RESOURCE MANAGEMENT ALGORITHMS FOR EDGE-BASED, LATENCY-AWARE DATA DISTRIBUTION AND PROCESSING

By

Shweta Prabhat Khare

Dissertation

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

February 29, 2020

Nashville, Tennessee

Approved:

Dr. Aniruddha Gokhale

Dr. Hongyang Sun

Dr. Kaiwen Zhang

Dr. Julien Gascon-Samson

Dr. Xenofon Koutsoukos

Dr. Akos Ledeczi

To my respected parents Dr. Kalpana Khare and Dr. Prabhat Khare,

my dear brother Ankit Khare,

and

my beloved husband Nishant Ranjan Das

ACKNOWLEDGMENTS

Pursuing my PhD has been a truly life-changing experience and I have many people to thank for their constant support, guidance and kindness without which this journey would not have been possible. First and foremost, I would like to express my heart-felt gratitude towards my advisor, Dr. Aniruddha Gokhale for his invaluable time, guidance and encouragement throughout my program. Dr. Gokhale believed in me and provided me with every opportunity to succeed. He was always available for research discussions and his guidance was paramount in shaping my research work. His humbleness, empathy and approachability made me feel very supported and I am very thankful for his graciousness. I thank my committee members Dr. Hongyang Sun, Dr. Kaiwen Zhang, Dr. Julien Gascon-Samson, Dr. Xenofon Koutsoukos and Dr. Akos Ledeczi for their time and advice. I feel very grateful for the weekly skype sessions with Dr. Zhang and Dr. Julien. Their time investment, input and guidance was tremendously helpful in evolving research ideas into concrete work. I thank Dr. Sun for the many hours he invested in discussing the problem formulation with me. I am inspired by his brilliance, expertise in optimization theory and his willingness to engage in a variety of different research problems. I am also very thankful to Dr. Koutsoukos for insightful discussions and incisive critique which helped in making our research work stronger. I thank Dr. Sumant Tambe for his collaboration, motivation and guidance in publishing my very first paper. Dr. Tambes superior command on programming languages and software engineering principles and his zeal for continuous learning is truly inspiring. I relied on him time and again for input during my studies and I thank him for his help. I thank my friends Anirban Bhattacharjee, Robert Canady, Shunxing Bao, Travis Brummett, Yogesh Barve, Zhuangwei Kang, and Ziran Min for their collaboration and encouragement. I thank Dr. Faruk Caglar, Dr. Kyoungho An, Dr. Shashank Shekhar, Dr. Prithviraj Patil, and Dr. Subhav Pradhan for their mentorship. Last but not the least, I thank my family: my parents Dr. Kalpana Khare and Dr. Prabhat Khare, my brother Ankit Khare, my uncle

Dr. Dinesh Verma, my brother-in-law Kaushik Ranjan, my parents-in-law Nina Ranjan and Prem Ranjan, and the rest of my family for their unwavering support. I especially thank my loving and supportive husband Nishant Ranjan for his encouragement, emotional presence, stability and support. Finally, I thank National Science Foundation (NSF) and Air Force Office of Scientific Research (AFOSR) for their financial support without which this research work would not have been possible.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Chapter 1

INTRODUCTION

1.1 Emerging Trends

The Internet of Things (IoT) [1] paradigm has enabled a large number of physical devices or "things" equipped with sensors and actuators to get connected over the Internet to exchange information. According to Gartner [2], $\sim$20 billion devices will be connected to the Internet by 2020. In many IoT applications, the sensor data produced by these devices is continuously processed in an online/streaming manner to gain insights, which are subsequently distributed to interested endpoints for intelligent control and actuation of the system. Often, both the data distribution and information processing needs to happen in near real time to meet the latency requirements of IoT applications. For example, SmartDriver [3] is a real-time personalized recommendation service deployed in the city of Seville in Spain to improve driving efficiency and road safety. Here, each driver publishes their biometric information, such as heart-rate and vehicle information every 10 seconds to the SmartDriver server. The SmartDriver processes this information to monitor each driver's stress level, and to provide personalized recommendations for safe and fuel-efficient driving. In addition to sending their own vehicle information, drivers also subscribe to receive information about all nearby drivers (within 100 meters) to assess traffic conditions. Given the time- and location-sensitivity of the provided information, IoT applications such as SmartDriver impose strict response time requirements on both: **data distribution and processing**.

To meet the data distribution needs of IoT applications, the publish/subscribe (pub/-sub) [4] communication paradigm is typically used [5] since it supports scalable, asynchronous and anonymous many-to-many communication between publishers (i.e., data producers) and subscribers (i.e., data consumers). In a pub/sub system, subscribers specify

their interest in the form of subscriptions whereas publishers send data publications. The underlying pub/sub system matches subscriptions and publications to distribute data to all interested subscribers. Many different types of pub/sub systems exist that can be classified based on the expressiveness of subscriptions [4]. Of these, the topic-based pub/sub is the most commonly used type due to its simplicity and ease of use. In topic-based pub/sub, a topic represents a logical channel of communication which is uniquely identified by its name. Subscribers specify their subscriptions for specific topic names and publishers tag their publications with topic names. A topic-based pub/sub system dispatches all messages sent on a topic to all interested subscribers. MQTT [6], Kafka [7], ActiveMQ [8], OMG Data Distributed Service (DDS) [9] etc. are examples of widely used topic-based pub/sub systems.

To meet the online, stream processing needs of IoT applications, normally the Distributed Stream Processing Systems (DSPS) are used [10]. In a DSPS, an application is structured as a Directed Acyclic Graph (DAG) composed of vertices, which represent operators that process incoming data, and directed edges, which represent the flow of data between operators. The operators can perform any user-defined computation. Apache Storm [11], Apache Spark [12], Apache Flink [13], Millwheel [14], etc. are examples of widely used cloud/cluster-based DSPSs.

Despite the availability and maturation of these technologies, employing existing cloud-based pub/sub and DSPS solutions to support the data distribution and processing needs of IoT applications will entail sending a huge amount of data produced by devices present at the network edge to data centers located at the core of the network [15]. For example, a shared ride safety application that analyzes in-vehicle video streams can easily require uploading in the order of petabytes of data each day [16]. A concrete example illustrates this challenge. For example, according to the statistics provided in [17], in 2017 around 45,787 Uber rides took place every minute. Assuming an average duration of 20 minutes for a ride, and 100MB size for a one minute video, roughly $\sim$9PB of data would need to

be uploaded each day [16].

Sending data over bandwidth-constrained WAN links can incur large unpredictable latencies and a heavy cost. Therefore, a cloud-only approach is not a viable solution to realize large-scale, latency-sensitive IoT applications [18]. To address this limitation, the edge computing [19] paradigm has been proposed, which allows computations to take place near the source of data on low-cost edge devices and small-scale data centers called cloudlets [20]. This eliminates the need to send all the data to the cloud; instead data can be processed near the source and local results can be disseminated quickly while only relevant information is sent to the remote cloud back-end. Edge-based deployment of several interactive applications have shown latency and network cost benefits [21, 22].

Despite these advances, a straightforward deployment of a cloud-based pub/sub and DSPS solution at the edge to meet the data distribution and processing requirements of latency-sensitive IoT applications does not work due to the following reasons:

1. The edge comprises resources that are fairly constrained in their resource capacities. Existing cloud-based pub/sub and DSPS solutions have been designed for a resource-rich environment and hence cannot be directly applied to the edge.

2. In order to support latency-sensitive IoT applications, the solutions must be designed to be **latency-aware**. The computation overhead of running a task on a resource-constrained device and the impact of interference due to co-location of tasks on the same resource-constrained device can quickly eclipse the network latency benefits of using the edge. This trade-off between computation overhead and network cost must be expressly accounted for in such solutions.

Developing an **edge-based, latency-aware data distribution and processing** solution for IoT applications is the objective of this proposed research. To that end, we first elaborate on the key research challenges that must be addressed (Section 1.2), followed by a brief description of our proposed solution to address each of these challenges (Section 1.3). Each

solution approach is then described in detail as a self-contained chapter in this dissertation as outlined in Section 1.4.

1.2 Key Research Challenges

The key research challenges addressed by this research in developing an edge-based, latency-aware data distribution and processing solution are described as below:

1.2.1 Challenge-1: Latency-Aware Data Distribution and Processing at the Edge

None of the existing, widely used topic-based pub/sub systems, such as Kafka [7], MQTT [6], ActiveMQ [8], RabbitMQ [23], DDS [9], etc. provide any Quality-of-Service (QoS) assurance on end-to-end latency of data delivery and processing all at once. Providing latency QoS assurance is critical for latency-sensitive IoT applications. Increasingly, pub/sub solutions deployed at the edge also support light-weight processing on incoming topic data streams. This allows some local data processing close to the source while only aggregated/anonymized data is sent to the cloud [24, 25, 26, 27, 28, 29]. These systems are referred to as publish-process-subscribe systems [24]. The end-to-end latency in a publish-process-subscribe system will be determined primarily by the processing load at the broker in addition to network transmission delay.

A large number of factors can affect a topic's observed end-to-end latency in a publish-process-subscribe system, which includes number of subscribers connected to the topic, number of publishers connected to a topic, a topic's cumulative publishing rate, duration of per-sample processing performed for each data sample received over a topic, impact of processing load of other co-located topics hosted at the same broker, network link characteristics, etc. A publish-process-subscribe system that provides assurance on end-to-end latency on a per-topic basis must consider all these factors while making its topic placement decisions, i.e., topics must be placed together on pub/sub brokers in a latency-aware manner such that none of the topics in the system violate their desired latency QoS. Addi-

tionally, due to constrained and fluctuating resource availability at the edge, the placement solution must be resource-efficient and make use of minimal number of pub/sub brokers.

1.2.2 Challenge-2: Realizing Effective Abstractions for Edge-Based DAG Stream Processing

Widely used DSPS such as Storm [11], Spark [12], Flink [13], etc., have been designed and optimized to run within a single data center. These systems have a master-worker architecture [30], where a master node distributes computations over a cluster of worker nodes for large-scale processing. However, with the advent of edge computing, stream-based processing of data will span across the entire geo-distributed resource spectrum from edge to cloud. Therefore, existing cloud-based DSPS solutions are not directly applicable [26, 31, 32] in an edge computing environment.

Geo-distributed, stream processing application can instead be viewed as a **distributed dataflow** [33, 34, 35], where both data distribution and processing occur seamlessly. This distributed dataflow assumes a DAG structure, where vertices represent individual processing stages (edge/cloud nodes) and directed edges represent the flow of data between these processing stages. Although the pub/sub pattern addresses the data distribution needs for such a distributed dataflow, the data processing aspects which are local to the individual stages are often not implemented as a dataflow. Due to a lack of generality and composability of Application Programming Interface (API) of pub/sub systems, local processing aspects are developed using the Observer pattern. The observer pattern is known to have many limitations [36] such as: (1) inversion of control, (2) non-composability of callbacks, (3) manual state management, and (4) unpredictability in the order of arrival of callbacks. Thus, there is a need for an intuitive programming model for pub/sub systems which preserves this end-to-end dataflow structure. This programming model for pub/sub systems must provide first-class abstraction for data streams, be composable and support reusable coordination primitives for joining, splitting and operating on data streams.

1.2.3 Challenge-3: Latency-Aware DAG Stream Processing at the Edge

To support latency-sensitive, stream-based IoT applications deployed at the edge, it is necessary that constituent operators of the application DAG are placed over resource-constrained edge devices intelligently in a manner that is both resource aware and satisfies the latency needs of the applications. Accordingly, a desirable operator placement is one that minimizes the end-to-end response time or makespan of the application DAG by intelligently trading off communication costs incurred due to distributed placement of operators across edge devices, and interference costs incurred due to co-location of operators on the same edge device [37].

Framework-specific solutions for operator placement [38, 39, 40] have been designed specifically for a given DSPS, such as storm [11]. As such, these solutions are not applicable at the edge. Existing framework-agnostic placement solutions [41, 42, 43, 37, 44] make simplifying assumptions about the interference cost of co-located operators. These solutions do not consider the impact of DAG structure-imposed execution semantics and the incoming data rate on DAG's response time. Due to these simplifying assumptions, their estimation of response time is less accurate and placement produced on the basis of this response time estimation is less effective.

Estimation of response times for arbitrary DAG structures is a hard problem. A large number of factors, such as DAG structure imposed execution semantics, processing interval of constituent operators, incoming data rate, impact of operator co-location and network link characteristics must be considered to make an accurate estimate of DAG's response time under a given placement. Moreover, there can be a very large number of possible placements for a given DAG. Searching through the entire space of possible placements to find one that minimizes the response time will be prohibitively expensive. Therefore, an efficient solution is needed which makes operator placement decisions for makespan minimization on the basis of accurate response time estimation.

1.2.4 Challenge-4: Application of Research Ideas to a Real-World Edge Use-Case

Contributions made to address research challenges 1.2.1, 1.2.2 and 1.2.3 have been presented in a stand-alone fashion, implemented using different technologies and tested on different hardware resources. The solution contribution for Challenge 1.2.3 has only been validated using synthetic application DAGs for single core edge devices. To overcome these limitations, there a need to demonstrate the applicability and effectiveness of presented research solutions in the context of a real-world, edge-based application. Additionally, solution for Challenge 1.2.3 must also be tested on multi-core edge devices to validate its generality.

1.3 Dissertation Research: Latency-Aware Data Distribution and Processing at the Edge

To meet the low-latency data distribution and processing needs of IoT applications deployed at the edge, solutions must be designed in a latency-aware manner. In this context, this doctoral research has identified four key research challenges as discussed in Section 1.2 and proposes the following solutions to address each challenge:

1.3.1 Contribution-1: Latency QoS Assurance for Topic-Based Pub/Sub

To address research Challenge-1 (Section 1.2.1), in Chapter 2 we have presented a solution to provide QoS specified as the desired per-topic 90th percentile latency for a publish-process-subscribe system. To incorporate the impact of various pub/sub features on a topic's 90th percentile latency, such as number of publishers, number of subscribers, cumulative publishing rate, impact of co-located topic load, etc., our solution first learns a latency prediction model for a publish-process-subscribe broker. Subsequently, the learned model is used to inform the placement of topics on brokers such that latency QoS requirement for all topics is met, while making efficient use of constrained system resources.

1.3.2 Contribution-2: Programming Model to Unify Data Distribution and Processing

To address research challenge-2 (Section 1.2.2), in Chapter 3 we have presented a solution which combines pub/sub-based data distribution with reactive programming to preserve the end-to-end distributed dataflow structure. Reactive programming provides a dedicated abstraction for data streams and a reusable set of stream processing operators. Thus, this integration allows even the local processing stages at edge/cloud sites to be implemented as a dataflow.

1.3.3 Contribution-3: Latency-Aware DAG Placement

To address research challenge-3 (Section 1.2.3), in Chapter 4 we have presented a solution which learns a data-driven latency prediction model that incorporates DAG-based execution semantics, incoming data rates and operator processing times to estimate the latency of all paths in a DAG. This latency prediction model is subsequently used by a heuristic-based solution to find a placement which minimizes the makespan of the DAG.

Learning a latency prediction model which predicts the latency of all paths in arbitrary DAG structures is complex. To reduce the model training cost, our solution learns a model to predict the latency of multiple co-located linear chains instead. Then, given any arbitrary DAG, our solution first linearizes the DAG into an equivalent set of linear chains only for the purposes of making placement decisions, and uses the latency prediction model for co-located linear chains to approximate the latency of all paths in the original DAG structure but deploys the original DAG according to the placement decisions.

1.3.4 Contribution-4: Bringing It All Together

To address research challenge-4 (Section 1.2.4), in Chapter 5 we have demonstrated the application of presented research solutions in the context of a real-world edge-based application for Automatic License Plate Recognition (ALPR). ALPR continuously processes

video streams to identify vehicle license plate numbers and can be used in a wide range of applications such as parking automation, ticket-less parking fee management, road toll collection, traffic surveillance, etc. Sending large volumes of video data to the cloud for processing can become prohibitively expensive, which makes ALPR a good edge-based application use-case for validating our proposed research solutions. ALPR was implemented using our unified programming model presented in Chapter 3 and our latency-aware solution for DAG placement presented in Chapter 4 was used to place ALPR's application DAG on a cluster of quad-core Raspberry Pi devices, which also helps to demonstrate the solution's applicability to multi-core hardware platforms.

1.4 Dissertation Organization

This research dissertation is organized as follows: Chapter 2 describes our solution for latency-aware data distribution at the edge; Chapter 3 describes our solution for end-to-end distributed dataflow programming; Chapter 4 describes our solution for latency-aware placement of DAG operators at the edge; and Chapter 5 presents our proposed solution for reliable edge stream processing. Finally, Chapter 6 summarizes the research contributions made by this dissertation.

Chapter 2

Scalable Edge Computing for Low Latency Data Dissemination in Topic-based Publish/Subscribe

2.1 Introduction

The Internet of Things (IoT) is a paradigm in which a plethora of devices across many domains are interconnected to provide and exchange data. Over the last few years, the IoT landscape has grown tremendously, with some studies estimating the number of connected devices to be in the range of tens of billions [45]. In many IoT applications, large amounts of data are produced by sensors deployed at scale, and rapidly consumed in order to provide fast decision-making. This is notably the case in many Smart City applications [46] in which data acquired from many sensors must be rapidly processed in an online/streaming manner to provide low-latency results for closed-loop actuation. Therefore, a scalable and low-latency solution for both data dissemination and in-network processing is needed for IoT applications.

The Publish/Subscribe (pub/sub) [4] communication pattern is considered highly suitable for the data dissemination needs of IoT applications [5], since it provides scalable, asynchronous and anonymous many-to-many communication between data producers (publishers) and data consumers (subscribers). In pub/sub systems, subscribers specify their interests in receiving data in the form of subscriptions. Publishers transmit publication messages that are disseminated by the underlying system to the relevant subscribers. The literature distinguishes between many flavors of pub/sub; the most prevalent being topic-based pub/sub, which is the subject of this research proposal. In topic-based pub/sub, subscriptions are expressed over *topics*, which can be used to model communication channels. Topics are specified by topic names, subscribers declare their subscriptions to specific topic names, and publishers tag their messages with topic names. The topic-based pub/sub

system dispatches all the messages published on a topic to all interested subscribers.

In addition to data dissemination, IoT applications also require real-time processing of streaming data. Traditionally, data produced at the network edge is sent to the cloud for processing. However, this approach can consume very high bandwidth and incur unpredictable and large latencies. Therefore, cloud-based processing and dissemination is not the best choice for latency-critical IoT applications [18]. Recently, edge [19, 20, 21], fog [47] and mobile-cloud computing [48] models have been proposed to address these concerns and support execution of computations near the source of data on low-cost edge devices and small-scale datacenters called cloudlets [20].

Edge computing, combined with the pub/sub communication model, which together is referred to as the *publish-process-subscribe* [24] paradigm, provides a promising approach for enabling low-latency data distribution and processing for IoT systems. In this model, computations take place on published streams of data directly at the pub/sub brokers deployed near the edge. This approach has several advantages: (1) results can be disseminated to local subscribers quickly; (2) only aggregated results are sent to the cloud backend to reduce bandwidth consumption; and (3) data can be anonymized before being sent to the cloud for privacy.

Although pub/sub brokers that perform streaming analytics are increasingly being used to enable latency-critical IoT applications, existing solutions seldom provide any Quality-of-Service (QoS) assurance on latencies experienced by the system. Providing some measure of response time assurance is imperative for the practical utility of many IoT applications. To address these concerns, in this chapter, we present a solution to provide QoS specified as the desired per-topic 90th percentile latency for a publish-process-subscribe system. Per-topic 90th percentile latency QoS implies that 90% of the messages received by all subscribers for a topic will have latencies below the specified QoS value. To ensure more reliable system performance, we use QoS specified as the 90th percentile latency as opposed to average latency [49]. Our solution first learns a latency prediction model of

the publish-process-subscribe broker, and subsequently uses the learned model to determine the number of edge-based pub/sub brokers needed, as well as the placement of topics on these brokers, so that the QoS requirement is met, while making efficient use of the constrained system resources.

In this regard, we makes the following key contributions:

- **Sensitivity analysis:** We present a sensitivity analysis of the impact of different pub/sub features including number of subscribers, number of publishers, publishing rate and per-sample processing interval, on a topic's 90th percentile latency, both in an isolated case where no other topics are hosted at the broker and in a co-located case where other topics are simultaneously hosted at the broker.

- **Latency prediction model:** We present a model for predicting a topic's 90th percentile latency based on its publishing rate, per-sample processing interval, as well as a characterization of the background load imposed by other co-located topics on a broker. Neural network regression is used to learn a separate model for hosting a different number of co-located topics on a broker up to a maximum *degree of co-location* k. The learned models are demonstrated to have $\sim$97% accuracy and experimental results show that only up to $\sim$10% of the messages in the system are not able to meet the desired latency QoS as a result of prediction error and subsequent incorrect topic placement.

- **Topic co-location heuristics:** We formulate a *k-Topic Co-location Problem* (k-TCP) of finding a resource-efficient co-location scheme for a collection of topics on brokers such that their desired QoS in terms of 90th percentile latency is not violated. Here, the degree of co-location k specifies the maximum number of topics that can be hosted by any broker. We show that k-TCP is NP-hard for $k \geq 3$ and present three heuristics that use the latency prediction model for placement of topics on brokers. The performance of these heuristics is evaluated and compared through extensive

experiments.

The rest of this chapter is organized as follows: Section 2.2 presents related work and compares our solution to some existing pub/sub systems. Section 2.3 gives a formal statement of the problem we are studying. Section 4.3 shows the results of a sensitivity analysis and the learned latency prediction model. Section 4.2.3 presents the complexity analysis of the topic co-location problem and the proposed heuristic-based solutions. Section 4.4 presents experimental results to validate our solutions. Finally, Section 5.5 offers concluding remarks and describes future work.

2.2 Related Work

Based on the expressiveness of subscriptions supported, a pub/sub system can be: (1) Content-based [50], where subscribers specify arbitrary boolean functions on the content of the messages; (2) Attribute-based [51], where subscribers specify predicates over attribute values associated with the messages; or (3) Topic-based [52], where messages are tagged with a topic name and subscribers that are interested in a specific topic receive all messages associated with that topic.

Matching published data with subscriptions for data dissemination occurs over an overlay network of pub/sub brokers. Brokers in a pub/sub system may be organized into a tree-based overlay [53], cluster-based overlay [54], structured/unstructured peer-to-peer overlay [55, 56] or cloud-based overlay [51, 52]. Tree-based, cluster-based and peer-to-peer overlays incur multi-hop routing latencies, lack reconfiguration flexiblity and require maintenance of costly state information. Increasingly, single-hop, topic-based pub/sub systems, such as MQTT [6], ActiveMQ [8] , Amazon IoT [57], are being used for developing IoT applications. These systems comprise a single flat layer of pub/sub brokers that are generally deployed in the cloud. Therefore, in our solution we focus on topic-based, single-hop, pub/sub systems similar to MQTT.

Many well-known and commercially-available, topic-based pub/sub systems, such as

MQTT, Redis [58], Kafka [7] and ActiveMQ have been used to build IoT applications. For example, the SmartSantander [59] IoT testbed uses ActiveMQ to distribute a variety of sensor data. MQTT is used to create a smart parking application [60] and the SmartDriver application described previously uses Kafka.

Very few pub/sub systems provide QoS guarantees [61, 62] on latency, which is much desirable for supporting latency critical IoT applications. IndiQoS [62] reserves network level resources over a peer-to-peer overlay of brokers to ensure QoS of data delivery. However, it is not always practical to make network-level resource reservations. Harmony [63] is a peer-to-peer pub/sub system which continuously monitors link quality and adapts routing paths for low-latency data dissemination. Harmony can also make use of priority-based scheduling of messages if the underlying network supports it. DCRD [64] dynamically switches among next-hop downstream nodes for reliable and time-bound data delivery. Brokers in DCRD maintain a sorted list of next-hop nodes for each subscriber on the basis of expected delay and reliability of delivery via the next-hop node. Although these solutions support QoS for latency of data delivery, they are designed for peer-to-peer, multi-hop networks and are not directly applicable for single-hop, topic-based pub/sub systems like MQTT, ActiveMQ, etc. Moreover, these solutions primarily focus on re-routing paths for data delivery in response to changes in network link characteristics. They do not consider the impact of existing broker load on latency.

With the adoption of edge computing concepts of processing near the source of data, many pub/sub sytems have emerged that implement the publish-process-subscribe [24, 26] pattern and additionally support computation at the pub/sub brokers. Latencies in publish-process-subscribe systems will be affected significantly by processing delays at the broker in addition to network link characteristics. Therefore, managing the load at pub/sub brokers is important for ensuring acceptable performance. Typically, load in topic-based pub/sub systems is managed by placing the topics on multiple brokers and distributing the connected endpoints across these brokers. Kafka supports manual rebalancing of topic

load, while Dynamoth [52] performs this rebalancing dynamically when the empirically set network thresholds are exceeded. However, both Kafka and Dynamoth do not perform load-balancing in a latency aware manner for QoS assurance. MultiPub [65] finds an optimal placement of topics across geographically distributed datacenters for ensuring per-topic 90th percentile latency of data delivery, but it only considers inter-datacenter network latencies and assumes that each datacenter has a local load balancing algorithm. On the contrary, FogMq [66] uses a distributed flocking algorithm to migrate the entire pub/sub broker between edge sites to ensure bounded tail latency of computation.

To addresss the need to balance loads at publish-process-subscribe brokers for providing latency QoS of data delivery, our solution learns a latency model for pub/sub broker load and uses the learned model for distributing the topic load across brokers such that data delivery QoS is provided in a resource efficient manner.

We use a data-driven approach instead of closed-form, analytical solutions, to model the impact of broker load on a topic's latency. Closed-form, analytical solutions, such as queueing models have been used extensively for performance modeling [67] [68][69]. However, we use a data-driven approach since simple queueing models do not incorporate the impact of interference by other co-located topics on a topic's latency and typically assume Poisson arrivals. In IoT deployments, it is more likely that sensors publish information at constant rate. Practical use of queueing models requires us to explicitly measure the processing capacity of the broker per topic. Although it can be indirectly estimated by measuring the number of queued samples per topic, many commercial off-the-shelf pub/-sub libraries do not expose this metric. To the best of our knowledge, a machine-learning based approach for modeling the performance of publish-process-subscribe systems has not been presented before.

2.3 Problem Statement

In this section, we first describe a use case to motivate the need for latency-bounded, edge-based publish-process-subscribe systems (Section 2.3.1). We then present the system model (Section 2.3.2) and assumptions made (Section 2.3.3). Finally, we provide the formal statement of k-Topic Co-location (k-TCP) optimization problem that meets the QoS requirements while making efficient use of the broker resources (Section 2.3.4).

2.3.1 Motivational Use Case

We use the DEBS Grand Challenge dataset [70] on New York taxi trips as our motivational use case — a near real-time, city-wide taxi navigation and dispatch service. The service divides New York into 500m$\times$500m regions and taxis within each region send their location updates on its region's *gps* topic. Additionally, taxis also subscribe to its region's *update* topic to receive processed information such as most profitable regions of operation, traffic and dispatch information. All topics are hosted by a publish-process-subscribe system running on brokers near the edge, called *edge brokers*, inside a small-scale datacenter. The RIoTBench paper [71] has benchmarked some stream processing pipelines built for the New York taxi dataset, such as ETL (Extract Transform Load), prediction, model training and statistical aggregation. While ETL and prediction pipelines take 10-40ms, statistical summarization and model training take $\sim$50 seconds. Model training and statistical summarization are good examples of latency-insensitive processing that can be offloaded to a more resourceful cloud backend, while ETL and prediction can be performed at the edge brokers to provide low-latency inference.

Given the time-sensitive nature of GPS position, traffic and dispatching information [72], we consider the response time requirement for the application to be sub-second, i.e., preprocessing of data on the *gps* topic should happen within one second and the updates published on the *update* topic should also be disseminated to all taxis within one second.

2.3.2 System Model and Notations

We now introduce the system model and notation used in this chapter. Consider a system where the cloud provider operates a set of homogeneous server brokers that are deployed on fog/edge resources. Let $T = \{t_1, t_2, \cdots, t_n\}$ be a collection of n pub/sub topics that need to be allocated on the brokers. Each topic $t_i \in T$ is characterized by several parameters, including the number of publishers, overall publishing rate, per-sample processing interval in the broker, number of subscribers, etc. Since each topic may only occupy a fraction of resources in a broker – the amount of which will be determined by a combination of its parameters described above – multiple topics can be co-located on the same broker for better resource utilization. Co-located topics, however, affect each other's performance [73, 74], thus increasing their end-to-end delays and hence 90th percentile latencies. We allow a maximum of k topics to be co-located, where $k \geq 1$ is a constant parameter that represents the *degree of co-location*. The value of k can be determined empirically by examining the overhead of managing multiple co-located topics as well as the severity of interference in terms of the latency degradation.

Let τ denote the desired 90th percentile latency that should not be exceeded by all topics. Given τ and k, to solve the proposed k-Topic Co-location (k-TCP) problem, we consider the following two sub-problems:

(1) Design a *latency prediction model* for the 90th percentile latencies of up to k co-located topics based on their input parameters.

(2) Find a *topic co-location scheme* to minimize the number of brokers used, which is needed due to the resource-constrained nature of the edge yet ensuring that all topics satisfy the desired 90th percentile latency τ.

2.3.3 Assumptions

Our system model makes the following assumptions:

(1) We only consider the impact of a broker load's on a topic's latency. In practice, a topic's latency will also be influenced by the fluctuating network conditions. We assume constant network latency and bandwidth.

(2) For simplicity of discourse, we assume that all topics have the same latency QoS requirement τ, although our system can support differentiated per-topic QoS requirements.

(3) We assume that the per-sample processing performed at the broker is CPU-bound.

(4) We assume that all edge brokers are homogeneous, i.e., they have the same hardware specification.

2.3.4 K-Topic Co-location Problem (k-TCP)

We present a formal definition of the topic co-location problem. For a collection $T = \{t_1, t_2, \ldots, t_n\}$ of n topics, a degree of co-location k, and a latency bound τ, a topic *co-location scheme* $\mathscr{S} : T \to B$ assigns the topics to a set $B = \{b_1, b_2, \ldots\}$ of edge brokers. The goal is to minimize the number $|B|$ of edge brokers used while ensuring that each topic satisfies the desired latency τ.

Under a particular co-location scheme, let y_j to be a binary variable that indicates whether broker b_j is used, i.e.,

$$y_j = \begin{cases} 1 & \text{if broker } b_j \text{ is used} \\ 0 & \text{otherwise} \end{cases}$$

and let x_{ij} be a binary variable that indicates the assignments of topics to brokers, i.e.,

$$x_{ij} = \begin{cases} 1 & \text{if topic } t_i \text{ is assigned to broker } b_j \\ 0 & \text{otherwise} \end{cases}$$

Also, for each topic $t_i \in T$, let $T_i \subseteq T$ denote its set of co-located topics (including t_i itself) on the same broker, i.e., $T_i = \{t_{i'} \in T | x_{ij} = x_{i'j} = 1\}$, and let $\ell_i(T_i)$ denote its 90th percentile latency, which can be computed by the latency predictive model (Section 4.3). If topic t_i is assigned to a server alone without other co-located topics, we simply use ℓ_i to denote its 90th percentile latency. The following describes a natural property on the latency model.

Property 1. *(a)* $\ell_i \leq \tau$ *for all* $t_i \in T$*; (b)* $\ell_i(T_i'') \leq \ell_i(T_i')$ *if* $T_i'' \subseteq T_i'$ *for all* $t_i \in T$.

In particular, the property states that: (a) each topic always satisfies the latency requirement when assigned alone to a broker[1]; and (b) removing a topic from a set of co-located topics on a broker will not increase the latency for any of the remaining topics.

Now, we formulate the ***k*-Topic Co-location Problem (*k*-TCP)** as the following integer linear program (ILP):

$$\text{Minimize } |B| = \sum_j y_j$$

$$\text{Subject to } \sum_j x_{ij} = 1, \quad \forall t_i \in T \tag{2.1}$$

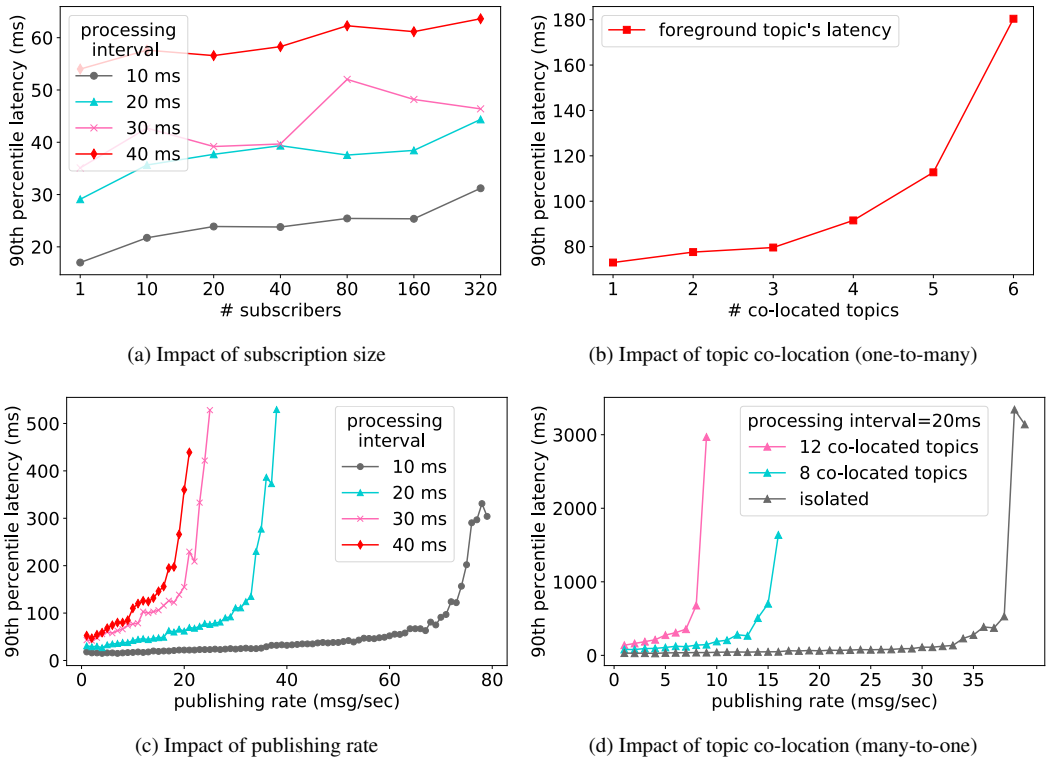
$$\sum_i x_{ij} \leq k, \quad \forall b_j \in B \tag{2.2}$$

$$\ell_i(T_i) \leq \tau, \quad \forall t_i \in T \tag{2.3}$$

$$x_{ij}, y_j \in \{0,1\}, \quad \forall t_i \in T, \forall b_j \in B \tag{2.4}$$

In the above formulation, Constraint (2.1) requires each topic to be assigned to exactly one broker, Constraint (2.2) allows no more than k co-located topics on each broker, Constraint (2.3) ensures the latency satisfiability for all topics, and Constraint (2.4) requires the decision variables to be binary. Section 4.2.3 shows the complexity of k-TCP and presents several heuristic solutions.

[1] Otherwise, the topic must be split into two or more topics, e.g., by splitting publishing rate [52] for any solution to be feasible. The design of topic splitting policies is out of the scope of this work.



(a) Impact of subscription size

(b) Impact of topic co-location (one-to-many)

(c) Impact of publishing rate

(d) Impact of topic co-location (many-to-one)

Figure 2.1: Sensitivity analysis for latency modeling

2.4 Latency Prediction Model and its Sensitivity Analysis

Solutions to the proposed *k*-TCP problem rely on an accurate understanding of the 90th percentile end-to-end latency values per topic under different topic co-location scenarios. To that end, we build a latency prediction model to determine the latency satisfiability of any set of topics to be placed on a broker. However, building such a prediction model first requires a critical understanding of the impact of pub/sub features and topic co-location on per-topic latencies. Therefore, we first conduct a set of sensitivity analysis experiments to study the impact of several pub/sub features, such as number of subscribers or subscription size, publishing rate, per-sample processing interval and background load on a topic's performance. This helps us to identify the dominant pub/sub features that should be used to build the latency prediction model.

Accordingly, we first describe our pub/sub system and the experimental testbed used to conduct the sensitivity analysis experiments in Section 2.4.1, following which Section 2.4.2

presents the sensitivity analysis results. Section 2.4.4 describes our latency prediction model, followed by discussions about its limitations in Section 2.4.5.

2.4.1 Experimental Setup

We implemented our pub/sub system using the Java language binding of the ZMQ [75] sockets library. Our system architecture is similar to Kafka, where topics are hosted on a flat layer of pub/sub brokers managed by Zookeeper [76], which is a centralized service for distributed coordination and state maintenance. Publishers and subscribers connect to the broker that hosts their topics of interest to send and receive data, respectively. We have used the matrix-product CPU stressor provided by the stress-ng [77] tool to emulate variable intervals of per-sample processing performed at the broker in accordance with the publish-process-subscribe paradigm. All experiments are performed for a broker node hardware with four 2.5GHz Intel Xeon E5420 cores, 4GB RAM and 1Gb/sec network capacity. Separate machines were used for hosting the publisher and subscriber endpoints. Network Time Protocol (NTP) [78] was used for time synchronization of all machines. Publishers tag their messages with a time-stamp and subscribers upon reception of the message use this time-stamp to compute the end-to-end latency of data delivery.

2.4.2 Sensitivity Analysis

In our motivational use-case described in Section 2.3.1, IoT workload is either of type one-to-many or of type many-to-one. In the one-to-many type, a single/few publishers send processed data to a large number of subscribers for actuation. For example, in our motivational use case, taxis receive information about the most profitable region of operation on the *update* topic. In the many-to-one type, a large number of publishers send their data to a few subscribers for processing. For example, all taxis in a region send their gps coordinates on the *gps* topic. The publishing rate in the one-to-many case is expected to be low, while the cumulative publishing rate for the many-to-one case is expected to be much higher.

We first study the impact of number of subscribers on a topic's performance for one-to-many type of data dissemination (recall that a topic's performance is characterized by the end-to-end 90th percentile latency experienced by its subscribers). Figure 2.1a shows the impact on latency when we increase the number of subscribers connected to a topic hosted at a broker in isolation (i.e., there are no other topics hosted along with this topic at the broker) for different values of per-sample processing interval. As this is a one-to-many type of workload, we connect a single publisher, which sends messages with a payload size of 4KB at a low rate of 1 message/second. We observe that although latency increases with increasing number of subscribers, the impact on latency is very small, especially if sub-second delivery bounds are considered, such as in our taxi use case. From our analysis of the New York taxi data-set over a five-day period, the maximum number of taxis in a region was found to be 240. Figure 2.1a shows that 240 subscribers/taxis can easily be sustained without incurring a significant performance penalty.

However, a topic will seldom be hosted in isolation at the broker given the resource-constrained nature of edge-based systems. Therefore, we also study how co-located topics can impact the performance of a topic of type one-to-many. We refer to the topic under consideration as the *foreground* topic, while the other co-located topics are referred to as the *background* topics. Figure 2.1b shows how the latency of the foreground topic with 200 connected subscribers and per-sample processing of 50ms is affected as we increase the number of background topics. Here, each background topic is of type many-to-one with 10 connected publishers- each publishing at the rate of one message/second, one connected subscriber and 50ms per-sample processing interval. The number of background topics is increased until CPU utilization at the broker saturates. We see that the latency of the foreground topic increases significantly with increasing load at the broker, but it is still well below the sub-second latency bound despite broker CPU saturation.

We perform similar sensitivity analysis experiments for the many-to-one scenario. Figure 2.1c shows how an isolated topic's latency is impacted as publishing rate or number of

connected publishers increases for different per-sample processing intervals. Here, each topic has one connected subscriber and each publisher publishes at the rate of one message/second. We observe that a topic's latency increases linearly up to a threshold rate after which the increase in latency becomes exponentially large. Beyond the threshold publishing rate, the processing capacity of the broker is exceeded by the incoming rate of messages, which results in large queuing delay at the broker and therefore, an exponential increase in the observed latency [79]. The threshold rate for a given per-sample processing interval decreases due to the impact of background load imposed by other co-located topics as shown in Figure 2.1d. This shows that the threshold rate for a topic with per-sample processing interval of 20ms reduces from 38 messages/second in the isolated case to 15 messages/sec when co-located with 7 other background topics and to 8 messages/second when co-located with 11 other background topics. In these experiments, the background topics were of type many-to-one, with randomly chosen publishing rates and per-sample processing intervals.

2.4.3 Key Insights from Sensitivity Analysis

The sensitivity analysis results show that a topic's latency in a publish-process-subscribe system increases with increasing subscription size, publishing rate, per-sample processing interval and background load. Depending on the broker hardware capacity, the measured values, such as, latency, threshold rate of publication, etc. may be different. However, observed behaviors will remain the same.

The sensitivity analysis experiments show that number of subscribers, publishing rate, per-sample processing interval and background load all impact a topic's latency. For sub-second latency requirements, the results show that even for 200 connected subscribers, the latency for a topic is not significantly impacted despite the broker being saturated. For applications where the number of connected subscribers per topic is much larger than 200, a topic can be replicated and the number of connected subscribers can be distributed [52]

to ensure that the latency QoS is met. Topic partitioning and replication is beyond our solution's scope and we assume that a topic can be safely placed at a broker in isolation. For sub-second latency bounds, as the number of subscribers does not significantly impact latency, we did not include it in the latency prediction model. It is important to note, however, that the number of subscribers may need to modeled for systems with stricter latency requirements. Figure 2.1b shows a 147% increase in latency from 72ms to 180ms as the background load increases for the foreground one-to-many topic with 200 connected subscribers.

2.4.4 Latency Prediction Model

The sensitivity analysis experiments show that a topic's per-sample processing interval and publishing rate prominently impact its latency. Hence, we have considered these two pub/sub features and features derived from them for learning our latency prediction model. More concretely, we used six input features for learning the model, of which the first three characterize the foreground topic and the remaining three characterize the background load.

These input features are described below, where t_f denotes the foreground topic and T_B denotes the set of background topics at a broker:

- p_f, i.e., per-sample processing interval of t_f;
- r_f, i.e., publishing rate of t_f;
- d_f, i.e., foreground load which is the product of per-sample processing interval p_f and publishing rate r_f;
- $\sum_{t_b \in T_B} p_b$, i.e., the sum of per-sample processing intervals of all background topics;
- $\sum_{t_b \in T_B} r_b$, i.e., the sum of publishing rates of all background topics;
- $\sum_{t_b \in T_B} d_b$, i.e., the sum of load (product of per-sample processing interval and publishing rate) of all background topics.

For a topic in isolation, the background load is zero. Therefore, to learn the latency model for a topic in isolation, we have only considered the per-sample processing interval p and publishing rate r of the topic as input features. We found that polynomial regression of degree 4 accurately models the latency curve for a topic in isolation. We describe the isolated topic model accuracy results in more detail in Section 2.6.2.

However, for two or more co-located topics, i.e., $k \geq 2$, we found that simple regression techniques could no longer capture the complex, non-linear impact of the background topics' loads on a foreground topic's latency. Since neural networks generally perform well in capturing non-linear functions of a problem [80], we use it to learn latency models for $k \geq 2$. Neural networks comprise multiple layers, namely, an input layer, one or more intermediate layers called hidden layers and an output layer. Each layer comprises nodes called neurons which are linked to neurons in another layer by weighted connections. The input layer simply feeds the input features of the training data to the network via these weighted connections. Neurons of the hidden and output layer sum the incoming weighted input signals, apply a bias term and an activation function to produce the output for the next layer (in case of a hidden layer) or the output of the network (in case of the output layer). Hidden layers and/or non-linear activation functions are used for modeling the non-linearity of the problem.

The architecture of a neural network, specifically the number of hidden layers, number of neurons in each hidden layer and the regularization factor, greatly impact the performance of the model. If the chosen architecture is too complex, it may result in over-fitting the data and may not generalize to perform well outside of the training data. In this case, training error is very low, but the error on the validation data is high and the model is said to suffer from high variance [81]. On the other hand, if the chosen architecture is too simple, it fails to learn from the data (under-fitting) and performs badly on both the training and validation data. In this case, the model is said to suffer from high bias [81]. Learning curves [82], which plot the training and validation errors as functions of the training data

size can be used to select the right architecture for reducing both the bias and the variance of the model. Specifically, the network architecture for which both training and validation errors converge to a low value is typically chosen.

We learn a separate k-topic co-location model using neural networks for each $k \geq 2$ as opposed to a unified model for reasons of higher accuracy. We plot learning curves for several different neural network architectures for each k and select the one which minimizes both bias and variance. While simpler network architectures perform well for lower values of k, more complex architectures are needed for higher values of k as the search space increases. Section 2.6.2 shows the learning curves and accuracy of the learned models.

2.4.5 Limitations of the Model

It is important to note the limitations of the k-topic co-location models. The learned latency models are specific to a broker hardware type. Therefore, separate latency models need to be learned for each new hardware type. Model learning overhead for different hardware architectures can be reduced by incorporating hardware-specific input features in the learned models so that a single model can be used across different architectures. Transfer learning [83] can also be used for learning the latency models for different broker architectures on the basis of existing models for a specific hardware architecture. Finally, our model also assumes that the per-sample processing performed at each topic is CPU-bound.

The learned latency model is used by our topic placement heuristics for k-TCP (see Section 4.2.3) so that the latency QoS of the topics is not violated on a broker. However, subject to the inaccuracy of the latency prediction model, the produced placement for some topics in the system may be incorrect and may result in QoS violation for those topics. Hence, our approach does not provide hard guarantees on meeting the specified latency QoS. To address this issue, our approach can be augmented with a feedback-based mechanism where subscribers experiencing QoS violations can inform the system, which can

then place this topic on another broker and also use this information to update the learned latency model. Employing a latency model as opposed to relying solely on a subscriber's feedback has the following benefits: 1) QoS violations can be prevented proactively for most of the cases where the latency model makes accurate predictions; and 2) subscriber feedback-based mechanism can incur a large overhead as the system scales.

2.5 NP-Completeness of k-TCP and Heuristics-Based Solutions

In this section, we analyze the computational complexity of k-TCP and show that it is NP-hard for $k \geq 3$, which represents a fairly small degree of co-location. We then propose some heuristics to solve k-TCP sub-optimally.

2.5.1 Feasibility Function

Before analyzing the complexity and proposing a solution for k-TCP, we first rely on the latency prediction model to define a *feasibility function* $\mathscr{F}$, which indicates whether a given set of at most k topics can be feasibly co-located on a broker. Specifically, for any $T^{'} \subseteq T$ and $|T^{'}| \leq k$, we can rely on the k' co-location model for $k' = |T^{'}|$ to predict the latencies for all topics in $T^{'}$, while using the individual parameters for each topic, i.e., the per-sample processing interval p_i and publishing rate r_i, as the input features for the model as described in Section 2.4.4. We define:

$$\mathscr{F}(T^{'}) = \begin{cases} 1 & \text{if } \ell_i(T^{'}) \leq \tau \text{ for all } t_i \in T^{'} \\ 0 & \text{otherwise} \end{cases}$$

Hence, according to Property 1, we have:

$$\mathscr{F}(T^{'}) = 1 \text{ for any } |T^{'}| = 1 \tag{2.5}$$

$$\mathscr{F}(T^{'}) = 1 \text{ implies } \mathscr{F}(T^{''}) = 1 \text{ for any } T^{''} \subseteq T^{'} \tag{2.6}$$

Note that, for a constant degree of co-location k, the set of all possible inputs to the feasibility function $\mathscr{F}$ can be encoded by at most $\sum_{k'=1}^{k} \binom{n}{k'} = O(n^k)$ bits, i.e., polynomial in the number of topics.

2.5.2 Complexity Analysis

We now show the computational complexity of k-TCP.

Theorem 1. *For $k \leq 2$, k-TCP can be solved in polynomial time.*

Proof. The claim is obvious for $k = 1$ (i.e., 1-TCP). In this case, each topic must be assigned to a broker alone, and the number of required brokers is therefore $|B| = |T|$.

For $k = 2$ (i.e., 2-TCP), construct a graph $G = (V, E)$, where $|V| = |T|$ and each vertex $v_i \in V$ represents a topic $t_i \in T$. An edge e_{ij} exists between two vertices v_i and v_j if the corresponding two topics t_i and t_j can be feasibly co-located, i.e., $\mathscr{F}(\{t_i, t_j\}) = 1$. Finding a maximum matching M of G, which can be computed in polynomial time [84], will lead to an optimal solution, where each pair of matched vertices (topics) are co-located on a broker and the unmatched ones are each assigned to a broker alone. The optimal number of required brokers is in this case $|B| = |T| - |M|$. □

Theorem 2. *For $k \geq 3$, k-TCP is NP-hard.*

Proof. We prove the NP-completeness for the decision version of k-TCP, which for a given instance asks whether the collection of topics can be co-located on m or fewer brokers. The problem is clearly in NP: given a co-location scheme, we can verify in polynomial time that it takes at most m brokers and that the set of co-located topics on any broker has a cardinality at most k and form a feasible set (via the feasibility functions).

To show that the problem is NP-complete, we use a reduction from k-Dimensional Matching (k-DM), which is a generalization of the well-known 3-Dimensional Matching (3-DM) problem. For k-DM, we are given k disjoint sets $X_1, X_2, \ldots, X_k$, where all X_j's have the same number m of elements. Let M be a subset of $X_1 \times X_2 \times \cdots \times X_k$, that is, M consists

of k-dimensional vectors $(x_1, x_2, \ldots, x_k)$ such that $x_j \in X_j$ for all $1 \leq j \leq k$. The question is whether M contains a perfect matching $M' \subseteq M$, that is, $|M'| = m$, and for any distinct vectors $(x'_1, x'_2, \ldots, x'_k) \in M'$ and $(x''_1, x''_2, \ldots, x''_k) \in M'$, we have $x'_j \neq x''_j$ for all $1 \leq j \leq k$. It is known k-DM is NP-complete for $k \geq 3$ [85].

Given an instance I_1 of k-DM, we construct an instance I_2 of k-TCP by creating km topics, each corresponding to an element in I_1. For each vector $(x_1, x_2, \ldots, x_k) \in M$ in I_1, we set the corresponding set of topics to be feasible in I_2, i.e., $\mathscr{F}(\{x_1, x_2, \ldots, x_k\}) = 1$, and derive the other feasible sets using Equations (2.5) and (2.6) while leaving all the remaining sets to be infeasible. Finally, the bound on the number of brokers in I_2 is set to be m. Clearly, if I_1 admits a perfect matching of size m, then we can co-locate the corresponding sets of topics in I_2 using m brokers. On the other hand, if all topics in I_2 can be co-located using m brokers, since there are km topics in total and each broker can accommodate at most k topics, then each broker must contain exactly k distinct topics, which based on the reduction must come from the elements in one of the k-dimensional vectors of I_1. Thus, using the sets of co-located topics in I_2, we can get a perfect matching M' for I_1. □

2.5.3 Heuristics

Given the NP-hardness result for $k \geq 3$, we propose heuristic solutions to solve k-TCP sub-optimally. Recall that the goal is to find a co-location scheme for a collection T of topics on a minimum set B of brokers. Equivalently, the topics could be considered to form a partition of B disjoint subsets $\{T(b_1), T(b_2), \ldots, T(b_{|B|})\}$ such that each broker b_j hosts a feasible subset $T(b_j) \subseteq T$ of topics, i.e., $\bigcup_{b_j \in B} T(b_j) = T$ and $T(b_j) \bigcap T(b_{j'}) = \emptyset$ for any $b_j \neq b_{j'}$.

In the following, we first describe two heuristics that are inspired by the greedy algorithms in bin packing and set cover problems, respectively, and apply them to the k-TCP context. We then present a hybrid heuristic that combines the two algorithms.

Algorithm 1: FirstFitDecreasing (FFD$_k$)

Input: Collection $T = \{t_1, t_2, \ldots, t_n\}$ of n topics, latency ℓ_i for each topic $t_i \in T$ when assigned to a broker in isolation, degree of co-location k, and feasibility function $\mathscr{F}$

Output: A partition of topics $\{T(b_1), T(b_2), \ldots, T(b_{|B|})\}$ for a set B of brokers with each broker $b_j \in B$ hosting a subset $T(b_j) \subseteq T$ of topics

```
1  begin
2      Sort the topics in decreasing order of latency when assigned to a broker in isolation, i.e.,
       ℓ_1 ≥ ℓ_2 ≥ ··· ≥ ℓ_n;
3      Initialize |B| ← 0;
4      for topic t_i (i = 1...n) do
5          mapped ← false;
6          for broker b_j (j = 1...|B|) do
7              if |T(b_j)| = k then
8                  continue;
9              end
10             if 𝓕(T(b_j) ⋃ {t_i}) = 1 then
11                 T(b_j) ← T(b_j) ⋃ {t_i};
12                 mapped ← true;
13                 break;
14             end
15         end
16         if mapped = false then
17             |B| ← |B| + 1;
18             Start a new broker b_|B| with T(b_|B|) = {t_i};
19         end
20     end
21 end
```

2.5.3.1 First Fit Decreasing

The first heuristic is inspired by a greedy algorithm in the bin packing problem, which we call First Fit Decreasing, or FFD$_k$ for a given degree of co-location k, and its pseudocode is presented in Algorithm 1. First, the algorithm sorts all topics in decreasing order of latency when they are assigned to a broker in isolation (line 2). Then, it considers each topic in sequence and finds the first broker that can feasibly host it together with the existing topics that have already been assigned to the broker (lines 6-15). If no such broker can be found, it starts a new broker and assigns the topic there (lines 16-19), which according to Property 1(a) is always feasible. The complexity of the algorithm is $O(n \log n + n|B|)$, where $|B|$ is the total number of brokers in the solution. Since $|B| \leq n$, the algorithms runs in $O(n^2)$ time in the worst case.

Algorithm 2: LargestFeasibleSet (LFS$_k$)

Input: Collection $T = \{t_1, t_2, \ldots, t_n\}$ of n topics, degree of co-location k, and feasibility function $\mathscr{F}$
Output: A partition of topics $\{T(b_1), T(b_2), \ldots, T(b_{|B|})\}$ for a set B of brokers with each broker $b_j \in B$ hosting a subset $T(b_j) \subseteq T$ of topics

```
1  begin
2      Initialized |B| ← 0;
3      while T ≠ ∅ do
4          while ∃T' ⊆ T s.t. 𝓕(T') = 1 and |T'| = k do
5              |B| ← |B| + 1;
6              Start a new broker b_|B| with T(b_|B|) = T';
7              T ← T \ T';
8          end
9          k ← k − 1;
10         if k = 2 then
11             MaximumMatching(T);
12         end
13     end
14 end
```

2.5.3.2 Largest Feasible Set

The second heuristic is inspired by the greedy algorithm in the set cover problem and the set packing problem. We call it Largest Feasible Set, or LFS$_k$ for a given degree of co-location k, and its pseudocode is presented in Algorithm 2. Specifically, the algorithm works in iterations. At each iteration, it finds any largest feasible set of k topics and co-locates them on a new broker (lines 4-8). If no such set can be found anymore, the maximum degree of co-location k is then decremented by 1 (line 9), and the process continues until k is reduced down to 2, in which case we can run the maximum matching algorithm (lines 10-12) as described in the proof of Theorem 1 that guarantees to co-locate the remaining topics in an optimal fashion. The complexity of the algorithm is $O(n^k)$ dominated by enumerating all possible subsets of k topics in the worst case for the feasibility test. Note that although the complexity is polynomial in the number n of topics, the running time can be prohibitive for a high degree of co-location (e.g., $k > 4$) on even moderate n. We resolve this problem below with a hybrid heuristic.

2.5.3.3 Hybrid Solution

We now present a heuristic that combines the benefits of top-down search of LFS and low complexity of FFD. In particular, the algorithm takes a parameter $k' \leq k$ as input, and Algorithm 3 shows its pseudocode. We call the algorithm LFS$_{k'}$+FFD$_k$. Similarly to LFS$_k$, this hybrid solution also works in iterations, but an iteration now consists of two steps. In the first step, any feasible set of k' topics (if any) are found and co-located on a new broker (lines 4-7). This is followed by the second step, which uses the first fit heuristic to maximize the degree of co-location up to k on this broker (lines 8-16) based on a sorted sequence of the remaining topics (line 2). This two-step iteration continues until no feasible set of k' topics can be found. In this case, the algorithm resorts to LFS with parameter $k'-1$ for assigning the remaining topics (line 18). The overall complexity of the algorithm is $O(n^{k'} + n^2/k')$, with the two parts coming from running LSF initially (using parameter k') and FFD (on at most n/k' brokers), respectively. Note that when the parameter satisfies $k' = k$ the algorithm becomes exactly LFS$_k$, and when $k' = 1$ it becomes FFD$_k$. Thus, for a suitable choice of k', the algorithm combines the two previous heuristics while offering a lower complexity solution to the problem.

2.6 Experiments

In this section, we present experimental results to validate our proposed solution for providing latency QoS of data delivery in publish-process-subscribe systems. We first describe the testbed used for conducting the experiments, and then present the accuracy results of the k-topic co-location model and the performance results for the proposed k-TCP heuristics.

2.6.1 Experimental Testbed and Setup

Our testbed comprises 25 heterogeneous machines running Ubuntu 14.04, of which 13 are homogeneous machines with four 2.5GHz Intel Xeon E5420 cores, 4GB RAM and

Algorithm 3: $\text{LFS}_{k'}$+FFD_k

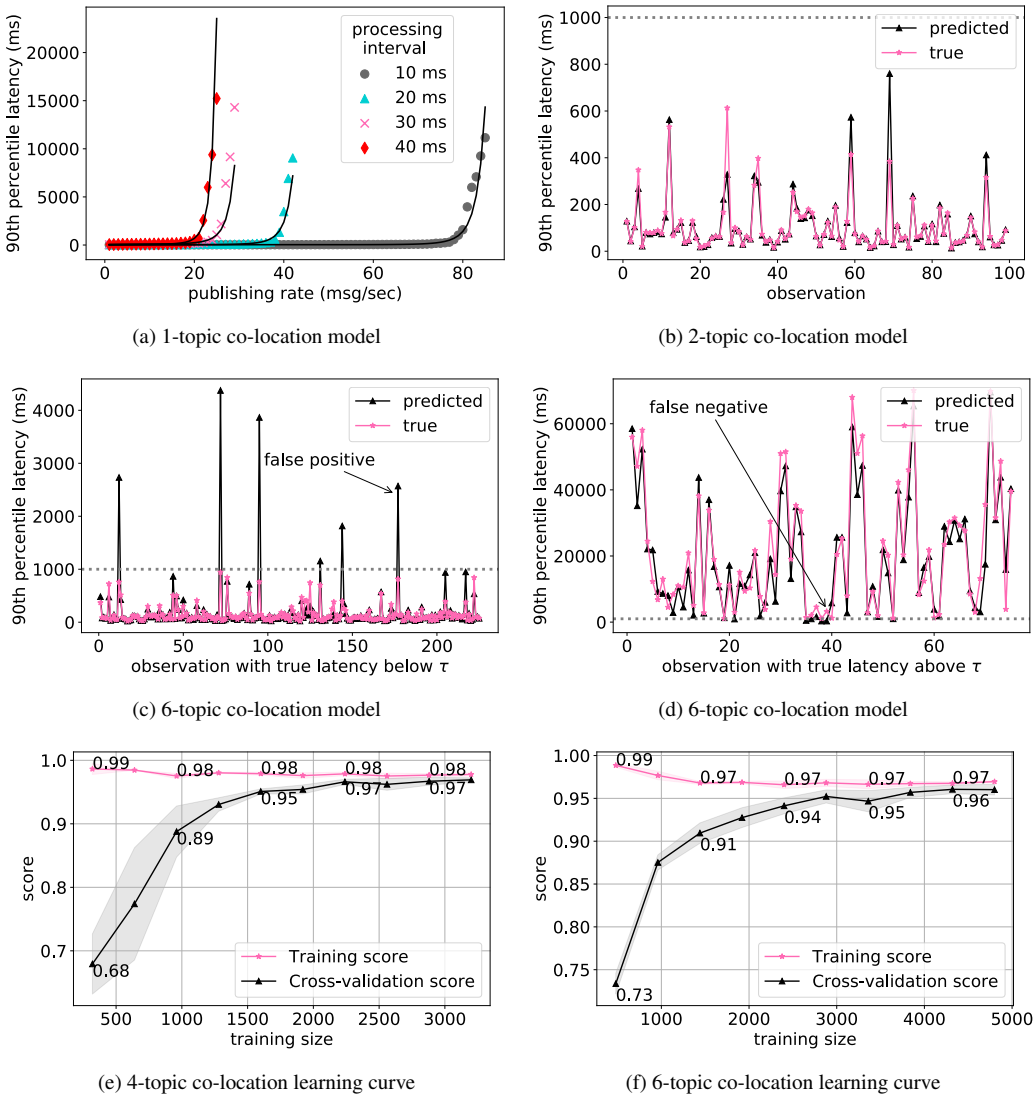
Input: Collection $T = \{t_1, t_2, \ldots, t_n\}$ of n topics, latency ℓ_i for each topic $t_i \in T$ when assigned to a broker in isolation, degree of co-location k, parameter $k' \leq k$, and feasibility function $\mathscr{F}$

Output: A partition of topics $\{T(b_1), T(b_2), \ldots, T(b_{|B|})\}$ for a set B of brokers with each broker $b_j \in B$ hosting a subset $T(b_j) \subseteq T$ of topics

```
begin
    Sort the remaining topics in decreasing order of latency when assigned to a broker in isolation;
    Initialized |B| ← 0;
    while ∃T' ⊆ T s.t. 𝒻(T') = 1 and |T'| = k' do
        |B| ← |B| + 1;
        Start a new broker b_|B| with T(b_|B|) = T';
        T ← T\T';
        for topic t_i (i = 1 ... |T|) do
            if |T(b_|B|)| = k then
                break;
            end
            if 𝒻(T(b_|B|) ⋃ {t_i}) = 1 then
                T(b_|B|) ← T(b_|B|) ⋃ {t_i};
                T ← T\{t_i};
            end
        end
    end
    LargestFeasibleSet(T, k' − 1);
end
```

1Gb/s network adapter, which were used for running the brokers. The k-topic co-location models are learned for this hardware type. The remaining machines were used to host the publisher/subscriber endpoints and the Zookeeper coordination service. We benchmarked the machines used to run the endpoints to find the maximum number of endpoints that can be run on them reliably. This was done to minimize the effect of resource contention on the experimental results. All machines were time synchronized using NTP.

Drawing from our motivational use case (Section 2.3.1) and the RIoTBench [71] results in which the ETL and prediction stream processing pipelines for the New York taxi data were benchmarked to take between 10ms and 40ms, the per-sample processing interval for any topic in our experiments was set to be either 10ms, 20ms, 30ms or 40ms. Publisher endpoints send 4KB messages at the rate of 1 message/second for two minutes (i.e., a total of 120 messages per publisher) to ensure that any experiment runs for a reasonable length of time. If a topic t in an experiment is configured with per-sample processing



(a) 1-topic co-location model

(b) 2-topic co-location model

(c) 6-topic co-location model

(d) 6-topic co-location model

(e) 4-topic co-location learning curve

(f) 6-topic co-location learning curve

Figure 2.2: Performance of latency prediction model

interval p and publishing rate r, then the broker is configured to execute `stress-ng matrix-product` so that it takes p per-sample processing time. Additionally, one subscriber and r publisher endpoints for topic t are created. All subscriber and publisher endpoints connect to the system before starting the experiment to ensure the fidelity of experimental results. In computing the 90th percentile latency of a topic, the latency values for some initial messages on the topic are not considered since they are observed to be very high due to initialization and connection setup.

Table 2.1: Accuracy of k-topic co-location model

k	#datapoints (training)	accuracy (training)	accuracy (test)	#datapoints (validation)	accuracy (validation)
2	2000	.987	.985	100	.972
3	3000	.985	.978	150	.976
4	4000	.983	.979	200	.984
5	5000	.981	.978	250	.951
6	6000	.981	.956	300	.968

2.6.2 K-Topic Co-location Model Learning

In order to learn the k-topic co-location model for $k \geq 2$, we first learn the latency model for a topic in isolation, i.e., when $k = 1$. In particular, the 1-topic co-location model takes the per-sample processing interval p and publishing rate r of a topic t as inputs and predicts its 90th percentile latency ℓ. We can use this model to estimate the maximum sustainable publishing rate $r_{\max}$ for the topic with per-sample processing time p beyond which the 90th percentile latency for the topic will violate its desired QoS τ. This maximum rate $r_{\max}$ is used to ensure Property 1(a), i.e., a topic can always be placed on a broker in isolation, otherwise topic replication and partitioning of publishers over topic replicas [52] will be needed.

We found that polynomial regression of degree 4 provides the best fit for the 1-topic co-location model with a training and test accuracy of .975 and .97, respectively. We used a dataset with 180 datapoints; 60% of which were used for training and the remaining 40% were used for testing. Figure 2.2a shows the fit of the polynomial curve in degree 4 over experimentally observed 90th percentile latency values. Here, the x-axis shows the publishing rate r in messages/second and the y-axis shows the 90th percentile latency in milliseconds for p values of 10ms, 20ms, 30ms and 40ms. Using the model, we found the $r_{\max}$ for sub-second 90th percentile latencies to be 78 messages/second, 37 messages/second, 24 messages/second and 20 messages/second for p values of 10ms, 20ms, 30ms and 40ms, respectively.

We then used $r_{\max}$ found under the 1-topic co-location model to create the training dataset for k-topic co-location models with $k \geq 2$. To create the training dataset, for each topic, p was uniformly randomly chosen from the set $\{10\text{ms}, 20\text{ms}, 30\text{ms}, 40\text{ms}\}$ and r was uniformly randomly chosen from the range $[1, r_{\max}]$. For each k-topic co-location model, we trained over 1,000 different randomly generated test configurations, and each configuration contains k datapoints, one for each of the k topics. This gives $1000k$ datapoints for each k-topic co-location model. A test runs for $\sim$3 mins and it took $\sim$11 days to collect the training data to learn these offline latency models. In all of these experiments, the network utilization was kept well below the 1Gb/sec network capacity of the broker to make sure that network saturation does not impact the gathered results.

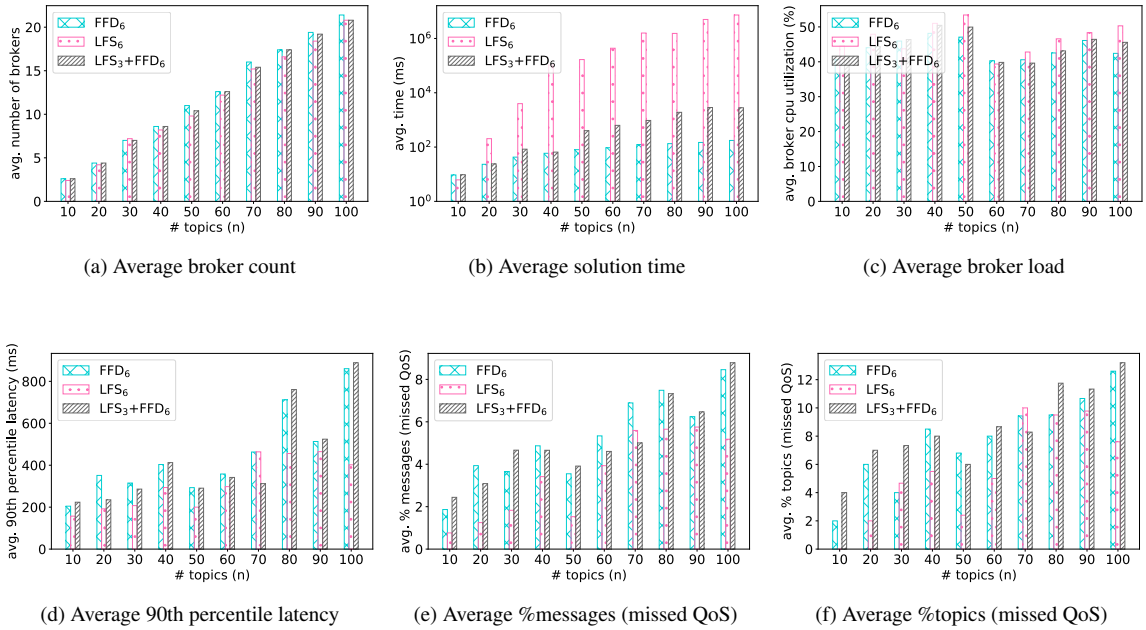
We tested different neural network architectures for each k-topic co-location model, and found that a neural network with two hidden layers composed of 40 neurons each performed well for $k \leq 5$. Figure 2.2e shows the learning curve for $k = 4$. The learning curve shows that the chosen neural network architecture has low bias and variance since both training and validation errors converge to a low value of $\sim$3%. A more complex neural network architecture was needed for $k = 6$ as the parameter space increases. In this case, a neural network with two hidden layers composed of 100 neurons each performed well. Figure 2.2f shows the learning curve for the 6-topic co-location model. Again, we see that the chosen neural network architecture has both low bias and low variance.

As described in Section 4.3, the input features for the model are p_f, r_f, d_f, $\sum_{t_b \in T_B} p_b$, $\sum_{t_b \in T_B} r_b$ and $\sum_{t \in T_B} d_b$. Table 4.2 shows the accuracy of the learned models for k up to 6. We used the logarithm of the 90th percentile latency as the output for the model as it performed better than using the 90th percentile latency value itself. Rectified Linear Units (ReLu) was used as the activation function, the limited memory Broyden-Fletcher-Goldfarb-Shanno (`lbfgs`) solver was used and the L2 regularization factor was set to 0.1. We used 95% of the datapoints for training and the remaining 5% for testing. A separate validation dataset was created by running 50 different test configurations for each k. The performance of the

learned models on the validation dataset is also shown in Table 4.2. We see that the learned models have an accuracy of ∼97%.

Figure 2.2b shows the performance of the 2-topic co-location model on the validation dataset. We see that the predicted latency tracks the experimentally observed latency values closely. Similarly, Figure 2.2c and Figure 2.2d show the performance of the 6-topic co-location model on test data points for which the experimentally observed latency values are below and above τ, respectively. There are cases where the model makes inaccurate latency predictions, resulting in both false-positives and false-negatives. Figure 2.2c shows a false-positive occurrence where the predicted value is greater than τ and the experimentally observed 90th percentile latency is below τ. Figure 2.2d shows a false-negative occurrence where the predicted 90th percentile latency is below τ and the experimentally observed 90th percentile latency is above τ. False negatives result in QoS violations and false positives result in inefficient resource utilization.

2.6.3 Performance of k-TCP Heuristics



(a) Average broker count (b) Average solution time (c) Average broker load

(d) Average 90th percentile latency (e) Average %messages (missed QoS) (f) Average %topics (missed QoS)

Figure 2.3: Performance of k-TCP heuristics for varying n

We now study how each of the three k-TCP heuristics, namely FFD$_k$, LFS$_k$, and LFS$_{k'}$+

FFD_k, perform for $k = 6$ as the number of n topics to be placed increases. We set the parameter k' of the hybrid heuristic to be $k' = 3$. In Figure 2.3, we present results averaged over 5 random placement requests for each value of n. The performance of these heuristics is compared along the following six dimensions: 1) number of brokers needed for hosting n topics; 2) time to find a placement solution for n topics; 3) average CPU utilization of all brokers used for hosting n topics; 4) average 90th percentile latency of all n topics; 5) percentage of all messages across n topics with latency greater than τ (1 second); and 6) percentage of n topics whose 90th percentile latency is greater than τ.

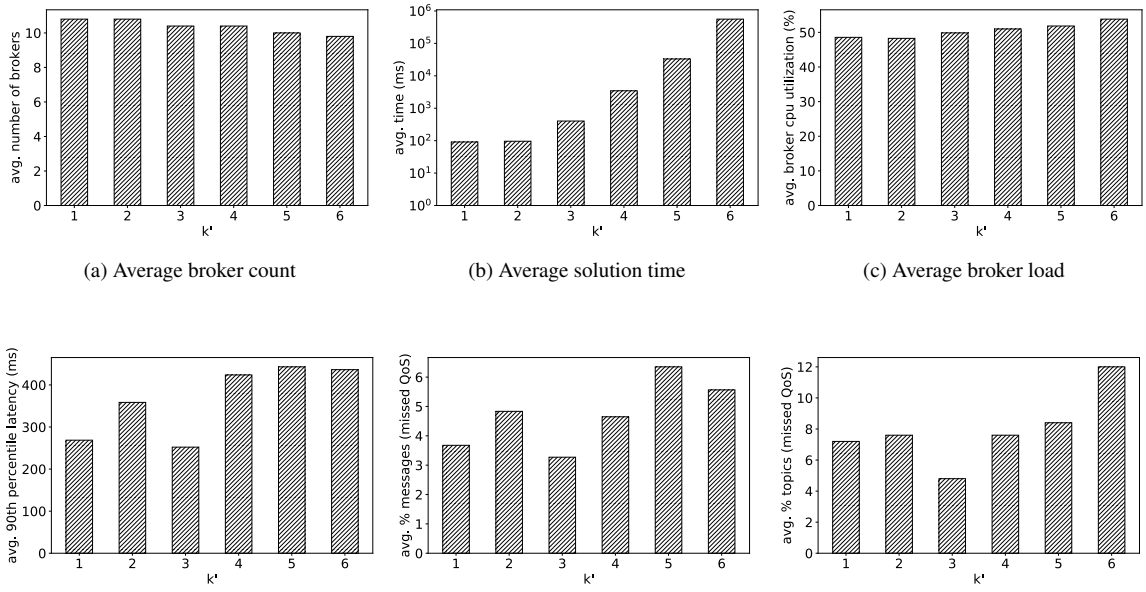
In Figure 2.3a, we observe that the three heuristics perform similarly to each other in terms of the number of brokers used for placing the topics. LFS_k is able to find a placement which uses less number of brokers than both FFD_k and $\text{LFS}_{k'}$+FFD_k for most of the cases. However, as seen in Figure 2.3b, LFS_k takes a much longer time to find a placement than FFD_k. As expected, $\text{LFS}_{k'}$+FFD_k, being a hybrid of the other two, takes less time than LFS_k but more time than FFD_k. Average CPU load of the brokers in the system for the placement produced by FFD_k, LFS_k and $\text{LFS}_{k'}$+FFD_k as seen in Figure 2.3c, does not show a wide variation.

Figure 2.3d shows the average 90th percentile latency across all n topics in the system for the placements produced by the three heuristics. FFD_k and $\text{LFS}_{k'}$+FFD_k have comparable performance in most cases, while LFS_k yields a lower average 90th percentile latency for all values of n. Figure 2.3e shows that up to 9% of all messages in the system are not able to meet their latency QoS. The percentage of messages that miss their QoS is comparable for both FFD_k and $\text{LFS}_{k'}$+FFD_k in most cases, while LFS_k performs better with a lower percentage of messages with QoS violations. Similarly in Figure 2.3f, we see that the percentage of topics that miss their QoS is comparable for FFD_k and $\text{LFS}_{k'}$+FFD_k in most cases, while LFS_k yields a lower percentage of topics with missed QoS in almost all cases except for $n = 70$. It shows that up to 13% of the topics in the system miss their QoS due to incorrect broker assignment, and we are able to meet the QoS requirements for 87%

of the topics in the system. As discussed in Section 4.3, our solutions can be used along with a subscriber feedback mechanism to place the topics experiencing QoS violation on another broker.

These results show that while LFS_k heuristic performs better than FFD_k and $\text{LFS}_{k'}$+FFD_k, it has a prohibitively large running time. On the other hand, FFD_k takes much less time to compute the placement and performs comparably well with $\text{LFS}_{k'}$+FFD_k. Hence, by tolerating some degradation in performance, simpler heuristics such as FFD_k can be employed in favor of computationally more expensive heuristics like LFS_k.

2.6.4 Performance of $\text{LFS}_{k'}$+FFD_k



(a) Average broker count (b) Average solution time (c) Average broker load

(d) Average 90th percentile latency (e) Average %messages (missed QoS) (f) Average %topics (missed QoS)

Figure 2.4: Performance of $\text{LFS}_{k'}$+FFD_k for varying k'

Finally, we study how the hybrid heuristic $\text{LFS}_{k'}$+FFD_k performs with varying value of k'. As discussed earlier in Section 4.2.3, $\text{LFS}_{k'}$+FFD_k behaves as FFD_k for $k' = 1$ and as LFS_k for $k' = k$. Figure 2.4 shows the results when k' varies from 1 to 6. For each value of k', we present results averaged over the same 5 random placement requests for $n = 50$ topics.

As expected, for higher values of k', LFS$_{k'}$+FFD$_k$ finds a placement that uses fewer brokers, as seen in Figure 2.4a. However, the time to find the solution also increases with k' as seen in Figure 2.4b. Average CPU utilization of the brokers does not show much variation for different values of k', as seen in Figure 2.4c. Average 90th percentile latency increases for higher values of k' as seen in Figure 2.4d, since the placement of topics produced is more compact. Once again, the QoS requirement is not always met in the placement solution produced by the hybrid heuristic. Up to 6% of the messages and up to 12% of the topics experience QoS violations as seen in Figure 2.4e and Figure 2.4f.

2.7 Conclusion and Discussions

Many emerging IoT applications are latency critical in nature and require both real-time data dissemination and information processing. The Publish/Subscribe (pub/sub) pattern for many-to-many communications is often used to meet the scalable data dissemination needs of IoT applications. With the emergence of edge computing that promotes processing near the source of data, the pub/sub system has been extended to support processing at the edge-based pub/sub brokers, making it the *publish-process-subscribe* pattern. It is in this context that end-to-end quality of service (QoS) for data dissemination and processing must be satisfied to realize the next generation of edge-based, performance-sensitive IoT applications.

To provide desired latency QoS for data dissemination and processing, our solution learns a latency prediction model for a set of co-located topics on an edge broker and uses this model to place the topics in a latency-aware manner. Our solution makes the following contributions: (a) a sensitivity analysis on the impact of different pub/sub features including the number of subscribers, number of publishers, publishing rate, per-sample processing interval and background load, on a topic's 90th percentile latency; (b) a latency prediction model for a topic's 90th percentile latency, which was then used for the latency-aware placement of topics on brokers; and (c) an optimization problem formulation for k-topic

co-location to minimize the number of brokers used while providing QoS guarantees.

The following lessons were learned from and insights into different dimensions of future work were informed by this research:

- The accuracy of the k-topic co-location model has a significant impact on QoS satisfaction and resource efficiency. More advanced machine learning methods for learning the k-topic co-location model could be investigated. For higher values of k, training over a larger search space is needed for good accuracy, but this will require significant additional resources for model learning. Online methods for updating the prediction model can also be explored, e.g., via reinforcement learning [86]. Transfer learning [83] of the k-topic co-location models for different hardware architectures on the basis of some learned models is another direction that can be explored.

- Currently, our load balancing decisions are made statically including the topic placement decisions. Future work can therefore involve dynamic load balancing decisions including elastic auto-scaling of the number of brokers used. Proactive provisioning of resources can also be performed on the basis of workload forecasting. Finally, network link state can also be incorporated.

The source code and experimental apparatus used in the research is made available in open source at https://github.com/doc-vu/edgent.

Chapter 3

REACTIVE STREAM PROCESSING FOR DATA-CENTRIC PUBLISH/SUBSCRIBE

3.1 Introduction

The *Internet of Things* (IoT) is a significant expansion of the Internet to include physical devices; thereby bridging the divide between the physical world and cyberspace. These devices or "things" are uniquely identifiable, fitted with sensors and actuators, which enable them to gather information about their environment and respond intelligently [87]. IoT has helped realize critical applications, such as smart-grids, intelligent transportation systems, advanced manufacturing, health-care tele-monitoring, etc. These applications share several key cross-cutting aspects. First, they are often large-scale, distributed systems comprising several, potentially mobile, publishers of information that produce large volumes of asynchronous events. Second, the resulting unbounded asynchronous streams of data must be combined with one-another and with historical data and analyzed in a responsive manner. While doing so, the distributed set of resources and inherent parallelism in the system must be effectively utilized. Third, the analyzed information must be transmitted downstream to a heterogeneous set of subscribers. In essence, the emerging IIoT systems can be understood as a *distributed asynchronous dataflow*. The key challenge lies in developing a dataflow-oriented programming model and a middleware technology that can address both *distribution* and *asynchronous processing* requirements adequately.

The distribution aspects of dataflow-oriented systems can be handled sufficiently by data-centric publish/subscribe (pub/sub) technologies [88], such as Object Management Group (OMG)'s Data Distribution Service (DDS) [9]. DDS is an event-driven publish-subscribe middleware that promotes asynchrony and loose-coupling between data publishers and subscribers which are decoupled with respect to (1) *time* (*i.e.*, they need not be present at the same time), (2) *space* (*i.e.*,they may be located anywhere), (3) *flow* (*i.e.*, data

publishers must offer equivalent or better quality-of-service (QoS) than required by data subscribers), (4) *behavior* (*i.e.*, business logic independent), (5) platforms, and (6) programming languages. In fact, as specified by the Reactive Manifesto [89], event-driven design is a pre-requisite for building systems that are *reactive*,*i.e.* readily responsive to incoming data, user interaction events, failures and load variations- traits which are desirable of critical IoT systems. Moreover, asynchronous event-based architectures unify scaling up (e.g., via multiple cores) and scaling out (e.g., via distributed compute nodes) while deferring the choice of the scalability mechanism at deployment-time without hiding *the network* from the programming model. Hence, the asynchronous and event-driven programming model offered by DDS makes it particularly well-suited for demanding IoT systems.

However, the data processing aspects, which are local to the individual stages of a distributed dataflow, are often not implemented as a dataflow due to lack of sufficient composability and generality in the application programming interface (API) of the pub/sub middleware. DDS offers various ways to receive data such as, listener callbacks for push-based notification, read/take functions for polling, waitset and read-condition to receive data from several entities at a time, and query-conditions to enable application-specific filtering and demultiplexing. These primitives, however, are designed for data and meta-data *delivery* [1] as opposed to *processing*. Further, the lack of proper abstractions forces programmers to develop event-driven applications using the observer pattern– disadvantages of which are well documented [36].

A desirable programming model is one that provides a first-class abstraction for streams; and one that is composable. Additionally, it should provide an exhaustive set of reusable coordination primitives for reception, demultiplexing, multiplexing, merging, splitting, joining two or more data streams. We go on to argue that a dataflow programming model that provides the coordination primitives (combinators) implemented in functional program-

[1] Strictly, DDS API is designed for retrieving the state of an object rather than individual updates about an object

ming style as opposed to an imperative programming style yields significantly improved expressiveness, composability, reusability, and scalability. [2] A desirable solution should enable an end-to-end dataflow model that unifies the local as well as the distribution aspects.

To that end we have focused on composable event processing inspired by Reactive Programming [90] and blended it with data-centric pub/sub. Reactive programming languages provide a dedicated abstraction for time-changing values called *signals* or *behaviors*. The language runtime tracks changes to the values of signals/behaviors and propagates the change through the application by re-evaluating dependent variables automatically. Hence, the application can be visualized as a *data-flow*, wherein data and respectively changes thereof implicitly flow through the application [91, 92]. Functional Reactive Programming (FRP) [93] was originally developed in the context of pure functional language, Haskell. and has since been implemented in other languages, for example, Scala.React (Scala) [36], FlapJax (Javascript) [94], Frappe (Java) [95].

Composable event processing–a modern variant[3] of FRP–is an emerging new way to create scalable reactive applications [96], which are applicable in a number of domains including HD video streaming [97] and UIs. It offers a declarative approach to event processing wherein program specification amounts to "what" (i.e., declaration of intent) as opposed to "how" (looping, explicit state management, etc.). State and control flow are hidden from the programmers, which enables programs to be visualized as a data-flow. Furthermore, functional style of programming elegantly supports composability of asynchronous event streams. It tends to avoid shared mutable state at the application-level, which is instrumental for multicore scalability. Therefore, there is a compelling case to systematically blend reactive programming paradigm with data-centric pub/sub mechanisms for realizing emerging IoT applications.

[2] Microsoft Robotics Coordination and Concurrency Runtime (CCR) and Robot Operating System (ROS) http://wiki.ros.org/

[3] without continuous time abstraction and denotation semantics

To demonstrate and evaluate our research ideas, we have combined concrete instances of pub/sub technology and reactive programming. The data-centric pub/sub instance we have used is OMG's DDS, more specifically the DDS implementation provided by Real Time Innovations Inc; while the reactive programming instance we have used is Microsoft's .NET Reactive Extensions (Rx.NET) [98]. We make the following contributions:

1. We show the strong correspondence between the distributed asynchronous dataflow model of DDS and the local asynchronous dataflow model of Rx. We integrated the two technologies in the Rx4DDS.NET open-source library. The remarkable overlap between the two technologies allows us to substitute one for the other and overcome the missing capabilities in both, such as the lack of a composable data processing API in DDS and the lack of interprocess communication and back-pressure support in .NET Rx; [4]

2. We present the advantages of adopting functional style of programming for real-time stream processing. Functional *stream* abstractions enable seamless composability of operations and preserve the conceptual "shape" of the application in the actual code. Furthermore, state management for sliding time-window, event synchronization and concurrency management can be delegated to the run-time which is made possible by the functional tenets, such as the immutable state.

3. We evaluate the Rx4DDS.NET library using a publicly available high-speed sensor data processing challenge [101]. We present the ease and the effect of introducing concurrency in our functional implementation of "queries" running over high-speed streaming data. Our dataflow programming admits concurrency very easily and improves performance (up to 35%).

4. Finally, we compare our functional implementation with our imperative implementation of the same queries in C#. We highlight the architectural differences and the

[4] Reactive Streams project [99], RxJava [100] support backpressure

lessons learned with respect to "fitness for a purpose" of stream processing, state management, and configurability of concurrency.

The remaining chapter is organized as follows: Section 3.2 compares our proposed solution with prior efforts; Section 3.3 describes our reactive solution that integrates Rx and DDS; Section 3.4 reports on both our qualitative and quantitative experience building a reactive solution to solve a specific case study problem; and finally Section 3.5 provides concluding remarks and lessons learned.

3.2 Related Work

A research roadmap towards applying reactive programming in distributed event-based systems has been presented in [102]. In this work the authors highlight the key research challenges in designing distributed reactive programming systems to deal with "data-in-motion". Our work on Rx4DDS.NET addresses the key open questions raised in this prior work. In our case we are integrating Reactive Programming with DDS that enables us to build a loosely coupled, highly scalable and distributed pub/sub system, for reactive stream processing.

Nettle is a domain-specific language developed in Haskell, a purely-functional programming language, to solve the low-level, complex and error-prone problems of network control [103]. Nettle uses Functional Reactive Programming (FRP) including both the discrete and continuous abstractions and has been applied in the context of OpenFlow software defined networking switches. Although the use case of Nettle is quite different from our work in Rx4DDS.NET, both approaches aim to demonstrate the integration of reactive programming with an existing technology: we use DDS where as Nettle uses OpenFlow.

The ASEBA project demonstrates the use of reactive programming in the event-based control of complex robots [104]. The key reason for using reactive programming was the need for fast reactivity to events that arise at the level of physical devices. Authors of the ASEBA work argue that a centralized controller for robots adds substantial delay and

presents a scalability issue. Consequently, they used reactive programming at the level of sensors and actuators to process events as close to the source as possible

Our work on Rx4DDS.NET is orthogonal to the issues of where to place the reactive programming logic. In our case such a logic is placed with every processing element, such as the subscriber that receives the topic data.

Prior work on Eventlets [105] comes close to our work on Rx4DDS.NET. Eventlets provides a container abstraction to encapsulate the complex event processing logic inside a component so that a component-based service oriented architecture can be realized. The key difference between Eventlets and Rx4DDS.NET is that the former applies to service oriented architectures and component-based systems, while our work is used in the context of publish/subscribe systems. Although this distinction is evident, there are ongoing efforts to merge component abstractions with pub/sub systems such that we may be able to leverage component abstractions in our future work.

Functional programming style (akin to Rx) has been used effectively in Spark Streaming [106] in the context of Lambda Architecture (LA) [107] to write business logic just once using functional combinator libraries and reuse that implementation for both real-time and batch processing of data. In a typical LA, the batch layer maintains the master data whereas the "speed layer" compensates for the high latency of the batch layer and also trades accuracy for speed. Business queries represented using the functional style abstract away the source of data (batch/streaming) and improve code reuse.

An ongoing project called Escalier [108] has very similar goals as our work. Escalier provides a Scala language binding for DDS. The future goals of the Escalier project are to provide a complete distributed reactive programming framework, however, we have not yet found sufficient related publications nor are we able to determine from their github site whether this project is actively maintained or not. Similarly, OpenDDS [109] and OpenSplice [110] describe integration of DDS with Rx and other functional-style stream processing technologies. However, to the best of our knowledge, our work includes the

most comprehensive comparison and evaluation of the two technologies together.

3.3 Design of the Rx4DDS.NET Library

We now describe our approach to realizing Rx4DDS.NET. To better understand our solution, we first provide a brief overview of DDS and Rx. We then illustrate some drawbacks of our imperative solution implemented only using DDS, which motivates the need for Rx4DDS.NET.

3.3.1 Overview of OMG DDS Data-Centric Pub/Sub Middleware

OMG DDS is a *data-centric* middleware that understands the schema/structure of "data-in-motion". The schemas are explicit and support keyed data types much like a primary key in a database. Keyed data types partition the global data-space into logical streams (*i.e.*, instances) of data that have an observable lifecycle.

DDS *DataWriters* (belonging to the publisher) and *DataReaders* (belonging to the subscriber) are endpoints used in DDS applications to write and read typed data messages (DDS samples) from the global data space, respectively. DDS ensures that the endpoints are compatible with respect to the topic name, data type, and the QoS policies.

3.3.2 Microsoft Reactive Extensions (Rx)

Microsoft Reactive Extensions (Rx) [98] is a library for composing asynchronous and event-based programs. Using Rx, programmers represent asynchronous data streams with *Observables*, query asynchronous data streams using a library of composable functional *Operators*, and parameterize the concurrency in the asynchronous data streams using *Schedulers*. Rx offers many built-in primitives for filtering, projecting, aggregating and composing multiple sources of events. Rx has been classified as a "cousin of reactive programming" [90] since Rx does not provide a dedicated abstraction for time-changing values which can be used in ordinary language expressions (*i.e.* automatic lifting of operators to work on behaviors/signals); rather it provides a container (*observable*) and the programmer

needs to manually extract the values from this container and encode dependencies between container values explicitly (*i.e.* manual lifting of operators).

3.3.3 Challenges in our Imperative Solution

We implemented the DEBS 2013 grand-challenge queries [101] in an imperative style using DDS and C#. This experience highlighted a number of challenges with *our* imperative solution which motivates our work on Rx4DDS.NET. We describe these challenges below:

- **Lack of built-in streaming constructs –** We had to manually code the logic and maintain relevant state information for merging, joining, multiplexing, de-multiplexing and capturing data dependencies between multiple streams of data.

- **Lack of a concurrency model to scale up event processing by employing multiple cores –** Since DDS utilizes a single dedicated thread for a DataReader to receive an input event, there was a need to manually create threads or a thread pool to exploit available cores for concurrent data processing.

- **Lack of a reusable library for sliding time windows –** A system for complex event processing typically requires handling events based on different sliding time-windows (*e.g.*, last one hour or one week). A reusable library for sliding time-windows which also operates with other streaming constructs is required. In our imperative approach, we had to reinvent the solution every time it was needed.

- **Lack of flexibility in component boundaries –** In DDS, data-writers/readers are used for publishing/subscribing intermediate results between processing stages. However, this approach incurs overhead due to serialization and de-serialization of DDS samples across the data writer-reader boundary, even if event processing blocks are deployed on the same machine. The use of data-writers/readers imposed a hard component boundary and there was no way to overcome that transparently.

3.3.4 Rx4DDS.NET: Integrating Rx and DDS

To address the challenges with our imperative approach, we designed our reactive programming solution that integrates .NET Reactive Extensions (Rx) framework with DDS. This solution is made available as a reusable library called Rx4DDS.NET. We describe our design by illustrating the points of synergy between the two.

In Rx, asynchronous data streams are represented using Observables. For example, an `IObservable<T>` produces values of type T. Observers subscribe to data streams much like the *Subject-Observer* pattern. Each Observer is notified whenever a stream has a new data using the observer's `OnNext` method. If the stream completes or has an error, the `OnCompleted,` and `OnError` operations are called, respectively. `IObservable<T>` supports chaining of functional operators to create pipelines of data processing operators (a.k.a. combinators).

Some common examples of operators in Rx are `Select, Where, SelectMany, Aggregate, Zip,` etc. Since Rx has first-class support for streams, Observables can be passed and returned to/from functions. Additionally, Rx supports streams of streams where every object produced by an Observable is another Observable (*e.g.*,`IObservable<IObservable<T>>`). Some Rx operators, such as `GroupBy,` demultiplex a single stream of T into a stream of keyed streams producing `IObservable<IGroupedObservable<Key,T>>.` The keyed streams (`IGroupedObservable<Key,T>`) correspond directly with DDS instances as described next.

In DDS, a topic is a logical data stream in the global data-space. DataReaders receive notifications when an update is available on a topic. Therefore, a topic of type T maps to Rx's `IObservable<T>.` DDS supports a key field in a data type that represents a unique identifier for data streams defined in a topic. A data stream identified by a key is called instance. If a DataReader uses a keyed data type, DDS distinguishes each key in the data as a separate instance. An instance can be thought of as a continuously changing row in a database table. DDS provides APIs to detect instance lifecycle events including

Table 3.1: Mapping of DDS concepts to Rx concepts

DDS Concept	Corresponding Rx Concept and the Rx4DDS.NET API
Topic of type T	An IObservable<T> created using DDSObservable.fromTopic<T>(...). Produces a hot observable. Internally creates a DataReader<T>.
Topic of type T with key-type=Key	An IObservable<IGroupedObservable<Key,T≫ created using DDSObservable.fromKeyedTopic<Key, T>(keySelector) where keySelector maps T to Key. Internally uses a DataReader<T>. Produces a hot observable.
A new instance in a topic of type T	An IGroupedObservable<Key,T> with Key==instance's key. Notified using IObserver<IGroupedObservable<Key,T≫.OnNext(IGroupedObservable<Key,T≫)
Disposal an instance (graceful)	Notified using IObserver<IGroupedObservable<Key,T≫.OnCompleted()
Dispose an instance (not alive, no writers)	Notified using IObserver<IGroupedObservable<Key,T≫.OnError(err)
DataReader<T>.take()	Push new values of T using IObserver<T>.OnNext(T). The fromTopic<T>() and fromKeyedTopic<Key,T>() factories produce hot observables.
DataReader<T>.read()	Push potentially repeated values of T using IObserver<T>.OnNext(T). The readFromDataReader<T>() and readFromDataReader<Key,T>() factories produce cold observables.
A transient local DataReader<T> with history = N	IObservable<T>.Replay(N) which caches the last N samples.
Hard error on a DataReader	Notified using Observer.OnError(err)
Entity status conditions (e.g., deadline missed, sample lost etc.)	Separate IObservable<T> streams per entity where T is communication status types. For example, IObservable<DDS::SampleLostStatus>.
Built-in discovery topics	Keyed observables for each built-in topic. For example, IObservable<IGroupedObservable¡Key, T≫ where T=Subscription/Publication/Participant BuiltInTopicData and Key=BuiltinTopicKey.
Read Conditions (parameterizes sample state, view state, and instance state)	IObservable<T>.Where() for filtering on sample state; New IGroupedObservable<Key,T> instance for new view state; and IObserver<IGroupedObservable<Key,T≫.OnCompleted() for disposed instance state.
Query Conditions	IObservable<T>.Where() for content-based filtering.
SELECT * in content-based filter topic (CFT) expression	IObservable<T>.Select(elementSelector) where elementSelector maps T to *
FROM "Topic" in CFT expression	DDSObservable.FromTopic<T>("Topic") or DDSObservable.FromKeyedTopic<Key, T>("Topic") if keyed
WHERE in CFT expression	IObservable<T>.Where(...)
ORDER BY in CFT expression	IObservable<T>.OrderBy(...)
MultiTopic (INNER JOIN)	IObservable<T>.selectMany(nestedSelector) where nestedSelector maps T to and IObservable<U>. Other alternatives are Join, CombineLatest, and Zip
Time-based filter	IObservable<T>.Sample(...)

Create, Read, Update, and Delete (CRUD). Since each instance is a logical stream by itself, a keyed topic can be viewed as a stream of keyed streams thereby mapping to Rx's `IObservable<IGroupedObservable<Key,T≫.`

Thus, when our Rx4DDS.NET library detects a new key, it reacts by producing a new `IGroupedObservable<Key,T>` with a new key. Subsequently, Rx operations can be composed on the newly created `IGroupedObservable<Key, T>` for instance-specific processing. As a result, pipelining and data partitioning can be implemented very elegantly using our integrated solution.

Table 3.1 summarizes how various DDS concepts map naturally to a small number of

Rx concepts. DDS provides various events to keep track of communication status, such as deadlines missed and samples lost between DataReaders and DataWriters. For discovery of DDS entities, the DDS middleware uses special types of DDS entities to exchange discovery events with remote peers using predefined *built-in topics*. As introduced in the table, discovery events using built-in topics and communication status events can be received and processed by Rx4DDS.NET API, but they are currently not implemented in our library and forms part of our ongoing improvements to the library.

Due to the similarity in the dataflow models, Rx and DDS are quite interchangeable. Table 3.1 forms the basis of our integration and the Rx4DDS.NET library. The contract between any two consecutive stages composed with Rx Observables is based on only two notions: (1) the static type of the data flowing across and (2) and the pair of `IObservable` and `IObserver` interfaces that represents the lifecycle of a data stream. These notions can be mapped directly to DDS in the form of strongly typed *topics* and the notion of *instance lifecycle*. No more (or less) information is required for a successful mapping as long as default QoS are used in DDS. The converse is also true, however, only a subset of QoS attributes can be mapped to Rx operators as of this writing. For example, DDS time-based filters can be mapped to Rx's `Sample` operator; Durability QoS with history maps to the `Replay` operator.

3.4 Evaluating Rx4DDS.NET Based Solution

This section reports on our qualitative and quantitative experience in evaluating our Rx4DDS.NET based solution. For the evaluations we have used a case study, which we also describe briefly.

3.4.1 Case Study: DEBS 2013 Grand Challenge Problem

The ACM International Conference on Distributed Event-based Systems (DEBS) 2013 Grand Challenge problem comprises real-life data from a soccer game and queries in event-

based systems [101]. Although the data is recorded in a file for processing, this scenario reflects IoT use cases where streamed data must be processed at runtime and not as a batch job.

The sensors are located near each player's cleats, in the ball, and attached to each goal keeper's hands. The sensors attached to the players generate data at 200Hz while the ball sensor outputs data at 2,000Hz. Each data sample contains the sensor ID, a timestamp in picoseconds, and three-dimensional coordinates of location, velocity, and acceleration. The challenge problem consists of four distinct queries that must be executed on the incoming streams of data. Figure 3.1 shows the high-level view of the four query components and the flow of data between them. For brevity we only describe queries 1 and 3 for which we also present experimental results later.

Query 1: The goal of query 1 is to calculate the running statistics for each player. Two sets of results – current running statistics and aggregate running statistics must be returned. Current running statistics should return the distance, speed and running intensity of a player, where running intensity is classified into six states (stop, trot, low, medium, high and sprint) based on the current speed. Aggregate running statistics for each player are calculated from the current running statistics and must be reported for four different time windows: 1 minute, 5 minutes, 20 minutes and entire game duration.

Query 3: Query 3 requires heat map statistics capturing how long each player stays in various pre-defined regions of the field. The soccer field is divided into four grids with x rows and y columns (8x13, 16x25, 32x50, 64x100) and results should be generated for each grid type. Moreover, distinct calculations are required for different time windows. As a result, query 3 must output 16 result streams (a combination of 4 different grid sizes and 4 time windows).

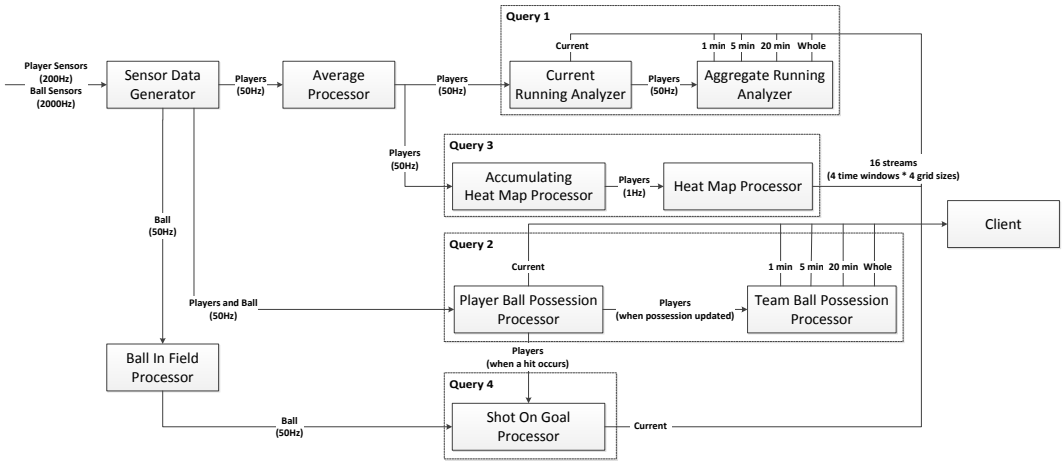


Figure 3.1: High Level Data Flow Architecture of DEBS 2013 Grand Challenge

3.4.2 Qualitative Evaluation of the Rx4DDS.NET Solution

We now evaluate our Rx4DDS.NET based solution along the dimensions of challenges expounded in Section 3.3.3 and compare it qualitatively with our imperative solution for the case study.

3.4.2.1 Automatic State Management

Recall that the imperative approach requires additional logic to maintain state and dependencies. For example, in the case study, to calculate average sensor data for a player from the sensor readings, we had to cache the sensor data for each *sensor_id* as it arrives in a map of *sensor_id* to sensor data. If the current data is for *sensor_id* 13, then the corresponding player name is extracted and a list of other sensors also attached to this player is retrieved. Subsequently using the retrieved *sensor_ids* as keys, the sensor data is retrieved from the map and used to compute the average player data.

In the functional style, there is no need to store the sensor values. We can obtain the latest sample for each sensor attached to the player with the `CombineLatest` function and then calculate the average sensor values. `CombineLatest` stream operator can be used to synchronize multiple streams into one by combining a new value observed on a stream with the latest values on other streams.

In Listing 3.1, *sensorStreamList* is a list that contains references to each sensor stream associated with sensors attached to a player. For example, for player Nick Gertje with attached *sensor_ids* (13, 14, 97, and 98), *sensorStreamList* for Nick Gertje holds references to sensor streams for sensors (13, 14, 97 and 98). Doing a `CombineLatest` on *sensorStreamList* returns a list (*lst* in Listing 3.1) of latest sensor data for each sensor attached to this player. *returnPlayerData* function is then used to obtain the average sensor values. The Marble diagram[5] for CombineLatest is shown in Figure 3.2.

Listing 3.1: CombineLatest Operator Example Code

```
List<IObservable<SensorData>> sensorStreamList =
        new List<IObservable<SensorData>>();
Observable
        .CombineLatest(sensorStreamList)
        .Select(lst => returnPlayerData(lst));
```

As another example of automatic state management, in query 1 the current running statistics need to be computed from average sensor data for each player (*PlayerData*). The distance traveled and average speed of a player (observed in the interval between the arrivals of two consecutive *PlayerData* samples) is calculated. Since our computation depends on the previous and current data samples, we can make use of the built-in `Scan` function and avoid maintaining previous state information manually. `Scan` is a runtime accumulator that will return the result of the accumulator function (optionally taking in a seed value) for each new value of source sequence. Figure 3.3 shows the marble diagram of the `Scan` operator. In the imperative approach, we employed the middleware cache to maintain previous state.

[5]Marble diagrams are a way to express and visualize how the operators in Rx work. For details see http://rxwiki.wikidot.com/marble-diagrams.

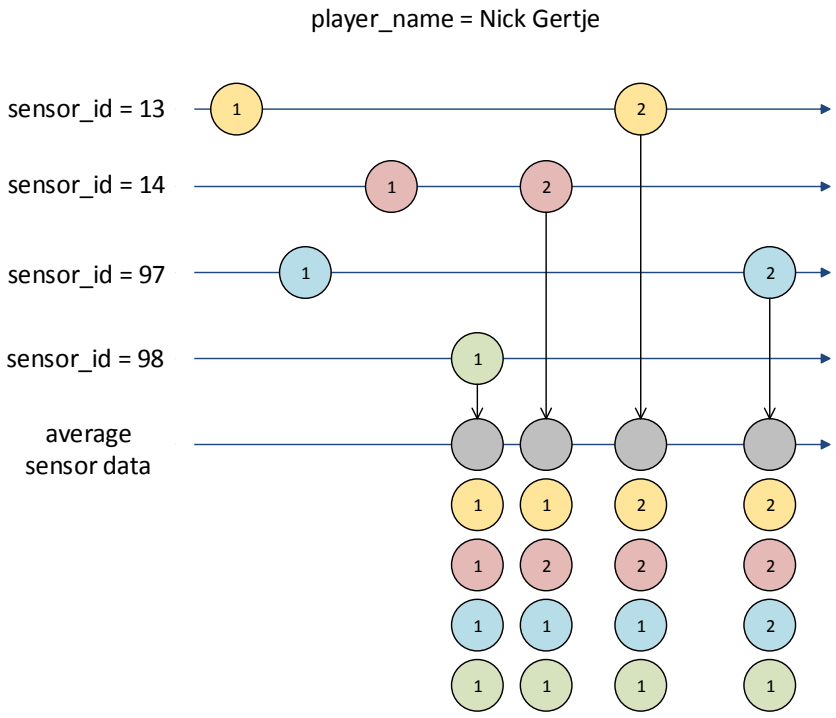


Figure 3.2: Marble Diagram of CombineLatest Operator

3.4.2.2 Concurrency Model to Scale-Up Multi-Core Event Processing

Rx provides abstractions that make concurrency management declarative, thereby removing the need to make explicit calls to create threads or thread pools. Rx has a free threading model such that developers can choose to subscribe to a stream, receive notifications and process data on different threads of control with a simple call to `subscribeOn` or `ObserveOn`, respectively. Delegating the management of shared state to stream operators also makes the code more easily parallelizable. Implementing the same logic in the imperative approach incurred greater complexity and the code was more verbose with explicit calls for creating and managing the thread pools.

In Query 1, the current running statistics and aggregate running statistics get computed for each player independently of the other players. Thus, we can use a pool of threads to perform the necessary computation on a per-player stream basis. In Listing 3.2,

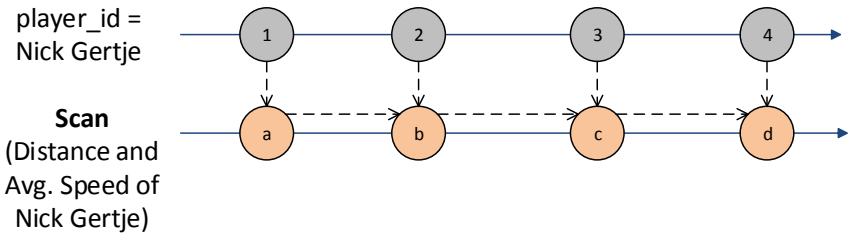


Figure 3.3: Marble Diagram of Scan Operator

player_streams represents a stream of all player streams *i.e.*, an *IObservable<IGroupedObservable<String,PlayerData>>*. Each player stream, which is an *IGroupedObservable<String,PlayerData>* keyed on player's name, is then processed further on a separate thread by using `ObserveOn`.

Listing 3.2: Concurrent Event Processing with Multi-threading

```
player_streams.selectMany( player_stream =>
{
        return player_stream
                        .ObserveOn(Scheduler.Default)
                        .CurrentRunningAnalysis();
}).Subscribe();
```

3.4.2.3 Library for Computations based on Time-Windows

One of the recurrent patterns in stream processing is to calculate statistics over a moving time window. All four queries in the case study require this support for publishing aggregate statistics collected over different time windows. In the imperative approach we had to reimplement the necessary functionality and manually maintain pertinent previous state information for the same because DDS does not support a time-based cache which can cache samples observed over a time-window.

Rx provides the "window abstraction" which is most commonly needed by stream pro-

cessing applications, and it supports both discrete (*i.e.*, based on number of samples) and time-based windows. Figure 3.4 depicts aggregation performed over a moving time window.

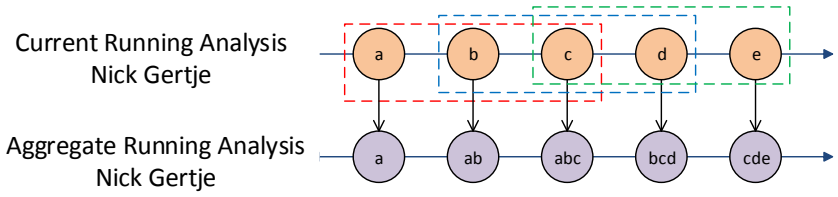


Figure 3.4: Marble Diagram of Time-window Aggregator

3.4.2.4 Flexible Component Boundaries

Interchangeability of Rx and DDS provides incredible flexibility to the developer in demarcating their component boundaries or points of data distribution. In fact, the points of distribution can be chosen at deployment-time. The imperative solution often does not possess a composable dataflow-oriented structure. Hence, more often than not, developers tend to over-commit to various interprocess communication mechanisms by hard-coding the dependency and eliminating the choice of an alternative mechanism. If scale-out or placement of these components on different machines is required, then this design is desirable, otherwise overcommitment to a specific distribution mechanism isolates the components and imposes "hard" component boundaries. The resulting structure is very rigid and hard to co-locate efficiently. For example, each query processor in our imperative solution is a component. Moving the functionality of one into another is intrusive and cannot be easily accomplished.

In Rx4DDS.NET, a stream of intermediate results can either be distributed over a DDS topic for remote processing or can be used for local processing by chaining stream operators. The details of whether the "downstream" processing happens locally or remotely can be abstracted away using the Dependency Injection pattern [111]. As a consequence,

component boundaries become more agile and the decision of data distribution need not be taken at design time but can be deferred until deployment.

In our implementation, developers may choose to distribute data over DDS by simply passing a DDS DataWriter to the `Subscribe` method. Alternatively, for local processing, a `Subject<T>` could be used in place of DDS DataWriter. The choice of a Subject versus a DataWriter is configurable at deployment-time.

Table 3.2 summarizes the key distinctions between our imperative and Rx4DDS.NET based solution for the case-study along each dimension of the challenges.

Table 3.2: Comparison of Our Imperative and Reactive Solutions

	Imperative Solution	**Reactive Solution**
State Management	Manual state management	State-management can be delegated to stream operators
Concurrency Management	Explicit management of low level concurrency	Declarative management of concurrency
Sliding Time-window Computation	Manual implementation of time window abstraction	Built-in support for both discrete and time-based window
Component Boundaries	Inflexible and hard component boundaries	Flexible and more agile component boundaries

3.4.2.5 Program Structure

The composability of operators in Rx allows us to write programs that preserve the conceptual high-level view of the application logic and data-flow. For example, Query 1 computes the *AggregateRunningData* for each player for 1 minute, 5 minutes, 20 minutes and full game duration, as shown in Listing 3.3.

In Listing 3.3, *player_streams* is a stream of streams (*e.g. IObservable<IGroupedObservable<String,PlayerData>>* comprises a stream for each player). Each player stream, represented by the variable *player_stream* is processed on a separate pooled thread by means of a single code statement, *ObserveOn(ThreadPoolScheduler.Instance)*. The *CurrentRunningData* for each player (*curr_running* stream in Listing 3.3) is computed by the function *CurrentRunningAnalysis()* and is subsequently used by *AggregateRunning*()* to compute the *AggregateRunningData* for each player for 1 minute, 5 minutes, 20 minutes and full

game durations, respectively. The use of *Publish() and Connect()* pair ensures that a single underlying subscription to *curr_running* stream is shared by all subsequent *AggregateRunning*()* computations otherwise the same *CurrentRunningData* will get re-computed for each downstream *AggregateRunning*()* processing pipeline.

Listing 3.3: Program Structure of Query 1

```
player_streams.Subscribe(player_stream =>
{
    var curr_running=
          player_stream
         .ObserveOn(ThreadPoolScheduler.Instance)
         .CurrentRunningAnalysis()
         .Publish();
   curr_running.AggregateRunningTimeSpan(1);
   curr_running.AggregateRunningTimeSpan(5);
   curr_running.AggregateRunningTimeSpan(20);
   curr_running.AggregateRunningFullGame();

   curr_running.Connect();
}
```

3.4.2.6 Backpressure

Integration of Rx with DDS allows us to leverage DDS QoS configurations and Real Time Publish Subscribe (RTPS) protocol to implement backpressure across the data reader-writer boundary. DDS offers a variety of QoS policies like *Reliability*, *History* and *Resource Limits* QoS policies that can be tuned to implement the desired backpressure strategy. *Reliability* QoS governs the reliability of data delivery between DataWriters and DataReaders. It can be set to either `BEST_EFFORT` or `RELIABLE` reliability. `BEST_EFFORT` configuration does not use any cpu/memory resources to ensure guaranteeded delivery

of data samples. `RELIABLE` configuration, on the other hand, uses ack/nack based protocol to provide a spectrum of reliability guarantees from strict to best-effort. Reliability can be configured with *History* QoS, which specifies how many data samples must be stored by the DDS middleware cache for the DataReader/DataWriter subject to *Resource Limits* QoS settings. It controls whether DDS should deliver only the most recent value (*i.e.*, history depth=1), attempt to deliver all intermediate values (*i.e.*, history depth=unlimited), or anything in between. *Resource Limits* QoS controls the amount of physical memory allocated for middleware entities. If `BEST_EFFORT` QoS setting is used, the DataWriter will drop the samples when the writer side queue (queue size determined by *History* and *Resource Limits* QoS) becomes full. We can use this strategy to cope with a slow subscriber or bursty input data rates if the application semantics support transient loss of data. On the other hand, if we use `RELIABLE` configuration, backpressure can be supported across the data reader-writer boundary. If the DataReader is not fast enough, it will start buffering the incoming samples upto a pre-configured limit (including unlimited, as configured using *History* and *Resource Limits* QoS) before throttling down the DataWriter in accordance with the reliability protocol semantics. However, this backpressure is only limited to work across two DDS entities. Local processing stages implemented in Rx .NET do not support backpressure. Hence, if operators with unbounded buffer sizes (e.g., *ObserveOn*, *Zip*) are used then we may observe an unbounded increase in queue lengths, arbitrarily large response times or out-of-memory exceptions.

Unlike Rx NET., the Reactive-Streams specification [99] implements a *dynamic push-pull model* for implementing backpressure between local processing stages. Their model can shift dynamically from being push-based (when the consumer can keep up with incoming data rate) to a pull-based model if the consumer is getting overwhelmed. The consumer specifies its "demand" using a backpressure channel to throttle the source. The producer can also use the "demand" specifications of downstream operators to perform intelligent load-distribution. In the future, we plan to integrate the Reactive-Streams specification

with DDS for end-to-end backpressure semantics.

3.4.3 Quantitative Evaluation of Rx4DDS.NET

To assess and compare the performance of Rx4DDS.NET library with that of the imperative approach, we implemented the DEBS 2013 Grand Challenge queries in an imperative style using C# so that both implementations used C#. Specifically, we compare the performance of our imperative and Rx4DDS.NET solutions under single threaded and multi-threaded query implementations. All the tests have been performed on a host with two 6-core AMD Opteron 4170 HE, 2.1 GHz processors and 32 GB RAM. The raw sensor stream was published by a DDS publisher by reading out the sensor data file in a separate process, while the queries were executed in another process by subscribing to the raw sensor stream published over DDS. Interprocess communication happens over shared-memory.

Table 3.3: Performance Comparison of Rx4DDS.NET over Imperative Solution

Rx Scheduler over Imperative Strategy	**query_1 %throughput difference**	**query_1 Std. Dev.**	**query_3 %throughput difference**	**query_3 Std. Dev.**	**query_1_3 %throughput difference**	**query_1_3 Std. Dev**
Rx single-thread over Imperative single-thread	-9.26	6.29	-4.3	7.3	1.19	5.67
Rx NewThread scheduler over Imperative NewThread-PlayerData Strategy	-6.7	4.44	-8.61	2.9	-3.75	3.28
Rx ThreadPool scheduler over Imperative ThreadPool-SensorData Strategy	-8.73	6.55	-5.47	4.56	-5.3	6.05
Rx Partitioner Eventloop over Imperative NewThread-SensorData Strategy	-13.87	7.04	-15.93	6.43	-10.87	3.67

We implemented the following strategies in the imperative solution for parallelizing query execution along the lines of available Rx schedulers: `SeparateThread`, `ThreadPool-SensorData`, `ThreadPool-PlayerData`, `NewThread-SensorData` and `NewThread-PlayerData`. In the `SeparateThread` strategy, `SensorData` is received on one thread while the entire query execution is offloaded to a separate thread (all player streams are processed on this separate thread). The `ThreadPool-SensorData`

strategy offloads the received SensorData on a threadpool such that each player's PlayerData calculation and subsequent player-specific processing happens on the threadpool. In the ThreadPool-PlayerData strategy, the PlayerData is calculated from SensorData on the thread that receives SensorData from DDS; thereafter the calculated PlayerData is offloaded to a threadpool for further player specific processing. The NewThread-SensorData strategy creates a designated thread for processing each player's data. The received SensorData is dispatched to its specific player thread, which computes that player's PlayerData and processes it further. The NewThread-PlayerData strategy also creates a separate thread for processing each player's data. However, the PlayerData is calculated from SensorData on the thread that receives data from DDS, which is then dispatched to the player-specific thread for further processing.

Figure 3.5 presents the performance of different imperative strategies over single-threaded implementations of query_1, query_3 and query_1_3 (runs both queries 1 and 3 together). Each query was run ten times and the error bars in the graphs denote one standard-deviation of values. For query_1, the average throughput gains per strategy over the single threaded implementation of query_1 are, respectively, 29% for the SeparateThread strategy, 35% for the ThreadPoolSensorData strategy, 23% for the ThreadPool-PlayerData strategy, 32% for the NewThread-SensorData strategy and 24% for the NewThread-PlayerData strategy. For query_3, the SeparateThread strategy shows an average of 40%, the ThreadPool-SensorData strategy shows an average of 12%, the ThreadPool-PlayerData strategy shows an average of 3%, the NewThread-SensorData shows an average of 15% and the NewThread-PlayerData strategy shows an average of 7% higher throughput than single-threaded implementation of query_3. For query_1_3, the SeparateThread strategy shows an average of 34%, the ThreadPool-SensorData strategy shows an average of 45%, the ThreadPool-PlayerData strategy shows an average of 42%, the NewThread-SensorData strategy shows an average of 47% and the NewThread-PlayerData strat-

egy shows an average of 39% higher throughput than single-threaded implementation of `query_1_3`.

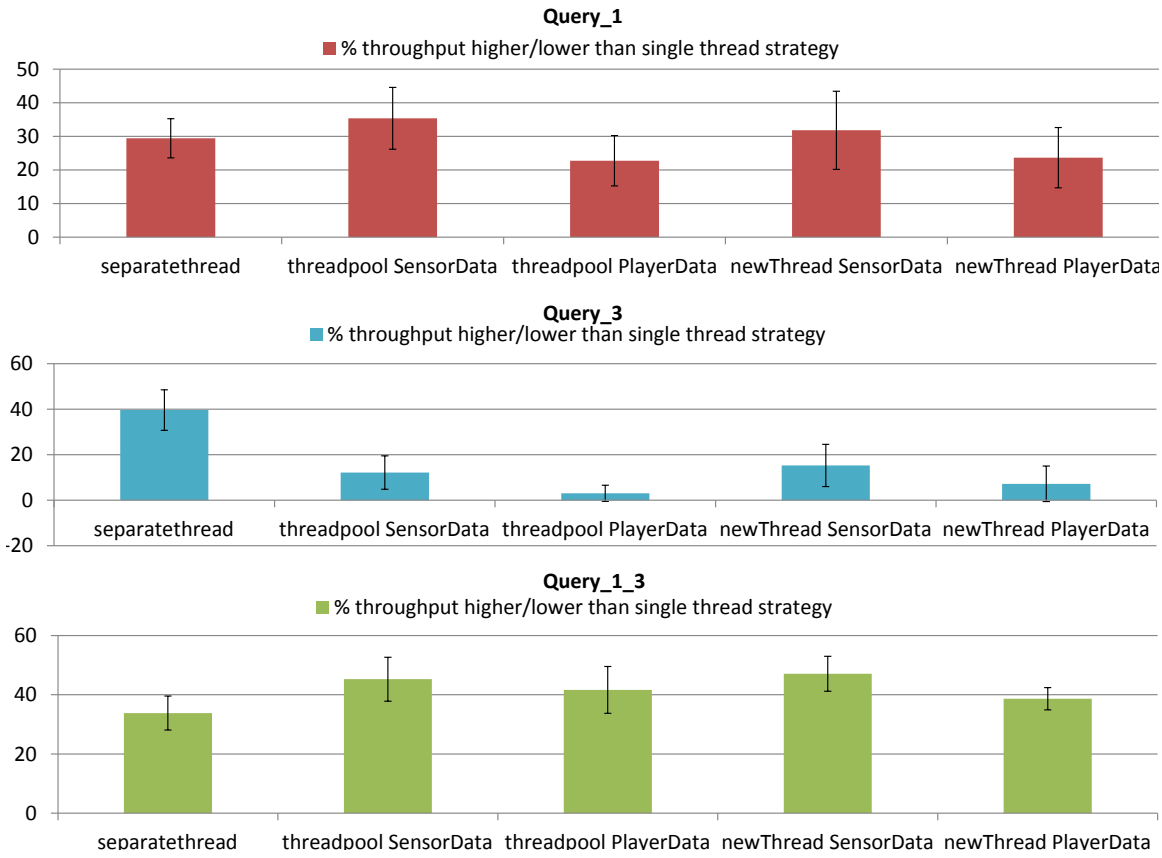


Figure 3.5: Performance of Imperative Strategies over Single Threaded implementation

To evaluate multi-threaded query implementations in our Rx4DDS.NET solution, we made use of the built-in Rx schedulers as shown in Listing 3.3. `ObserveOn` in Listing 3.3 causes each player stream(`player_stream`)'s data to get offloaded on the specified scheduler and all downstream processing of that player's `PlayerData` takes place on the specified scheduler passed to `ObserveOn`. Hence, in this case `PlayerData` gets calculated on the thread which receives data from DDS, but subsequent processing is offloaded to the specified `ObserveOn` scheduler. Rx offers many built-in schedulers such as the `EventLoopScheduler`, `NewThreadScheduler`, `ThreadPoolScheduler`, `TaskPoolScheduler`, etc. for parameterizing the concurrency of the application. `EventLoopScheduler` provides a dedicated thread which processes scheduled tasks in a

FIFO fashion; NewThreadScheduler processes each scheduled task on a new thread; ThreadPoolScheduler processes the scheduled tasks on the default threadpool while TaskPoolScheduler processes scheduled tasks on the default taskpool. Apart from using ObserveOn to process each player's PlayerData using a different scheduler, we also tested a variant test-case named Partitioner EventLoop, wherein the incoming SensorData is de-multiplexed and offloaded onto a player-specific EventLoop (each player has its own EventLoop) which will first calculate PlayerData and then perform further processing. This is similar to imperative NewThreadSensorData strategy, wherein each player thread is also responsible for calculating PlayerData from received SensorData.

Figure 3.6 presents the performance of different Rx schedulers over single-threaded implementations of query_1, query_3 and query_1_3. For query_1, EventLoopScheduler shows an average of 3%, NewThreadScheduler shows an average of 27%, ThreadPoolScheduler shows an average of 23%, TaskPoolScheduler shows an average of 25% and Partitioner EventLoop shows an average of 25% increase in throughput over single threaded implementation. For query_3, EventLoopScheduler shows an average of 20% lower performance, while NewThreadScheduler shows an average of 3%, ThreadPoolScheduler shows an average of 2%, TaskPoolScheduler shows an average of .04% and Partitioner EventLoop shows an average of 1% increase in performance over single threaded implementation. For query_1_3, EventLoopScheduler shows an average of 12%, NewThreadScheduler shows an average of 32%, ThreadPoolScheduler shows an average of 33%, TaskPoolScheduler shows an average of 30% and Partitioner EventLoop shows an average of 30% increase in performance over single threaded solution.

Query_1 processes each player's aggregate running data for all four time-windows, i.e. 1 min, 5 mins, 20 mins and full-game duration, which is updated for each input PlayerData sample. Query_1 shows inherent parallelism wherein each player's data

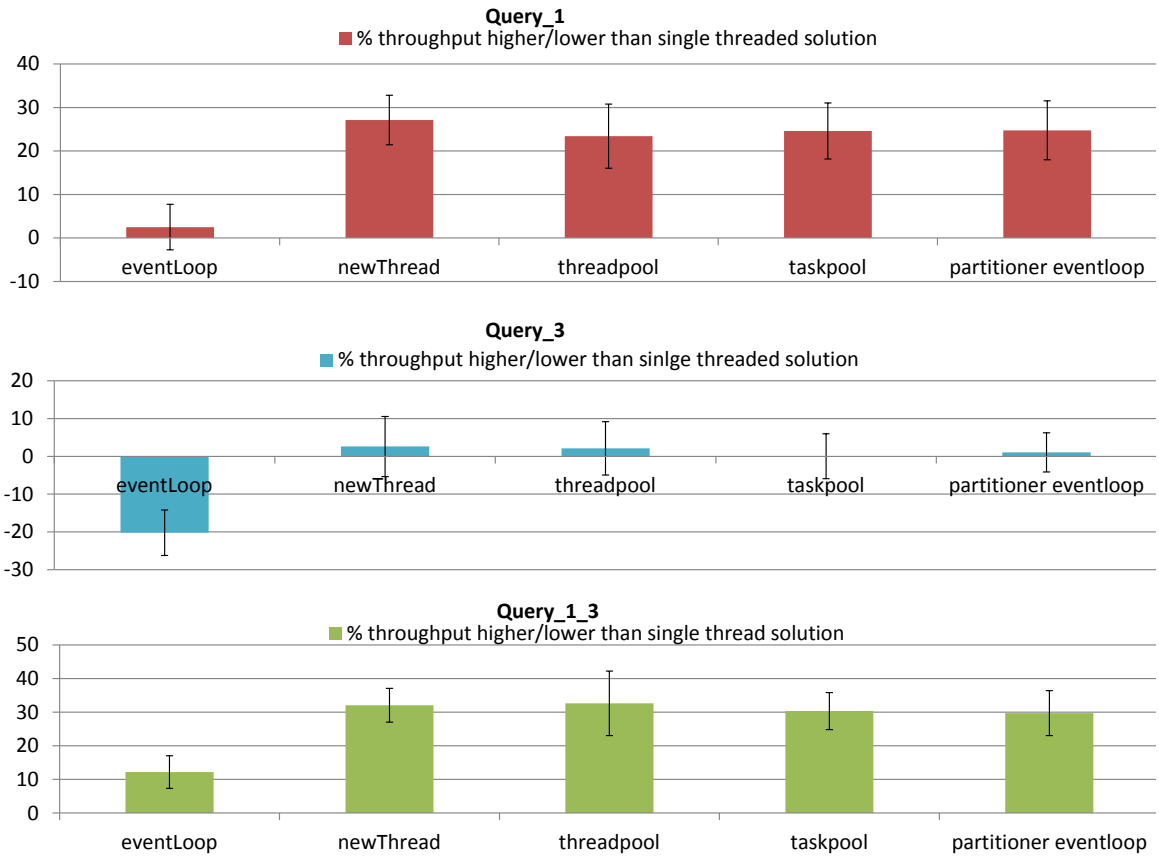


Figure 3.6: Performance of Different Rx schedulers over Single Threaded implementation

can be processed independently of each other in parallel. The multi-threaded implementation of `query_1` shows an increased performance with a maximum performance gain of 35% over single threaded implementation. `Query_3`, wherein each player's heatmap is calculated for all four time-windows(1min, 5mins, 10mins and full-game duration), also shows inherent parallelism in that each player's heatmap information can be calculated independently. However, `query_3` is only required to furnish an update after every 1 second (based on sensor timestamps) unlike `query_1` which furnishes an update for each input sample. Hence in case of `query_3` we find that introducing parallelism imposes a greater overhead without significant performance gain. Figure 3.7 presents the difference in input and output data-rates for `query_1` and `query_3` in our Rx4DDS.NET based implementation parameterized with `ThreadPoolScheduler`.

Table 3.3 compares the difference in the performance of Rx schedulers over its corre-

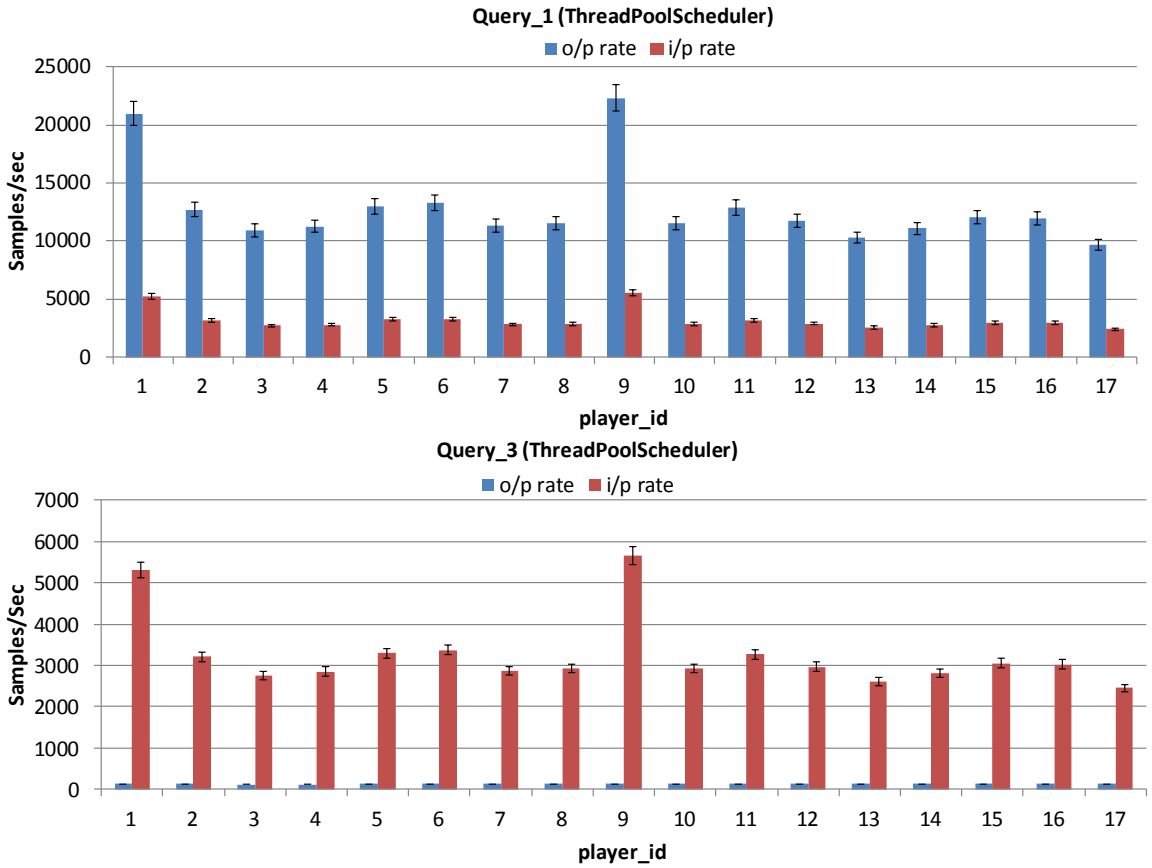


Figure 3.7: Input Vs Output data rate for Rx4DDS.NET implementation of Query_1 and Query_3 with ThreadPoolScheduler

sponding imperative solution strategy. While it is expected that the Rx library will impose some overhead, it offers several advantages due to its declarative approach towards system development, improved expressiveness and composability. Since Rx provides abstractions which make concurrency management declarative, testing different concurrency options for an application requires negligible effort. By changing a few lines of code we can test whether introducing parallelism provides increased performance gain (e.g., query 1) which is worth the added overhead or degrades it due to greater overhead (e.g., query 3). In contrast, gaining such insights by testing different implementation alternatives in the imperative approach was more complex, requiring a fair amount of changes in the code.

3.5 Conclusions

Reactive programming is increasingly becoming important in the context of real-time stream processing for big data analytics. While reactive programming supports event-driven design, most of the generated data must be disseminated from a large variety of sources (*i.e.*, publishers) to numerous interested entities, called subscribers while maintaining anonymity between them. These properties are provided by pub/sub solutions, such as the OMG DDS, which is particularly suited towards performance-sensitive applications. Bringing these two technologies together helps solve both the scale-out problem (*i.e.*, by using DDS) and scale-up using available multiple cores on a single machine (*i.e.*, using reactive programming).

This chapter described a concrete realization of blending the Rx .NET reactive programming framework with OMG DDS, which resulted in the Rx4DDS.NET library. Our solution was evaluated and compared against an imperative solution we developed using DDS and C# in the context of the DEBS 2013 grand challenge problem. The following lessons were learned, which allude to future directions of research:

- The integration of Rx with DDS as done in the Rx4DDS.NET library unifies the local and distributed stream processing aspects under a common dataflow programming model. It allows highly composable and expressive programs that achieve data distribution using DDS and data processing using Rx with a seamless end-to-end dataflow architecture that is closely reflected in the code.

- Our quantitative results indicate that Rx parameterizes concurrency and avoids application level shared mutable state that makes multi-core scalability substantially easier. We showed increase (up to 35%) in performance of `Query_1` by simply configuring the schedulers in Rx.

- Rx4DDS.NET library can be enhanced to map all available DDS features with Rx. Most commonly used stream processing constructs can be identified and made part

of this reusable library.

The Rx4DDS.Net framework and the implementation of the case study queries are available for download from https://github.com/rticommunity/rticonnextdds-reactive.

Chapter 4

LINEARIZE, PREDICT AND PLACE: MINIMIZING THE MAKESPAN FOR EDGE-BASED STREAM PROCESSING OF DIRECTED ACYCLIC GRAPHS

4.1 Introduction

The Internet of Things (IoT) paradigm has enabled a large number of physical devices or "things" equipped with sensors and actuators to connect over the Internet to exchange information. IoT applications typically involve continuous processing of data streams produced by these devices for the control and actuation of intelligent systems. In most cases, such processing needs to happen in near real-time to gain insights and detect patterns of interest. For example, in smart grids, energy usage data from millions of smart meters is continuously assimilated to identify critical events, such as demand-supply mismatch, so that corrective action can be taken to maintain grid stability [112]. Similarly, in video surveillance systems, video streams are continuously analyzed to detect traffic violations, such as jay walking and collisions [113].

Distributed Stream Processing Systems (DSPS) are used for scalable and continuous processing of data streams, such as sensor data streams produced by IoT applications [10]. In DSPS, an application is structured as a Directed Acyclic Graph (DAG), where vertices represent operators that process incoming data and directed edges represent the flow of data between operators. The operators perform user-defined computations on the incoming stream(s) of data. Storm [11], Spark [114], Flink [13], Millwheel [14], etc. are examples of widely used DSPSs. These systems have, however, been designed for resource-rich, cloud/cluster environments, where a master node distributes both data and computation over a cluster of worker nodes for large-scale processing (e.g., as in Storm). Using such cloud-centric solutions for IoT applications will require transferring huge amounts of sensor data produced by devices at the network edge to data-centers located at the core of

the network [15]. However, moving data over a bandwidth-constrained backhaul network incurs high latency cost, which makes such cloud-centric solutions infeasible for latency-sensitive IoT applications.

To address this concern, the edge computing paradigm has been proposed [19] to enable computations to execute near the source of data on low-cost edge devices and small-scale data-centers called cloudlets [20]. Edge-based stream processing systems, such as Frontier [115], Amazon Greengrass [116], Microsoft Azure IoT [117] and Apache Edgent [118], support data stream processing on multiple edge devices thereby reducing the need for costly data transfers. However, to meet the low response time requirements of latency-sensitive applications, it is also important to distribute the constituent operators of the DSPS over resource-constrained edge devices intelligently. An optimal placement approach should minimize the end-to-end response time or *makespan* of a stream processing DAG while trading-off communication costs incurred due to distributed placement of operators across edge devices, and interference costs incurred due to co-location of operators on the same edge device [37].

The above-mentioned edge-based stream processing platforms, however, provide only the mechanisms for IoT stream processing but not the solution for optimal operator placement. As such, framework-specific solutions for operator placement [38, 39, 40] are not directly applicable for edge-based stream processing. Likewise, existing framework-agnostic solutions [41, 42, 43, 37, 44] for operator placement make simplifying assumptions about the interference costs of co-located operators. These solutions do not consider the impact of incoming data rates and DAG-based execution semantics on the response time of an application. Due to these simplifying assumptions, their estimation of response time for a DAG execution is less accurate and the produced placement of operators on the basis of this response time estimation is less effective.

To address these limitations, we present our solution, in which we formulate the DAG placement as an optimization problem and solve it using a greedy heuristic algorithm. The

heuristic algorithm, however, needs to conduct a *what-if* analysis in making its placement decisions; i.e., it must predict the potential outcome on a (partial) DAG's latency if it were to take a certain operator placement decision. To enable the heuristic algorithm in making these look-ahead decisions, we develop a data-driven latency prediction model that incorporates DAG-based execution semantics, incoming data rates and operator processing times on edge nodes to estimate the latency of all paths in a DAG.

Learning a latency prediction model for arbitrary DAG structures, however, has significantly high overhead in terms of computational costs and model training time, and moreover, formulating the model training problem itself is very complex. This is compounded by the use of edge devices which are extremely sensitive to operator co-location due to their limited resources (e.g., with a single core). Therefore, our solution introduces a novel transformation of a DAG into an equivalent set of linear chains, which makes learning a latency prediction model for co-located operators significantly less expensive and easier to construct than learning a model for arbitrary DAG structures. Accordingly, to estimate the latency of a path in a given DAG, we first linearize the DAG into multiple linear chains and then use the latency prediction model for co-located linear chains to approximate the response time of the path in the original DAG. This learned latency prediction model is subsequently used by our greedy placement heuristic to inform its operator placement decisions that in turn minimizes the DAG makespan.

Our solution makes the following key contributions:

- **DAG Linearization:** We present an algorithm that transforms any given DAG into an equivalent set of linear chains in order to approximate the latency of a path in the DAG. This set includes the target path of the DAG, whose latency we are interested in approximating, in addition to other mapped linear chains. Upon execution of this equivalent set of linear chains, the latency of the path we are interested in is observed to be very close to the measured latency along that path when the original DAG structure is executed. The transformation algorithm considers both the split (or fork),

and join (or merge) points in DAGs.

- **Latency Prediction Model:** We present a model for predicting the 90th percentile latency of a linear chain of operators on the basis of its length (i.e., number of operators in the linear chain), the incoming data rate, the sum of execution times of all operators in the linear chain and a characterization of background load imposed by other co-located linear chains. For higher accuracy, we learn a separate prediction model for different numbers k of co-located chains present at an edge device. All the learned models have a prediction accuracy of at least 92%.

- **Greedy Placement Heuristic:** We present a greedy placement heuristic for makespan minimization, which leverages the DAG linearization algorithm and the latency prediction model to guide its placement decisions. Experimental results show that, compared with two baseline approaches, our placement heuristic significantly reduces both the prediction error *and* the number of edge nodes needed to deploy the DAG, while achieving low makespan.

The rest of this chapter is organized as follows: Section 4.2 gives a formal statement of the problem we are studying and provides a greedy heuristic to solve it. Section 4.3 presents our approach for DAG linearization and the latency prediction model to estimate the 90th percentile latency of co-located linear chains. These predictions are needed for our greedy heuristic. Section 4.4 presents experimental results to validate our solution. Section 4.5 presents related work and compares our operator placement solution to existing solutions for makespan minimization. Finally, Section 5.5 offers concluding remarks, lessons learned and outlines future work.

4.2 Problem Formulation and Heuristic Solution

In this section, we formally describe the operator placement problem by first introducing the models and assumptions. We then demonstrate the complex trade-offs between

communication and interference induced costs, and show the complexity. Finally, we present a greedy heuristic solution for the problem.

4.2.1 Models and Assumptions

A stream processing application can be represented by a Directed Acyclic Graph (DAG) $\mathbb{G} = (\mathbb{O}, \mathbb{S})$, where the set of operators $\mathbb{O} = \{o_i\}$ form the vertices of $\mathbb{G}$ and the set of data streams $\mathbb{S} = \{s_{ij}\}$, connecting the output of an operator o_i to its downstream operator o_j, form the directed edges of $\mathbb{G}$. Source operators, $\mathbb{O}_{src} \subset \mathbb{O}$, do not have any incoming edges and publish data into $\mathbb{G}$, i.e., $\mathbb{O}_{src} = \{o_i | \nexists s_{ji} \in \mathbb{S}, o_j \in \mathbb{O}\}$. Sink operators, $\mathbb{O}_{snk} \subset \mathbb{O}$, do not have any outgoing edges and receive the final results of $\mathbb{G}$, i.e., $\mathbb{O}_{snk} = \{o_i | \nexists s_{ij} \in \mathbb{S}, o_j \in \mathbb{O}\}$. All source and sink operators are *no-op* operators, i.e., they do not perform any computation. Each intermediate operator, i.e., $\mathbb{O}_{int} = \{o_i | o_i \notin \mathbb{O}_{src}, o_i \notin \mathbb{O}_{snk}\}$, performs computation and is characterized by its: 1) *execution time*, $\rho(o_i)$, which defines the average time interval of processing that o_i performs on every input message, and 2) *incoming rate*, $\lambda(o_i)$, which defines the rate at which o_i receives incoming messages.

The problem requires finding a *placement* $\mathscr{P} : \mathbb{O}_{int} \rightarrow \mathbb{E}$ for the set of intermediate operators $\mathbb{O}_{int}$ over a cluster of homogeneous edge nodes $\mathbb{E} = \{e_j\}$, such that the *makespan* of $\mathbb{G}$, specified by its maximum end-to-end latency[1] is minimized. Formally, the makespan of a graph $\mathbb{G}$ under a placement $\mathscr{P}$ is defined as:

$$\ell_{\mathscr{P}}(\mathbb{G}) = \max_{p \in \Pi} \ell_{\mathscr{P}}(p) \tag{4.1}$$

where Π represents the set of all paths in $\mathbb{G}$ and $\ell_{\mathscr{P}}(p)$ represents the latency of a path $p \in \Pi$ under placement $\mathscr{P}$. Suppose the path p has n intermediate operators, i.e., $p = (o_s, o_1, o_2, \ldots, o_n, o_k)$, where $o_s \in \mathbb{O}_{src}$, $o_k \in \mathbb{O}_{snk}$, and $o_i \in \mathbb{O}_{int}$ for $1 \leq i \leq n$. Given a

[1]While the model is flexible enough to incorporate different definitions of latency, we consider the 90th percentile end-to-end latency.

placement $\mathscr{P}$, the latency of path p can be expressed as:

$$\ell_{\mathscr{P}}(p) = \sum_{i=1}^{n} \omega_{\mathscr{P}}(o_i) + \sum_{i=1}^{n-1} d_{\mathscr{P}}(o_i, o_{i+1}) \tag{4.2}$$

Here, $\omega_{\mathscr{P}}(o_i)$ denotes the processing delay experienced by an operator o_i under placement $\mathscr{P}$, which may be higher than the operator's isolated execution time $\rho(o_i)$ due to potential interference with other co-located operators [119, 120]. Typically, the more co-located operators on the same edge node, the higher the processing delay. Also, $d_{\mathscr{P}}(o_i, o_{i+1})$ denotes the communication delay between an upstream operator o_i and its downstream operator o_{i+1} in the path. If o_i and o_{i+1} are placed on the same edge node under $\mathscr{P}$, then no network delay will be incurred. Otherwise, we assume a constant network delay c between any two edge nodes, i.e.,

$$d_{\mathscr{P}}(o_i, o_{i+1}) = \begin{cases} 0 & \text{if } \mathscr{P}(o_i) = \mathscr{P}(o_{i+1}) \\ c & \text{if } \mathscr{P}(o_i) \neq \mathscr{P}(o_{i+1}) \end{cases}$$

4.2.2 Cost Trade-Off and Complexity

The optimal solution to the makespan minimization problem described above depends on delicately exploiting the trade-off between the communication costs incurred by dependent operators located on different edge nodes and the interference cost due to the co-location of multiple operators on the same edge nodes. For example, consider a linear chain of n operators, i.e., $\langle o_1, o_2, \ldots, o_n \rangle$. On the one hand, placing each operator separately on different edge nodes has zero interference, but incurs a large communication cost between each pair of consecutive operators. On the other hand, placing all operators on one edge node incurs zero communication cost, but incurs a large processing delay due to performance interference among the co-located operators.

It turns out that, to place a set of n operators that form a linear chain, an optimal solution that balances the two costs can be obtained, as illustrated by the following dynamic

programming formulation:

$$\ell_i^* = \min_{i \leq j \leq n} \left(\omega_{i,j} + d(o_j, o_{j+1}) + \ell_{j+1}^* \right) \tag{4.3}$$

where ℓ_i^* denotes the optimal latency for placing the sub-chain $\langle o_i, o_{i+1}, \ldots, o_n \rangle$, and $\omega_{i,j} = \sum_{h=i}^{j} w_h$ denotes the cumulative latency of all operators in the sub-chain $\langle o_i, o_{i+1}, \ldots, o_j \rangle$ when they are co-located on the same edge node.

For placing general DAGs, however, the problem is more difficult. Many prior works (e.g., [121, 37, 44]) have shown the NP-hardness of the problem when there is a limited number of edge nodes. Here, we consider a model in which the amount of edge resource is unrestricted. Indeed, additional edge nodes can always be recruited in practice to accommodate the operators in order to minimize the response time of a streaming service, which is often a more important objective than the resource usage. Even in this case, the problem can be shown to be NP-hard via a simple reduction from a multiprocessor scheduling problem with communication delays[2] [122, 123]. Hence, we will focus on designing heuristic-based solution with the primary objective of minimizing the makespan while considering the number of edge nodes used as a secondary metric.

4.2.3 Greedy Placement Heuristic

In this section, we present a greedy placement heuristic for the makespan minimization problem formulated in Section 4.2.1. Algorithm 4 shows the pseudocode of the greedy heuristic. Specifically, the heuristic places the operators in the intermediate set $\mathbb{O}_{int}$ one after another in a Breadth-First Search (BFS) order (Line 2), which preserves the spatial locality of the dependent operators, thus reducing the communication cost. For each operator o_i to be placed, the heuristic tries two different options: (1) co-locate o_i with other

[2]The multiprocessor scheduling problem concerns mapping an arbitrary task graph with communication delays onto a set of m identical processors in order to minimize the makespan. The problem is NP-hard even when $m = \infty$ and all communication costs are uniform [122, 123]. This corresponds to a special case of our problem without any performance interference due to co-located operators, thus establishing the NP-hardness of the problem.

operators that have already been placed on an existing edge node (Lines 6-13); and (2) place o_i on a new edge node (Lines 14-19). In both options, the latencies of all paths that go through operator o_i will be estimated (Line 8 and Line 15). This estimation is done using our novel DAG linearization scheme and latency prediction model development described in Section 4.3. Note that, when operator o_i is co-located with other operators on an edge node, the latencies of all paths that go through those co-located operators also need to be estimated due to the interference caused by the placement of o_i [124]. Then, the resulting partial makespan for the sub-graph $\mathbb{G}_i = (\mathbb{O}_i, \mathbb{S}_i)$ that contains the set of operators $\mathbb{O}_i = \{o_1, o_2, \ldots, o_i\}$ up to operator o_i and the set of associated data streams $\mathbb{S}_i = \{s_{jk} | o_j \in \mathbb{O}_i, o_k \in \mathbb{O}_j\}$ will be updated (Line 9 and Line 16). Finally, the option that results in the minimum predicted makespan for $\mathbb{G}_i$ is selected for placing operator o_i (Line 23).

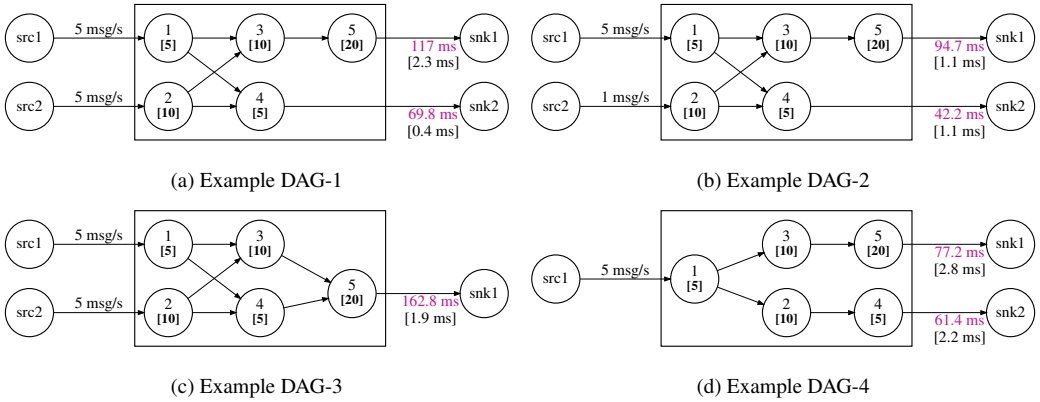


Figure 4.1: Impact of incoming data rate and DAG structure on the observed latency

Since each edge node hosts at least one operator, Algorithm 4 deploys at most n operators, where $n = |\mathbb{O}_{int}|$ denotes the number of intermediate operators in the graph. Let f denote the total number of paths in the graph. As the placement of each operator examines at most n edge nodes and updates at most f paths, the complexity of the algorithm is therefore $O(n^2 f)$.

Algorithm 4: Greedy Placement Heuristic

Input: Operator graph $\mathbb{G} = (\mathbb{O}, \mathbb{S})$
Output: A placement $\mathscr{P}_{\text{greedy}}$ of the intermediate operator set $\mathbb{O}_{int} \in \mathbb{O}$ onto a set $\mathbb{E}$ of edge nodes

begin
  Reorder all operators in the intermediate set $\mathbb{O}_{int} = \{o_1, o_2, \ldots, o_n\}$ in BFS order, where $n = |\mathbb{O}_{int}|$;
  $\mathbb{E} \leftarrow \emptyset$;
  for each operator o_i $(i = 1 \ldots n)$ **do**
    $\ell^* \leftarrow \infty$ and $j^* \leftarrow 0$;
    `// try to place on each existing edge node`
    for each edge node $e_j \in \mathbb{E}$ **do**
      $\mathscr{P}(o_i) \leftarrow e_j$;
      Predict and compute latencies of all paths that contain operators co-located in e_j;
      Update the partial makespan $\ell_{\mathscr{P}}(\mathbb{G}_i)$;
      if $\ell_{\mathscr{P}}(\mathbb{G}_i) < \ell^*$ **then**
        $\ell^* \leftarrow \ell_{\mathscr{P}}(\mathbb{G}_i)$ and $j^* \leftarrow j$;
      end
    end
    `// try to place on a new edge node`
    $\mathscr{P}(o_i) \leftarrow e_{|\mathbb{E}|+1}$;
    Compute latencies of all paths that contain operator o_i;
    Update the partial makespan $\ell_{\mathscr{P}}(\mathbb{G}_i)$;
    if $\ell_{\mathscr{P}}(\mathbb{G}_i) < \ell^*$ **then**
      $\ell^* \leftarrow \ell_{\mathscr{P}}(\mathbb{G}_i)$ and $j^* \leftarrow j$;
    end
    if $j^* = |\mathbb{E}| + 1$ **then**
      $\mathbb{E} \leftarrow \mathbb{E} \bigcup \{e_{|\mathbb{E}|+1}\}$; `// start a new edge node`
    end
    $\mathscr{P}_{\text{greedy}}(o_i) \leftarrow e_{j^*}$;
  end
end

4.3 Developing a Latency Prediction Model

In this section, we describe the development of a data-driven latency prediction model that is used by our greedy placement heuristic (see Section 4.2.3). We first justify the rationale for selecting the input feature vectors used in building the model. We then show a novel approach to simplify the training for DAGs. To that end, we explain our DAG linearization approach for transforming any arbitrary DAG into an equivalent set of linear chains and the *k*-chain co-location latency prediction model used for predicting the latency of multiple co-located linear chains.

4.3.1 Critical Considerations for Model Building

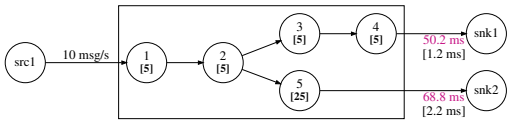
When training any data-driven model, it is important to identify the key input features that must be used. In our case, the goal is to train a latency prediction model that can predict the impact on the DAG latency if one were to co-locate an operator of that DAG on an edge node when one or more other operators (belonging to the same or other DAGs) already exist on that node by incorporating the impact of performance interference caused by co-location.

Existing solutions make simplifying assumptions to estimate the cost of interference. Some solutions [41, 42, 43] assume that the execution time of each operator becomes the sum of execution times of all co-located operators when the underlying physical node is single core and uses round robin scheduling. Other solutions [37, 44] ignore the impact of operator co-location and assume constant execution time. These solutions do not consider the impact of incoming data rate and DAG structure imposed execution semantics, both of which have a significant impact on the observed latency as illustrated in Figure 4.1.

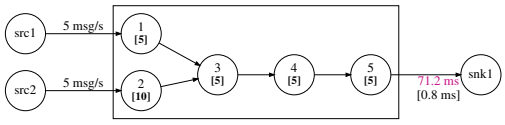
To that end, we first set out to pinpoint the key considerations when building a model. We used a single core Beagle Bone Black (BBB) board [125] to run the DAGs depicted in Figure 4.1. All intermediate vertices which process incoming data, namely `vertex-1` to `vertex-5`, were hosted on the same BBB board while the source and sink vertices were hosted on a separate 2.83 GHz Intel Q9550 quad core server. Source vertices send 64 Byte, time-stamped messages at a configurable rate shown on their outgoing edges. Intermediate vertices perform a configurable amount of processing on each incoming message. This execution time of intermediate vertices, measured in milliseconds, is depicted within brackets below the `vertex-ID`. For intermediate vertices with multiple incoming edges, we assume interleaving semantics [42] wherein the vertex performs processing whenever it receives a message on any of its incoming edges. Sink vertices log the time-stamp of reception of messages after being processed by the intermediate vertices.

Each DAG was allowed to execute for two minutes in an experimental run. Average

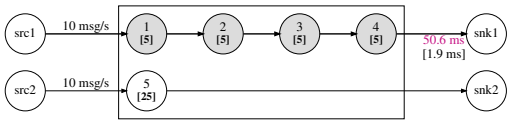
90th percentile latency and standard deviation (shown in brackets) recorded by each sink vertex across 5 runs are shown along the incoming edge of the sink vertex. This 90th percentile latency value recorded by a sink vertex implies that 90 percent of all messages received along all the paths which end at that sink vertex were observed to have an end-to-end path latency below the 90th percentile value. For example, in DAG-1 (Figure 4.1a), 90 percent of all messages received along paths $\langle src1, 1, 3, 5, snk1 \rangle$ and $\langle src2, 2, 3, 5, snk1 \rangle$, which end at *snk1*, were observed to have an end-to-end latency below 117 ms (on average across 5 experimental runs). Similarly, 90 percent of all messages received along paths $\langle src1, 1, 4, snk2 \rangle$ and $\langle src2, 2, 4, snk2 \rangle$, which end at *snk2*, were observed to have an end-to-end latency below 69.8 ms (again on average across 5 runs). The makespan (i.e., response time) of a DAG is the maximum 90th percentile latency across all paths, which is 117 ms for DAG-1.



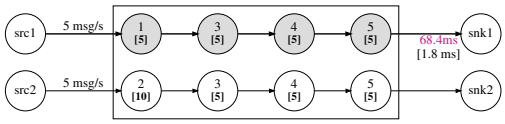
(a) Original DAG with fork operator



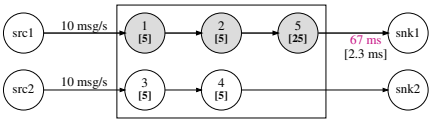
(a) Original DAG with join operator



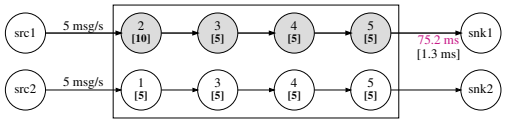
(b) Linearization for path-1



(b) Linearization for path-1



(c) Linearization for path-2



(c) Linearization for path-2

Figure 4.2: Linearization rule for fork operator

Figure 4.3: Linearization rule for join operator

Based on our experiments, we made the following two critical observations:

- **Impact of data rate (i.e., publishing rate):** DAG-1 in Figure 4.1a and DAG-2 in Figure 4.1b are the same, except for the publishing rate of source vertex *src2*, which generates data at 5 messages/sec in DAG-1 and at 1 message/sec in DAG-2. In DAG-1, the 90th percentile latency at sink vertex *snk1* is 117 ms and at sink vertex *snk2*

is 69.8 ms. However, due to the lower publishing rate of *src2* in DAG-2, sink vertices *snk1* and *snk2* show lower 90th percentile latencies of 94.7 ms and 42.2 ms, respectively.

- **Impact of DAG Structure:** DAG-1, DAG-3 and DAG-4 in Figure 4.1a, Figure 4.1c and Figure 4.1d, respectively, are composed of the same set of intermediate vertices, although they are structurally different. All three DAGs show markedly different response times on account of this difference in their DAG structures. The simplifying assumptions of prior works that do not consider DAG structure imposed execution semantics, such as assuming constant execution time or sum of execution times of all co-located vertices, can respectively underestimate or overestimate a DAG's makespan. For example, if we assume constant execution time for each vertex, the latency of path $\langle src1, 1, 3, 5, snk1 \rangle$ in DAG-4 will be $\sim$35 ms, which is much less than the observed path latency of 77.2 ms. Similarly, if we assume each vertex's execution time to be the sum of execution times of all 5 co-located vertices given that we are using a single core BBB board, then the path latency would be $\sim$150 ms, which is twice the experimentally observed path latency of 77.2 ms.

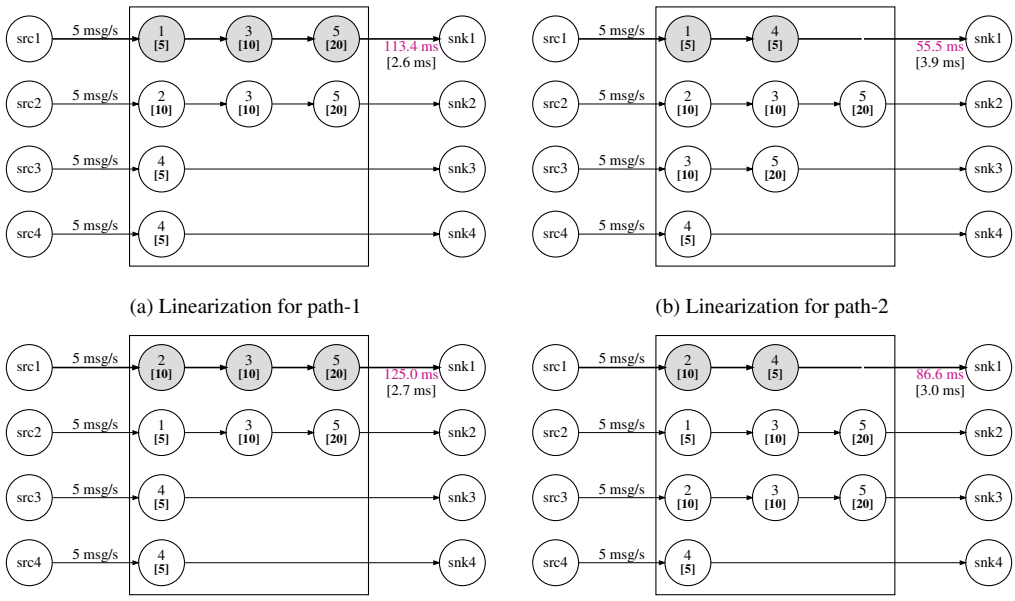
Thus, it is critical to take both the data rate and the DAG-based execution semantics into account in order to accurately estimate the response time of a DAG, which is used in our approach for building a latency prediction model.

4.3.2 DAG Linearization Transformation Rules

Since it is expensive to train a latency prediction model for arbitrary DAG structures, we propose a linearization-based approach, which transforms any given DAG into multiple sets of linear chains, whose latencies will then be predicted to approximate the end-to-end latencies of the original DAG structure. Due to the simplicity of the linear structures, the proposed approach is able to significantly reduce the space over which the latency prediction model needs to be learned. We arrived at these transformation rules based on

multiple different empirical observations.

Suppose the operators in a connected graph[3] $\mathbb{G}'$ are all co-located on one edge node, and suppose $\mathbb{G}'$ contains a collection $\{p_1, p_2, \ldots, p_f\}$ of f paths from its source operator(s) to its sink operator(s). The linearization scheme first transforms graph $\mathbb{G}'$ into f sets of linear chains, denoted as $\{L_1(\mathbb{G}'), L_2(\mathbb{G}'), \ldots, L_f(\mathbb{G}')\}$. For each $1 \leq k \leq f$, the set $L_k(\mathbb{G}')$ contains f linear chains in it, including a target linear chain corresponding to the path p_k in the original DAG, as well as $f-1$ auxiliary linear chains to simulate the performance interference for path p_k. The latency prediction model (Section 4.3.3) is then used to predict the latency of the target path p_k in each set $L_k(\mathbb{G}')$. Finally, the predicted latencies for all the paths in $\{p_1, p_2, \ldots, p_f\}$ that share the same sink operator are averaged to approximate the end-to-end latency for messages that exit that sink operator in graph $\mathbb{G}'$.



(a) Linearization for path-1

(b) Linearization for path-2

(c) Linearization for path-3

(d) Linearization for path-4

Figure 4.4: Linearization and latency prediction results for the DAG shown in Figure 4.1a.

We now present our approach to transform graph $\mathbb{G}'$ into f sets of linear chains $\{L_1(\mathbb{G}'), L_2(\mathbb{G}'), \ldots, L_f(\mathbb{G}')\}$. Algorithm 5 shows the pseudocode of the linearization procedure. Since an operator in the original graph $\mathbb{G}'$ may be replicated in a set $L_k(\mathbb{G}')$, we first com-

[3]If a graph contains several connected components, the linearization can be done separately for each connected component.

Algorithm 5: DAG Linearization

Input: Operator graph $\mathbb{G}' = (\mathbb{O}', \mathbb{S}')$ that contains f paths $\{p_1, p_2, \ldots, p_f\}$ from its source operator(s) to its sink operator(s).

Output: f sets of linear chains $\{L_1(\mathbb{G}'), L_2(\mathbb{G}'), \ldots, L_f(\mathbb{G}')\}$, each with a target path p_k from $\mathbb{G}'$ whose latency will be predicted.

```
1  begin
2      Identify the set O'_src of source operators of G';
3      t_i ← 1 for ∀o_i ∈ O'_src, and t_i ← 0 for ∀o_i ∈ O'\O'_src;
4      Initialize an empty queue Q ← ∅;
5      Q.enqueue(O'_src);
6      while Q ≠ ∅ do
7          o_i ← dequeue(Q);
8          for each o_j ∈ o_i.childen() do
9              t_j ← t_j + 1;
10             Q.enqueue(o_j);
11         end
12     end
13     for k = 1 to f do
14         p'_h ← p_h, for ∀1 ≤ h ≤ f;
15         t'_i ← t_i, for ∀o_i ∈ O';
16         t'_i ← t'_i − 1, for ∀o_i ∈ p'_k;
17         for each p'_h ∈ {p'_1, p'_2, ..., p'_f}\{p'_k} do
18             for each o_i ∈ p'_h do
19                 if t'_i = 0 then
20                     remove o_i from p'_h;
21                 else
22                     t'_i ← t'_i − 1;
23                 end
24             end
25         end
26         L_k(G') ← {p'_1, p'_2, ..., p'_k};
27     end
28 end
```

pute the number of times each operator is replicated. To that end, we describe below the linearization rules for two types of operators in a DAG structure, namely, fork and join operators.

- *Fork operator*: All paths in a DAG that originate from a fork operator can be executed concurrently. Hence, we can reason about these paths as independent linear chains. The fork operator only executes once, so it is included in one of the multiple paths that originate from the fork vertex. Figure 4.2 illustrates this rule.

- *Join operator*: Join operators have multiple incoming edges. Therefore, we can

argue that the join operator and all its downstream operators execute as many times as the number of incoming edges of the join vertex. Hence, a join operator and all downstream operators are replicated into multiple linear chains. Figure 4.3 illustrates this rule.

Generalizing the rules above, for each operator o_i in $\mathbb{G}'$, we can obtain the number of times t_i it should appear in any set $L_k(\mathbb{G}')$ of linear chains as the total number of paths from the source operator(s) of $\mathbb{G}'$ to o_i. This can be computed by a simple breadth-first traversal of the graph (Lines 2-12).

The algorithm then constructs the set $L_k(\mathbb{G}')$ of linear chains for each $1 \leq k \leq f$ (Lines 13-27). Specifically, it first adds the target path p_k into the set $L_k(\mathbb{G}')$, and then examines the operators from the remaining paths in sequence. If an operator o_i has already appeared t_i times from the previously examined paths, it will be removed from the current and subsequent paths. This ensures the correct number of replicas for each operator in the set. The complexity of the algorithm is $O(f^2|\mathbb{O}'|)$.

Figure 4.4 shows the linearization results for the DAG shown in Figure 4.1a, which contains two source operators, two sink operators, and four different paths: $\langle 1,3,5\rangle$, $\langle 2,3,5\rangle$, $\langle 1,4\rangle$ and $\langle 2,4\rangle$. Therefore, four corresponding sets of linear chains are constructed as shown in Figure 4.4a to Figure 4.4d. In each set, the chain highlighted in grey represents the target path whose latency will be predicted, and the other chains represent auxiliary paths to simulate the performance interference.

4.3.3 Training the *k*-Chain Co-location Latency Prediction Model

In this section, we describe the k-chain co-location latency prediction model we trained for predicting the latencies of k co-located linear chains. Given a set of k linear chains, our latency prediction model first employs a classification model to determine if the placement of these k linear chains at an edge device is feasible; i.e., the placement does not saturate an edge node's resources. In case of resource saturation, the observed latency values become

significantly high and unpredictable. If the classification model predicts that the placement is feasible, then a regression model is used to predict each linear chain's 90th percentile latency.

Both the classification and regression models for a k-chain co-location take the same set of 7 input features. Of these 7 input features, the first 3 features characterize the foreground (target) linear chain, or the chain under observation, and the remaining 4 features characterize the background load imposed by the set of background (auxiliary) chains co-located along with the foreground chain. These input features are described below, where c_f denotes the foreground linear chain and $\mathbb{C}_B$ denotes the set of background linear chains.

- $n(c_f)$: number of operators in c_f;
- $\sum_{o \in c_f} \rho(o)$: sum of execution intervals of operators in c_f;
- $\lambda(c_f)$: incoming data rate for c_f;
- $\sum_{c_b \in \mathbb{C}_B} n(c_b)$: sum of number of operators in all background chains;
- $\sum_{c_b \in \mathbb{C}_B} \sum_{o \in c_b} \rho(o)$: sum of execution intervals of all operators in all background chains;
- $\sum_{c_b \in \mathbb{C}_B} \lambda(c_b)$: sum of incoming data rates over all background chains;
- $\sum_{c_b \in \mathbb{C}_B} \lambda(c_b) \cdot \sum_{o \in c_b} \rho(o)$: sum of the product of $\lambda(c_b)$ and $\sum_{o \in c_b} \rho(o)$ over all background chains.

The classification model takes these input features and outputs a binary value: 0 to indicate that the placement is feasible and 1 otherwise. The regression model outputs the predicted 90th percentile latency of the foreground chain co-located with a given background load. For $k = 1$, i.e., only one chain exists, the four input features characterizing the background load is set to 0.

We used neural networks for learning both the classification and regression models. For higher accuracy, we learned separate models for different numbers k of co-located chains.

The neural networks comprise an input layer, one or more hidden layers and an output layer, where each layer is composed of nodes called neurons [81]. Neurons belonging to different layers are interconnected with each other via weighted connections. The input layer feeds the input features to the network. Neurons belonging to the intermediate and output layers sum the incoming weighted signals, apply a bias term and an activation function to produce their output for the next layer. The architecture of a neural network, i.e., the number of hidden layers and number of neurons per hidden layer, choice of the activation function, regularization factor, and the solver greatly determine the accuracy of the learned model.

Typically, learning curves [82] are plotted to understand the performance of a neural network and to guide the selection of various parameters, such as number of layers, number of neurons per layer, regularization factor, etc. A learning curve shows the training and validation errors as functions of the training data size. If the learning curve shows that the training error is low, but the validation error is high, the model is said to suffer from high variance [82], i.e., it is over-fitting the training data and may not generalize well. If the learning curve shows high training and validation errors, the model is said to suffer from high bias [82], i.e., it fails to learn from the data or is under-fitting the data.

A neural network model for which both the training and validation errors converge to a low value is selected. Such a model neither over-fits (low variance) nor under-fits (low bias) the data and is expected to perform well. We plotted learning curves for different architectures for each value of k in both classification and regression models to help us select a model with low bias and variance. Section 4.4 shows the learning curves and accuracy results for the selected k-chain co-location classification and regression models.

4.4 Experimental Validation

In this section, we present experimental results to validate our DAG linearization-based approach for predicting the latency of arbitrary DAG structures and to evaluate our greedy placement heuristic for minimizing the DAG makespan, which relies on the latency predic-

tion approach. We describe our experiment testbed first, followed by the accuracy results for the learned k-chain co-location latency prediction models and performance results for our greedy placement heuristic.

4.4.1 Experiment Testbed and Setup

Our experiment testbed comprises 8 Beagle Bone Black (BBB) boards running Ubuntu 18.04, where each board has one AM335x 1GHz ARM processor, 512 MB RAM and 1 Gb/s network interface card. Intermediate vertices are hosted on BBB boards whereas the source and sink vertices are hosted on a separate quad-core 2.83 GHz Intel Q9550 server with 8GB RAM and 1Gb/s network interface card, also running Ubuntu 18.04. We used RTI DDS [9], a peer-to-peer topic-based publish-subscribe messaging system to model the directed edges interconnecting the vertices. Each edge is implemented as a DDS topic over which messages are sent and received by the upstream and downstream vertices, respectively. Source vertices send 64 Byte time-stamped messages at a configurable publishing rate λ up to 20 messages/second. Intermediate vertices process each incoming message by performing recursive fibonacci computations for a configurable execution interval ρ, whose value can either be 1 ms, 5 ms, 10 ms, 15 ms or 20 ms. Sink vertices log the time-stamp of reception of processed messages to compute the end-to-end latencies. To ensure the fidelity of experimental results, a DAG is executed for two minutes. Some initial end-to-end latency values logged by a sink vertex are ignored while computing the 90th percentile latency value, since they are observed to be high on account of initialization and connection setup.

4.4.2 Validating the k-Chain Co-location Latency Prediction Model

As discussed in Section 4.3, to predict the end-to-end path latency of k co-located linear chains, we rely on two prediction models: k-chain co-location classification and k-chain co-location regression models. First, the classification model is used to assess if the placement

of given k linear chains is feasible. If the classification model predicts that the placement is feasible, then the k-chain co-location regression model is used to predict each linear chain's 90th percentile latency.

Empirically, we observed that a BBB's CPU gets saturated if more than 12 vertices are placed on the node. Therefore, to create the training dataset for k-chain co-location models, the number of vertices in each linear chain is randomly chosen such that the sum of number of vertices across all k chains is not more than 12. The execution interval ρ for each vertex is uniformly randomly chosen from the set {1ms, 5ms, 10ms, 15ms, 20ms} and the incoming data rate for each chain is uniformly randomly chosen in the range [1, 20] messages/sec. We learned k-chain co-location models for k up to 4. As k increases further, the range of possible values for the input features increases and more training data is needed to get good prediction accuracy. We ran 600, 1500, 1950 and 1950 experiments for $k = 1, 2, 3$ and 4, respectively. Additionally, a separate validation dataset was created by running 50 experiments for each k.

When an experiment for k-chain co-location is run, we get k latency data-points, one corresponding to each linear chain. Therefore, the training dataset size becomes k times the number of experiments, i.e., 600, 3000, 5850 and 7800 data-points for $k = 1, 2, 3$ and 4, respectively, as shown in Table 4.1. To learn the classification model, data-points for which the observed 90th percentile latency is greater than twice the sum of execution intervals of all vertices in all k chains are categorized as infeasible placements. While the entire dataset comprising both feasible and infeasible data-points is used for training the classification model, the regression model is trained only over the feasible subset of data-points. For example, in case of $k = 1$, all 600 data-points are used for training the classification model. Out of these, 186 data-points were categorized as infeasible placements and the remaining 416 data-points as feasible placements. To learn the regression model for $k = 1$, we therefore used the 416 feasible data-points as shown in Table 4.2.

We tested different neural network architectures for k-chain co-location classification

Table 4.1: Accuracy of k co-location classification model

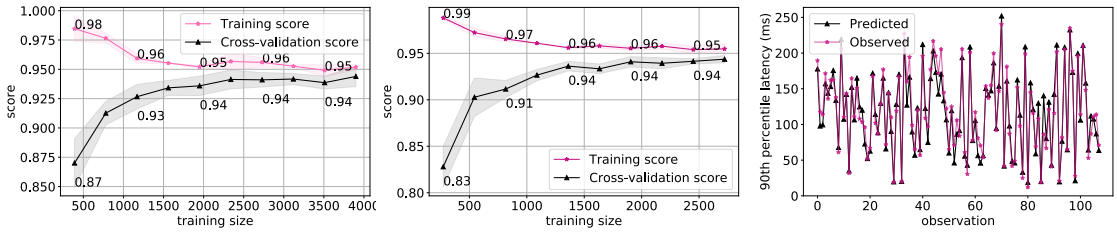
k	#datapoints (training)	accuracy (training)	accuracy (test)	#datapoints (validation)	accuracy (validation)
1	600	.98	.97	50	.98
2	3000	.98	.96	100	.98
3	5850	.96	.96	150	.97
4	7800	.96	.95	200	.94

Table 4.2: Accuracy of k co-location regression model

k	#datapoints (training)	accuracy (training)	accuracy (test)	#datapoints (validation)	accuracy (validation)
1	416	.99	.99	38	.99
2	2268	.98	.98	84	.96
3	4083	.96	.95	108	.94
4	5376	.95	.94	128	.92

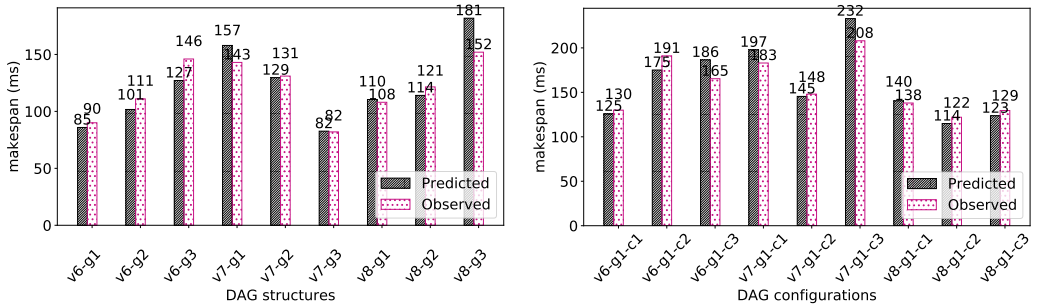
and regression models. For classification, we found that a neural network with one hidden layer composed of 50 neurons performs well for $k = 1$ and $k = 2$, while a neural network with one hidden layer composed of 100 neurons performs well for $k = 3$ and $k = 4$. For regression, we found that a neural network with one hidden layer composed of 50 neurons performs well for $k = 1$. However, for $k = 2, 3$ and 4, a neural network regressor with three hidden layers composed of 50 neurons each gives good accuracy. For all the models, Rectified Linear Units (ReLu) was used as the activation function, limited memory Broyden-Fletcher-Goldfarb-Shanno (lbfgs) was used as the solver and L2 regularization factor was set to 0.1. Figure 4.5a and Figure 4.5b show the learning curves for $k = 3$ classification and $k = 3$ regression models, respectively. Here, we see that the chosen neural network architectures have low bias and variance since the training and cross-validation errors converge to a reasonably low value that is at most 8%. Therefore, the model neither over-fits nor under-fits the training data and is expected to generalize well.

Table 4.1 and Table 4.2 show the performance of trained k-chain co-location classification and regression models on training, test and validation datasets. We used 90% of



(a) Classification learning curve for $k = 3$ (b) Regression learning curve for $k = 3$ (c) Validation results for $k = 3$

Figure 4.5: Performance of k-chain co-location latency prediction model for $k = 3$



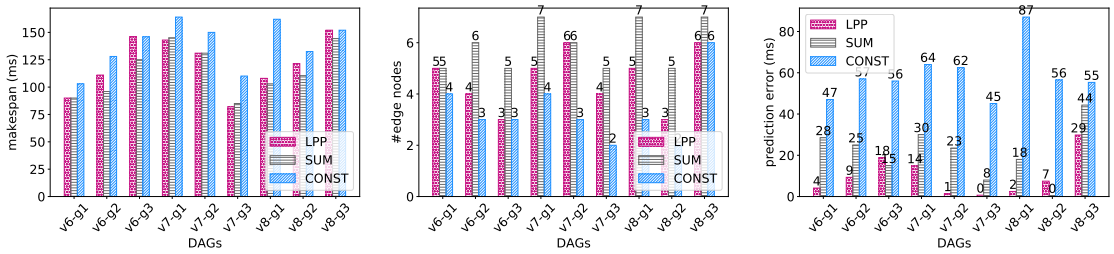
(a) LPP predicted vs. observed makespan for different DAG structures (b) LPP predicted vs. observed makespan for different DAG configurations

Figure 4.6: LPP makespan prediction accuracy (for various randomly generated DAGs)

data-points for training and the remaining 10% for testing. We observed that all learned models have an accuracy of at least 92%. Figure 4.5c shows the performance of k-chain co-location regression model on the validation dataset for $k = 3$. We see that the predicted latencies track the experimentally observed values closely and the average difference between the predicted and observed latencies over all 108 validation data-points is 10.8 ms.

4.4.3 Performance Evaluation of the LPP Approach

To assess the performance of our Linearize, Predict and Place (LPP) operator placement solution for makespan minimization, we generated 9 random test DAGs with three different structures per intermediate vertex count of 6, 7 and 8. These nine test DAGs were generated using Fan-In-Fan-Out [126] method for task-graph generation. We refer to the DAG structure 1 with 6 intermediate vertices as `v6-g1`, which also exemplifies the naming convention used to identify these test DAGs. For our experiments, we set a constant network delay of 10ms for communication between any two edge nodes.



(a) Observed makespan (b) Number of edge nodes (c) Prediction error

Figure 4.7: Comparison of LPP with SUM and CONST approaches (for various randomly generated DAGs)

4.4.3.1 LPP Prediction Results

Figure 4.6a compares the makespan predicted by LPP with experimentally observed makespan upon DAG execution, for the same test DAG structure, parameter configuration (data rate and execution intervals of all vertices) and operator placement. We see that the LPP approach based on DAG linearization is able to make a fairly good prediction for the makespan of all 9 test DAGs with an average prediction error of 9.8 ms. For the same DAG structure, different DAG parameter configurations, such as incoming data rate and execution intervals of the constituent vertices, also impact DAG latency. Figure 4.6b shows the variation in the makespan of the same DAG structure across three different DAG configurations *c1*, *c2* and *c3* (which are also randomly generated). The LPP approach incorporates these differences in DAG parameter configurations while predicting a DAG's makespan. As seen in Figure 4.6b, LPP is able to make good predictions for different parameter configurations with a mean prediction error of 11 ms across these 9 DAGs.

4.4.3.2 LPP Placement Results

We compared LPP with two solution variations: (1) *SUM:* Similar to LPP, this approach uses the concept of DAG linearization to approximate arbitrary DAG structures and the k-chain co-location classification model to assess if the placement of k linear chains is feasible. However, unlike the LPP approach, which uses k-chain co-location regression model to predict the path latencies, the SUM approach makes the simplifying assumption

that a vertex's execution time becomes the sum of all co-located vertices' execution times; and (2) *CONST:* This approach makes the simplifying assumption that a vertex's execution time remains unchanged despite co-location.

Figure 4.7a compares the makespan of the placements produced by the LPP, SUM and CONST approaches. Figure 4.7b compares the number of edge nodes used in the placements produced by the LPP, SUM and CONST approaches. Figure 4.7c shows the error in predicting the makespan of a DAG by the LPP, SUM and CONST approaches. We see that, in many cases, the CONST approach underestimates the path latencies and inaccurately favors co-locating the vertices, which results in a higher makespan in comparison to LPP as seen in Figure 4.7a. The placement produced by CONST uses fewer edge nodes than LPP as seen in Figure 4.7b since CONST inaccurately favors co-location due to underestimation of path latencies. The makespan predicted by CONST is also much lower than the observed makespan upon DAG execution, which results in high prediction errors as seen in Figure 4.7c.

The SUM approach, on the other hand, overestimates path latencies and inaccurately favors distributed placement of operators in many cases, which results in more edge nodes being used in comparison to LPP, as seen in Figure 4.7b. Due to such overestimation, the error in makespan prediction by the SUM approach is higher than that of the LPP approach, which uses a latency prediction model, as seen in Figure 4.7c. The observed makespans produced by LPP and SUM are similar as seen in Figure 4.7a, but LPP achieves this with less amount of edge resources. Overall, these results show that LPP makes a more accurate prediction of the path latencies and thereby a more effective placement of operators than the other approaches that make simplifying (either overestimating or underestimating) assumptions to estimate the cost of interference.

4.5 Related Work

In this section, we compare our work to the related work along several key dimensions, including operator placement for makespan reduction, graph transformations, operator placement at the edge, and degenerate forms of DAG placement at the edge, all of which are key considerations in our work.

4.5.1 Operator Placement for DAG Makespan Minimization

Operator placement problem has been studied extensively in the literature [127]. A generic formulation of the operator placement problem has been presented in [37]. The authors show how their formulation can be used for optimizing different QoS parameters, such as response time, availability, network use, etc. They formulate the problem as an integer linear optimization problem and use the CPLEX solver to find an optimal placement. Existing solutions have varied objectives, such as minimizing network use [128, 129], minimizing inter-node traffic [130], minimizing the makespan or response time of an operator graph [44, 42, 43, 41]. However, to the best of our knowledge, existing works on makespan minimization do not consider the impact of operator co-location and hence the interference effects on response time, while our solution expressly considers such an impact. For example, in [44], authors have used a queueing theory-based model for estimating the response time of paths in a DAG. However, their model also does not consider the impact of co-location. Our solution uses a data-driven latency prediction model to estimate path latencies in a DAG which also incorporates the impact of operator co-location on observed path latencies.

4.5.2 Operator Graph Transformation

In [131], authors leverage the technique of operator replication to provide better performance for processing incoming data streams. Apart from the replication, the authors also propose an algorithm for placement of these operators on the runtime platform. Similar

graph transformation using operator replication strategy has been applied in [132]. In contrast to these works, we use DAG transformations as a means to simplify model learning. In [133], the authors decompose a series-parallel-decomposable (SPD) graph into a SPD tree structure to aid allocation of tasks to physical resources. While their solution is applicable only for SPD graphs, our solution is applicable to any arbitrary DAG structure.

4.5.3 Edge-Based Operator Placement

In [134], authors present a constraint satisfaction problem for placement of operators across heterogeneous edge-fog-cloud resources for maximizing resource usage. In comparison to this work, we do not consider heterogeneous resources and it will be important to extend our work to cover the range of edge-fog-cloud resources. However, unlike this work, our aim is to minimize the makespan. In [135], authors present an optimization framework that formulates the placement of operators across cloud and edge resources as a constraint satisfaction problem for makespan minimization. The evaluation of their proposed scheme is, however, conducted through a simulation study implemented using OMNET++ simulator. In contrast, we have validated our research on an actual IoT testbed. DROPLET [136] formulates operator placement problem using the shortest path problem, in which the operators are placed in such a fashion that it minimizes the total completion time of the graph. Although their goal is similar to ours, we believe that this work and several of the other works outlined above have not considered the performance interference issue which can result due to resource contention happening among the participating operators.

4.5.4 Latency Minimization for Publish/Subscribe Systems

A publish-subscribe system which involves some processing at an intermediate broker is a degenerate form of a DAG that we consider in our work. There are some recent works that consider minimizing end-to-end latencies for such degenerate DAG topologies. For instance, in [137], the authors present an approach to minimize end-to-end latencies for

competing publish-process-subscribe chains and balancing the topic loads on intermediate brokers hosted in edge computing environments. Unlike this work which focuses on only a 3-node chain and focuses more on balancing the load on the brokers, our work focuses on the placement arbitrary DAGs comprising a workflow of stream processing operators on edge resources. MultiPub [65] is an effort to find the optimal placement of topics across geographically distributed datacenters for ensuring per-topic 90th percentile latency of data delivery. Although we are also interested in 90th percentile latencies, this work only considers inter-datacenter network latencies for making placement decisions.

4.6 Conclusions

4.6.1 Summary of Research Contributions

With the growing importance of the Internet of Things (IoT) paradigm, there is increasing focus on realizing a range of smart applications that must ingest the continuous streams of generated data from different sources, process this data within the stream processing application which is often structured as a directed acyclic graph (DAG) of operators, and make informed decisions in near real-time. The low latency response time requirements of these applications require that the computations of the DAG operators be performed closer to the data sources, i.e., on the IoT edge devices. However, resource constraints on these devices require careful considerations for multi-tenancy, i.e., co-location of operators on the edge devices, which must be done in a way that minimizes the DAG makespan, i.e., the end-to-end latency for the longest path in the DAG.

To that end, we present an optimization problem formulation for DAG makespan minimization. The NP-hardness of the general problem and the need to realize an efficient, runtime deployment decision engine inform the design of our greedy heuristic. Our heuristic-based solution makes its operator placement decisions using a look-ahead scheme wherein the algorithm predicts the potential impact on a DAG's makespan latency if a specific operator co-location decision is made. To aid in this look-ahead phase of our placement

algorithm, we present a data-driven latency prediction model, which is a machine learning model that is trained using empirical data generated from conducting a variety of operator co-location experiments on an edge computing testbed. A novel trait of the prediction model is the use of linearized chains of operators created using a transformation algorithm that we developed to closely approximate the performance of the original DAG structure. In summary, we presented a novel *Linearize-Predict-Place (LPP)* approach for DAG makespan minimization. Our empirical results comparing LPP with two separate baselines called SUM and CONST reveal that LPP makes a more accurate prediction of path latencies and thereby a more effective placement of operators than the SUM and CONST approaches, which otherwise make simplifying assumptions to estimate the cost of interference caused due to co-location.

4.6.2 Discussions and Directions for Future Work

Although operator placement is a well-known problem and many prior efforts exist, the DAG placement on edge resources to minimize the makespan while promoting co-location is a problem that remains to be solved. Moreover, applying data-driven machine learning models to use as predictive models in solving the placement problem is another dimension of novelty. Finally, the critical insights into how the model should be developed, the key input features to use, and the use of linear chains as an approximation of the original DAGs that makes it easier to build the prediction models are fundamental and novel contributions of this work.

There are many dimensions along which additional work will need to be performed, and these needs stem from the current assumptions we made and the limitations of the edge computing resources we used for this study.

- The DAG linearization algorithm was informed by empirical observations that we made on our Beagle Black Bone (BBB) testbed. To prove the correctness of this approach, there is a need to build a theoretical model possibly using queuing the-

ory. Secondly, the linearization transformation for the fork/join operators currently assumes interleaving semantics, i.e., OR semantics. The rules need to be extended when the join operators require AND semantics.

- In current work, we have focused primarily on CPU computations alone at the operator when making the co-location decisions. Other resources, such as memory, also need to be accounted for. Moreover, we have assumed stable environments with significant resource fluctuations, these aspects must be considered.

- Our edge computing testbed is made up of homogeneous resource types (i.e., BBB boards with a single core), but IoT can illustrate significant heterogeneity in resource type, and it will become important to explore this dimension.

- The scale of our experiments and the size of the DAGs considered in this work is limited, and moreover, the applications used were mostly synthetic. More work is needed to use larger-scale and real-world IoT applications.

- Finally, uncertainty quantification remains to be incorporated into our models. These uncertainties can result from network fluctuations, bursty workloads, resource failures among many other issues.

The source code and experimental apparatus used in this research is made available in open source at https://github.com/doc-vu/dag-placement

Chapter 5

BRINGING IT ALL TOGETHER

5.1 Introduction

Internet of Things (IoT) [1] applications such as intelligent transportation [3], smart grids [138], remote infrastructure monitoring and control [139], etc. involve continuous collection and near real-time processing of data streams produced by a large number of sensors in order to gain insights and detect events of interest, which are subsequently distributed to interested endpoints for intelligent control and actuation of the system. Such applications can be visualized as a *distributed data-flow*, where data is collected, processed and disseminated across a vast range of distributed resources spanning from the edge of the network, where sensors reside to remote cloud back-ends. Many of these applications are latency sensitive in nature and impose low response time requirements on both-data distribution and data processing. Performing computation near the source of data or Edge computing [19] offers an attractive solution for meeting the low response time requirements these applications. However, the high performance overhead of processing on typically resource-constrained edge devices can quickly eclipse the network latency benefits of processing near the source. Developing an edge-based and latency-aware unified solution for meeting the data distribution and stream data processing needs of IoT applications which takes this trade-off between computation and network overhead into account, is the objective of this dissertation research.

Previous chapters in this dissertation offer solutions towards this goal in the following ways. First, Chapter 2 presents a novel data-driven solution for distributing the publish/subscribe (pub/sub) based data distribution and processing workload over edge servers to meet application specified end-to-end latency requirements while making use of minimal number of scarce edge resources. Second, Chapter 3 presents a unified programming model which

combines pub/sub based data distribution with reactive stream data processing to support end-to-end development of an IoT application's distributed data-flow. Third, Chapter 4 presents Linearize, Predict and Place (LPP), a novel data-driven algorithm for placing the constituent operators/vertices of a data-flow application which is generally structured as a Directed Acyclic Graph (DAG), over edge devices such that the end-to-end latency of processing of the application is minimized. However, these solutions have been presented in a stand-alone fashion, implemented using different technologies and tested on different hardware resources. The applicability of these solutions together in the context of a real world edge-based use-case has not been presented. Moreover, LPP placement algorithm was developed on the basis of empirical observations that were made on a single core Beagle Black Bone device. Although LPP is not specific to single-core edge devices, its performance on multi-core edge devices has not been tested. To address these concerns, this chapter presents the applicability of our research ideas in the context of a real-world edge-based application for Automatic License Plate Recognition (ALPR) and tests it on a cluster of quad-core Raspberry Pi boards which serve as edge devices in our experiments.

Video processing has been identified as the most suitable application for edge computing [140]. Video processing applications such as surveillance, traffic control, Augmented Reality (AR), etc. have very low response time requirements and can benefit from computation near the source of data. Moreover, sending video data streams collected from a large scale deployment of video cameras to the cloud for processing would incur a huge bandwidth cost. Automatic License Plate Recognition (ALPR) is another example of a video processing application in which video streams are continuously processed to identify license plate numbers of vehicles being monitored. ALPR can be used in a wide range of applications [141] such as automated parking fee management, road toll collection, identifying traffic violators, etc. and thus, makes a good real-world, edge-based application use-case for validating our proposed research solutions.

We implemented our ALPR application using our unified programming model pre-

sented in Chapter 3 and we demonstrate its benefits such as declarative and composable style of programming, declarative management of concurrency and flexibility in defining component boundaries for developing distributed data-flows in the context of ALPR. To keep the response time of ALPR low, we used our Linearize, Predict and Place (LPP) algorithm presented in Chapter 4 for placing the operators of ALPR's application DAG over a cluster of Raspberry Pi devices. Experiment results show that LPP model performs well even on a multi-core edge device like Raspberry Pi and is able to come up with a placement with minimal end-to-end latency in comparison to some simpler placement heuristics.

This chapter is organized as follows: Section 5.2 describes our ALPR application use-case, Section 5.3 describes the implementation of ALPR using our unified programming model presented in Chapter 3, Section 5.4 describes the application of LPP for latency-aware placement of ALPR and finally, Section 5.5 concludes the chapter.

5.2 Use-Case: Automatic License Plate Recognition Application

Automatic License Plate Recognition (ALPR) continuously processes video streams to identify vehicle license plate numbers. ALPR can be used in a wide range of applications [141] such as parking automation, ticket-less parking fee management, road toll collection, traffic surveillance, identifying traffic violators and finding stolen vehicles. Sending large volumes of video data to the cloud for processing can become prohibitively expensive, therefore ALPR can benefit from an edge based deployment, wherein video streams are processed near the source of data on low-cost edge devices.

ALPR involves several processing stages [142, 143, 144]. First, potential regions within an image which may contain a license plate are identified for which several detection techniques exist, for example, in edge-based detection, hough transformation is used to detect pairs of parallel lines which are considered as potential plate candidates [145], and in case of texture-based detection, a license plate is considered to have a distinct visual description/texture on the basis of which potential plate candidates are identified [146]. After

candidate license plate regions are identified, character segmentation is performed within the region to extract individual characters. Finally, Optical Character Recognition (OCR) is performed either by using pattern matching [147] or neural network based classification [148] to identify each character.

OpenALPR [149] is a widely used, open-source library for ALPR which uses a texture-based method for license plate detection and pattern matching for OCR. While OpenALPR performs well on images taken under ideal lighting conditions with high contrast [150], it is not able to recognize license plates in images taken under cloudy, dim and rainy conditions. To increase the likelihood of license plate detection in such cases, several preprocessing techniques [151] can be applied, such as, noise reduction with Bilateral Filtering and contrast enhancement with Histogram Equalization or Contrast Limited Adaptive Histogram Equalization (CLAHE).

(a) License Plate-1 (b) License Plate-2 (c) License Plate-3 (d) License Plate-4

Figure 5.1: Example Images for License Plate Detection Application

Figure 5.1 shows four gray-scale sample images taken from a public dataset comprising of over 500 images of vehicle license plates taken under different lighting conditions [151]. Table 5.1 shows the detected license plate number with openALPR after preprocessing an image with the following techniques: 1) **none**: no image preprocessing is performed and openALPR is run directly on the original image, 2) **bf**: Bilateral Filter is applied to reduce noise, 3) **eqh**: Histogram Equalization is performed, 4) **clahe**: Contrast Limited Adaptive Histogram Equalization (CLAHE) is performed, 5) **bf-eqh**: Histogram Equalization is performed after Bilateral Filtering, and 6) **bf-clahe**: CLAHE is performed after Bilateral Filtering.

We see that different preprocessing techniques give different results under varied light-

Table 5.1: Image Preprocessing for ALPR

license plate#	none	bf	eqh	clahe	bf-eqh	bf-clahe
1	551AG	550AG	K551A	VK55	551AG	551AG
2	S771	SQ77	771AE	-	771E	S0771
3	-	-	-	BA056	-	-
4	-	-	162LC	12LC	-	-

ing conditions. Image of License plate-3 in Figure 5.1c was taken under dim light and only **clahe** is able to detect the plate number, whereas for License plate-2 in Figure 5.1b and plate-4 in Figure 5.1d, **eqh** gives the best result. For License plate-1 in Figure 5.1a with number VK551AG, **clahe** detects the first part (VK55) while **bf-clahe** and **none** detect the second part (551AG). Therefore, to maximize the chance of correct license plate number detection, different types of preprocessing techniques can be applied in parallel and results can be tallied for higher confidence. Such an application data-flow is depicted in Figure 5.2, where the source `vertex-src` sends license plate images to be processed in parallel by employing three different preprocessing techniques, namely, bilateral filtering performed by `vertex-bf`, histogram equalization performed by `vertex-eqh` and CLAHE performed by `vertex-clahe`, before license plate recognition is performed by `vertex-lpr` with openALPR. Additionally, here after license plate detection, `Vertex-seg` marks the detected plate region with a rectangular bounding box and `vertex-post` performs some kind of post-processing such as saving the results to a database or looking up a database of stolen car license plates.

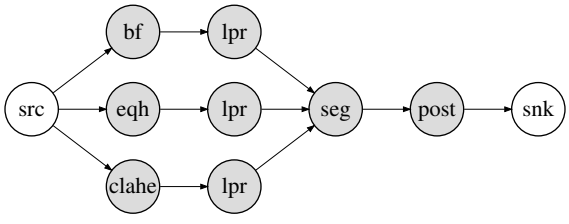


Figure 5.2: ALPR Application DAG-1

5.3 Implementation of ALPR as a Distributed Data-Flow

In this section we discuss the implementation of ALPR as a distributed data-flow using our unified programming model presented in Chapter 3. We developed RxZmq library which unifies data distribution using ZMQ pub/sub library [75] and stream data processing with Microsoft's Reactive Extensions [152]. Section 5.3.1 describes the implementation of ALPR using RxZmq and the benefits of using RxZmq for distributed data-flow programming. We also describe our experiment setup and test-bed used for running our experiments in Section 5.3.2.

5.3.1 RxZmq

As discussed in Section 3.3.4, Rx4DDS.NET integrates DDS [9], a publish/subscribe middleware technology, with Microsoft's Reactive Extensions (Rx) library [152] to unify data distribution and local stream data processing for seamless end-to-end development of data-flow applications. Rx provides a first-class abstraction for data streams called `observables`; a reusable set of composable `operators` for joining, multiplexing, demultiplexing and transforming data-streams; and `schedulers` for declarative management of concurrency. Rx4DDS library maps data received over a DDS pub/sub topic to an `observable` data stream in Rx so that local data processing aspects can be programmed in a declarative and composable manner using Rx `operators`. Similar to Rx4DDS, to implement our ALPR application as a distributed data-flow, we developed RxZmq library which integrates ZMQ pub/sub library [75] with RxPy [153], a python binding for Reactive Extensions. We developed RxZmq instead of using Rx4DDS.Net since python enables rapid prototyping and is readily supported on Raspberry Pi boards, which were used as edge devices in our experiments. Image processing functions used in our ALPR data-flow such as bilateral filtering, histogram equalization, CLAHE and segmentation were implemented using OpenCV, license plate recognition was performed using OpenALPR and post-processing such as looking up a database/saving to a remote database was simulated

by performing recursive fibonacci computations for 25 ms.

Using RxZmq for developing our ALPR data-flow offers several advantages such as declarative and composable style of programming, declarative management of concurrency and flexibility in defining points of data distribution/component boundaries. Listing 5.1 shows how $\langle eqh, lpr, seg, post \rangle$ processing pipeline is implemented using RxZmq. RxZmq library function `from_topic` maps a ZMQ topic (`input_zmq_topic`) to an `observable` in Rx, which can be processed by composing various built-in or user-defined `operators`. For example, `map` is a built-in Rx operator which transforms an input data-stream to an output data-stream by applying a mapping function. In Listing 5.1, the input stream of images received over `input_zmq_topic` is transformed by performing histogram equalization by the first `map` function. This transformed data-stream is further processed by downstream `map` operators which carry out `lpr`, `seg` and `post` functionality. Finally, the result stream is published over a ZMQ topic (`output_zmq_topic`) using RxZmq's `to_topic` function which maps an `observable` data-stream in Rx to a ZMQ topic. RxZmq's composable and declarative style of programming preserves the conceptual flow of the application and makes it readily visible as can be seen in Listing 5.1

Listing 5.1: Sample Implementation of ALPR's Data-flow with RxZmq

```
from_topic(input_zmq_topic).pipe(
        map(lambda img: cv2.equalizeHist(img)),
        map(lambda img: alpr.recognize_ndarray(img)),
        map(lambda res: cv2.rectangle(res.img,res.plate_x1,
                res.plate_y1,res.plate.x2,res.plate.y2,
                border_color,border_thickness)),
        map(lambda img: post_processing(img)),
        to_topic(output_zmq_topic)
).subscribe()
```

RxZmq also provides a lot of flexibility in defining the points of data distribution or component boundaries in our application. For example, in Listing 5.2, the transformed stream of input images after histogram equalization is published over a ZMQ topic `intermediate_zmq_topic` so that further processing can be offloaded on another device. These points of data distribution can be easily configured and chosen at the time of deployment.

Listing 5.2: Flexible Component Boundaries with RxZmq

```
from_topic(src).pipe(
        map(lambda img: cv2.equalizeHist(img)),
        to_topic(intermediate_zmq_topic)
).subscribe()
```

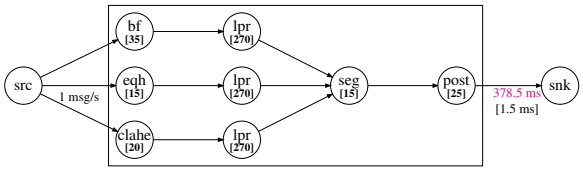
In addition scaling out the application for distributed processing, RxZmq also provides declarative management of concurrency for scaling up the application to use multiple cores available on a single node. Rx supports several built-in `schedulers`, such as thread-pool, task-pool, new thread, etc. for concurrent execution. For example, in Listing 5.3, the computationally expensive task of `lpr` has been offloaded on a `ThreadPool` scheduler using Rx's built-in function `observe_on` which offloads all downstream processing on the specified scheduler.

Listing 5.3: Scaling-up ALPR's computation

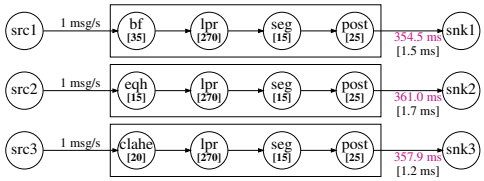
```
from_topic(src).pipe(
        map(lambda img: cv2.equalizeHist(img)),
        observe_on(ThreadPool),
        map(lambda img: alpr.recognize_ndarray(img)),
        ...
).subscribe()
```

5.3.2 Experiment Setup and Testbed

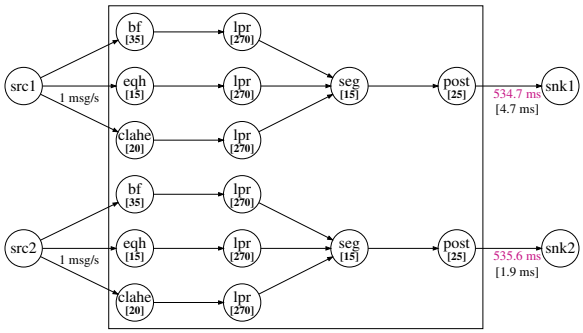
Our experiment testbed comprises of 8 Raspberry Pi 3 Model B boards with 1.2 GHz quad-core Broadcom BCM2837 64-bit processor, 1 Gb RAM and 100 Base Ethernet capacity [154]. Of these, one Raspberry Pi board was used for hosting the source and sink vertices, while the remaining 7 boards were used for hosting the intermediate vertices which process license plate images.



(a) Latency of ALPR DAG-1 placed on one Raspberry Pi



(b) Latency of constituent paths in ALPR DAG-1 in isolation



(c) Latency of two ALPR DAG-1s placed on one Raspberry Pi

Figure 5.3: Impact of DAG Structure and Co-location on Latency

Figure 5.3a shows constituent intermediate vertices of ALPR DAG-1 (Figure 5.2) placed on a single Raspberry Pi board. Isolated execution interval of each vertex, measured in milliseconds, has been shown in brackets below each vertex's type. `Vertex-lpr` is computationally intensive and was measured to take ∼270 ms on average to process a 240 x 320 gray-scale image. Therefore, not more than 3 frames/second can be processed by `vertex-lpr`. For our experiments, we set the publication rate of source vertices to 1 frame/second and a `vertex-src` sends 200 time-stamped 240 x 320 gray-scale images during each experiment's run, which lasts for ∼4 minutes. Join vertices or intermediate vertices with more than one incoming edge, are assumed to follow interleaving semantics [42] wherein the vertex performs its computation whenever it receives a message on

any of its incoming edge. Sink vertices log the time-stamp of reception of each image after it has been processed by the intermediate vertices. Average 90th percentile latency and its standard deviation (shown in brackets) recorded by a sink vertex across 3 runs has been shown on its incoming edge. This implies that 90% of all messages received along all paths that end at a given sink vertex were observed to have an end-to-end path latency below the 90th percentile value. For example, in Figure 5.3a, 90% of all images processed along paths $\langle bf, lpr, seg, post \rangle$, $\langle eqh, lpr, seg, post \rangle$ and $\langle clahe, lpr, seg, post \rangle$ that end at `vertex-snk1` have an end-to-end latency below 378.5 ms.

5.4 Application of LPP for Latency-Aware Placement of ALPR

This section describes the application of Linearize Predict and Place (LPP) algorithm presented in Chapter 4 for the placement of ALPR such that the response time of processing is minimized. Response time, end-to-end latency or makespan of an application DAG is defined as the maximum latency of all paths in the application DAG. Response time of a DAG is impacted by several factors, such as, DAG based execution semantics and performance interference due to co-location of multiple vertices on the same physical node. Figure 5.3b shows the latency of constituent paths of DAG-1, namely, $\langle bf, lpr, seg, post \rangle$, $\langle eqh, lpr, seg, post \rangle$ and $\langle clahe, lpr, seg, post \rangle$, when these paths are run in isolation on a single Pi board. We see that the end-to-end 90th percentile latencies of constituent paths are lower than the 90th percentile latency of these paths when the original DAG structure is executed. Hence, we see that DAG structure imposed execution semantics impact the observed path latencies. Figure 5.3c shows two ALPR DAG-1 applications placed on a single Pi board. It can be assumed that each application processes images from two different surveillance video cameras. We see that the latency of each application DAG increases by $\sim$40% due to performance interference on account co-location of large number of vertices on the same Pi board.

LPP uses a latency prediction model which can estimate the path latencies of all paths

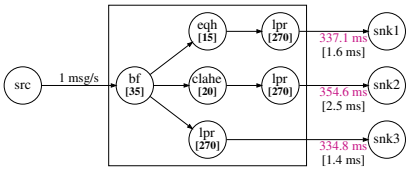
in a DAG and uses this information to guide the placement of constituent vertices such that the response time of the DAG is minimized. However, developing a latency prediction model which can predict the end-to-end path latencies of arbitrary DAG structures while incorporating the impact of performance interference due to co-location of vertices is a hard problem. To substantially simplify this overhead of model learning, LPP first linearizes a DAG into an approximate set of linear chains using a set of linearization rules. When this set of approximate linear chains are executed, the observed latencies are very similar to that observed when the original DAG structure is executed. Now, a latency prediction model for co-located linear chains can be used instead to estimate the path latencies in the original DAG structure. Linearization based approximation helps to reduce the overhead of learning a model for arbitrary DAG structures since a latency prediction model for co-located linear chains can be used instead to approximate the path latencies in the original DAG structure. It is important to note that the original DAG structure is what gets placed using LPP. Linearization and the prediction model for co-located linear chains are only used for estimating the path latencies in the original DAG structure which helps to guide the placement of vertices for makespan minimization.

Section 5.4.1 describes the application of LPP Linearization rules for approximating the path latencies of ALPR application, Section 5.4.2 describes the latency prediction models learned for co-located linear chains to guide the placement of ALPR and Section 5.4.3 shows the performance of LPP in predicting the latency of ALPR application DAG and compares LPP's placement results with some other simple heuristics for placement.

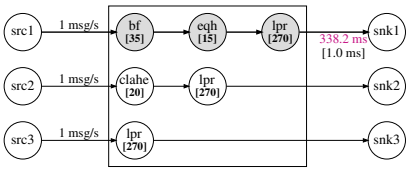
5.4.1 Linearization Rules

As described in Section 4.3.2, in order to reduce the overhead of training a latency prediction model for arbitrary DAG structures, LPP linearizes a DAG into a set of linear chains to approximate the latency of each path in the original DAG structure. This set of linear chains contains the target path for which the latency is being approximated along

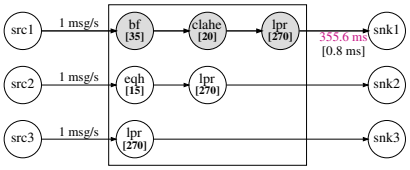
with some auxiliary chains to simulate the performance interference for the target path in the original DAG structure. Linearization is performed by applying distinct linearization rules for fork and join operators in the original DAG structure. Figure 5.4 and Figure 5.5 show the application of these linearization rules for fork and join operators respectively as applied to ALPR use-case.



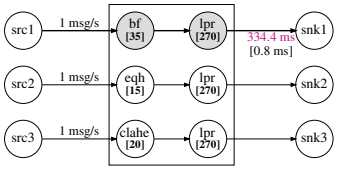
(a) Original DAG with fork operator



(b) Linearization for path-1

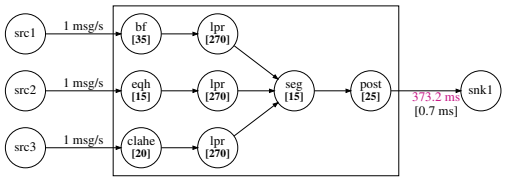


(c) Linearization for path-2

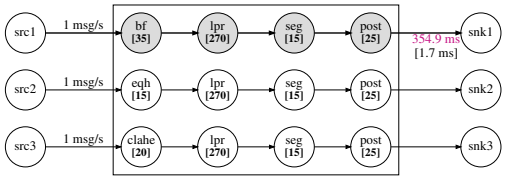


(d) Linearization for path-3

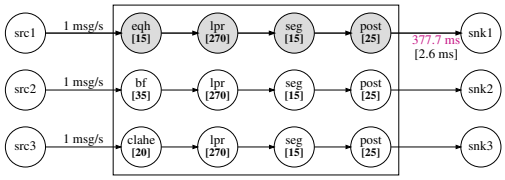
Figure 5.4: Linearization for fork operator



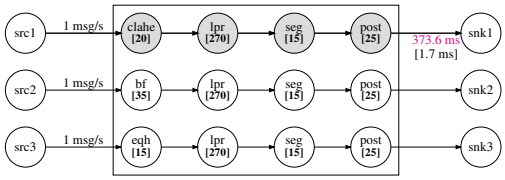
(a) Original DAG with join operator



(b) Linearization for path-1



(c) Linearization for path-2



(d) Linearization for path-3

Figure 5.5: Linearization for join operator

- *Fork operator*: Figure 5.4a shows the original DAG structure where `vertex-bf` is the fork operator. `Vertex-bf` executes once and distributes the image after bilateral filtering to three downstream paths which execute concurrently for further processing. Since the fork operator executes once, it is only made part of the target path while remaining concurrent paths (without the fork operator) constitute the auxiliary chains for performance interference. Figure 5.4b, shows the set of linear chains

produced to approximate the latency of path $\langle bf, eqh, lpr \rangle$ by applying this linearization rule for fork `vertex-bf`. The observed latency for target path $\langle bf, eqh, lpr \rangle$ highlighted in gray in Figure 5.4b, i.e., 338.2 ms is very similar to 337.1 ms, i.e., its observed latency when the original DAG structure is executed as seen in Figure 5.4a. Similarly, Figure 5.4c and Figure 5.4d show the approximate set of linear chains produced for target paths $\langle bf, clahe, lpr \rangle$ and $\langle bf, lpr \rangle$ respectively.

- *Join operator*: Figure 5.5a shows the original DAG structure where `vertex-seg` is the join operator. `Vertex-seg` processes each image that it receives from its three upstream paths (interleaving semantics) by marking the region where license plates are detected with a rectangular bounding box. Join operator and all its downstream operators execute once for each incoming path at the join operator. Therefore, the join operator and all its downstream operators are made part of each incoming path in the set of approximate linear chains. This is illustrated in Figure 5.5b, where to approximate the latency of target path $\langle bf, lpr, seg, post \rangle$ highlighted in gray, `vertex-seg` and `vertex-post` are also made part of the remaining incoming paths, $\langle eqh, lpr \rangle$ and $\langle clahe, lpr \rangle$.

  The latencies of all paths that end at the same sink operator are averaged to approximate the latency observed at that sink. The average of approximate latencies of paths $\langle bf, lpr, seg, post \rangle$, $\langle eqh, lpr, seg, post \rangle$ and $\langle clahe, lpr, seg, post \rangle$ is 368.7 ms which is very similar to 373.2 ms, i.e., the observed latency at sink `vertex-snk1` in the original DAG structure seen in Figure 5.5a.

5.4.2 k-Chain Co-location Latency Prediction Model

As discussed in Section 5.4.1, to approximate the latency of a path in the original DAG structure, a set of linear chains is produced which upon execution gives a latency value for the target path which is similar to that path's latency when the original DAG structure is executed. Hence, arbitrary DAG structures can be linearized and a latency prediction

model for co-located linear chains can be used to estimate the latency of a path in the original DAG structure. For example, a latency prediction model for 3 co-located linear chains can be used to predict the latency for target path $\langle bf, eqh, lpr \rangle$ in Figure 5.4b. DAG linearization greatly reduces the overhead of model learning for arbitrary DAG structures since it allows for a much simpler model for co-located linear chains to be learned and used instead.

Constituent operators of ALPR, namely, `vertex-eqh`, `vertex-seg`, `vertex-cl-ahe`, `vertex-post`, `vertex-bf` and `vertex-lpr` take 15 ms, 15 ms, 20 ms, 25 ms, 35 ms and 270 ms on average respectively when executed in isolation on a Raspberry Pi board. Therefore, to learn latency prediction models for co-located linear chains which can be applied for placing ALPR, we created a randomly generated dataset where each linear chain's constituent operator has an execution interval ρ uniformly randomly chosen from the set $\{$15 ms, 20 ms, 25 ms, 35 ms, 270 ms$\}$. Since `vertex-lpr` is computationally expensive and takes 270 ms on average for processing a 240 x 320 gray-scale input image, the source vertex is restricted to send image files at 1 frame/second.

Similar to the model learning process described in Section 4.3, some features were used to characterize the foreground chain/target chain denoted by c_f, while others were used to characterize the background load imposed by the set of auxiliary/background chains denoted by $\mathbb{C}_B$. The following set of features were used for the learning the k-chain co-location latency prediction model:

- $n(c_f)$: number of operators in c_f;
- $\sum_{o \in c_f} \rho(o)$: sum of execution intervals of operators in c_f;
- $\sum_{c_b \in \mathbb{C}_B} n(c_b)$: sum of number of operators in all background chains;
- $\sum_{c_b \in \mathbb{C}_B} \sum_{o \in c_b} \rho(o)$: sum of execution intervals of all operators in all background chains;

- $n(c_f)/\sum_{c \in \mathbb{C}_B + c_f} n(c)$: fraction of total number of operators in the foreground chain
- $\sum_{o \in c_f} \rho(o)/[\sum_{c \in \mathbb{C}_B + c_f} \sum_{o \in c} \rho(o)]$: fraction of the total sum of execution intervals of all operators in the foreground chain

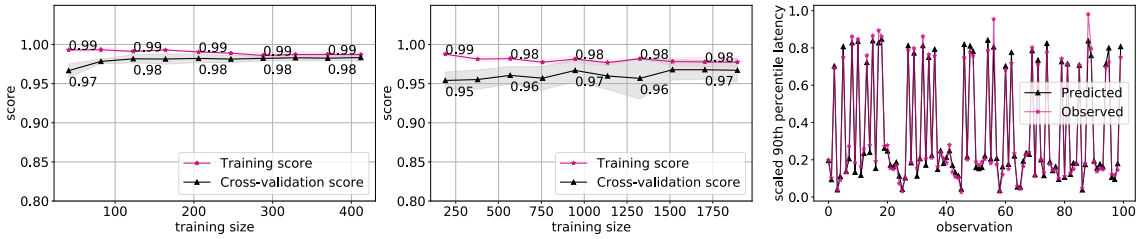
For higher accuracy, a separate latency prediction model for each k co-location value was learned using Neural Network regression, for k up to 8. We ran 200 experiments for $k \leq 6$ and 400 experiments for $k = 7$ and $k = 8$ to generate the training dataset. Up to 100 additional tests were performed to generate a separate validation dataset for each k. During an experiment run, a source vertex sends 200 gray-scale 240 x 320 images at 1 frame/second and an experiment took ~ 4 minutes to execute. When an experiment for k co-located chains is run, we get k latency data-points, one corresponding to each linear chain. Therefore, the training dataset size becomes k times the number of experiments, i.e., 200, 400, 600, 800, 1000, 1200, 2800 and 3200 for $k = 1$ to $k = 8$ as shown in Table 5.2. 90% of this dataset was used for training the neural network and the remaining 10% was used for testing.

Table 5.2: Accuracy of k-chain co-location regression models for placing ALPR

k	#datapoints (training)	accuracy (training)	accuracy (test)	#datapoints (validation)	accuracy (validation)
1	200	.98	.99	100	.99
2	400	.99	.97	100	.97
3	600	.99	.98	100	.99
4	800	.98	.98	100	.98
5	1000	.98	.98	100	.98
6	1200	.98	.98	100	.98
7	2800	.98	.98	100	.98
8	3200	.98	.98	100	.98

We tested different neural network architectures for learning each k-chain co-location model and found that a neural network with one layer comprising 80 neurons performed best for $k \leq 6$ and a neural network architecture with two layers comprising 40 neurons each performed best for $k = 7$ and $k = 8$. Rectified Linear Units (ReLu) was used as the activation function, limited memory Broyden-Fletcher-Goldfarb-Shanno (lbfgs) was

used as the solver and L2 regularization factor was set to 0.1 for learning all the models. Figure 5.6a and Figure 5.6b show the learning curves for $k = 3$ and $k = 7$ co-location latency prediction models respectively. The learning curves show that the training and validation errors converge to a low value of at most 3%. Therefore, the models neither under-fit nor over-fit the data and are expected to generalize well. Performance of $k = 7$ chain co-location model on the validation dataset is shown in Figure 5.6c. Here, we see that the predicted values track the experimentally observed latency values quite well with an average error of 9.9 ms over all 100 data-points. Table 5.2 summarizes the performance of all k-chain co-location regression models on training, test and validation datasets. All learned models were observed to have an accuracy of at least 97%.



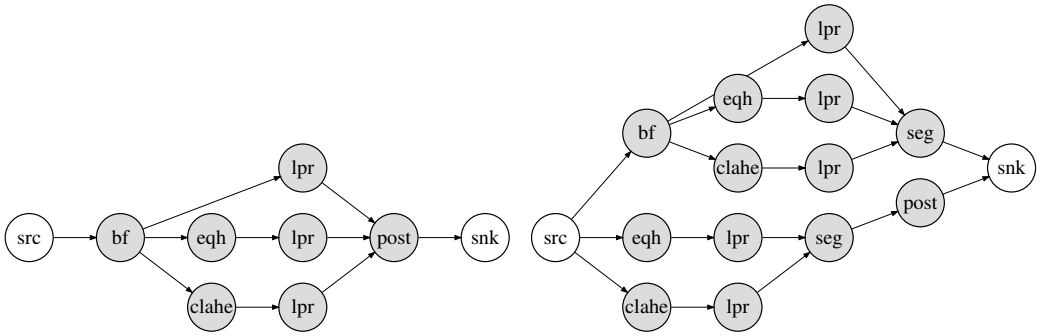
(a) Learning curve for $k = 3$ (b) Learning curve for $k = 7$ (c) Validation results for $k = 7$

Figure 5.6: Performance of k-chain co-location latency prediction models

5.4.3 Performance Evaluation of LPP for Placement of ALPR

To assess the performance of LPP for placement of ALPR application over a cluster of Raspberry Pis such that its latency of processing is minimized, we apply LPP to generate the placement for three ALPR application DAG structures as depicted in Figure 5.2, Figure 5.7a and Figure 5.7b. All three DAG structures are composed of the same component operator types, such as `vertex-bf`, `vertex-clahe`, `vertex-lpr`, etc., but they are structurally different from each other. Figure 5.8 shows the predicted vs observed latency of these three DAG structures after they are placed by LPP. We observe that LPP is able to predict the latency fairly well with a mean prediction error of 19 ms.

We also compared the placement produced by LPP with two other heuristics- SUM and



(a) ALPR Application DAG-2 (b) ALPR Application DAG-3

Figure 5.7: Application DAG Structures

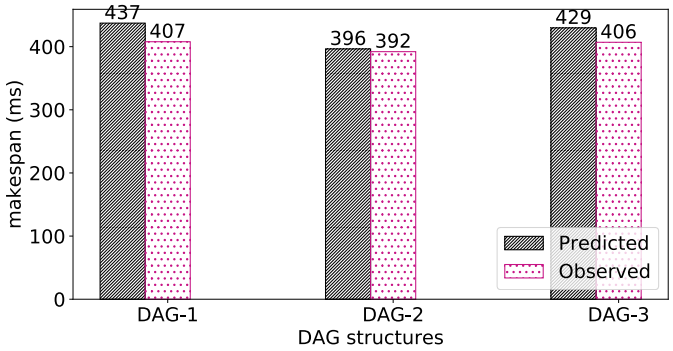
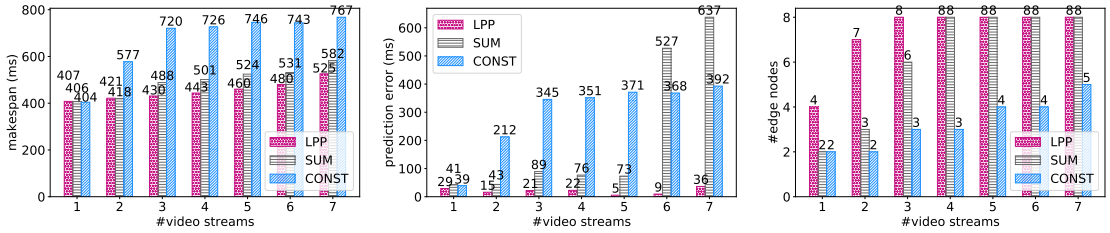


Figure 5.8: LPP Predicted vs Observed for different DAG Structures

CONST. SUM and CONST use the same DAG linearization approach used by LPP to estimate the number of co-located linear chains on a physical node/Raspberry Pi upon operator placement, but unlike LPP which uses the k-chain co-location latency prediction model to predict each path's latency, SUM and CONST make some simplifying assumptions to estimate a path's latency. CONST approach assumes that there is no impact of co-location of background chains on a DAG's path latency and it estimates each path's latency to be the sum of execution intervals of its constituent operators. SUM approach assumes that a path's latency is the sum of its constituent operators so long as the number of co-located paths on the same Raspberry Pi is ≤ 4 since the Raspberry Pi has 4 cores and can execute 4 paths concurrently. However, if the number of co-located paths is more than 4, then SUM estimates that each operator's execution interval becomes equal to the sum of execution intervals of all co-located operators divided by 4, on account of performance interference.



(a) Observed makespan (b) Prediction Error (c) Number of nodes

Figure 5.9: Comparison of LPP with SUM and CONST approaches (Application DAG-1)

Since latency prediction models have only be learned for $k \leq 8$, all three approaches consider an operator's placement on a physical node as infeasible if that operator's placement on the node results in more than 8 co-located linear chains on the same physical node.

Assuming that one ALPR DAG-1 structure processes images from one video camera, an increasing number of ALPR DAG-1 structures for processing up to 7 video camera streams are placed on our test-bed of 8 Raspberry Pis using LPP, SUM and CONST approaches for comparison. Figure 5.9a compares the makespan of placements produced by LPP, SUM and CONST for placing up to 7 ALPR DAG-1 structures. Figure 5.9b shows the error in predicting the makespan when up to 7 ALPR DAG-1 structures are placed using LPP, SUM and CONST. Figure 5.9c shows the number of physical nodes used by LPP, SUM and CONST for placing up to 7 ALPR DAG-1 structures. We observe that LPP is able to find a placement with minimal makespan in comparison to SUM and CONST approaches. Latency prediction error for LPP is also substantially less than that of SUM and CONST approaches. This shows that LPP's linearization and k-chain co-location models perform fairly well in estimating path latencies. CONST underestimates path latencies since it ignores the impact of operator co-location and favors placing more operators on the same physical node. This results in much higher DAG latencies due to higher resource contention, although it uses the least number of physical nodes for operator placement. SUM overestimates path latencies to a great extent and therefore, its prediction error is very high. Due to this overestimation SUM distributes the operators out over more physical nodes in comparison to CONST approach. LPP uses same or more number of nodes than SUM, but

is still able to find a placement of operators which results in a lower makespan than SUM approach. LPP is able to make a better trade-off between network cost and cost of operator co-location than CONST and SUM since it uses a more accurate latency prediction model.

5.5 Conclusion

In this chapter we demonstrated the application of our research ideas in the context of a real world, edge-centric application for Automatic License Plate Recognition (ALPR). ALPR can benefit from an edge based deployment since processing images near the source can help in fast identification of traffic violations and real-time, automatic toll and parking fee management. Moreover, sending video streams from a large/city scale deployment of surveillance cameras to the cloud for processing would be very expensive. We implemented our ALPR application using RxZmq library which unifies pub/sub data distribution and local data stream processing for seamless end-to-end development of distributed data-flows. RxZmq offers several benefits such as composability, flexibility in defining component boundaries and declarative management of concurrency, which eased the development of ALPR to a great extent. To ensure that ALPR benefits from an edge-based deployment in terms of response time, our Linearize Predict and Place (LPP) algorithm was used to place the constituent operators of ALPR over a cluster of Raspberry Pi boards which served as edge devices. Experiment results show that LPP was able to predict the latency of several different ALPR application graph structures fairly well and was able to come up with a placement with minimal response time in comparison to simpler heuristics which either overestimate or underestimate path latencies. Linearization rules for approximating path latencies of arbitrary graph structures were developed on the basis of empirical observations that were made on a single-core Beagle Black Bone device. However, LPP's low prediction error in placing ALPR over quad-core Raspberry Pi boards demonstrates that linearization can be used as a good approximation, even for estimating path latencies for application graphs placed on multi-core devices.

Chapter 6

SUMMARY

6.1 Summary of Research Contributions

To meet the data distribution and on-line processing needs of latency-sensitive IoT applications deployed at the edge, this doctoral research makes the following contributions:

1. **Latency-Aware Data Distribution at the Edge:** Topic-based pub/sub communication pattern is widely used to meet the large scale data distribution needs of IoT applications. However, none of the existing open-source topic-based pub/sub systems provide any QoS assurance on the end-to-end latency of data delivery. Increasingly, pub/sub brokers deployed at the edge also support light-weight processing on the incoming data streams in addition to data distribution, making it the publish-process-subscribe pattern. It is in this context that end-to-end latency QoS for data dissemination and processing must be provided to support latency-sensitive applications.

   To provide latency QoS assurance for publish-process-subscribe systems, we propose our solution which learns a latency prediction model for a set of co-located topics on an edge broker and uses this model to balance the processing and data-dissemination load to provide the desired QoS, specified as per-topic 90th percentile latency. In this context, we make the following key contributions: (a) a sensitivity analysis to understand the impact of features such as publishing rate, number of subscribers, per-sample processing interval and background load on a topic's performance; (b) a latency prediction model for a set of co-located topics, which is then used for the latency-aware placement of topics on brokers; and (c) an optimization problem formulation for k-topic co-location to minimize the number of brokers while meeting each topic's QoS requirement. Here, k denotes the maximum number of topics that

can be placed on a broker.

2. **Support for DAG Stream Processing at the Edge:** With the advent of edge computing, stream based processing of data can span across the entire resource spectrum from edge to the cloud. Such a hierarchical and geo-distributed stream processing application can be viewed as a distributed dataflow. Pub/sub pattern can adequately meet the distribution needs of this dataflow. However, due to lack of generality and composability in the API of pub/sub systems, data processing is seldom implemented as a dataflow. Therefore, there is a need for a unified programming model for data distribution and processing which can help preserve the end-to-end dataflow structure. To address this need, we have proposed our solution which blends reactive programming with pub/sub based data distribution. Reactive programming provides a dedicated abstraction for data streams and supports a variety of composable and reusable operators for stream-based processing. Therefore, this integration allows even the local processing stages of a pub/sub application to be structured as a dataflow.

   To demonstrate our research ideas, we integrated a specific instance of reactive programming, namely, Microsoft .NET Reactive Extensions (Rx) with a specific instance of pub/sub, namely, Real Time Innovation's (RTI) Data Distribution Service (DDS), into an open-source library called Rx4DDS.NET. We compared a stream processing application developed using Rx4DDS.NET with an imperative solution developed using DDS and C#. Our qualitative evaluation results show that our integrated Rx4DDS.NET solution unifies local processing and data distribution aspects to preserve the end-to-end dataflow structure.

3. **Latency-Aware DAG Stream Processing at the Edge:** In order to support latency-sensitive stream-based applications deployed at the edge, it is important that the constituent operators of the application DAG are placed over resource constrained edge devices intelligently. An optimal placement is one that minimizes the the end-to-end

response time of the DAG by intelligently trading off inter-operator communication cost incurred to do distributed placement of operators across edge devices with the cost of interference that is incurred due to co-location of operators on the same resource-constrained edge device.

Although operator placement problem has been studied extensively in the literature, existing solutions do not consider the impact of DAG structure imposed execution semantics, incoming data rates and performance interference due to co-location on path latencies. To address these limitations, we have presented our solution which uses a latency prediction model to estimate path latencies accurately by taking all the aforementioned factors, i.e., DAG structure, data rate and operator co-location into account. This latency prediction model is subsequently used by a greedy placement algorithm to inform the placement of operators such that the end-to-end response time of the DAG is minimized.

Since learning a model which predicts the latency of all paths in any arbitrary DAG structure is complex, we present a novel solution which first linearizes the DAG into an equivalent set of linear chains. Thereafter, a simple latency prediction model for multiple co-located linear chains is used to approximate the path latencies in the original DAG.

4. **Application of Presented Research to a Real-World Edge Use-case :** We demonstrated the applicability of our research solutions in the context of a real-world, edge-based application on Automatic License Plate Recognition (ALPR). ALPR continuously monitors video data streams to identify vehicle license plate numbers and can be used in a variety of applications such as video surveillance, automated parking fee management and automated toll collection. Sending video data streams from a large/city scale deployment of video cameras can become prohibitively expensive and incur a huge latency cost, which makes ALPR is a good edge-based application

use-case. We implemented ALPR using our unified programming language for pub-/sub based data distribution and reactive stream data processing. We demonstrate the benefits of using this unified programming model, such as composable, declarative style of programming, declarative management of concurrency and flexibility in defining the component boundaries in the context of ALPR application use-case. To ensure that ALPR's latency of processing is kept low when deployed on edge devices, we applied our Linearize, Predict and Place (LPP) algorithm for latency-aware placement of ALPR on a cluster of Raspberry Pi devices. Although LPP model was developed using empirical latency observations made on a single-core Beagle Black Bone device, successful application of LPP to place ALPR on quad-core Raspberry Pi devices demonstrates the generality of LPP model. Experiment results show that LPP is able to find a good placement of ALPR which minimizes the end-to-end latency of data processing.

6.2 List of Publications

JOURNAL PUBLICATIONS

1. Subhav Pradhan, Abhishek Dubey, **Shweta Khare**, Saideep Nannapaneni, Aniruddha Gokhale, Sankaran Mahadevan, Douglas C. Schmidt, and Martin Lehofer. "Chariot: Goal-driven orchestration middleware for resilient iot systems." In ACM Transactions on Cyber-Physical Systems, 2018.

2. Swetasudha Panda, Andrew J. Asman, **Shweta Khare**, Lindsey Thompson, Louise A. Mawn, Seth A. Smith, and Bennett A. Landman. "Evaluation of multiatlas label fusion for in vivo magnetic resonance imaging orbital segmentation." In Journal of Medical Imaging, 2014.

CONFERENCE PUBLICATIONS

1. **Shweta Khare**, Hongyang Sun, Julien Gascon-Samson, Kaiwen Zhang, Yogesh Barve, Anirban Bhattacharjee, Aniruddha Gokhale and Xenofon Koutsoukos. " Linearize, Predict and Place: Minimizing the Makespan for Edge-based Stream Processing of Directed Acyclic Graphs." In proceedings of the 4th ACM/IEEE Symposium on Edge Computing (SEC), 2019.

2. Yogesh Barve, Shashank Shekhar, Ajay Dev Chhokra, **Shweta Khare**, Anirban Bhattacharjee, Zhuangwei Kang, Hongyang Sun and Aniruddha Gokhale. "FECBench: A Holistic Interference-aware Approach for Application Performance Modeling." in the IEEE International Conference on Cloud Engineering (IC2E), 2019.

3. Yogesh Barve, Shashank Shekhar, **Shweta Khare**, Anirban Bhattacharjee, and Aniruddha Gokhale. "UPSARA: A Model-Driven Approach for Performance Analysis of Cloud-Hosted Applications." In proceedings of the 11th IEEE/ACM International Conference on Utility and Cloud Computing (UCC), 2018.

4. **Shweta Khare**, Hongyang Sun, Kaiwen Zhang, Julien Gascon-Samson, Aniruddha S. Gokhale, Xenofon D. Koutsoukos and Hamzah Abdelaziz. "Scalable Edge Computing for Low Latency Data Dissemination in Topic-Based Publish/Subscribe." In proceedings of the 3rd ACM/IEEE Symposium on Edge Computing (SEC), 2018.

5. Kyoungho An, **Shweta Khare**, Aniruddha S. Gokhale and Akram Hakiri. "An Autonomous and Dynamic Coordination and Discovery Service for Wide-Area Peer-to-peer Publish/Subscribe: Experience Paper." In proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems (DEBS), 2017.

6. **Shweta Khare**, Kyoungho An, Aniruddha S. Gokhale, Sumant Tambe and Ashish Meena. "Reactive stream processing for data-centric publish/subscribe." In proceedings of the 9th ACM International Conference on Distributed Event-Based Systems

(DEBS), 2015.

7. Yurui Gao, Kurt G Schilling, **Shweta Khare**, Swetasudha Panda, Ann S Choe, Iwona Stepniewska, Xia Li, Zhoahua Ding, Adam Anderson and Bennett A Landman. "A brain MRI atlas of the common squirrel monkey, Saimiri Sciureus." In Medical Imaging 2014: Biomedical Applications in Molecular, Structural, and Functional Imaging, 2014.

SHORT PAPERS, WORKSHOPS AND POSTERS

1. Anirban Bhattacharjee, Barve, Yogesh, **Shweta Khare**, Shunxing Bao, Aniruddha Gokhale, and Thomas Damiano. "Stratum: A Serverless Framework for the Lifecycle Management of Machine Learning-based Data Analytics Tasks." To Appear in the 2019 USENIX Conference on Operational Machine Learning (OpML 19), USENIX, 2019

2. **Shweta Khare**, Hongyang Sun, Kaiwen Zhang, Julien Gascon-Samson, Aniruddha S. Gokhale and Xenofon D. Koutsoukos. "Poster Abstract: Ensuring Low-Latency and Scalable Data Dissemination for Smart-City Applications." In proceedings of the 3rd International Conference on Internet-of-Things Design and Implementation (IoTDI), 2018.

3. Yogesh Barve, Shashank Shekhar, Ajay Dev Chhokra, **Shweta Khare**, Anirban Bhattacharjee, and Aniruddha Gokhale. "Poster: FECBench: An extensible framework for pinpointing sources of performance interference in the cloud-edge resource spectrum." In Proceedings of the Third ACM/IEEE Symposium on Edge Computing, 2018.

4. Yogesh Barve, Shashank Shekhar, Ajay Dev Chhokra, **Shweta Khare**, Anirban Bhattacharjee, and Aniruddha Gokhale. "Demo Paper: FECBench A Framework for Measuring and Analyzing Performance Interference Effects for Latency-Sensitive

Applications." RTSS Works Demo Session of the 39th IEEE Realtime Systems Symposium (RTSS), 2018.

5. **Shweta Khare**, Janos Sallai, Abhishek Dubey and Aniruddha S. Gokhale. "Short Paper: Towards Low-Cost Indoor Localization Using Edge Computing Resources." In proceedings of the 20th IEEE International Symposium on Real-Time Distributed Computing (ISORC), 2017.

6. Subhav Pradhan, Abhishek Dubey, **Shweta Khare**, Fangzhou Sun, Janos Sallai, Aniruddha S. Gokhale, Douglas C. Schmidt, Martin Lehofer and Monika Sturm. "Poster Abstract: A Distributed and Resilient Platform for City-Scale Smart Systems." In proceedings of the 1st IEEE/ACM Symposium on Edge Computing, 2016.

TUTORIALS AND TALKS

1. Aniruddha Gokhale, Yogesh Barve, Anirban Bhattacharjee and **Shweta Khare**. "Software defined and Programmable CPS/IoT-OS: Architecting the Next Generation of CPS/IoT Operating Systems." To Appear in the 1st International Workshop on Next-Generation Operating Systems for Cyber-Physical Systems (NGOSCPS), 2019.

2. Anirban Bhattacharjee, Yogesh Barve, **Shweta Khare** and Aniruddha Gokhale. "Investigating Dynamic Resource Management Solutions for Cloud Infrastructures using Chameleon Cloud." In 2nd Chameleon Users Meeting, 2019.

3. Shashank Shekhar, Yogesh Barve, **Shweta Khare**, Anirban Bhattacharjee and Aniruddha Gokhale. "FECBench: An Extensible Framework for Pinpointing Sources of Performance Interference in Cloud-to-Edge hosted Applications." Tutorial at IEEE International Conference on Cloud Engineering (IC2E), 2018.

DOCTORAL SYMPOSIUM

1. **Shweta Khare**. ”Towards Scalable Edge Computing with Latency Assurance.” In Proceedings of the Third ACM/IEEE Symposium on Edge Computing, 2018.

2. **Shweta Khare**. ”Distributed Reactive Processing: Research Roadmap.” In proceedings of the 9th ACM International Conference on Distributed Event-Based Systems (DEBS), 2015.

BIBLIOGRAPHY

[1] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Computer networks*, vol. 54, no. 15, pp. 2787–2805, 2010.

[2] "Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016," https://www.gartner.com/technology/pressRoom.do?id=3598917, 2017.

[3] J. Y. Fernndez-Rodrguez, J. A. lvarez Garca, J. Arias Fisteus, M. R. Luaces, and V. Corcoba Magaa, "Benchmarking real-time vehicle data streaming models for a smart city," *Inf. Syst.*, vol. 72, no. C, pp. 62–76, Dec. 2017. [Online]. Available: https://doi.org/10.1016/j.is.2017.09.002

[4] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, Jun. 2003. [Online]. Available: http://doi.acm.org/10.1145/857076.857078

[5] D. Happ, N. Karowski, T. Menzel, V. Handziski, and A. Wolisz, "Meeting iot platform requirements with open pub/sub solutions," *Annals of Telecommunications*, vol. 72, no. 1, pp. 41–52, Feb 2017. [Online]. Available: https://doi.org/10.1007/s12243-016-0537-4

[6] "MQTT," http://mqtt.org/.

[7] "Kafka," https://kafka.apache.org/.

[8] "ActiveMQ," http://activemq.apache.org/.

[9] P. Bellavista, A. Corradi, L. Foschini, and A. Pernafini, "Data distribution service (dds): A performance comparison of opensplice and rti implementations," in

2013 IEEE Symposium on Computers and Communications (ISCC), July 2013, pp. 000 377–000 383.

[10] A. Shukla and Y. Simmhan, "Benchmarking distributed stream processing platforms for iot applications," in *Performance Evaluation and Benchmarking. Traditional - Big Data - Internet of Things*, R. Nambiar and M. Poess, Eds. Cham: Springer International Publishing, 2017, pp. 90–106.

[11] "Apache Storm," http://storm.apache.org/.

[12] "Apache Spark," https://spark.apache.org/.

[13] "Apache Flink," https://flink.apache.org.

[14] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, "Millwheel: Fault-tolerant stream processing at internet scale," in *Very Large Data Bases*, 2013, pp. 734–746.

[15] E. G. Renart, J. Diaz-Montes, and M. Parashar, "Data-driven stream processing at the edge," in *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*, May 2017, pp. 31–40.

[16] L. Liu, X. Zhang, M. Qiao, and W. Shi, "Safeshareride: Edge-based attack detection in ridesharing services," in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2018, pp. 17–29.

[17] "Data Never Sleeps 5.0," https://www.domo.com/learn/data-never-sleeps-5, 2017.

[18] 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*, ser. HotCloud'15. Berkeley, CA, USA: USENIX Association, 2015, pp. 21–21. [Online]. Available: http://dl.acm.org/citation.cfm?id=2827719.2827740

[19] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30–39, Jan 2017.

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

[21] D. Fesehaye, Y. Gao, K. Nahrstedt, and G. Wang, "Impact of cloudlets on interactive mobile cloud applications," in *2012 IEEE 16th International Enterprise Distributed Object Computing Conference*, Sept 2012, pp. 123–132.

[22] Z. Chen, W. Hu, J. Wang, S. Zhao, B. Amos, G. Wu, K. Ha, K. Elgazzar, P. Pillai, R. Klatzky, D. Siewiorek, and M. Satyanarayanan, "An empirical study of latency in an emerging class of edge computing applications for wearable cognitive assistance," in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, ser. SEC '17. New York, NY, USA: ACM, 2017, pp. 14:1–14:14. [Online]. Available: http://doi.acm.org/10.1145/3132211.3134458

[23] "RabbitMQ," https://www.rabbitmq.com/.

[24] B. Krishnamachari and K. Wright, "The publish-process-subscribe paradigm for the internet of things," 2017.

[25] A. Kapsalis, P. Kasnesis, I. S. Venieris, D. I. Kaklamani, and C. Z. Patrikakis, "A co-operative fog approach for effective workload balancing," *IEEE Cloud Computing*, vol. 4, no. 2, pp. 36–45, March 2017.

[26] B. Cheng, A. Papageorgiou, and M. Bauer, "Geelytics: Enabling on-demand edge analytics over scoped data sources," in *2016 IEEE International Congress on Big Data (BigData Congress)*, June 2016, pp. 101–108.

[27] "PubNub," https://www.pubnub.com/docs/tutorials/pubnub-functions.

[28] D. Happ and A. Wolisz, "Towards gateway to cloud offloading in iot publish/subscribe systems," in *2017 Second International Conference on Fog and Mobile Edge Computing (FMEC)*, May 2017, pp. 101–106.

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

[30] H. P. Sajjad, K. Danniswara, A. Al-Shishtawy, and V. Vlassov, "Spanedge: Towards unifying stream processing over central and near-the-edge data centers," in *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, Oct 2016, pp. 168–178.

[31] K. Hong, D. Lillethun, U. Ramachandran, B. Ottenwälder, and B. Koldehofe, "Mobile fog: A programming model for large-scale applications on the internet of things," in *Proceedings of the Second ACM SIGCOMM Workshop on Mobile Cloud Computing*, ser. MCC '13. New York, NY, USA: ACM, 2013, pp. 15–20. [Online]. Available: http://doi.acm.org/10.1145/2491266.2491270

[32] P. Ravindra, A. Khochare, S. Reddy, S. Sharma, P. Varshney, and Y. Simmhan, "ECHO: an adaptive orchestration platform for hybrid dataflows across cloud and edge," *CoRR*, vol. abs/1707.00889, 2017. [Online]. Available: http://arxiv.org/abs/1707.00889

[33] N. K. Giang, M. Blackstock, R. Lea, and V. C. M. Leung, "Distributed data flow: A programming model for the crowdsourced internet of things," in *Proceedings of the Doctoral Symposium of the 16th International Middleware Conference*, ser. Middleware Doct Symposium '15. New York, NY, USA: ACM, 2015, pp. 4:1–4:4. [Online]. Available: http://doi.acm.org/10.1145/2843966.2843970

[34] D. Hilley and U. Ramachandran, "Persistent temporal streams," in *Proceedings*

of the 10th ACM/IFIP/USENIX International Conference on Middleware, ser. Middleware '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 17:1–17:20. [Online]. Available: http://dl.acm.org/citation.cfm?id=1656980.1657003

[35] "streamCoCo," http://sumanttambe.com/documents/pubs/rti-edge-streaming.pdf.

[36] I. Maier and M. Odersky, "Deprecating the Observer Pattern with Scala.react," Tech. Rep., 2012.

[37] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli, "Optimal operator placement for distributed stream processing applications," in *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, ser. DEBS '16. New York, NY, USA: ACM, 2016, pp. 69–80. [Online]. Available: http://doi.acm.org/10.1145/2933267.2933312

[38] T. Li, J. Tang, and J. Xu, "Performance modeling and predictive scheduling for distributed stream data processing," *IEEE Transactions on Big Data*, vol. 2, no. 4, pp. 353–364, Dec 2016.

[39] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell, "R-storm: Resource-aware scheduling in storm," in *Proceedings of the 16th Annual Middleware Conference*, ser. Middleware '15. New York, NY, USA: ACM, 2015, pp. 149–161. [Online]. Available: http://doi.acm.org/10.1145/2814576.2814808

[40] T. Li, Z. Xu, J. Tang, and Y. Wang, "Model-free control for distributed stream data processing using deep reinforcement learning," *Proc. VLDB Endow.*, vol. 11, no. 6, pp. 705–718, Feb. 2018. [Online]. Available: https://doi.org/10.14778/3199517.3199521

[41] X. Wei, X. Wei, H. Li, Y. Zhuang, and H. Yue, "Topology-aware task allocation for distributed stream processing with latency guarantee," in *Proceedings of the*

2Nd International Conference on Advances in Image Processing, ser. ICAIP '18. New York, NY, USA: ACM, 2018, pp. 245–251. [Online]. Available: http://doi.acm.org/10.1145/3239576.3239621

[42] R. Ghosh and Y. Simmhan, "Distributed scheduling of event analytics across edge and cloud," *ACM Trans. Cyber-Phys. Syst.*, vol. 2, no. 4, pp. 24:1–24:28, Jul. 2018. [Online]. Available: http://doi.acm.org/10.1145/3140256

[43] R. Ghosh, S. P. R. Komma, and Y. L. Simmhan, "Adaptive energy-aware scheduling of dynamic event analytics across edge and cloud resources," *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pp. 72–82, 2018.

[44] X. Cai, H. Kuang, H. Hu, W. Song, and J. Lü, "Response time aware operator placement for complex event processing in edge computing," in *International Conference on Service-Oriented Computing*. Springer, 2018, pp. 264–278.

[45] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of things (iot): A vision, architectural elements, and future directions," *Future generation computer systems*, vol. 29, no. 7, pp. 1645–1660, 2013.

[46] A. Zanella, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi, "Internet of things for smart cities," *IEEE Internet of Things Journal*, vol. 1, no. 1, pp. 22–32, Feb 2014.

[47] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, ser. MCC '12. New York, NY, USA: ACM, 2012, pp. 13–16. [Online]. Available: http://doi.acm.org/10.1145/2342509.2342513

[48] N. Fernando, S. W. Loke, and W. Rahayu, "Mobile cloud computing: A survey," *Future Generation Computer Systems*, vol. 29, no. 1, pp.

84 – 106, 2013, including Special section: AIRCC-NetCoM 2009 and Special section: Clouds and Service-Oriented Architectures. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X12001318

[49] "Averages can be misleading: try a percentile," https://www.elastic.co/blog/averages-can-dangerous-use-percentile.

[50] R. Barazzutti, T. Heinze, A. Martin, E. Onica, P. Felber, C. Fetzer, Z. Jerzak, M. Pasin, and E. Rivire, "Elastic scaling of a high-throughput content-based publish/subscribe engine," in *2014 IEEE 34th International Conference on Distributed Computing Systems*, June 2014, pp. 567–576.

[51] M. Li, F. Ye, M. Kim, H. Chen, and H. Lei, "A scalable and elastic publish/subscribe service," in *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE, 2011, pp. 1254–1265.

[52] J. Gascon-Samson, F. P. Garcia, B. Kemme, and J. Kienzle, "Dynamoth: A scalable pub/sub middleware for latency-constrained applications in the cloud," in *2015 IEEE 35th International Conference on Distributed Computing Systems*, June 2015, pp. 486–496.

[53] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Design and evaluation of a wide-area event notification service," *ACM Transactions on Computer Systems (TOCS)*, vol. 19, no. 3, pp. 332–383, 2001.

[54] F. Cao and J. P. Singh, "Efficient event routing in content-based publish-subscribe service networks," in *IEEE INFOCOM 2004*, vol. 2, March 2004, pp. 929–940 vol.2.

[55] I. Aekaterinidis and P. Triantafillou, "Pastrystrings: A comprehensive content-based publish/subscribe dht network," in *26th IEEE International Conference on Distributed Computing Systems (ICDCS'06)*, 2006, pp. 23–23.

[56] S. Voulgaris, E. Riviere, A.-M. Kermarrec, M. Van Steen *et al.*, "Sub-2-sub: Self-organizing content-based publish subscribe for dynamic large scale collaborative networks." in *IPTPS*, 2006.

[57] "Amazon IoT," https://aws.amazon.com/iot/.

[58] "Redis," https://redis.io/.

[59] L. Sanchez, L. Muñoz, J. A. Galache, P. Sotres, J. R. Santana, V. Gutierrez, R. Ramdhany, A. Gluhak, S. Krco, E. Theodoridis, and D. Pfisterer, "Smartsantander: Iot experimentation over a smart city testbed," *Comput. Netw.*, vol. 61, pp. 217–238, Mar. 2014. [Online]. Available: http://dx.doi.org/10.1016/j.bjp.2013.12.020

[60] A. Khanna and R. Anand, "Iot based smart parking system," in *2016 International Conference on Internet of Things and Applications (IOTA)*, Jan 2016, pp. 266–270.

[61] P. Bellavista, A. Corradi, and A. Reale, "Quality of service in wide scale publish-subscribe systems," *IEEE Communications Surveys & Tutorials*, 2014.

[62] N. Carvalho, F. Araujo, and L. Rodrigues, "Scalable qos-based event routing in publish-subscribe systems," in *Network Computing and Applications, Fourth IEEE International Symposium on*. IEEE, 2005, pp. 101–108.

[63] H. Yang, M. Kim, K. Karenos, F. Ye, and H. Lei, "Message-oriented middleware with qos awareness," in *ICSOC/ServiceWave*, 2009.

[64] S. Guo, K. Karenos, M. Kim, H. Lei, and J. Reason, "Delay-cognizant reliable delivery for publish/subscribe overlay networks," in *2011 31st International Conference on Distributed Computing Systems*, June 2011, pp. 403–412.

[65] J. Gascon-Samson, J. Kienzle, and B. Kemme, "Multipub: Latency and cost-aware

global-scale cloud publish/subscribe," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, June 2017, pp. 2075–2082.

[66] S. Abdelwahab and B. Hamdaoui, "Fogmq: A message broker system for enabling distributed, internet-scale iot applications over heterogeneous cloud platforms," *arXiv preprint arXiv:1610.00620*, 2016.

[67] I. Awan, M. Younas, and W. Naveed, "Modelling qos in iot applications," in *2014 17th International Conference on Network-Based Information Systems*, Sept 2014, pp. 99–105.

[68] G. Bouloukakis, N. Georgantas, A. Kattepur, and V. Issarny, "Timeliness evaluation of intermittent mobile connectivity over pub/sub systems," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ser. ICPE '17. New York, NY, USA: ACM, 2017, pp. 275–286. [Online]. Available: http://doi.acm.org/10.1145/3030207.3030220

[69] P. Nguyen and K. Nahrstedt, "Resource management for elastic publish subscribe systems: A performance modeling-based approach," in *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, June 2016, pp. 561–568.

[70] Z. Jerzak and H. Ziekow, "The debs 2015 grand challenge," in *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '15. New York, NY, USA: ACM, 2015, pp. 266–268. [Online]. Available: http://doi.acm.org/10.1145/2675743.2772598

[71] A. Shukla, S. Chaturvedi, and Y. Simmhan, "Riotbench: An iot benchmark for distributed stream processing systems," vol. 29, 11 2017.

[72] J. Y. Fernndez-Rodrguez, J. A. lvarez Garca, J. Arias Fisteus, M. R. Luaces, and V. Corcoba Magaa, "Benchmarking real-time vehicle data streaming models for a

smart city," *Inf. Syst.*, vol. 72, no. C, pp. 62–76, Dec. 2017. [Online]. Available: https://doi.org/10.1016/j.is.2017.09.002

[73] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: ACM, 2015, pp. 450–462. [Online]. Available: http://doi.acm.org/10.1145/2749469.2749475

[74] C. Delimitrou and C. Kozyrakis, "ibench: Quantifying interference for datacenter applications," in *2013 IEEE international symposium on workload characterization (IISWC)*. IEEE, 2013, pp. 23–33.

[75] "ZeroMQ," http://zeromq.org/.

[76] "ZooKeeper," https://zookeeper.apache.org/.

[77] "Stress-Ng," http://kernel.ubuntu.com/~cking/stress-ng/.

[78] "NTP," http://www.ntp.org/.

[79] J. F. Shortle, J. M. Thompson, D. Gross, and C. M. Harris, *Fundamentals of queueing theory*. John Wiley & Sons, 2018, vol. 399.

[80] S. P. Curram and J. Mingers, "Neural networks, decision tree induction and discriminant analysis: An empirical comparison," *The Journal of the Operational Research Society*, vol. 45, no. 4, pp. 440–450, 1994. [Online]. Available: http://www.jstor.org/stable/2584215

[81] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006.

[82] A. Ng, "Learning curves," https://www.coursera.org/lecture/machine-learning/learning-curves-Kont7.

[83] S. J. Pan, Q. Yang *et al.*, "A survey on transfer learning," *IEEE Transactions on knowledge and data engineering*, vol. 22, no. 10, pp. 1345–1359, 2010.

[84] C. H. Papadimitriou, *Computational Complexity*. Addison-Wesley, 1994.

[85] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1990.

[86] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.

[87] L. Coetzee and J. Eksteen, "The internet of things - promise for the future? an introduction," in *IST-Africa Conference Proceedings, 2011*.

[88] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Comput. Surv.* [Online]. Available: http://doi.acm.org/10.1145/857076.857078

[89] "The Reactive Manifesto," http://www.reactivemanifesto.org, 2013.

[90] E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx, and W. d. Meuter, "A survey on reactive programming," *ACM Comput. Surv.*, vol. 45, no. 4, pp. 52:1–52:34, Aug. 2013. [Online]. Available: http://doi.acm.org/10.1145/2501654.2501666

[91] G. Salvaneschi, P. Eugster, and M. Mezini, "Programming with implicit flows," *IEEE Software*, vol. 31, no. 5, pp. 52–59, 2014. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/MS.2014.101

[92] G. H. Cooper and S. Krishnamurthi, "Embedding Dynamic Dataflow in a Call-by-value Language," in *Programming Languages and Systems*. Springer, 2006, pp. 294–308.

[93] C. Elliott and P. Hudak, "Functional Reactive Animation," in *ACM SIGPLAN Notices*, vol. 32, no. 8. ACM, 1997, pp. 263–273.

[94] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi, "Flapjax: A Programming Language for Ajax Applications," in *ACM SIGPLAN Notices*, vol. 44, no. 10. ACM, 2009, pp. 1–20.

[95] A. Courtney, "Frappe: Functional reactive programming in java," in *Proceedings of the Third International Symposium on Practical Aspects of Declarative Languages*, ser. PADL '01. London, UK, UK: Springer-Verlag, 2001, pp. 29–44. [Online]. Available: http://dl.acm.org/citation.cfm?id=645771.667929

[96] D. Synodinos, "Reactive Programming as an Emerging Trend," http://www.infoq.com/news/2013/08/reactive-programming-emerging, 2013.

[97] "Reactive Programming at Netflix," http://techblog.netflix.com/2013/01/reactive-programming-at-netflix.html, 2013.

[98] "The Reactive Extensions (Rx)," http://msdn.microsoft.com/en-us/data/gg577609.aspx.

[99] "Reactive-Streams," http://www.reactive-streams.org/.

[100] "RxJava," https://github.com/ReactiveX/RxJava.

[101] Z. Jerzak and H. Ziekow, "The ACM DEBS 2013 Grand Challenge," http://www.orgs.ttu.edu/debs2013/index.php?goto=cfchallengedetails, 2013.

[102] G. Salvaneschi, J. Drechsler, and M. Mezini, "Towards Distributed Reactive Programming," in *Coordination Models and Languages*, ser. Lecture Notes in Computer Science, R. Nicola and C. Julien, Eds. Springer Berlin Heidelberg, 2013, vol. 7890, pp. 226–235. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-38493-6_16

[103] A. Voellmy and P. Hudak, "Nettle: Taking the Sting Out of Programming Network Routers," in *Practical Aspects of Declarative Languages*, ser. Lecture Notes in Computer Science, R. Rocha and J. Launchbury, Eds. Springer Berlin Heidelberg, 2011, vol. 6539, pp. 235–249. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-18378-2_19

[104] S. Magnenat, P. Rétornaz, M. Bonani, V. Longchamp, and F. Mondada, "ASEBA: A Modular Architecture for Event-Based Control of Complex Robots," *Mechatronics, IEEE/ASME Transactions on*, vol. 16, no. 2, pp. 321–329, April 2011.

[105] S. Appel, S. Frischbier, T. Freudenreich, and A. Buchmann, "Eventlets: Components for the Integration of Event Streams with SOA," in *Service-Oriented Computing and Applications (SOCA), 2012 5th IEEE International Conference on*, Dec 2012, pp. 1–9.

[106] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized Streams: An Efficient and Fault-tolerant Model for Stream Processing on Large Clusters," in *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing*, ser. HotCloud'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=2342763.2342773

[107] N. Marz and J. Warren, *Big Data: Principles and Best Practices of Scalable Real-time Data Systems*. O'Reilly Media, 2013.

[108] A. Corsaro, "Escalier: The Scala API for DDS," https://github.com/kydos/escalier.

[109] "Reactive Open DDS," http://www.ociweb.com/resources/news/2014/11/05/reactive-opendds-part-i.

[110] "Building Reactive Data-centric Applications with Vortex, Apache Spark and ReactiveX," http://www.prismtech.com/products/vortex/resources/youtube-videos-slideshare/building-reactive-data-centric-applications-vort.

[111] "Dependency Injection Pattern," https://msdn.microsoft.com/en-us/magazine/cc163739.aspx.

[112] Y. Simmhan, S. Aman, A. Kumbhare, R. Liu, S. Stevens, Q. Zhou, and V. Prasanna, "Cloud-based software platform for big data analytics in smart grids," *Computing in Science Engineering*, vol. 15, no. 4, pp. 38–47, July 2013.

[113] C.-C. Hung, G. Ananthanarayanan, P. Bodk, L. Golubchik, M. Yu, V. Bahl, and M. Philipose, "Videoedge: Processing camera streams using hierarchical clusters," October 2018.

[114] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: ACM, 2013, pp. 423–438. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522737

[115] D. O'Keeffe, T. Salonidis, and P. Pietzuch, "Frontier: resilient edge processing for the internet of things," *Proceedings of the VLDB Endowment*, vol. 11, no. 10, pp. 1178–1191, 2018.

[116] "AWS IoT Greengrass," https://aws.amazon.com/greengrass/.

[117] "Azure IoT Edge," https://azure.microsoft.com/en-us/services/iot-edge/.

[118] "Apache Edgent," http://edgent.apache.org/.

[119] D. Novaković, N. Vasić, S. Novaković, D. Kostić, and R. Bianchini, "Deepdive: Transparently identifying and managing performance interference in virtualized environments," in *USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 219–230. [Online]. Available: http://dl.acm.org/citation.cfm?id=2535461.2535489

[120] R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-clouds: managing performance interference effects for qos-aware clouds," in *Proceedings of the 5th European conference on Computer systems*. ACM, 2010, pp. 237–250.

[121] R. Eidenbenz and T. Locher, "Task allocation for distributed stream processing," in *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, April 2016, pp. 1–9.

[122] C. H. Papadimitriou and M. Yannakakis, "Towards an architecture-independent analysis of parallel algorithms," *SIAM J. Comput.*, vol. 19, no. 2, pp. 322–328, 1990.

[123] J. Hoogeveen, J. Lenstra, and B. Veltman, "Three, four, five, six, or the complexity of scheduling with communication delays," *Operations Research Letters*, vol. 16, no. 3, pp. 129–137, 1994.

[124] C. Delimitrou and C. Kozyrakis, "ibench: Quantifying interference for datacenter applications," in *Workload Characterization (IISWC), 2013 IEEE International Symposium on*. IEEE, 2013, pp. 23–33.

[125] "Beagle Bone Black," https://beagleboard.org/black.

[126] D. Cordeiro, G. Mounié, S. Perarnau, D. Trystram, J.-M. Vincent, and F. Wagner, "Random graph generation for scheduling simulations," *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*, 2010. [Online]. Available: http://eudl.eu/doi/10.4108/ICST.SIMUTOOLS2010.8667

[127] G. T. Lakshmanan, Y. Li, and R. Strom, "Placement strategies for internet-scale data stream systems," *IEEE Internet Computing*, vol. 12, no. 6, pp. 50–60, Nov 2008.

[128] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer, "Network-aware operator placement for stream-processing systems," in *22nd International Conference on Data Engineering (ICDE'06)*, April 2006, pp. 49–49.

[129] S. Rizou, F. Durr, and K. Rothermel, "Solving the multi-operator placement problem in large-scale operator networks," in *2010 Proceedings of 19th International Conference on Computer Communications and Networks*, Aug 2010, pp. 1–6.

[130] J. Xu, Z. Chen, J. Tang, and S. Su, "T-storm: Traffic-aware online scheduling in storm," in *2014 IEEE 34th International Conference on Distributed Computing Systems*, June 2014, pp. 535–544.

[131] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli, "Optimal operator replication and placement for distributed stream processing systems," *ACM SIGMETRICS Performance Evaluation Review*, vol. 44, no. 4, pp. 11–22, 2017.

[132] G. T. Lakshmanan, Y. Li, and R. Strom, "Placement of replicated tasks for distributed stream processing systems," in *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*. ACM, 2010, pp. 128–139.

[133] R. Eidenbenz and T. Locher, "Task allocation for distributed stream processing," in *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*. IEEE, 2016, pp. 1–9.

[134] J. Gedeon, M. Stein, L. Wang, and M. Muehlhaeuser, "On scalable in-network operator placement for edge computing," in *2018 27th International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 2018, pp. 1–9.

[135] G. Amarasinghe, M. D. de Assunçao, A. Harwood, and S. Karunasekera, "A data stream processing optimisation framework for edge computing applications," in *2018 IEEE 21st International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE, 2018, pp. 91–98.

[136] T. Elgamal, A. Sandur, P. Nguyen, K. Nahrstedt, and G. Agha, "Droplet: Distributed operator placement for iot applications spanning edge and cloud resources," in *2018*

IEEE 11th International Conference on Cloud Computing (CLOUD). IEEE, 2018, pp. 1–8.

[137] S. Khare, H. Sun, K. Zhang, J. Gascom-Samson, A. Gokhale, and X. Koutsoukos, "Scalable Edge Computing Architectures for Low Latency Data Dissemination in Topic-based Publish/Subscribe," in *3rd ACM/IEEE Symposium on Edge Computing (SEC)*, Bellevue, WA, USA, Oct. 2018, pp. 214–227.

[138] A. Al-Ali *et al.*, "Role of internet of things in the smart grid technology," *Journal of Computer and Communications*, vol. 3, no. 05, p. 229, 2015.

[139] W. Z. Khan, M. Y. Aalsalem, M. K. Khan, M. S. Hossain, and M. Atiquzzaman, "A reliable internet of things based architecture for oil and gas industry," in *2017 19th International conference on advanced communication Technology (ICACT)*. IEEE, 2017, pp. 705–710.

[140] G. Ananthanarayanan, V. Bahl, P. Bodk, K. Chintalapudi, M. Philipose, L. R. Sivalingam, and S. Sinha, "Real-time video analytics the killer app for edge computing," *IEEE Computer*, October 2017. [Online]. Available: https://www.microsoft.com/en-us/research/publication/real-time-video-analytics-killer-app-edge-computing/

[141] "ALPR Applications," http://www.anpr.net/anpr_09/anpr_applicationareas.html.

[142] C. A. Rahman, W. Badawy, and A. Radmanesh, "A real time vehicle's license plate recognition system," in *Proceedings of the IEEE Conference on Advanced Video and Signal Based Surveillance, 2003.*, July 2003, pp. 163–166.

[143] "OpenALPR Design," https://github.com/openalpr/openalpr/wiki/OpenALPR-Design.

[144] T. D. Duan, T. L. H. Du, T. V. Phc, and N. V. Hoàng, "Building an automatic vehicle license-plate recognition system."

[145] Y. Yanamura, M. Goto, D. Nishiyama, M. Soga, H. Nakatani, and H. Saji, "Extraction and tracking of the license plate using hough transform and voted block matching," in *IEEE IV2003 Intelligent Vehicles Symposium. Proceedings (Cat. No.03TH8683)*, June 2003, pp. 243–246.

[146] K. I. Kim, K. Jung, and J. H. Kim, "Color texture-based object detection: An application to license plate localization," in *Pattern Recognition with Support Vector Machines*, S.-W. Lee and A. Verri, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 293–309.

[147] Yo-Ping Huang, Shi-Yong Lai, and Wei-Po Chuang, "A template-based model for license plate recognition," in *IEEE International Conference on Networking, Sensing and Control, 2004*, vol. 2, March 2004, pp. 737–742 Vol.2.

[148] X. Zhai, F. Bensaali, and R. Sotudeh, "Ocr-based neural network for anpr," in *2012 IEEE International Conference on Imaging Systems and Techniques Proceedings*, July 2012, pp. 393–397.

[149] "openALPR," https://github.com/openalpr/openalpr.

[150] " Car Detection and Recognition Using DNN Networks ," https://medium.com/swlh/car-detection-recognition-using-dnn-networks-3ac7603d2e9b.

[151] "Contrast Enhancement Techniques," http://www.zemris.fer.hr/projects/LicensePlates/english.

[152] "Reactive Extensions," http://reactivex.io/.

[153] "RxPy," https://github.com/ReactiveX/RxPY.

[154] “Raspberry Pi 3 Model B,” https://www.raspberrypi.org/products/raspberry-pi-3-model-b/.