. 922–926, 2012.

[5] W. Tärneberg, V. Chandrasekaran, and M. Humphrey, *Experiences creating a framework for smart traffic control using aws iot*, in *Proceedings of the 9th International Conference on Utility and Cloud Computing*, UCC '16, (New York, NY, USA), p. 63–69, Association for Computing Machinery, 2016.

[6] F. Zhu, Y. Lv, Y. Chen, X. Wang, G. Xiong, and F.-Y. Wang, *Parallel transportation systems: Toward iot-enabled smart urban traffic control and management*, *IEEE Transactions on Intelligent Transportation Systems* **21** (2020), no. 10 4063–4071.

[7] Y. Lee, J. Kim, H. Lee, and K. Moon, *Iot-based data transmitting system using a uwb and rfid system in smart warehouse*, in *2017 Ninth International Conference on Ubiquitous and Future Networks (ICUFN)*, pp. 545–547, 2017.

[8] H. Chourabi, T. Nam, S. Walker, J. R. Gil-Garcia, S. Mellouli, K. Nahon, T. A. Pardo, and H. J. Scholl, *Understanding smart cities: An integrative framework*, in

2012 45th Hawaii International Conference on System Sciences, pp. 2289–2297, 2012.

[9] Y. Sun, H. Song, A. J. Jara, and R. Bie, *Internet of things and big data analytics for smart and connected communities*, *IEEE Access* **4** (2016) 766–773.

[10] K. Su, J. Li, and H. Fu, *Smart city and the applications*, in *2011 International Conference on Electronics, Communications and Control (ICECC)*, pp. 1028–1031, 2011.

[11] N. Golubovic, R. Wolski, C. Krintz, and M. Mock, *Improving the Accuracy of Outdoor Temperature Prediction by IoT Devices*, in *IEEE Conference on IoT*, 2019.

[12] C. Krintz, R. Wolski, N. Golubovic, and F. Bakir, *Estimating Outdoor Temperature from CPU Temperature for IoT Applications in Agriculture*, in *International Conference on the Internet of Things*, Oct., 2018.

[13] A. R. Elias, N. Golubovic, C. Krintz, and R. Wolski, *Where's the Bear? – Automating Wildlife Image Processing Using IoT and Edge Cloud Systems*, in *Internet-of-Things Design and Implementation (IoTDI), 2017 IEEE/ACM Second International Conference on*, pp. 247–258, IEEE, 2017.

[14] M. S. Mekala and P. Viswanathan, *A novel technology for smart agriculture based on iot with cloud computing*, in *2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*, pp. 75–82, 2017.

[15] A. Botta, W. de Donato, V. Persico, and A. Pescapé, *On the integration of cloud computing and internet of things*, in *2014 International Conference on Future Internet of Things and Cloud*, pp. 23–30, 2014.

[16] "Amazon Web Service." `https://aws.amazon.com/`. [Online; accessed 6-May-2021].

[17] "Google Cloud Platform." `https://aws.amazon.com/`. [Online; accessed 6-May-2021].

[18] "Microsoft Azure." `http://azure.microsoft.com/`. [Online; accessed 6-May-2021].

[19] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, *The eucalyptus open-source cloud-computing system*, in *Cluster Computing and the Grid, 2009. CCGRID'09. 9th IEEE/ACM International Symposium on*, pp. 124–131, IEEE, 2009.

[20] "OpenStack." `https://www.openstack.org/`. [Online; accessed 6-May-2021].

[21] "Apache CloudStack." `https://cloudstack.apache.org/`. [Online; accessed 6-May-2021].

[22] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, *Maui: Making smartphones last longer with code offload*, in *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, (New York, NY, USA), p. 49–62, Association for Computing Machinery, 2010.

[23] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, *Clonecloud: Elastic execution between mobile device and cloud*, in *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, (New York, NY, USA), p. 301–314, Association for Computing Machinery, 2011.

[24] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, *Fog computing and its role in the internet of things*, in *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pp. 13–16, ACM, 2012.

[25] B. Zhang, N. Mor, J. Kolb, D. S. Chan, N. Goyal, K. Lutz, E. Allman, J. Wawrzynek, E. Lee, and J. Kubiatowicz, *The cloud is not enough: Saving iot from the cloud*, in *Proceedings of the 7th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'15, (USA), p. 21, USENIX Association, 2015.

[26] "Internet of Things Solutions - Google Cloud Platform." `https://cloud.google.com/solutions/iot/`. [Online; accessed 12-Sep-2017].

[27] "AWS IoT Core." "https://aws.amazon.com/iot-core/" [Online; accessed 12-Sep-2017].

[28] "Azure Internet of Things." `https://www.microsoft.com/en-us/cloud-platform/internet-of-things-azure-iot-suite`. [Online; accessed 22-Aug-2016].

[29] K. Dolui and S. K. Datta, *Comparison of edge computing implementations: Fog computing, cloudlet and mobile edge computing*, in *2017 Global Internet of Things Summit (GIoTS)*, pp. 1–6, 2017.

[30] S. Singh, *Optimize cloud computations using edge computing*, in *2017 International Conference on Big Data, IoT and Data Science (BID)*, pp. 49–53, 2017.

[31] J. Ren, D. Zhang, S. He, Y. Zhang, and T. Li, *A survey on end-edge-cloud orchestrated network computing paradigms: Transparent computing, mobile edge computing, fog computing, and cloudlet*, *ACM Comput. Surv.* **52** (Oct., 2019).

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

[33] T. Verbelen, P. Simoens, F. De Turck, and B. Dhoedt, *Cloudlets: bringing the cloud to the mobile user*, in *ACM workshop on Mobile cloud computing and services*, ACM, 2012.

[34] B. I. Ismail, E. Mostajeran Goortani, M. B. Ab Karim, W. Ming Tat, S. Setapa, J. Y. Luke, and O. Hong Hoe, *Evaluation of docker as edge computing platform*, in *2015 IEEE Conference on Open Systems (ICOS)*, pp. 130–135, 2015.

[35] M. Jia, J. Cao, and W. Liang, *Optimal cloudlet placement and user to cloudlet allocation in wireless metropolitan area networks*, *IEEE Transactions on Cloud Computing* **5** (2017), no. 4 725–737.

[36] Y. Zhang, D. Niyato, and P. Wang, *Offloading in mobile cloudlet systems with intermittent connectivity*, *IEEE Transactions on Mobile Computing* **14** (2015), no. 12 2516–2529.

[37] V. Cozzolino, A. Y. Ding, and J. Ott, *Fades: Fine-grained edge offloading with unikernels*, in *Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems*, HotConNet '17, (New York, NY, USA), p. 36–41, Association for Computing Machinery, 2017.

[38] L. Ma, S. Yi, and Q. Li, *Efficient service handoff across edge servers via docker container migration*, in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, SEC '17, (New York, NY, USA), Association for Computing Machinery, 2017.

[39] X. Wei, S. Wang, A. Zhou, J. Xu, S. Su, S. Kumar, and F. Yang, *Mvr: An architecture for computation offloading in mobile edge computing*, in *2017 IEEE International Conference on Edge Computing (EDGE)*, pp. 232–235, 2017.

[40] "AWS Greengrass." "https://aws.amazon.com/greengrass/" [Online; accessed 12-Sep-2017].

[41] "AWS Lambda." `https://aws.amazon.com/lambda/`. [Online; accessed 8-Apr-2021].

[42] "Azure IoT Edge." `https://azure.microsoft.com/en-us/services/iot-edge/`. [Online; accessed 22-Aug-2018].

[43] "Docker." `https://www.docker.com` [Online; accessed 12-Sep-2017].

[44] "Azure IoT Hub." `https://azure.microsoft.com/en-us/services/iot-hub/` [Online; accessed 22-Aug-2018].

[45] "Google IoT Core." https://cloud.google.com/iot-core/. [Online; accessed 12-Sep-2019].

[46] A. Das, S. Patterson, and M. Wittie, *Edgebench: Benchmarking edge computing platforms*, in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pp. 175–180, 2018.

[47] H. Chang, A. Hari, S. Mukherjee, and T. V. Lakshman, *Bringing the cloud to the edge*, in *2014 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pp. 346–351, 2014.

[48] S. Suryavansh, C. Bothra, M. Chiang, C. Peng, and S. Bagchi, *Tango of edge and cloud execution for reliability*, in *Proceedings of the 4th Workshop on Middleware for Edge Clouds & Cloudlets*, MECC '19, (New York, NY, USA), p. 10–15, Association for Computing Machinery, 2019.

[49] D. R. Vasconcelos, R. M. C. Andrade, V. Severino, and J. N. D. Souza, *Cloud, fog, or mist in iot? that is the question*, *ACM Trans. Internet Technol.* **19** (Mar., 2019).

[50] R. Jain and S. Tata, *Cloud to edge: Distributed deployment of process-aware iot applications*, in *2017 IEEE International Conference on Edge Computing (EDGE)*, pp. 182–189, 2017.

[51] "Serverless Framework." https://en.wikipedia.org/wiki/Serverless_Framework [Online; accessed 12-Sep-2017].

[52] "FaaS, PaaS, and the Benefits of the Serverless Architecture." https://www.infoq.com/news/2016/06/faas-serverless-architecture [Online; accessed 1-Nov-2016].

[53] "Amazon Serverless computing or Google Function as a Service (FaaS) vs Microservices and Container Technologies." https://cloudramblings.me/2016/09/12/amazon-serverless-computing-or-google-function-as-a-service-faas-vs-microservices-and-container-technologies [Online; accessed 12-Sep-2020].

[54] "Linux LXC." "https://en.wikipedia.org/wiki/LXC" [Online; accessed 22-Aug-2016].

[55] "Linux Containers." "https://linuxcontainers.org" [Online; accessed 22-Aug-2016].

[56] *Aws lambda for web services*, 2019. "https://aws.amazon.com/getting-started/projects/build-serverless-web-app-lambda-apigateway-s3-dynamodb-cognito/" [Online; accessed on 12-Aug-2019].

[57] "Google Cloud Functions." https://cloud.google.com/functions. [Online; accessed 7-May-2021].

[58] "Azure Functions." https://azure.microsoft.com/en-us/services/functions/. [Online; accessed 7-May-2021].

[59] "IBM Cloud Functions." https://cloud.ibm.com/functions/. [Online; accessed 7-May-2021].

[60] "OpenFaaS." "https://www.openfaas.com" [Online; accessed 12-Sep-2017].

[61] "IBM OpenWhisk." https://developer.ibm.com/openwhisk/. [Online; accessed 12-Sep-2017].

[62] "Kubeless." http://kubeless.io/. [Online; accessed 11-Sept-2017].

[63] "Knative." https://knative.dev/. [Online; accessed 7-May-2021].

[64] "Fission." http://fission.io/. [Online; accessed 11-Sept-2017].

[65] K. Kritikos and P. Skrzypek, *A review of serverless frameworks*, in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pp. 161–168, 2018.

[66] S. K. Mohanty, G. Premsankar, and M. di Francesco, *An evaluation of open source serverless computing frameworks*, in *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 115–120, 2018.

[67] A. Palade, A. Kazmi, and S. Clarke, *An evaluation of open source serverless computing frameworks support at the edge*, in *2019 IEEE World Congress on Services (SERVICES)*, vol. 2642-939X, pp. 206–211, 2019.

[68] "MQTT." https://mqtt.org/. [Online; accessed 28-Sep-2018].

[69] A. Stanford-Clark and H. Truong, *MQTT for sensor networks (MQTT-S) protocol specification*, *International Business Machines Corporation version* **1** (2008).

[70] "Apache ActiveMQ." https://activemq.apache.org/. [Online; accessed 7-May-2021].

[71] "RabbitMQ." https://www.rabbitmq.com [Online; accessed 1-Nov-2016].

[72] S. Chun, S. Shin, S. Seo, S. Eom, J. Jung, and K. Lee, *A pub/sub-based fog computing architecture for internet-of-vehicles*, in *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 90–93, 2016.

[73] *Apache Kafka*, 2019. `http://kafka.apache.org` [Online; accessed Sep. 2019].

[74] G. J. Chen, J. L. Wiener, S. Iyer, A. Jaiswal, R. Lei, N. Simha, W. Wang, K. Wilfong, T. Williamson, and S. Yilmaz, *Realtime data processing at facebook*, in *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, (New York, NY, USA), p. 1087–1098, Association for Computing Machinery, 2016.

[75] "Apache Flume." `http://flume.apache.org/`. [Online; accessed 11-Mar-2021].

[76] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, *The Hadoop Distributed File System*, in *IEEE Symposium on Mass Storage Systems and Technologies*, 2010.

[77] P. R. Pietzuch and J. M. Bacon, *Hermes: a distributed event-based middleware architecture*, in *Proceedings 22nd International Conference on Distributed Computing Systems Workshops*, pp. 611–618, 2002.

[78] M. Castro, P. Druschel, A. . Kermarrec, and A. I. T. Rowstron, *Scribe: a large-scale and decentralized application-level multicast infrastructure*, *IEEE Journal on Selected Areas in Communications* **20** (2002), no. 8 1489–1499.

[79] T. Chang, S. Duan, H. Meling, S. Peisert, and H. Zhang, *P2s: A fault-tolerant publish/subscribe infrastructure*, in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, (New York, NY, USA), p. 189–197, Association for Computing Machinery, 2014.

[80] L. Lamport, *The part-time parliament*, *ACM Trans. Comput. Syst.* **16** (May, 1998) 133–169.

[81] D. Happ and A. Wolisz, *Limitations of the pub/sub pattern for cloud based iot and their implications*, in *2016 Cloudification of the Internet of Things (CIoT)*, pp. 1–6, 2016.

[82] B. Mishra and A. Kertesz, *The use of mqtt in m2m and iot systems: A survey*, *IEEE Access* **8** (2020) 201071–201086.

[83] E. Brewer, *Cap twelve years later: How the "rules" have changed*, *Computer* **45** (2012), no. 2 23–29.

[84] R. Boichat, P. Dutta, S. Frølund, and R. Guerraoui, *Deconstructing paxos*, *SIGACT News* **34** (Mar., 2003) 47–67.

[85] T. D. Chandra, R. Griesemer, and J. Redstone, *Paxos made live: An engineering perspective*, in *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, (New York, NY, USA), p. 398–407, Association for Computing Machinery, 2007.

[86] B. W. Lampson, *How to build a highly available system using consensus*, in *Distributed Algorithms* (Ö. Babaoğlu and K. Marzullo, eds.), (Berlin, Heidelberg), pp. 1–17, Springer Berlin Heidelberg, 1996.

[87] R. Van Renesse and D. Altinbuken, *Paxos made moderately complex*, *ACM Comput. Surv.* **47** (Feb., 2015).

[88] L. Lamport, *Paxos made simple*, *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (December, 2001) 51–58.

[89] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh, *Megastore: Providing scalable, highly available storage for interactive services*, in *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pp. 223–234, 2011.

[90] J. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, *et. al.*, *Spanner: Google's globally-distributed database*, in *OSDI*, pp. 251–264, 2012.

[91] M. Burrows, *The chubby lock service for loosely-coupled distributed systems*, in *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[92] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlfield, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte, *F1: A distributed sql database that scales*, in *VLDB*, 2013.

[93] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, *Large-scale cluster management at google with borg*, in *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, (New York, NY, USA), Association for Computing Machinery, 2015.

[94] D. Ongaro and J. Ousterhout, *In search of an understandable consensus algorithm*, in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, (Philadelphia, PA), pp. 305–319, USENIX Association, June, 2014.

[95] "Kubernetes." `https://kubernetes.io/`. [Online; accessed 7-May-2021].

[96] "etcd." `https://etcd.io/`. [Online; accessed 7-May-2021].

[97] "M3." `https://m3db.io/`. [Online; accessed 7-May-2021].

[98] "Trillian." `https://github.com/google/trillian`. [Online; accessed 7-May-2021].

[99] "CockroachDB." `https://www.cockroachlabs.com/`. [Online; accessed 7-May-2021].

[100] "Redis." `http://redis.io`. [Online; accessed 7-May-2021].

[101] E. Cohen and S. Shenker, *Replication strategies in unstructured peer-to-peer networks*, in *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '02, (New York, NY, USA), p. 177–190, Association for Computing Machinery, 2002.

[102] C. Ye and D. M. Chiu, *Peer-to-peer replication with preferences*, in *Proceedings of the 2nd International Conference on Scalable Information Systems*, InfoScale '07, (Brussels, BEL), ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2007.

[103] A. Brinkmann and S. Effert, *Data replication in p2p environments*, in *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, (New York, NY, USA), p. 191–193, Association for Computing Machinery, 2008.

[104] L. Gao, M. Dahlin, A. Nayate, J. Zheng, and A. Iyengar, *Application specific data replication for edge services*, in *Proceedings of the 12th International Conference on World Wide Web*, WWW '03, (New York, NY, USA), p. 449–460, Association for Computing Machinery, 2003.

[105] L. Gao, M. Dahlin, J. Zheng, L. Alvisi, and A. Iyengar, *Dual-quorum: A highly available and consistent replication system for edge services*, *IEEE Transactions on Dependable and Secure Computing* **7** (2010), no. 2 159–174.

[106] Y. Lin, B. Kemme, M. Patino-Martinez, and R. Jimenez-Peris, *Enhancing edge computing with database replication*, in *2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*, pp. 45–54, 2007.

[107] W. Hao, J. Fu, I.-L. Yen, and Z. Xia, *Achieving high performance web applications by service and database replications at edge servers*, in *2009 IEEE 28th International Performance Computing and Communications Conference*, pp. 153–160, 2009.

[108] H. Saxena and K. Salem, *Edgex: Edge replication for web applications*, in *2015 IEEE 8th International Conference on Cloud Computing*, pp. 1041–1044, 2015.

[109] Q. Xia, L. Bai, W. Liang, Z. Xu, L. Yao, and L. Wang, *Qos-aware proactive data replication for big data analytics in edge clouds*, in *Proceedings of the 48th International Conference on Parallel Processing: Workshops*, ICPP 2019, (New York, NY, USA), Association for Computing Machinery, 2019.

[110] "Amazon Simple Storage Service (Amazon S3)." `https://aws.amazon.com/s3/`. [Online; accessed 7-May-2021].

[111] M. Rosenblum and J. K. Ousterhout, *The design and implementation of a log-structured file system*, *ACM Transactions on Computer Systems (TOCS)* **10** (1992), no. 1 26–52.

[112] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, *Serverless computation with openlambda*, in *HotCloud*, 2016.

[113] M. S. Aslanpour, A. N. Toosi, C. Cicconetti, B. Javadi, P. Sbarski, D. Taibi, M. Assuncao, S. S. Gill, R. Gaire, and S. Dustdar, *Serverless edge computing: Vision and challenges*, in *2021 Australasian Computer Science Week Multiconference*, ACSW '21, (New York, NY, USA), Association for Computing Machinery, 2021.

[114] P. K. Gadepalli, G. Peach, L. Cherkasova, R. Aitken, and G. Parmer, *Challenges and opportunities for efficient serverless computing at the edge*, in *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, pp. 261–2615, 2019.

[115] A. Hall and U. Ramachandran, *An execution model for serverless functions at the edge*, in *Proceedings of the International Conference on Internet of Things Design and Implementation*, IoTDI '19, (New York, NY, USA), p. 225–236, Association for Computing Machinery, 2019.

[116] P. K. Gadepalli, S. McBride, G. Peach, L. Cherkasova, and G. Parmer, *Sledge: A serverless-first, light-weight wasm runtime for the edge*, in *Proceedings of the 21st International Middleware Conference*, Middleware '20, (New York, NY, USA), p. 265–279, Association for Computing Machinery, 2020.

[117] E. Paraskevoulakou and D. Kyriazis, *Leveraging the serverless paradigm for realizing machine learning pipelines across the edge-cloud continuum*, in *2021 24th Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, pp. 110–117, 2021.

[118] I. Wang, E. Liri, and K. K. Ramakrishnan, *Supporting iot applications with serverless edge clouds*, in *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*, pp. 1–4, 2020.

[119] S. Nastic, T. Rausch, O. Scekic, S. Dustdar, M. Gusev, B. Koteska, M. Kostoska, B. Jakimovski, S. Ristov, and R. Prodan, *A serverless real-time data analytics platform for edge computing*, *IEEE Internet Computing* **21** (2017), no. 4 64–71.

[120] E. Al-Masri, I. Diabate, R. Jain, M. H. L. Lam, and S. R. Nathala, *A serverless iot architecture for smart waste management systems*, in *2018 IEEE International Conference on Industrial Internet (ICII)*, pp. 179–180, 2018.

[121] L. Baresi, D. F. Mendonça, M. Garriga, S. Guinea, and G. Quattrocchi, *A unified model for the mobile-edge-cloud continuum*, *ACM Trans. Internet Technol.* **19** (Apr., 2019).

[122] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. M. Rabbah, A. Slominski, and P. Suter, *Serverless computing: Current trends and open problems*, *CoRR* (2017).

[123] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, *Serverless computing: One step forward, two steps back*, in *Conference on Innovative Data Systems Research (CIDR)*, 2019.

[124] I. Baldini, P. Cheng, S. J. Fink, N. Mitchell, V. Muthusamy, R. Rabbah, P. Suter, and O. Tardieu, *The serverless trilemma: Function composition for serverless computing*, in *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2017, (New York, NY, USA), p. 89–103, Association for Computing Machinery, 2017.

[125] S. Ginzburg and M. J. Freedman, *Serverless isn't server-less: Measuring and exploiting resource variability on cloud faas platforms*, in *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*, WoSC'20, (New York, NY, USA), p. 43–48, Association for Computing Machinery, 2020.

[126] A. Jangda, D. Pinckney, Y. Brun, and A. Guha, *Formal foundations of serverless computing*, *Proc. ACM Program. Lang.* **3** (Oct., 2019).

[127] D. Bardsley, L. Ryan, and J. Howard, *Serverless performance and optimization strategies*, in *2018 IEEE International Conference on Smart Cloud (SmartCloud)*, pp. 19–26, 2018.

[128] P. Vahidinia, B. Farahani, and F. S. Aliee, *Cold start in serverless computing: Current trends and mitigation strategies*, in *2020 International Conference on Omni-layer Intelligent Systems (COINS)*, pp. 1–7, 2020.

[129] M. Sewak and S. Singh, *Winning in the era of serverless computing and function as a service*, in *2018 3rd International Conference for Convergence in Technology (I2CT)*, pp. 1–5, 2018.

[130] R. Kotla, L. Alvisi, and M. Dahlin, *Safestore: A durable and practical storage system*, in *USENIX Annual Technical Conference*, pp. 129–142, 2007.

[131] F. Nawab, V. Arora, D. Agrawal, and A. El Abbadi, *Chariots: A scalable shared log for data management in multi-datacenter cloud environments.*, in *EDBT*, pp. 13–24, 2015.

[132] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobbler, M. Wei, and J. D. Davis, *Corfu: A shared log design for flash clusters*, in *USENIX NSDI*, 2012.

[133] E. A. Brewer, *Towards robust distributed systems (abstract)*, in *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, 2000.

[134] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, *Making data structures persistent*, *J. Comput. Syst. Sci.* **38** (Feb., 1989).

[135] D. Meissner, B. Erb, F. Kargl, and M. Tichy, *Retro-lambda: An event-sourced platform for serverless applications with retroactive computing support*, in *Intl. Conf. on Distributed and Event-based Systems*, 2018.

[136] "The linux mmap system call." `http://man7.org/linux/man-pages/man2/mmap.2.html`, 2018. [Accessed electronically, March 2018].

[137] "ZeroMQ - Distributed Messaging." "http://zeromq.org/" [Online; accessed 12-Sep-2017].

[138] "Espressif 8266 Microcontroller." "https://www.espressif.com/en/products/hardware/esp8266ex/overview" [Online; accessed 12-Sep-2018].

[139] "Raspberry pi zero." `https://www.raspberrypi.org/products/raspberry-pi-zero/`, 2018. [Accessed electronically, March 2018].

[140] "Intel NUC." `http://www.intel.com/content/www/us/en/nuc/overview.html`. [Online; accessed 12-Sep-2017].

[141] "NTP: The Network Time Protocol." `http://www.ntp.org/`. [Online; accessed 7-May-2021].

[142] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, *Dynamo: Amazon's Highly Available Key-Value Store*, in *Symposium on Operating System Principles*, 2007.

[143] J. Manner, M. Endreß, T. Heckel, and G. Wirtz, *Cold start influencing factors in function as a service*, in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pp. 181–188, 2018.

[144] R. van Renesse and F. B. Schneider, *Chain replication for supporting high throughput and availability*, in *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, (USA), p. 7, USENIX Association, 2004.

[145] "Apache Storm." `http://storm.apache.org/`. [Online; accessed 21-January-2016].

[146] "Apache Spark." `https://spark.apache.org/` [Online; accessed 14-Feb-2015].

[147] "Apache Flink." `https://flink.apache.org/`. [Online; accessed 11-Mar-2021].

[148] "Function as a Service." `https://en.wikipedia.org/wiki/Function_as_a_Service` [Online; accessed 12-Sep-2017].

[149] R. Wolski, C. Krintz, F. Bakir, G. George, and W.-T. Lin, *CSPOT: Portable, Multi-scale Functions-as-a-Service for IoT*, in *ACM Symposium on Edge Computing*, 2019.

[150] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, *Chord: A scalable peer-to-peer lookup service for internet applications*, in *Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2001.

[151] "SHA-1." `https://en.wikipedia.org/wiki/SHA-1`. [Online; accessed 12-Mar-2021].

[152] "Aristotle Cloud Federation." `https://federatedcloud.org` [Online; accessed 12-Sep-2017].

[153] "iPerf." `https://iperf.fr/`. [Online; accessed 8-Apr-2021].

[154] "AWS X-ray." "https://aws.amazon.com/xray/" [Online; accessed 12-Sep-2017].

[155] L. Lamport, *Time, clocks, and the ordering of events in a distributed system*, *Commun. ACM* **21** (July, 1978) 558–565.

[156] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto, *Causal memory: definitions, implementation, and programming*, *Distributed Computing* **9** (1995), no. 1 37–49.

[157] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, *Dapper, a large-scale distributed systems tracing infrastructure*, tech. rep., Google, Inc., 2010.

[158] "Serverless Framework." `https://serverless.com/`. [Online; accessed 7-May-2021].

[159] "Docker-lambda." `https://github.com/lambci/docker-lambda`. [Online; accessed 7-May-2021].

[160] J. Spillner, *Snafu: Function-as-a-service (faas) runtime design and implementation*, *CoRR* **abs/1703.07562** (2017).

[161] "New Relic Monitoring for AWS Lambda." https://newrelic.com/blog/best-practices/monitor-aws-lambda. [Online; accessed 7-May-2021].

[162] "Dashbird." https://dashbird.io/. [Online; accessed 7-May-2021].

[163] "Amazon CloudWatch." https://aws.amazon.com/cloudwatch/. [Online; accessed 7-May-2021].

[164] "Zipkin." http://zipkin.io/. [Online; accessed 7-May-2021].

[165] R. Schwarz and F. Mattern, *Detecting causal relationships in distributed computations: In search of the holy grail*, *Distrib. Comput.* **7** (Mar., 1994) 149–174.

[166] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, *The potential dangers of causal consistency and an explicit solution*, in *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, (New York, NY, USA), Association for Computing Machinery, 2012.

[167] W. Lloyd, M. Freedman, M. Kaminsky, and D. Andersen, *Don't settle for eventual: Scalable causal consistency for wide-area storage with cops*, in *ACM Symposium on Operating Systems Principles*, 2011.

[168] P. Bailis, A. Ghodsi, J. Hellerstein, and I. Stoica, *Bolt-on causal consistency*, in *ACM SIGMOD International Conference on Management of Data*, 2013.

[169] M. Bravo, L. Rodrigues, and P. V. Roy, *Saturn: A distributed metadata service for causal consistency*, in *European Conference on Computer Systems*, 2017.

[170] V. Budilov, "Use Amazon Rekognition to Build an End-to-End Serverless Photo Recognition System." "https://aws.amazon.com/blogs/ai/use-amazon-rekognition-to-build-an-end-to-end-serverless-photo-recognition-system/" Accessed 15-Sep-2017.

[171] Wikipedia, "Monkey Patch." "https://en.wikipedia.org/wiki/Monkey_patch" Accessed 15-Sep-2017.

[172] "AWS SDK for Python (Boto3)." https://aws.amazon.com/sdk-for-python/. [Online; accessed 7-May-2021].

[173] "Fleece." https://github.com/rackerlabs/fleece. [Online; accessed 7-May-2021].

[174] "Graphviz - Graph Visualization Software." http://www.graphviz.org. [Online; accessed 7-May-2021].

[175] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, *A comparison of approaches to large-scale data analysis*, in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, (New York, NY, USA), p. 165–178, Association for Computing Machinery, 2009.

[176] "Ad Hoc Big Data Processing Made Simple with Serverless MapReduce." `https://aws.amazon.com/blogs/compute/ad-hoc-big-data-processing-made-simple-with-serverless-mapreduce/`. [Online; accessed 7-May-2021].

[177] "AWS X-Ray Segment Documents." `http://docs.aws.amazon.com/xray/latest/devguide/xray-api-segmentdocuments.html`. [Online; accessed 7-May-2021].

[178] "Student's T-Test." `https://en.wikipedia.org/wiki/Student's_t-test`. [Online; accessed 7-May-2021].

[179] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh, *Megastore: Providing scalable, highly available storage for interactive services*, in *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pp. 223–234, 2011.

[180] R. Fonseca, G. Porter, R. Katz, S. Shenker, and I. Stoica, *X-trace: A pervasive network tracing framework*, in *USENIX NSDI*, 2007.

[181] R. Escriva, A. Dubey, B. Wong, and E. Sirer, *Kronos: The design and implementation of an event ordering service*, in *Eurosys*, 2014.

[182] H. T. Vo, S. Wang, D. Agrawal, G. Chen, and B. C. Ooi, *Logbase: a scalable log-structured database system in the cloud*, *VLDB* **5** (2012), no. 10.

[183] "gRPC." `https://grpc.io/`. [Online; accessed 7-May-2021].

[184] J. Mace, R. Roelke, and R. Fonseca, *Pivot tracing: Dynamic causal monitoring for distributed systems*, *ACM Trans. Comput. Syst.* **35** (Dec., 2018).

[185] D. Romano and M. Pinzger, *Using vector clocks to monitor dependencies among services at runtime*, in *QASBA*, 2011.

[186] P. Alvaro, S. Galwani, and P. Bailis, *Research for practice: Tracing and debugging distributed systems; programming by examples*, in *CACM*, Jan., 2017.

[187] I. Beschastnikh, P. Wang, Y. Brun, and M. Ernst, *Debugging distributed systems*, in *CACM*, June, 2016.

[188] I. Beschastnikh, Y. Brun, M. D. Ernst, A. Krishnamurthy, and T. E. Anderson, *Mining temporal invariants from partially ordered logs*, *SIGOPS Oper. Syst. Rev.* **45** (Jan., 2012).

[189] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica, *Friday: Global comprehension for distributed replay*, in *NSDI*, 2007.

[190] X. Liu, W. Lin, A. Pan, and Z. Zhang, *Wids checker: Combating bugs in distributed systems*, in *NSDI*, 2007.

[191] D. Meissner, B. Erb, F. Kargl, and M. Tichy, *Retro-lambda: An event-sourced platform for serverless applications with retroactive computing support*, in *Intl. Conf. on Distributed and Event-based Systems*, 2018.

[192] D. Geels, G. Altekar, S. Shenker, and I. Stoica, *Replay debugging for distributed applications*, in *ATC*, 2006.

[193] "CIMIS Weather Station Data." `https://data.cnra.ca.gov/dataset/cimis-weather-station-data`. [Online; accessed 7-May-2021].

[194] P. Hudak, *Conception, evolution, and application of functional programming languages*, *ACM Comput. Surv.* **21** (Sept., 1989) 359–411.

[195] K. Hinsen, *The promises of functional programming*, *Computing in Science Engineering* **11** (July, 2009) 86–90.

[196] L. Karczmarczuk, *Scientific computation and functional programming*, *Computing in Science Engineering* **1** (May, 1999) 64–72.

[197] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, *Efficiently computing static single assignment form and the control dependence graph*, *ACM Trans. Program. Lang. Syst.* **13** (Oct., 1991) 451–490.

[198] P. Briggs, K. D. Cooper, T. J. Harvey, and L. T. Simpson, *Practical improvements to the construction and destruction of static single assignment form*, *Softw. Pract. Exper.* **28** (July, 1998) 859–881.

[199] A. W. Appel, *Ssa is functional programming*, *SIGPLAN Not.* **33** (Apr., 1998).