Geo-distributed Edge and Cloud Resource Management for Low-latency Stream Processing

by

Jinlai Xu

M.S. in Software Engineering, China University of Geosciences, China, 2015

B.E. in Software Engineering, China University of Geosciences, China, 2012

Submitted to the Graduate Faculty of

the School of Computing and Information in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2021

UNIVERSITY OF PITTSBURGH

SCHOOL OF COMPUTING AND INFORMATION

This dissertation was presented

by

Jinlai Xu

It was defended on

November 23^{rd} 2021

and approved by

Dr. Balaji Palanisamy, School of Computing and Information, University of Pittsburgh

Dr. David Tipper, School of Computing and Information, University of Pittsburgh

Dr. Amy Babay, School of Computing and Information, University of Pittsburgh

Dr. Qingyang Wang, School of Electrical Engineering and Computer Science, Louisiana State University

Dissertation Director: Dr. Balaji Palanisamy, School of Computing and Information, University of Pittsburgh

Geo-distributed Edge and Cloud Resource Management for Low-latency Stream Processing

Jinlai Xu, PhD

University of Pittsburgh, 2021

The proliferation of Internet-of-Things (IoT) devices is rapidly increasing the demands for efficient processing of low latency stream data generated close to the edge of the network. Edge Computing provides a layer of infrastructure to fill latency gaps between the IoT devices and the back-end cloud computing infrastructure. A large number of IoT applications require continuous processing of data streams in real-time. Edge computing-based stream processing techniques that carefully consider the heterogeneity of the computing and network resources available in the geo-distributed infrastructure provide significant benefits in optimizing the throughput and end-to-end latency of the data streams. Managing geo-distributed resources operated by individual service providers raises new challenges in terms of effective global resource sharing and achieving global efficiency in the resource allocation process.

In this dissertation, we present a distributed stream processing framework that optimizes the performance of stream processing applications through a careful allocation of computing and network resources available at the edge of the network. The proposed approach differentiates itself from the state-of-the-art through its careful consideration of data locality and resource constraints during physical plan generation and operator placement for the stream queries. Additionally, it considers co-flow dependencies that exist between the data streams to optimize the network resource allocation through an application-level rate control mechanism. The proposed framework incorporates resilience through a cost-aware partial active replication strategy that minimizes the recovery cost when applications incur failures. The framework employs a reinforcement learning-based online learning model for dynamically determining the level of parallelism to adapt to changing workload conditions. The second dimension of this dissertation proposes a novel model for allocating computing resources in edge and cloud computing environments. In edge computing environments, it allows service providers to establish resource sharing contracts with infrastructure providers *apriori* in a latency-aware manner. In geo-distributed cloud environments, it allows cloud service providers to establish resource sharing contracts with individual datacenters *apriori* for defined time intervals in a cost-aware manner. Based on these mechanisms, we develop a decentralized implementation of the contract-based resource allocation model for geo-distributed resources using Smart Contracts in Ethereum.

Keywords: resource management, stream processing, edge computing, resource sharing, cloud computing, reinforcement learning, blockchain.

Table of Contents

List of Tables

List of Figures

Preface

When I started my PhD study at Pitt, I didn't anticipate the uncertainties and unknowns along the journey. All I expected was an incredible adventure and gradually I realized that PhD is such an exciting journey! There are both highs and lows and all the sweetness and bitterness have made the past six years a truly valuable experience for me to learn and grow. It would not have been possible for me to go through this journey without all the help and support I received throughout.

First and foremost, I am immensely grateful to my advisor, Dr. Balaji Palanisamy, for his supervision. He has been patiently mentoring me, directing my research, expanding my research vision, and encouraging me to explore new research directions. He never hesitates to brainstorm ideas, provide constructive suggestions and discuss research opportunities. How he communicates, discusses ideas and gives suggestions have set an example of excellence as an advisor, researcher, and instructor.

I would like to thank the dissertation committee members, Dr. David Tipper, Dr. Amy Babay, and Dr. Qingyang Wang for their insights and invaluable suggestions for future research directions in my dissertation. Also, I want to thank Dr. James Joshi and Dr. Vladimir Zadorozhny for serving on my comprehensive examination committee.

I would like to express my gratitude to Mr. Zhongwen Luo, who advised me through my undergraduate and graduate study. He never hesitates to point out the weaknesses of your work and has set an excellent example of researcher and instructor. I also want to thank Dr. Deze Zeng, and Dr. Hong Yao, who advised my research project during my graduate study when I was a noob just starting to know the basics of research.

Next, I am grateful to my collaborators and co-authors over the years: Dr. Qingyang Wang, Dr. Heiko Ludwig, Mr. Sandeep Gopisetty, Dr. Yuzhe Tang, Dr. S.D Madhu Kumar, Dr. Chao Li, Dr. Runhua Xu, Mr. Jingzhe Wang. They have inspired me in several ways.

Also, I thank my internship mentor in industry, Dr. Benjamin Heintz from Facebook. He showed me how much impact is possible by developing robust distributed systems with the intention to serve billions of people. I want to thank all my friends and colleagues when I interned in Facebook. They have set a great example of responsive and responsible engineers.

I am thankful to all the people I got to befriend with in *LERSAIS* and in the *University of Pittsburgh*. In particular, I would like to extend my special gratitude to Chao and Runhua, who took extra time to help me get familiar with the Pitt campus. We have had lunches and dinners together and they drove me to different places when I didn't have a car. I also want to thank all my other friends in Pittsburgh. You have really made me feel like at home when we were all away from home.

Last but not least, I would like to express my endless gratitude to my family: my parents, Wenjin and Hongying, for your unconditional love and perpetual support in my entire life; my grandfather, Shengjun, for being strict when I seem to lose directions and for always encouraging me to aim high; my wife, Jiaojiao, for always supporting me despite all the difficulties and uncertainties. This dissertation is dedicated to you.

1.0 Introduction

The proliferation of Internet-of-Things (IoT) devices is rapidly increasing the demands for efficient processing of low latency stream data generated close to the edge of the IoT network. IHS Markit forecasts that the number of IoT devices will increase to more than 125 billion by 2030[138]. Applications in the IoT era have strong demands for low latency computing. For instance, telesurgery applications have a critical latency requirement of 200 ms with less than 1 ms jitter [5][101]. Similarly, virtual reality applications (games) that use head-tracked systems require latencies less than 16 ms to achieve perceptual stability [120]. Connected autonomous vehicle applications (collision warning, autonomous driving, traffic efficiency, etc.) have latency requirements between 10 ms to 100 ms [14]. In addition, in the era of Big Data, with data growing massively in scale and velocity, geo-distributed cloud and edge computing provide effective solutions to process large amounts of data in real-time.

Stream processing engines executing on edge computing resources is a promising approach to support low latency big data processing. Stream data processing is an integral component of low-latency data analytic systems and several open-source systems (e.g., Apache Kafka [134], Apache Storm[135], and Apache Flink [133]) provide engines for efficient processing of data streams. These systems optimize the performance of stream data processing for achieving high throughput and low (or bounded) response time (latency) for the stream queries. However, these stream processing solutions are designed for cloud computing environments where there is no scarcity of computing resources, and hence, they are not suitable for edge computing environments that have resource and network bandwidth limitations. For example, the low-profile edge devices (such as smart gateways placed near IoT devices) may not be able to handle the same workload as regular servers in a cloud datacenter. In this dissertation, we present a distributed stream processing platform that optimizes the resource allocation for stream queries by carefully considering the data locality and resource constraints during physical plan generation and operator placement in edge computing environments [160]. It includes the following novel features to facilitate a system-wide optimization of computing and networking resource usage for processing large volumes of data streams near the edge. First, it employs a novel data locality-aware approach to optimize the resource allocation for the stream queries executing in resource-constrained edge computing environments. Second, the system schedules each operator of the queries by carefully considering the dynamically varying load conditions and resource requirement for each operation to further improve the query performance. Third, the system considers coflow [39] dependencies which group the dependent network flows that exist in the stream processing application. It prioritizes smaller coflows to complete first and increases the overall coflow completion rate which in turn improves the overall efficiency of the network resource usage. In addition to the above three aspects, the distributed stream processing platform also achieves system-wide fault tolerance while meeting the latency requirement for applications in edge computing environments [159]. The proposed approach employs a novel resilient

physical plan generation for stream queries, which carefully considers the fault tolerance resource budget and the risk of each operator in the query to partially actively replicate the high-risk operators to minimize the recovery time when there is a failure. The proposed techniques also consider the placement of the backup components (e.g. active replication) to further optimize the processing latency during recovery and reduce the overhead of checkpointing delays. Finally, we propose a reinforcement learning(RL)-based approach that learns how to dynamically scale the operations of the stream processing applications in an online fashion to adapt to workload variations [156]. Based on the recent developments in RL algorithms, we model the DSP scaling problem as a contextual Multi-Armed Bandit (MAB) problem, which is reduced from the original Markov Decision Process (MDP). With the above simplification, the elastic parallelism configuration can be efficiently solved using the state-of-art algorithms that work well on MAB problems [80]. It can automatically achieve tradeoffs between exploring the solution space to find an optimal solution (exploration) and utilizing the data gathered from the previous trials (exploitation). We investigate the use of LinUCB [80] algorithm to dynamically decide the parallelism configuration during the execution of the DSP application that aims to improve multiple QoS metrics including end-to-end latency upper bound, throughput and resource usage. We further improve the sample efficiency of LinUCB using a model-based method which is based on a queuing model simulation to pre-train the RL agent to improve the accuracy of the initial parameters.

Cloud Computing has been a cost-effective solution[76][74] to address computing needs, however, clouds fail to meet low latency requirements of modern computing applications that demand strict guarantees on response times. As large datacenters are often located at a longer distance from the source of data generation, moving data to remote cloud datacenters may lead to high latency (response time) and as a result, it cannot support latency-sensitive applications such as location-based augmented reality games, real-time smart grid management and real-time navigation using wearables. The Edge/Fog Computing model [21, 121, 87, 18, 6, 50, 143] provides an additional layer of computing infrastructure for storing and processing data at the edge, allowing low latency applications to meet their response time requirements effectively. There has been a few recent work [44, 1] addressing the resource management and resource provisioning challenges in edge computing. A fundamental assumption in these solutions includes a tight coupling of the management of the Edge Computing Infrastructures (ECIs) with the service management performed by Service Providers (SPs). As a result, the computing resources present at the edge micro datacenters (MDCs) are coupled and controlled directly by edge Service Providers (SPs). Such a coupled model for management of ECIs by SPs significantly limits the cost-effectiveness and opportunities for latency-optimized provisioning of edge infrastructure resources to applications [158]. When the management of the Edge computing infrastructures is controlled by the SPs, it results in an increased infrastructure cost and a decrease in the overall utilization of the system leading to poor cost-effectiveness. We propose a novel resource allocation model for allocating computing resources in edge computing platforms that allows edge service providers to establish resource sharing contracts with edge infrastructure providers *apriori*. It employs a decoupled model where the management of ECIs is independent of that of the SPs and as a result, it

provides increased resource utilization and minimizes job execution latency. We also extend the contracts-based resource sharing model for federated geo-distributed cloud environments that allows Cloud Service Providers (CSPs) to establish resource sharing contracts with individual datacenters *apriori* for defined time intervals in a cost-aware manner. Based on the established contracts, individual CSPs employ a contracts cost and duration aware job scheduling and provisioning algorithm that enables jobs to complete and meet their response time requirements while achieving both global resource allocation efficiency and local fairness in the profit earned [155, 157]. We enhance the resilience of these proposed centralized resource allocation and management mechanisms to prevent attacks by adversaries and system failures by designing a decentralized implementation of the contract-based resource sharing model using the Smart Contracts in Ethereum.

In the rest of this chapter, we first outline the key research tasks of this dissertation and then briefly present the organization of the remaining chapters.

1.1 Overview of research thrusts

This dissertation has two major research thrusts: (i) optimizing stream processing applications in edge computing, and (ii) resource allocation and management for geo-distributed edge and cloud resources. With the help of Figure 1.1, we discuss the research thrusts in detail.

1.1.1 Research Thrust 1: Optimizing stream processing applications in edge computing

This research thrust consists of three aspects of optimization for stream processing applications in edge computing. It is organized as three sub thrusts.

Research Thrust 1.1: Optimizing low-latency stream processing applications in edge computing

This research thrust focuses on optimizing low-latency stream processing applications in edge computing. Specifically, our objective is to minimize the end-to-end latency for the stream processing applications deployed in edge computing environments by minimizing the impact of bottlenecks. To achieve this goal, we consider various conditions that create bottlenecks in an edge computing environment. The data locality optimization feature in our approach maximizes the data locality by placing the selective operators near the *source*. We perform load optimization to avoid back-pressure to improve the overall latency by considering the computational load of each operator and the resource capacity of each node to improve the overall performance. Besides the above optimizations on scheduling, the proposed system also considers the coflows [39] that capture the dependency between the flows waiting to be scheduled in the network. It may also cause bottlenecks due to the flow dependency in the streams. The proposed optimization adjusts the stream rates to optimize the bandwidth allocation in order to improve the coflow completion rate to enhance the overall performance of the stream processing applications.

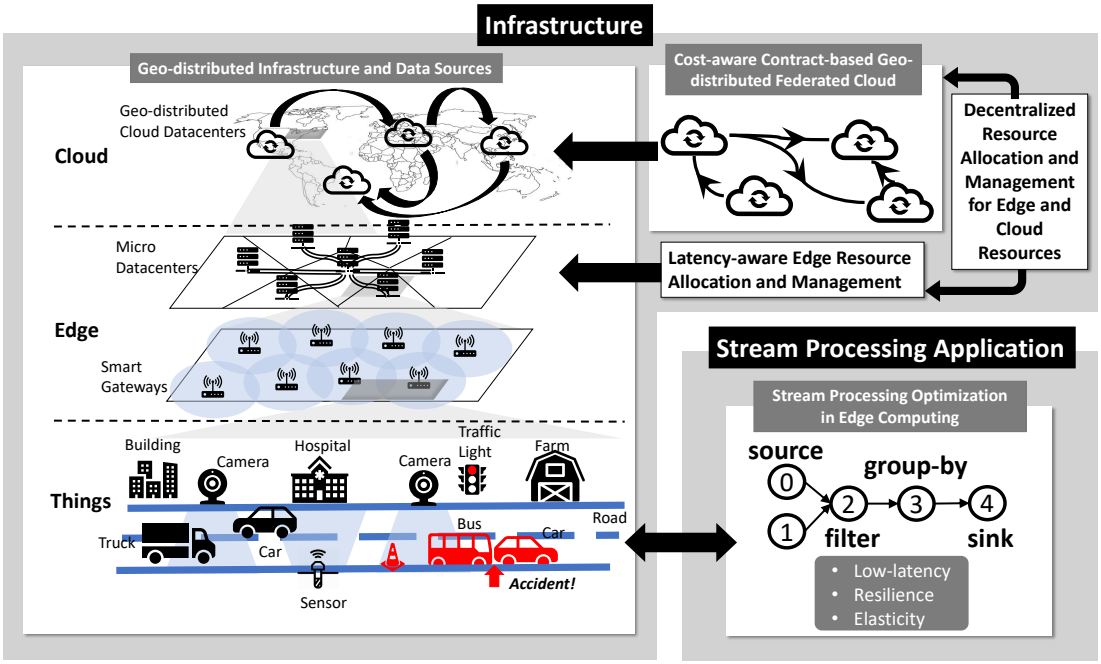


Figure 1.1: An overview of research thrusts

Research Thrust 1.2: Resilient stream processing in edge computing

Fault tolerance is an important aspect of edge computing as many IoT applications require both high accuracy and timeliness of results. As edge infrastructures may include some unreliable devices and components in a highly dynamic environment, failures are more of a norm than exception [81]. Thus, to support reliable delivery of low latency stream processing over edge computing, we need a highly fault-tolerant stream processing solution that understands the properties of the edge computing environment for meeting both the latency and fault tolerance requirements. Checkpointing[145] and replication [63] represent two classical techniques for fault-tolerant stream processing. The idea behind replication is to withstand the failure by using additional backup resources. The checkpointing mechanisms on the other hand periodically store the state of the operators in persistent storage to create timestamped snapshots of the application. Hybrid methods employ a combination of both checkpointing and replication. They are referred to as adaptive checkpointing and replication techniques. Adaptive checkpoint and replications schemes have been proposed in several domains [129, 89, 168, 142, 60, 128]. The goal of combining active replication and checkpoint mechanisms is to achieve seamless recovery compared to pure checkpointing. While adaptive checkpointing and replication is a promising approach, its application in edge computing is challenged in several aspects. The heterogeneous nature of both physical nodes and the network components in an edge computing environment significantly challenges. In this thrust, we develop a novel resilient stream processing framework

that achieves system-wide fault tolerance while meeting the latency requirement for the applications in the edge computing environment. The proposed approach employs a novel resilient physical plan generation for the stream queries, which carefully considers the fault tolerance resource budget and the risk of each operator in the query to partially actively replicate the high-risk operators in order to minimize the recovery time when there is a failure. The proposed techniques also consider the placement of the backup components (e.g. active replication) to further optimize the processing latency during recovery and reduce the overhead of checkpointing delays.

Research Thrust 1.3: Elastic stream processing in edge computing

In this thrust, we develop an elastic stream processing solution for edge computing. Elasticity in cloud computing services has been widely studied in the past [28, 69, 59]. These techniques adapt to the workload changes for different services (e.g., web services) by dynamically increasing or decreasing the number of instances (or virtual machines) to serve the users. However, for the current state-of-art distributed stream processing engines, adapting the number of tasks (parallelism) allocated to each operation needs to be manually tuned and it requires a lot of try-and-error efforts as well as experience to do it. This is further challenged by the heterogeneity of computing nodes and network resources in edge computing environments. Recently, RL-based methods were developed in the distributed system domain to enhance heuristic-based system optimization algorithms and provide more effective solutions to problems for which heuristic-based solutions are less effective. However, applying RL methods in the systems domain incurs several challenges. A key performance factor in effective RL-based methods is the sample efficiency of the method. Currently, most RL methods are based on deep learning techniques that use deep neural networks (DNNs) to handle the approximation of the environment dynamics and the reward distribution. The use of DNN enhances the models to handle more complex conditions. However, most of the DNN-based RL algorithms require a large amount of data to converge a good result which makes the optimization of the sample efficiency even harder. Therefore, improving the sample efficiency is a critical problem when applying RL-based algorithms for optimizing distributed systems management. In this thrust, we propose a reinforcement learning-based mechanism to learn how to dynamically adapt the workload changes through an online learning model and strategy. Compared to previous reinforcement learning-based methods that explore the solution space of the problem by randomly sampling the possible solutions, the proposed online learning method is based on LinUCB[80] as the base method and a predefined model of parallelism of each operation which improves the sample efficiency and decreases the cost (time) to converge to a good parallelism configuration.

1.1.2 Research Thrust 2: Resource allocation and management for geo-distribu-ted edge and cloud resources

In this thrust, we develop new mechanisms to efficiently allocate and manage geo-distributed edge and cloud resources. It is organized as three sub thrusts.

Research Thrust 2.1: Latency-aware resource allocation and management for geo-distributed edge resources

In this thrust, we propose a new resource allocation model for allocating edge computing resources that allows edge service providers to establish resource sharing contracts with edge infrastructure providers *apriori*. Based on the established contracts, service providers employ a latency-aware scheduling and resource provisioning algorithm that enables tasks to complete and meet their latency requirements while achieving both global and local resource allocation efficiency and fairness. In contrast to existing solutions, our proposed model decouples the infrastructure management from service management, enabling the ECIs to be managed by Edge Infrastructure Providers (EIPs) independently of the service provisioning and service management at the SPs. Such a decoupled model enables EIPs to join up to establish an Edge Computing Infrastructure Federation (ECIF) to provide resources to the Edge Computing applications provisioned and managed by the SPs. In addition, the model provides increased opportunities for resource consolidation and utilization as the geo-distributed ECIs can be jointly managed and allocated to maximize application utility and minimize cost. Specifically, we divide the geo-distributed locations into regions using Weighted Voronoi Diagrams (WVD)[65] which reduces the complexity to find the nearest computational resources. Then, for each region, based on the McAfee mechanism [90], we design auction mechanisms to let the ECIs and SPs trade resources. It includes a latency-aware bid strategy designed for the SPs and a cost-aware bid strategy designed for ECIs. Finally, the approach uses a latency-aware provisioning mechanism for the SPs to deploy their services on the geo-distributed resources.

Research Thrust 2.2: Cost-aware resource allocation and management for geo-distributed cloud resources

In this thrust, we propose a contracts-based resource sharing architecture for Cloud Service Providers (CSPs) to share resources across globally geo-distributed datacenters. The proposed approach allows to establish resource sharing contracts with individual datacenters. Based on the established contracts, individual CSPs employ a contracts cost and duration aware job scheduling and provisioning algorithm that enables jobs to complete and meet their response time requirements while achieving both global resource allocation efficiency and local fairness in the profit earned.

Research Thrust 2.3: Decentralized resource allocation and management for geo-distributed edge and cloud resources

The centralized resource allocation and management mechanisms described in the previous two thrusts may suffer from attacks by adversaries, system failures and their security is generally limited to a single point of trust by the coordinator. On one hand, the adversaries can tamper with the bids or the contracts to influence the integrity of both the bids and the generated contracts. On the other hand, the centralized auction system is prone to Denial of Service (DoS) attacks and system failures. Also, in a centralized mechanism, all participants need to trust the coordinator to handle the trading process which may not be always possible in real world. In this thrust, we design a decentralized resource allocation and sharing

mechanism for supporting a decentralized trusted platform for infrastructure providers and service providers to interact and trade resources.

1.2 Chapters overview

The rest of the dissertation is organized as follows: Chapter 2 discusses the related work. In Chapter 3, the optimization for low-latency stream processing in edge computing is presented. In Chapter 4, we present the techniques for resilient stream processing and in Chapter 5, we introduce our approach for elastic stream processing in edge computing. In Chapter 6, we discuss the techniques to support latency-aware resource allocation and management for geo-distributed edge. In Chapter 7, we discuss the proposed cost-aware resource allocation and management for geo-distributed clouds. In Chapter 8, we present the techniques for decentralized resource allocation for geo-distributed resources using smart contracts. Finally, we conclude and discuss some future work directions in Chapter 9.

2.0 Related Work and Preliminaries

In this chapter, we discuss the related work and preliminaries. We begin with the discussion of existing work on stream processing in edge computing. We then review the literature on resource management for geo-distributed edge and cloud computing. Finally, we discuss the preliminaries of stream processing.

2.1 Stream Processing in Edge Computing

2.1.1 Stream Processing Engine

As modern stream applications have strong requirements on latency and throughput, stream processing has gained considerable attention in recent years. Several open-source stream processing frameworks have been developed recently. Key examples include Kafka[134], Flink[133], Storm[135]. There have also been some efforts on developing stream processing engines for edge computing environments. For example, Edgent [132] is an incubating project in Apache. It provides a runtime environment for implementing a real-time data analytic system in the edge environment. Pisani et al. [113] proposed LMC that provides a runtime to deploy light operators on embedded devices at the edge.

2.1.2 Stream Processing Optimization

Wang et al. [146] proposed an algorithm to optimize the service entity placement for social virtual reality applications in edge computing by modeling and solving a combinatorial optimization problem considering the activation, placement, proximity, and colocation costs. In [147], the authors discuss the edge server placement problem in the Mobile Edge Computing environment for balancing the workloads of edge servers and minimizing the access delay between the mobile users and edge servers. Mencagli et al. [95] proposed a tool for supporting programmers during the design phase of data stream processing applications by modeling the backpressure effects. Fu et al. [49] proposed an edge-friendly stream processing engine for multi-core edge computing environments. Hiessl et al. [61] proposed a solution that extends an ILP (Integer Linear Programming) model [33] to optimize the reconfiguration of the operator placement in the edge environment.

2.1.3 Stream Processing Fault tolerance

Fault tolerance is a well-studied topic in the context of cloud computing and Big Data processing. In stream processing systems, fault tolerance mechanisms have primarily focused on developing two kinds of solutions namely (i) checkpointing and relaying techniques [31, 124] that have low resource overhead and higher recovery time and (ii) active replication techniques [63, 25] that incur high resource cost and

lower recovery time. These solutions do not optimize for latency and recovery time requirements that are critical in edge-based IoT applications. Su and Zhou [128] proposed a hybrid solution employing both checkpointing and active replication by selectively choosing the operators to be actively replicated using a minimal completion tree. A key limitation of this approach is that the operators that are actively replicated may not fail simultaneously which leads to higher resource usage cost. Heinze et al. [60] proposed an adaptive mechanism that enables the operator to switch between active and reserved status. The adaptive mechanism proposed by Upadhyaya et al. [142] provides fault tolerance for database queries by optimizing recovery latency. The hybrid solution proposed by Zhang et al. [168] allows the operator to change from passive backup to active replication when failure happens. Martin et al. [89] proposed an adaptive hybrid mechanism, which can alternate between six fault tolerance schema based on user-defined recovery time and cost. The adaptive hybrid mechanism for HPC systems proposed by Subasi et al. [129] selects partial tasks to be replicated using active replication.

2.1.4 Elastic Stream Processing

There have been several efforts in recent years to achieve elastic stream processing in cloud computing environments. To dynamically scale the application, several different approaches have been developed including techniques for re-configuring the execution graphs of the application or adjusting the parallelism by increasing the number of instances of certain operators. Cardellini et al.[34] proposed an elastic stream processing framework based on an ILP(Integer Linear Programming) model to reconfigure the stream processing application to make decisions on operator migration and scaling. Dhalion [48] provides a self-regulating capabilities on top of Twitter Heron that enables dynamic resource provisioning and auto-tuning for meeting throughput SLOs. However, given the heterogeneous nature of edge computing systems, these techniques may require substantial manual effort to tune the parameters to achieve a self-stabilizing status. There have also been several efforts on developing techniques to manage stream processing applications using RL methods. For example, Li et al. [82] proposed a model-free method to schedule the stream processing application based on an actor-critical RL method [98]. Ni et al. [104] developed a resource allocation mechanism based on GCN(graph Convolution network)-based RL method to group operators to different nodes. However, the above DNN-based method suffers from long training periods and low sampling efficiency and they need a large amount of data to build the model. There have been several efforts on leveraging the traditional RL method (such as Q-learning) to deal with the problem. For instance, Russo et al.[118] used the FA(Function Approximation)-based TBVI (Trajectory Based Value Iteration) to improve the sample efficiency of the traditional RL methods (such as Q-learning) to make operator scaling decisions in heterogeneous environments.

2.2 Geo-distributed Edge and Cloud Resource Management

2.2.1 Resource Management for Geo-distributed Edge Resources

Edge Computing has gained significant attention from the distributed systems community in the recent years. While the concept of edge and fog Computing is still in their early years of development, there has been several notable research efforts on this emerging topic. Bonomi et al.[22] discuss the concept of fog Computing and there has also been several other related developments in the broader area of edge and fog computing. Such efforts include the development of Cloudlets[121] proposed by Satyanarayanan et al. and the work on mobile edge computing [111] which is an extension of the effort on mobile cloud computing[78]. The primary benefit of edge computing comes from its ability to offer low latency computing resources on the fly for applications that have strict latency requirements. Edge computing is also beneficial in situations when a large number of small computing nodes need to deliver data to a cloud. Therefore applications such as IoT (Internet-of-Things), AR (Augmented reality) and VR (Virtual reality) benefit the most from modern edge computing solutions. There have been many research efforts studying the benefits of edge computing in these areas. Satyanarayanan et al. present GigaSight[122], an Internet-scale repository of crowd-sourced video content. Want et al.[148] discuss the technologies that enable IoT using edge computing. As the field is still emerging, there has been only few efforts addressing the problem of resource allocation in edge computing platforms. Aazam et al. [1] propose a model for SPs to estimate the amount of services for each MDC in the edge computing platform. Do et al.[44] propose a system for allocating fog computing resources to minimize the carbon footprint. The solution is based on a distributed algorithm that employs the proximal algorithm and alternating direction method of multipliers(ADMM).

2.2.2 Resource Management for Geo-distributed Clouds

Existing literature [3, 114, 154, 163, 117, 57, 169] have focused on optimizing the performance of cloud services in the Geo-distributed Cloud environment. This class of techniques builds Virtual Machines (VMs) for users to use computing resources across geo-distributed datacenters as a single logical virtual cluster. These techniques primarily optimize the data placement [3] [114], the latency of the services [114] [154][163] , the Quality of Service(QoS) [117] [57], the electricity cost [154] [169] across multiple datacenters. In the recent past, cloud Federation has gained significant focus from the cloud computing research community. Most of the works related to Cloud Federation primarily focus on two aspects: the first kind of research efforts focus on the architecture and the system model for enabling and deploying federated clouds [115, 27, 35, 47, 100]; the second class of existing work optimizes the performance of federated cloud through efficient job scheduling, job migration and resource allocation [91, 92, 79, 154, 24]. Rochwerger et al.[115] proposed an architecture called RESERVOIR to enable cloud providers to deal with each other in a P2P manner. Buyya et al.[27] proposed a centralized architecture named InterCloud which provides a market for the CSPs or cloud brokers

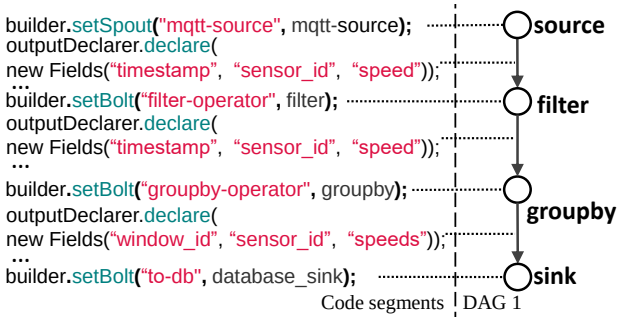


Figure 2.1: A Stream processing application example in Apache Storm

to share their resources. In [35], Carlini et al. proposed a centralized architecture for achieving federated cloud which provides single sign-on and both centralized and decentralized architecture for building the federated cloud. Moreno et al.[100] proposed a cloud-broker-based architecture to support the management of the federated cloud. In [47], Ferrer et al. proposed OPTIMIS which is a toolkit for implementing peer-to-peer Cloud Federation provisioning. McGough et al. [91, 92] analyzed and optimized the power consumption in a commercial framework for establishing Cloud Federations. In [79], Li et al. proposed a model which makes it possible for one Cloud Federation to consider both the workload and the electricity price and maximize the profit through an auction mechanism. Xu et al.[154] proposed a technique that combines the alternating direction method of multipliers (ADMM) method with the problem of how to place the cloud services with minimized electricity cost and latency to the client. In [24], Breitgand et al. proposed a method that uses policy similar to contracts to optimize the service placement problem in federated clouds.

2.2.3 Decentralized Resource Management for Geo-distributed Edge Resources

There have been several efforts on decentralized auction-based mechanisms to allocate edge resources. Zavodovski et al. [165] proposed DeCloud which uses blockchain for managing the auction using a weighted matching mechanism. However, the algorithm is not tested on a real blockchain network to validate the efficiency. Lin et al. [84] proposed a hierarchical real-time auction mechanism for allocating resources. However, the real-time operation of this auction may not be possible in the public blockchain network or may result in high transaction fees.

2.3 Preliminaries of Stream Processing

Many existing stream data processing systems such as Apache Storm [135] and Apache Flink [133] represent stream queries as a directed acyclic graph (DAG) of stateful and stateless operators (Figure 2.1).

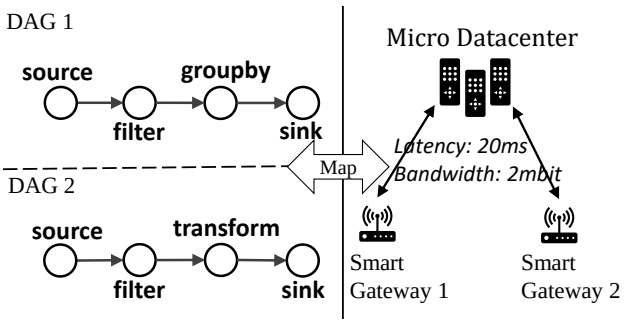


Figure 2.2: Example DAGs and cluster

In the DAG, the nodes represent the logic operators and the edges represent the streams of tuples between the operators. Data tuples flow from *source* operators to *sink* operators as shown in the DAG in Figure 2.1. The *source* operator and *sink* operator are responsible for generating the input streams and publishing the results respectively. The other operators perform a variety of computation on the streams, ranging from simple filtering to complex operations like grouping or joining streams in a time window. For example, in Figure 2.1, we see a stream query job, which aggregates the car speeds in a smart city application. The application has four operators and three streams connecting them, which first takes the car speed data generated in the sensors distributed on the roads in a *source* operator, then filters the speeds in the *filter* operator. It then groups the speed information by a time window using a *groupby* operator, and finally stores the results into a database by a *sink* operator. The above DAG is typically generated from source code written using the interfaces provided in a stream processing engine. We illustrate a JAVA code example of an Apache Storm application in Figure 2.1 to explain how the DAG is generated.

Here, the *builder* is an object that handles the DAG configuration. The *setSpout*() function indicates a *source* operator, and the *setBolt*() function defines the other kind of operators. The *outputDeclarer* is an object that handles the definition of the output stream for each operator and the *declare*() function takes a *Fields* object which defines the schema of the output stream. Thus, Apache Storm maps the stream processing application to a DAG that has four nodes and three edges as shown in Figure 2.1. Each operator sends and receives logically timestamped events (tuples) along directed edges. For example, in the output definition of the *source* operator and *filter* operator, there is a "timestamp" field that represents the generation time of the tuple (Figure 2.1).

After creating a DAG from the stream processing application, the stream processing engine schedules the DAGs on the physical nodes. As shown in Figure 2.2, the scheduling can be simplified to mapping the operators of the DAGs into each physical node controlled by the stream processing engine. Figure 2.2 shows an example edge computing environment consisting of one micro datacenter and two smart gateways connected to the micro datacenter with a link latency of 20 milliseconds and bandwidth 2 megabits per second.

In the next chapter, we present the proposed optimization techniques for low-latency stream processing applications in edge computing.

3.0 Optimizing low-latency stream processing applications in Edge Computing

In this chapter, we present Amnis, a distributed stream processing platform that optimizes the resource allocation for stream queries by carefully considering the data locality and resource constraints during physical plan generation and operator placement in an edge computing environment. Amnis includes the following novel features to facilitate a system-wide optimization of computing and networking resource usage for processing large volumes of data streams near the edge. First, *Amnis* employs a novel data locality-aware approach to optimize the resource allocation for the stream queries executing in resource-constrained edge computing environments. Second, *Amnis* schedules each operator of the queries by carefully considering the dynamically varying load conditions and resource requirement for each operation to further improve the query performance. Finally, *Amnis* considers coflow [39] dependencies which group the dependent network flows that exist in the stream processing application and prioritizes smaller coflows to complete first in order to increase the overall coflow completion rate which in turn improves the overall efficiency of the network resource usage. We evaluate Amnis by implementing a prototype on Apache Storm [135] using a real cluster test-bed on CloudLab[45]. Our evaluation performed using a range of edge computing stream processing applications shows that Amnis achieves more than 200X improvement on the end-to-end latency and as much as 10X throughput compared to the state-of-the-art distributed stream processing engine, namely, Apache Storm [135].

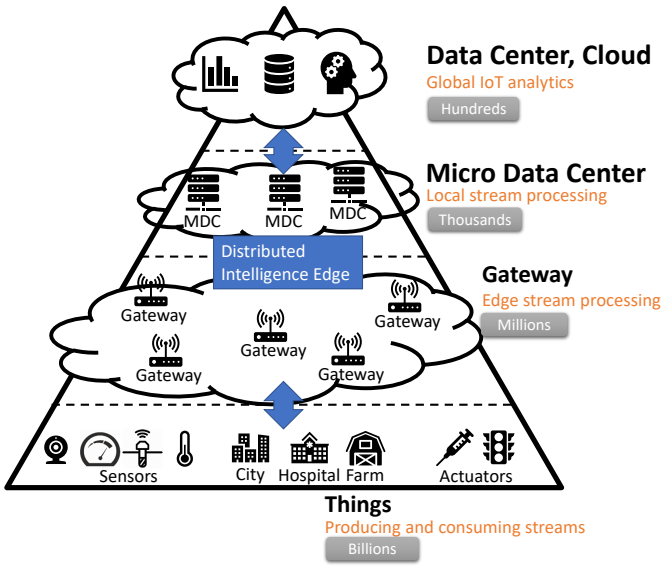


Figure 3.1: Edge/Fog Computing architecture

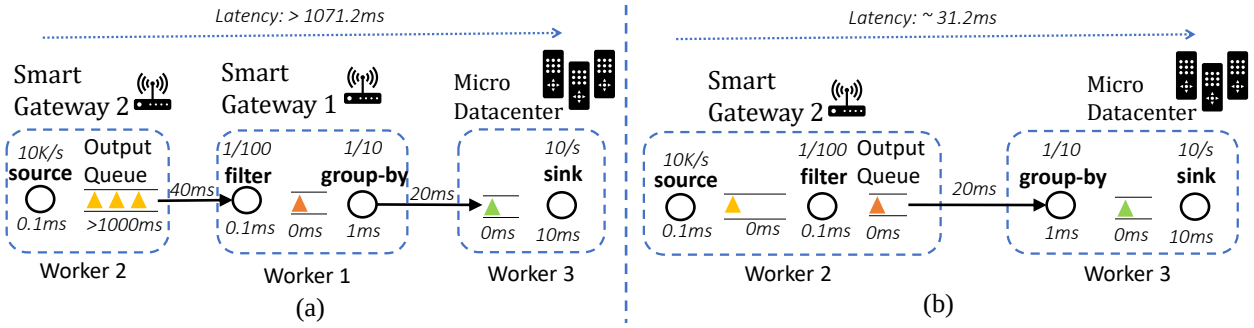


Figure 3.2: Scheduling example of DAG 1: (a) Data locality unaware scheduling causes the link being saturated, (b) Data locality aware optimization avoids the link being saturated

3.1 Background and preliminaries

As the number of IoT devices increases, large amounts of data get generated near the edge of the network in real-time. Transferring such data to the cloud may lead to longer processing times and hence, it may not be suitable for latency-sensitive IoT applications such as virtual reality (VR) and augmented reality (AR) applications and applications for the smart city such as connected vehicles and intelligent online traffic control[22].

Edge computing architecture: In a fog/edge computing architecture [22], we have different layers of heterogeneous computational resources as shown in Figure 3.1. The cloud datacenter represents the remote resources for storing the final results and it can be used for long term data storage and analysis. The micro datacenters (MDCs) and the smart gateways which are deployed near the edge of the network are used for processing the data locally and helping provide low latency computing to the IoT.

Scheduling: After creating a DAG, the stream processing engine schedules the DAGs on the physical nodes. As shown in Figure 2.2, the scheduling can be simplified to mapping the operators of the DAGs into each physical node controlled by the stream processing engine. In the scheduling phase, the stream processing engine tries to optimize the placement for each operator to achieve a good or predictable performance. However, any bottlenecks will cause performance degrading, which we will use two examples to illustrate in the remaining section with the possible directions to solve the problems.

3.1.1 Bandwidth Bottleneck

If a default round-robin scheduling algorithm is used, the scheduling may produce the output shown in Figure 3.2(a). In the example, the *sink* operator and *source* operator placements defined in the program are located at the smart gateway 2 and the micro datacenter respectively. For the remaining two operators, the

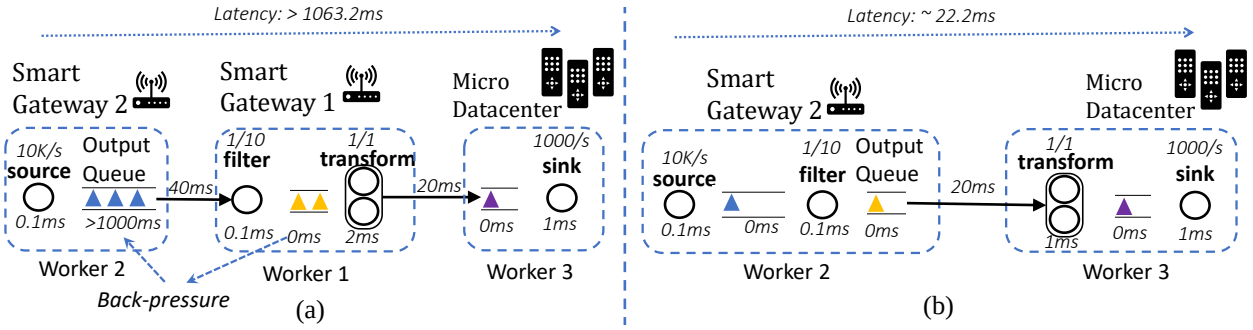


Figure 3.3: Scheduling example of DAG 2: (a) a load unaware scheduling which causes back-pressure, (b) a load aware optimization which avoids back-pressure

scheduler randomly chooses smart gateway 1 and finds that it has two free slots to place the two operators. Hence, the final operator placement is as shown in Figure 3.2(a). Here, the latency for each operation and the queuing latency between the operators are shown (e.g., the computational latency for the *sink* operator is 10ms). We can easily calculate the overall latency by adding the latency at each step from the *source* operator to the *sink* operator. When the bandwidth is sufficient to transfer all the tuples between the *source* and the *filter*, there are no bottlenecks and the tuples can be processed promptly along the DAG. It will result in a latency of around 71.2ms though it is not shown in the figure, we can calculate it by not including the 1000ms queuing latency in smart gateway 2. When the bandwidth is not sufficient to transfer all the tuples from the *source* to the *filter*, the tuples will be queued in the smart gateway 2 which will result in a much higher overall latency as shown in Figure 3.2(a). The latency will increase to more than 1000 ms due to the bottleneck (i.e., insufficient bandwidth) and will continue to increase if the transfer rate cannot catch up with the input rate.

However, when the data locality is considered in the scheduling, the *filter* operator which is both a selective operation and the immediate downstream operator of the *source* operator will be moved closer to the *source* operator in the smart gateway 2 and the group-by operator will be moved to the micro datacenter as shown in Figure 3.2(b). With the above optimization, the bandwidth becomes sufficient and hence, the overall latency is reduced significantly (around 31.2 ms).

3.1.2 Computational Bottleneck

The stream processing engines introduce a back-pressure mechanism to increase the availability of the stream processing (e.g., the *acker* in Apache Storm [135], and the *BufferPool* used as a blocking queue between the tasks in Apache Flink [133]). The back-pressure mechanism slows down the stream output rate at the *source* operator (queuing the tuples) when any of the down-stream operators cannot process as fast as

the tuples emit from the *source*. This feature helps guarantee the availability by not dropping tuples when there is a bottleneck but it will cause other issues when resources are constrained for example in the edge computing environment. For example, in Figure 3.3(a), when the *transform* operator is placed on the smart gateway 1 with the *filter* operator, the compute resource is not sufficient for the two operators and it causes back-pressure as the *transform* operator cannot process the tuples as fast as they come in (1000 tuples/second input rate but 500 tuples/second consuming rate). This will then cause the *source* operator to queue the tuples that it emits, which may further continuously increase the overall latency (e.g., more than 1000 ms latency in Figure 3.3(a) or may even cause out-of-memory (OOM) failure in smart gateway 2) due to the overall throughput is lower than the input rate, which is bounded by the bottleneck operator. However, if we adopt a load aware optimization as shown in Figure 3.3(b), the *filter* operator is moved to the smart gateway 2 and the *transform* operator is scheduled to the micro datacenter. Here the computational resources are sufficient for the two operators and the *transform* operator can work twice as fast as in the previous scenario and the *transform* operation can be done in 1 ms. Hence, there is no back-pressure (bottleneck) and the latency is much lower (around 22.2 ms as shown in Figure 3.3(b)).

The design of the proposed *edge-oriented* stream processing engine, Amnis is motivated by the above-mentioned observations that any bottlenecks that exist in the stream processing application may cause high latency due to different reasons such as lack of data locality awareness and lack of load awareness. Amnis is designed for resource-constrained edge computing environments to achieve low latency and high throughput while simultaneously eliminating as many bottlenecks as possible. The data locality optimization in Amnis maximizes the data locality by placing the selective operators near the *source*. The load optimization in Amnis avoids back-pressure to improve the overall latency by considering the computational load of each operator and the resource capacity of each node to improve the overall performance. Besides the above optimizations on scheduling, Amnis also considers the coflows [39] which captures the dependency between the flows waiting to be scheduled in the network, which may also cause bottleneck because of the flow dependency in the streams. Amnis shapes the stream rates to optimize the bandwidth allocation in order to improve the coflow completion rate to enhance the overall performance of the stream processing applications.

In the next section, we introduce the features of the Amnis system design.

3.2 Amnis: System Design

In this section, we present the stream processing optimization problem and illustrate how Amnis extends the current distributed stream processing solutions (e.g. Apache Storm) to tackle the challenges in edge computing environments. As discussed in Section 3.1, our objective is to minimize the end-to-end latency for the stream processing applications in edge computing environments by minimizing the impact of the bottlenecks. To achieve this goal, we consider various conditions that would create bottlenecks in an edge computing environment.

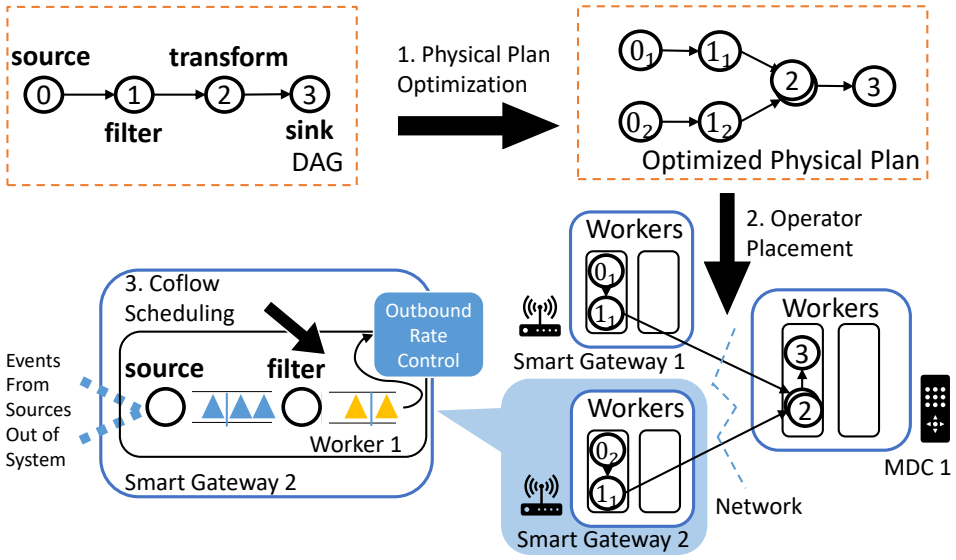


Figure 3.4: Amnis Optimization

3.2.1 Stream Processing Model

We assume that there are K stream processing applications deployed in the system. Each application k is translated from the source code to a DAG (Figure 2.1), denoted as $G^k_{dsp}(V^k_{dsp}, E^k_{dsp})$ (notations presented in Table 3.1). Here, the vertices V^k_{dsp} represent the operators and the edges E^k_{dsp} represent the streams. As shown in Figure 3.4, we assume that the physical plan is generated from the DAG, G^k_{dsp}, and it is also represented as a graph, $G^k_{phy}(V^k_{phy}, E^k_{phy})$. After the physical plan is generated from the original DAG, the physical plan is scheduled on to the cluster in the operator placement step. The computational and network resources in the cluster can be represented as a graph $G_{res}(V_{res}, E_{res})$. The vertices here are the computation nodes e.g., smart gateways and micro datacenters, and the edges represent the network links between the physical nodes. The operator placement plan can be seen as a map $X^k = \{x^v_i | i \in V^k_{phy}, v \in V_{res}\}$. If we represent the operator as $i \in V^k_{phy}$, and the physical node in the cluster as $v \in V^k_{res}$, we can use $x^v_i = 1$ to indicate that the operator i is placed on node v, vice versa.

As our objective is to improve the end-to-end latency of the stream processing application, it is important to understand how the end-to-end latency is composed of in the stream processing context. Based on the model proposed in [32], the end-to-end latency of a particular application can be modeled using the longest path (cumulative sum of the latency from the source to the sink). We denote the paths between all the pairs of sources and sinks as P^k. Any $\rho^k \in P^k$ indicates a path from a source to a sink. The end-to-end latency of an application k can be represented as:

$$l^k(G^k_{phy}, G_{res}, X^k) = \max_{\rho^k \in P^k} l^k_\rho(G^k_{phy}, G_{res}, X^k) \tag{1}$$

Table 3.1: Notations

$G^k_{dsp}(V^k_{dsp}, E^k_{dsp})$	DAG of application$k$	i, j	an operator
$G^k_{phy}(V^k_{phy}, E^k_{phy})$	the physical plan of application k	v, u	a physical node
$G_{res}(V_{res}, E_{res})$	cluster topology	X^k	operator placement plan
x^v_i	operator placement indicator	P^k	pairs of sources and sinks
ρ^k	path from a source to a sink	l^k	end-to-end latency
l^k_ρ	latency on path ρ^k	λ_i	input rate of operator i
w_i	mean output tuple size of operator i	θ_i	selectivity of operator i
$e_{(i,j)}$	data volume of stream (i, j)	$Y_{i,j}$	co-located Boolean indicator
$c_i(\lambda_i)$	resource requirement of operator i	O_v	resource capacity of node v
b	a a windowed aggregator	D_b	coflow of the aggregator b
P_b	upstream operators of the aggregator b	R_b	rate controlling plan of the aggregator b
ψ_i	input/output ratio of operator i	a	a source operator
S^k_a	data sources of a source operator a	k	an application

$$l^k_\rho(G^k_{phy}, G_{res}, X^k) = \sum_{v\in\rho} l_{(i,v)}(G^k_{phy}, G_{res}, X^k) + \sum_{(v,u)\in\rho} l_{(v,u)}(G^k_{phy}, G_{res}, X^k) \quad (2)$$

where $l_{(i,v)}$ is the computational latency of operator i when it is placed on node v, and $l_{v,u}$ is the network latency (including the network transmission latency and the queuing latency) between node v and u when there is a stream in the path ρ passes between them. For all the K applications, we assume they have the same priorities and consider the optimization simultaneously.

Based on the analysis of the end-to-end latency above, we can see that there are two major components, namely the computational latency caused by processing the tuples in the operator and the network latency caused by transmitting the tuples between two connected operators when they are scheduled to different nodes (when two operators are scheduled to the same node, we assume that the bandwidth between them is large and it does not become a bottleneck). The computational latency is often tackled using load balancing methods such as migrating the operator from a overloaded node to another idle node. The network latency is much harder to improve directly in the edge computing environment. Optimizing data locality is a promising direction to decrease the network utilization and the network latency. In addition, coflow [39] optimization provides additional opportunities for optimizing the efficiency of the network usage by considering the dependencies between the network flows.

Combining the examples illustrated in Section 3.1 and the above end-to-end latency model, we can see that either the computational bottleneck or the network bottleneck can cause a path to become the longest path, which influences the end-to-end latency and performance of an application. In Amnis, we propose a three-phase optimization to deal with the two types of bottlenecks as shown in Figure 3.4:

Physical plan optimization: extends the user-defined DAG of the logic plan into an optimized physical plan, which aims to optimize the data locality to decrease the potential network usage.

Operator placement optimization: decides the placement of the operators in the output of the physical plan optimization on to the physical nodes to further optimize the data locality and balance the load,

Coflow scheduling optimization: takes the result of operator placements of multiple applications to determine the output rate for each specific operator in order to reduce the overall impact of potential network bottlenecks.

The overall correctness of Amnis is ensured if the individual objective in each individual phase is achieved. In the remaining part of this section, we comprehensively explain the above-mentioned phases.

3.2.2 Data Locality Aware Physical Plan Generation and Operator Placement

As discussed above, the bottlenecks will influence the end-to-end latency and performance, and we use data locality as a key criterion in generating the physical plan and placement of the operators.

Input: this phase takes as input the program of the stream processing application which is translated to a DAG, G^k_{dsp}.

Output: the data locality optimized physical plan, G^k_{phy}, and the data locality optimized operator placement, X^k.

Objective: We use data locality to guide the generation of the physical plan and the first part of the operator placement. The objective of the data locality is to minimize the data volume transferred between different nodes, which is widely studied in MapReduce frameworks [43]. To optimize the end-to-end latency by considering the data locality, we borrow the terminology from the distributed batch processing related works. We denote the average data volume of a stream $(i, j) \in E^k_{phy}$ (from operator i to operator j) as $e_{(i,j)}$ and we model the input rate of operator i as a Poisson distribution with the arrival rate as λ_i. The average tuple size has a distribution with a mean value w_i, and the selectivity of i is θ_i and therefore, $e(i, j) = \lambda_i w_i \theta_i$. Next, we define the network cost as follows when the operator placement is determined:

$$NCost^k(G^k_{phy}, G_{res}, X^k) = \sum_{(i,j)\in E^k_{phy}} e_{(i,j)}(1 - Y_{i,j}) \tag{3}$$

Here, $Y_{i,j}$ is a co-located Boolean indicator, which is 1 when the downstream operator j of i is placed on the same node of i, $x^v_i = x^v_j = 1$. Due to the resource limitation of the physical nodes, we cannot place all the operators in a single node to reduce the network cost. Therefore, we design an algorithm to group the operators to minimize the data volume transmitted between the groups. It is discussed in detail in Section 3.3.1. However, there is a gap between optimizing the data locality by optimizing the operator placement and the input DAG, which is the ability of moving the data source near the place where the stream is generated. In a stream processing application, the *source* operator often subscribes multiple streams from different Message Queuing Telemetry Transport (MQTT) [62] (a lightweight publish/subscribe message transport service) services that provide similar stream sources and cannot be automatically split and placed near the stream sources. It is critical that the optimization technique is capable of increasing data locality by placing the *source* operator near the stream source and keeping the code simple for the users (e.g., define one logic source operator to fetch the streams from multiple sources by just mentioning the service

hosts and ports in one place). Amnis tackles this in the physical plan generation using an additional step. When there is a *source* operator fetching multiple streams that hold the same schema, Amnis automatically divides a logic *source* operator into multiple physical *source* operators, which can be mapped to the nodes that are co-located or near the stream sources as shown in Figure 3.4. The above step provides an additional benefit that the downstream *stateless selective* operators can be split and moved to co-locate with the *source* operator to further improve the data locality. We discuss the details of these algorithms in Section 3.3.1.

Parallelism Configuration: as in Figure 3.4, the parallelism of each operator is decided also in physical plan generation. The parallelism of the *transform* operator is set to be two in the example, which can be done using many existing optimization techniques such as [34] which decide the number of replicas of each operator by solving an ILP (Integer Linear Programming) problem. In our work, we assume that the parallelism of each operator is already optimized and defined as a *fixed* number in the user's program.

3.2.3 Load Aware Operator placement

By applying the data locality to both the physical plan and the operator placement plan, we reduce the network usage of the application. However, the optimization does not consider the resource requirements of the operators which may lead to overload the nodes. In this subsection, we discuss the load aware optimization on the operator placement to deal with this issue.

Input: the physical plan, G^k_{phy} and the operator placement plan X^k, from the output of the data locality optimization and the topology of the cluster, G_{res}.

Output: the load aware optimized operator placement, X^k.

Objective: The data locality is already considered in the above subsection. In this subsection, we focus on load balancing, which takes the output physical plan and operator placement from the data locality optimization to re-balance the overloaded nodes by considering the resource requirement of each operator and resource capacity of each node. As discussed in Section 3.1, the computational bottleneck (mostly caused by overloading) may influence the end-to-end latency performance. To mitigate that, we need to consider the load of each node to avoid overloading a node. We denote the resource capacity of a node v as O_v. The minimal resource requirement of an operator i is $c_i(\lambda_i)$, which means that the operator needs at least $c_i(\lambda_i)$ resource to process the tuples arriving based on a Poisson distribution with arrival rate λ_i. Both the resource capacity of the node and the resource requirement of the operator are simplified from multiple dimensions (CPU, Memory, etc.) to one dimension by identifying the dominant resource [52]. As most of the stream processing operations (deserializing, filtering, mapping, etc.) are computationally intensive, we assume that the default dominant resource is CPU and only for the operation which needs to maintain large states (windowed aggregation, join, etc.), we set the dominant resource as Memory. We define a resource constraint to ensure that the operator placement does not overload node v:

$$\sum_{i \in V^k_{phy}} c_i(\lambda_i) x^v_i \leq O_v \tag{4}$$

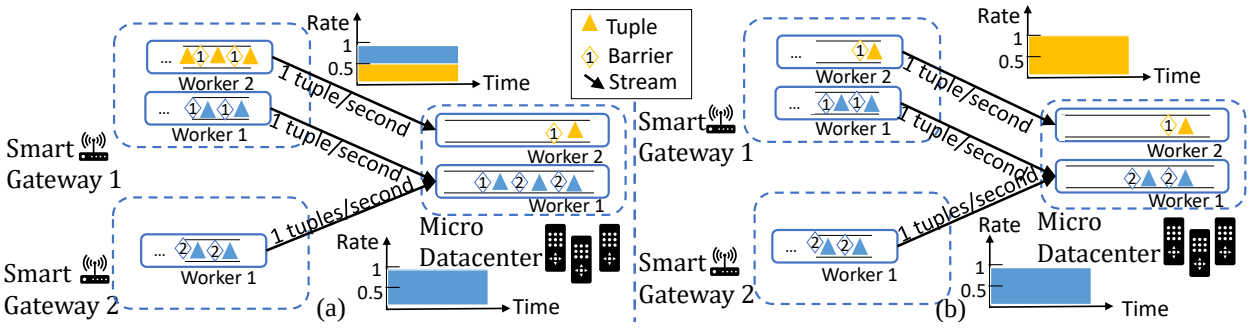


Figure 3.5: Coflow optimization example: (a) a coflow unaware fair sharing flow scheduling, (b) a coflow aware flow scheduling

The basic idea of load aware operator placement is to take the output from the data locality aware optimization and further detect the overloaded node by comparing the resource requirement of the operator and the resource capacity of the node. Then, the operator placement plan is modified by moving some of the operators from the overloaded node to the neighboring nodes with less utilization. We present the details of this approach in Section 3.3.2.

3.2.4 Coflow optimization

After the operator placement is decided, *Amnis* performs another optimization to control the data flows between the operators to shape the stream rates to improve the efficiency of using the network, which also reduces the network latency.

Input: the physical plan of all the K applications submitted to the system, $\mathbf{G_{phy}} = \{G^1_{phy}, ..., G^k_{phy}, ..., G^K_{phy}\}$, the topology of the cluster, G_{res}, and the operator placement plan for each application k, $\mathbf{X} = \{X^1, ..., X^k, ..., X^K\}$, from the output of the operator placement phase. In addition, the Coflow information is analyzed for each application and gathered for each aggregator, $\mathbf{D} = \{D_b | \forall b \text{ are aggregators}\}$

Output: the rate controlling plan, $\mathbf{R} = \{R_b | \forall b \text{ are aggregators}\}$.

Coflow: Before discussing the objective of the coflow scheduling, we first provide the definition of the coflow in the stream processing context. For each application k, if there is a windowed aggregator $b \in V^k_{phy}$, and if it needs to fetch the input from the upstream operators crossing the network, we use $P_b = \{1, 2, ..., p, ...\}$ to define the set of the upstream operators of the aggregator b. We also use a set $D_b = \{e_{(1,b)}, e_{(2,b)}, ..., e_{(p,b)}, ...\}$ to indicate the coflow of the aggregator, which indicates that the completion of all the flows in D_b refers to the completion of the coflow. As mentioned in the previous section, $e_{p,b}$ indicates the average stream data volume between the upstream operator p and operator b. If we want the upstream to arrive on time, we need to at least allocate $e_{(p,b)} \in D_b$ to all the corresponding streams between P_b and b to transmit the flows

so that it does not cause back-pressure. The reason why the coflow scheduling is needed is illustrated in Figure 3.5. Two applications are deployed in the cluster. We assume that they have the same priorities and latency requirements, and the computational of the operations do not lead to a bottleneck. In the example, the bandwidth between the micro datacenter and the two smart gateways is limited to one tuple/second. Application 1 is deployed on all the three nodes and Application 2 uses only smart gateway 1 and the micro datacenter. The tuple rate of application 1 is one tuple/second between the smart gateway 1 and the micro datacenter and one tuple per second between the smart gateway 2 and the micro datacenter. The tuple rate of application 2 is one tuple/second between the smart gateway 1 and the micro datacenter. As shown in Figure 3.5(a), if the bandwidth is allocated using a fair sharing (the default policy of TCP protocol) mechanism, both application 1 and 2 are bottle-necked at the link between the smart gateway 1 and the micro datacenter. Either of them builds the queue in both smart gateway 1 and the micro datacenter. This is because, in the micro data center, there is an aggregator for both applications requiring to wait for all the barriers to arrive in order to ensure the all tuples in the time window arrived to trigger the windowed function. Barriers are synchronization signals [31] used in Storm and Flink. It is also called *watermark* [7]. In the example shown in Figure 3.5(a), application 1 needs to wait for the barriers from smart gateway 1 and smart gateway 2 (noted as blue diamond 1 and 2) in order to trigger the processing function for a particular time window. However, the bandwidth bottleneck between the smart gateway 1 and micro datacenter makes the barrier from smart gateway 1 to be always later than the barrier from smart gateway 2, which builds the queue on both the smart gateway 1 (because of bandwidth bottleneck) and the micro datacenter (because of barrier synchronization).

Objective: In the coflow scheduling phase, Amnis targets the completion rate of the coflows to satisfy the network requirement for the dependent flows to avoid back-pressure. Given the rate controlling plan, $\mathbf{R}$, if the coflow D_b is satisfied, it means that the corresponding bandwidth allocation, $r_p >= e_{(p,b)}, \forall e_{(p,b)} \in D_b$. If the coflow D_b is satisfied, it is captured using $Z_b = 1$, otherwise, $Z_b = 0$. We can define the objective of maximizing the coflow completion rate as:

$$\max \sum_{b} Z_b \tag{5}$$

In our approach, we prioritize the small coflows to satisfy their bandwidth requirements first to maximize the coflow completion rate defined above. The detailed algorithm is presented in Section 3.3.3.

In the next section, we present the details of the techniques used in Amnis.

3.3 Amnis: Techniques

The proposed techniques in the Amnis system are organized along with three aspects: (i) Data-locality-aware optimization for physical plan generation and operator placement, (ii) Load-aware optimization for operator placement, and (iii) Coflow optimization for data rate control.

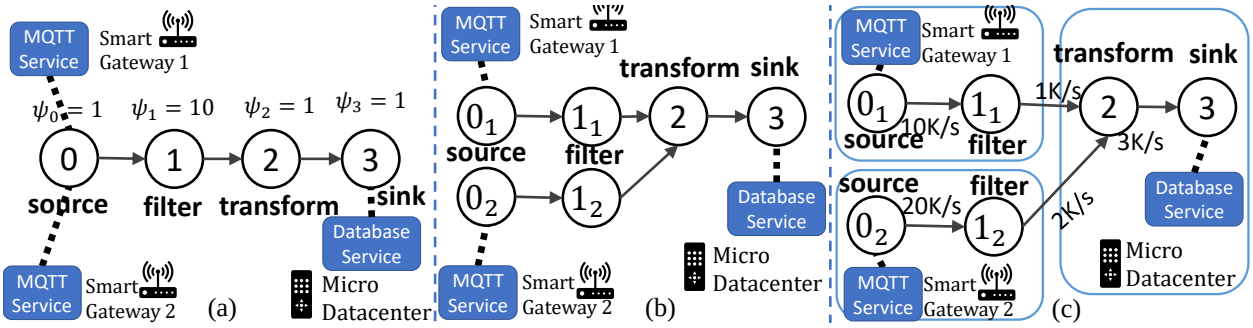


Figure 3.6: A data locality aware physical plan optimization: (a) the original DAG and the ψ_i of each operator, (b) an optimized physical plan calculated by Algorithm 1, (c) the operator placement plan calculated by Algorithm 2 and 3.

3.3.1 Data locality optimization

As discussed in Section 3.2.2, we define the data locality objective as optimizing the network cost $NCost(G^k_{dsp}, G_{res}, X^k)$. In this section, we present the detailed algorithms which aim to decrease the network cost.

On one hand, to improve data locality, if we move the *source* operator and the selective operators (e.g., a *filter* operator) to the same node where the stream source comes from, the data locality can be improved by reducing the bandwidth usage across the network. However, the *source* operator often fetches streams from multiple sources as shown in Figure 3.6. Here, if we want to move the *source* operator near the stream source, we need to first split the *source* operator to multiple operators, and each of them fetches only one of the stream sources. It is worth noting that if the downstream operator can be moved to co-locate with the *source* operator, it also needs to be split into multiple independent operators and each only connects to one of the *source* operators. Here we only consider to split *stateless* operators as *stateful* operators need the shuffling phase to ensure correctness. We discuss it in detail in Section 3.5. On the other hand, if we can move the operator which increases the output size compared to the input (e.g., a *join* operator) to the same node where the downstream operator is placed, it will further enhance the data locality at the downstream side.

In *Amnis*, we classify each operator i based on its input/output ratio ψ_i to optimize both the physical plan and the operator placement. We assume the input/output ratio can be obtained by profiling. It is defined as:

$$\psi_i = \frac{\sum_p e_{p,i}}{\sum_j e_{i,j}} \tag{6}$$

where p is an upstream operator of i and j is a downstream operator of i. Operators are categorized as one of the two types: (i) high input/output ratio ($\psi_i > 1$) operators, and (ii) low input/output ratio

($\psi_i \leq 1$) operators. For the operator with $\psi > 1$ (e.g., a *filter* operator that only allows the tuples meeting the condition to pass), we first split the upstream *source* operator to bind with the stream source and then split the operator with $\psi > 1$ to move the operator near the *source* operator to improve the data locality. For the operator with $\psi_i \leq 1$ (e.g., a *join* operator that joins a stream with a database table to produce the output), we can move it near the downstream operator (e.g., a *sink* operator) to improve the data locality.

In the remaining part of this subsection, we will describe the details of the algorithms in two steps namely (i) data locality aware physical plan generation that generates the optimized physical plan by improving the data locality for the two kinds of operators, (ii) data locality aware operator placement plan generation that generates the placement of the operators to the physical nodes based on its input/output ratio.

Algorithm 1: Data locality aware physical plan optimization for operators $\psi_i > 1$

Input : DAG: $G^k_{dag}(V^k_{dag}, E^k_{dag})$;
Output: Optimized Physical Plan: $G^k_{phy}(V^k_{phy}, E^k_{phy})$;
1 Initial: $G^k_{phy} = G^k_{dag}$;
2 A Queue $Q = \emptyset$;
3 **for** *each source operator* a **do**
4 Split each *source* operator a to its number of sources $|S^k_a|$ as $U_a = \{a_1, a_2, ..., a_s, ..., a_{|S^k_a|}\}$;
5 Replace a as U_a in G^k_{phy};
6 For all the operators i such that i the neighbor of a: add a tuple (i, U_a) to Q;
7 **while** Q *is not empty* **do**
8 Pop a tuple (i, U) from Q;
9 **if** *operator* i *is a stateless operator and* $\psi_i > 1$ *and* i *is not a sink operator* **then**
10 Split i to $|U|$ operators noted as $U_i = \{i_1, i_2, ..., i_s, ..., i_{|S^k_a|}\}$;
11 Set the upstream operator of each i_s to $i'_s \in U$;
12 Replace i as U_i in G^k_{phy};
13 For all the downstream operators i' of operator i: add a tuple (i', U_i) to Q;

3.3.1.1 Data locality aware physical plan generation

As discussed earlier, to optimize the data locality by modifying the physical plan, we need to place the *source* operator and its downstream operators in the optimized physical plan near the stream sources. Based on this intuition, we propose Algorithm 1. The algorithm starts from the *source* operators, and for each *source* operator a, it first splits them into the operators set U_a that each operator $a_s \in U_a$ binds with a stream source $s \in S^k_a$, where S^k_a is the data source set of application k. Then, the downstream operators are traversed by a breadth first search with the help of a queue Q. In the queue Q, each element contains the operator number i and its upstream operator set U. For each downstream operator i, if $\psi_i > 1$ and the operator is a *stateless* operator which can be split without shuffling the intermediate results (e.g., a *filter* operator as shown in Figure 3.6), then operator i is split into an operator set U_i and each operator $i_s \in U_i$ binds with one of the *source* operator $a_s \in U_a$. The time complexity of the algorithm is determined by Loop on line 7, which traverses the DAG by a breadth first method and the initial set is determined by the number of *source* operators. It is $O(|V^k_{dag}|^2)$ and in the worst case, half of the operators are *source* operators.

An example is shown in Figure 3.6(a) and (b) with four operators in the input DAG. The algorithm starts from the *source* operator, splits it into two and then traverses the DAG by a breadth first search. In the next round, the *filter* operator meets the condition ($\psi_i > 1$) and is split into two as the upstream *source* operator. The loop ends when it traverses to the *transform* operator which is $\psi_i = 1$. Finally, the physical plan $G^k_{phy}(V^k_{phy}, E^k_{phy})$ becomes the output.

Algorithm 2: Data locality aware operator placement for operators $\psi_i > 1$

Input : Optimized Physical Plan: $G^k_{phy}(V^k_{phy}, E^k_{phy})$;
Cluster: $G_{res}(V_{res}, E_{res})$;
Output: Operators Placement: $X^k = \{x^v_i | i \in V^k_{phy}, v \in V_{res}\}$;

```
1  Initiate: x_i^v = 0, ∀i, v;
2  A temporary queue Q;
3  for each source operator a do
4      For a single source operator a_s, we place it co-located with the source s, which is noted as
         x_{a_s}^v = 1, v = Location(s);
5      For all the operators i such that i is the neighbor of a: add i to Q;
6  while Q is not empty do
7      Pop an operator i from Q;
8      if ψ_i > 1 then
9          Place each operator i_s co-located with a_s that x_{i_s}^v = 1, v = Location(s);
10         For all the downstream operator i' of operator i: add i' to Q;
```

3.3.1.2 Data locality aware operator placement plan generation

To improve the data locality by generating the operator placement plan, the *source* operator and its downstream operators with $\psi > 1$ should be moved near the stream sources. The operator that increases the data size ($\psi \leq 1$) should be moved near the downstream operators. Based on the above intuition, we divide the heuristics into two parts namely (i) for operators with $\psi > 1$, and (ii) for operators with $\psi \leq 1$.

As shown in Algorithm 2, for the operators with $\psi_i > 1$, we bind the *source* operators to place each split *source* operator to the node where the source service is placed. Then, the algorithm traverses the downstream operators by a breadth first search with the help of a queue Q and tries to bind the operators with the source by identifying if $\psi > 1$ and if it can benefit by moving close to the source. The time complexity of the algorithm is determined by the second loop, which traverses the DAG by a breadth first method and the initial set is determined by the number of *source* operators. We note that it is $O(|V^k_{dag}|^2)$ and in the worst case, half of operators are *source* operators.

For the operators with $\psi \leq 1$, placing them near the stream source may not improve the data locality as the output stream rate does not decrease after the operation. Therefore, for these type of operators, *Amnis* places them near the downstream operator (often a *sink* operator) to improve the data locality. As shown in Algorithm 3, the operators are placed at the same node with the output service (e.g., a database) to improve the downstream-side data locality using a breadth first search from the *sink* operator to the upstream operators. If the operator has $\psi_i \leq 1$, then the algorithm places the operator in the same node as

the *sink* operator. The time complexity of the algorithm is determined by the number of operators, which is $O(|V^k_{dag}|)$.

Algorithm 3: Data locality aware operator placement for operators $\psi_i \leq 1$

Input : DAG: $G^k_{phy}(V^k_{phy}, E^k_{phy})$;
Cluster: $G_{res}(V_{res}, E_{res})$;
Output: Operators Placement: $X^k = \{x^v_i | i \in V^k_{phy}, v \in V_{res}\}$;

```
Initiate: a temporary queue Q;
Find the physical node v' where the sink service of G^k_phy places;
Add all the upstream operators i of the sink operator into Q;
while Q is not empty do
    Pop an operator i from Q;
    if ψ_i ≤ 1 then
        Set x_i^{v'} = 1;
        For all the upstream operator i' of operator i: add i' to Q;
```

We present an example of the data locality aware physical plan and operator placement optimization in Figure 3.6. The figure illustrates the input DAG and the result of Algorithm 2 and 3. We can see in the DAG that there are four operators: *source*, *filter*, *transform*, and *sink*. The stream rates and the input/output ratios are noted on the edge and the node respectively. The physical plan is first generated by Algorithm 1 as shown in Figure 3.6(b). Then the operator placement plan is generated by taking the physical plan and the cluster as input. It first goes through Algorithm 2 from the *source* operator, which co-locates the *source* operators with its binding stream source (the *source* operator 0_1 is placed on smart gateway 1 and the *source* operator 0_2 is placed on smart gateway 2). Then the algorithm identifies the selective downstream operators (the *filter* operator with $\psi_i > 1$) to be placed on the same node as the corresponding *source* operator as shown in Figure 3.6(c). After that, Algorithm 3 takes the output of the above steps as the input. The algorithm starts from the *sink* operator and places it on the same node as the *sink* output service (e.g., a database service). Then, it performs a breadth first search of the upstream operators. The *transform* operator is the next one and its input/output ratio is less than or equal to one and therefore, the operator is placed on the same node where the *sink* operator is placed. After that, the algorithm outputs the operator placement plan $X^k = \{x^v_i | i \in V^k_{phy}, v \in V_{res}\}$.

It is worth noting that after the data locality-aware physical plan generation and operator placement, there may be some operators whose placement decisions are not yet made even if the user has already optimized the stream processing application through some logic plan optimizations (e.g., predicate-pushdown [16] that push the operators with high input/output ratio towards the *source* operator). Therefore, after the data locality aware optimization, we use a network-aware operator placement [112] to decide the placement for the other operators that are not marked in this step by providing them as the input of the load-aware operator placement optimization in the next subsection.

3.3.2 Load aware operator placement optimization

As discussed in Section 3.2.3, the load aware operator placement takes the output from the data locality

optimization proposed in the above section to further make sure it satisfies the resource constraint.

We assume there is a preliminary operator placement X^k already calculated through the above steps namely: (i) data locality aware optimization on the physical plan, (Algorithm 1), and (ii) data locality aware optimization on the operator placement plan (Algorithm 2 and 3). As shown in Algorithm 4, it first estimates the input rate λ_i for each operator. The input rate of each operator is calculated by a breadth first search with the help of a queue Q from each *source* operator a to the *sink* operator in the line 3∼10 of Algorithm 4. The output rate of each operator is calculated as the product of the input rate λ_i and the selectivity θ_i. Then, as shown in the line 11∼17 of Algorithm 4, the total resource requirement of handling the operations on each node is estimated as C_v and compared with the resource capacity O_v of the node v. If the resource is not sufficient, we offload an operator which has the lowest impact of the data locality (the operator has the minimal input/output ratio) to a neighbor node v' which has the lowest resource congestion (the resource requirement/capacity ratio).

Algorithm 4: Algorithm for optimizing the operator placement by load awareness

Input : Optimized Physical Plan: $G^k_{phy}(V^k_{phy}, E^k_{phy})$;
Cluster: $G_{res}(V_{res}, E_{res})$;
Initial Operators Placement: $X^k = \{x^v_i | i \in V^k_{phy}, v \in V_{res}\}$;
Estimated source rate: $\lambda_s, \forall s \in S^k_a$ and $\forall a$
Output: Optimized Operator Placement: $X^{k\prime} = \{x^v_i | i \in V^k_{phy}, v \in V_{res}\}$;

```
1  Initial the input rate λ_i = 0,  ∀i ∈ V^k_phy;
2  A temporary queue Q = ∅;
3  for each source operator a do
4      Initial the input rate of the source operator to λ_a = Σ_s^{S^k_a} λ_s;
5      i = a;
6      while i is not sink operator do
7          for each downstream operator i' of i do
8              λ_{i'} = λ_{i'} + λ_i θ_i;
9              Add i' to Q;
10         Pop the first element of Q as i;
11 for each node v do
12     Calculate the total resource requirement: C_v = Σ c_i(λ_i), ∀i ∈ V^k_phy that x^v_i = 1;
13     while O_v < C_v do
14         Select an operator i = arg min_i ψ_i that i is not source or sink;
15         Select the neighbor v' of v that v' = arg min_v C_v/O_v;
16         Offload i to v' that x^v_i = 0 and x^{v'}_i = 1;
17         Update C_v and C_{v'};
```

The neighborhood can be identified when the cluster starts by clustering in the latency space [112] or other methods such as a predefined multi-tier network architecture as shown in Figure 3.1 (each node is assigned to a tier and contains a parent and multiple children in the network). The placement plan is changed by modifying $x^v_i = 0$ and $x^{v'}_i = 1$. The above step is repeated until the resource is sufficient on node v. The time complexity of the algorithm is determined by the two iterations. For the first iteration which is similar to the previous algorithms, the time complexity is $O(|V^k_{phy}|^2)$. For the second iteration, the worst-case time complexity is $O(|V_{res}|^2 \log |V_{res}|)$ when all the nodes are in the neighborhood and need to offload operators. Hence, the overall time complexity becomes $O(\max\{|V^k_{phy}|^2, |V_{res}|^2 \log |V_{res}|\})$.

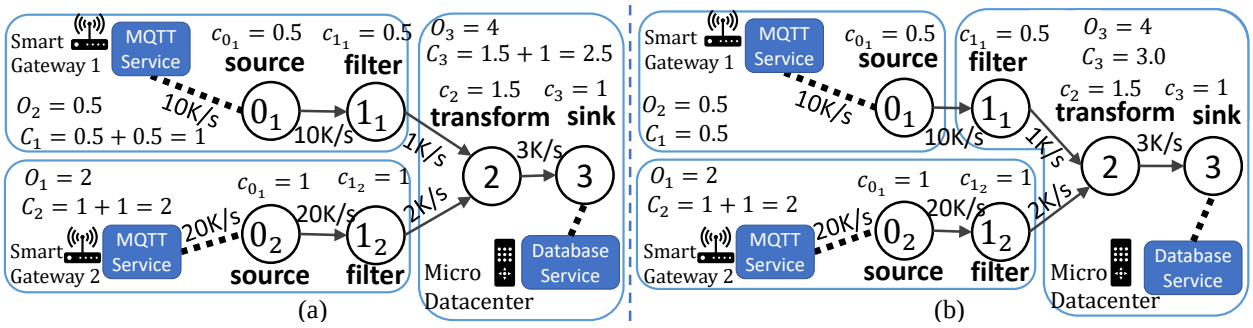


Figure 3.7: A load aware operator placement optimization example: (a) the data locality operator placement result, (b) the load aware optimization result based on (a)

In Figure 3.7, we present an example illustrating the steps in Algorithm 4. In figure (a), the physical plan and operator placement plan of the data locality optimization are given. Based on the optimized plan, we calculate the input rate for each operator starting from the *source* operator (e.g for the *sink* operator 0_1), the input rate is initialized as ten thousand per second, and for *source* operator 0_2, the input rate is twenty thousand per second. Then, the input rate is calculated through the graph one by one. Each output rate is calculated as the product of the input rate and the selectivity. For example, the output ratio of *filter* operator 1_1 is calculated by multiplying its input rate, ten thousand per second and the selectivity, one tenth. The result is one thousand per second. With the input rate information, we then estimate the resource requirement based on it (e.g., the resource requirement of the *transform* operator is 1.5 which is based on the estimated input rate, three thousand per second, and the non-decreasing function $c_i(\lambda_i)$, which can be estimated by a benchmark method). Then, the resource requirement for each node is calculated and compared with the resource capacity. We can see that for the smart gateway 2, the resource is insufficient and hence, we try to offload some operators. After applying the method described in the algorithm, we choose the *filter* operator 1_1 and move it to the neighbor node in the micro datacenter which has the lowest congestion ratio (2.5/4). The final result is shown in Figure 3.7(b).

It is worth noting that multiple applications can be optimized by Algorithm 4 simultaneously with the physical plans and the initial operator placement plans calculated by the data locality aware algorithms as the input. Then, for each application, the input rate is estimated by going through the line 3∼10. After that, for each node, the offloading is calculated by the lines 11∼17, but all of the operators (which may belong to multiple applications) are considered in the steps represented in line 12 and 14.

3.3.3 Coflow optimization

As discussed in Section 3.2.4, we define the objective of the coflow optimization as maximizing the coflow

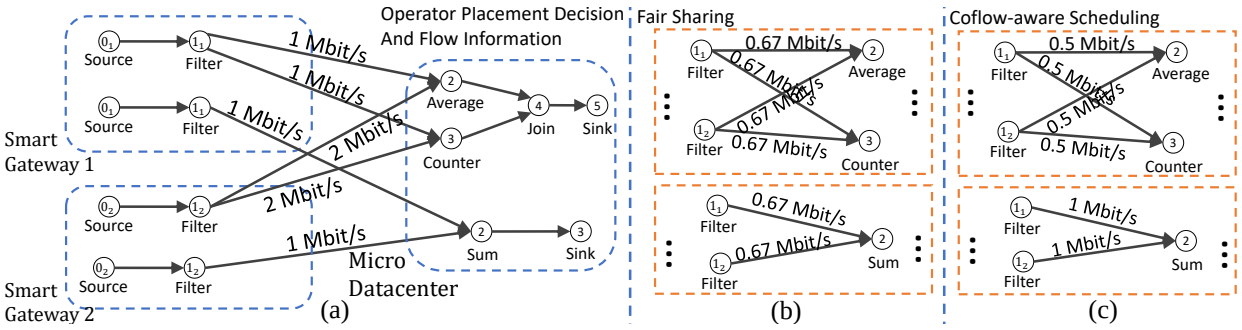


Figure 3.8: A coflow optimization: (a) the operator placement decision and flow information, (b) a flow scheduling decision by a fair sharing mechanism, (c) the flow scheduling decision of our coflow aware optimization

completion rate across all the applications deployed in the system. So in this section, we will propose the detailed algorithm whose key idea is to prioritize the smaller coflows to make sure their bandwidth requirements are satisfied and then the larger ones.

As shown in Algorithm 5, the proposed heuristic sorts the coflows by its size (the sum of the flow rates in the coflow) and allocates the bandwidth to suit the coflow requirement one by one from the smallest coflow. If any of the links cannot afford the allocation, we revoke the allocation, mark the current coflow as unsatisfied, and continue to the next coflow. After all of the coflows are traversed, we allocate the bandwidth to the remaining unsatisfied coflows by a fair sharing mechanism to equally allocate the remaining bandwidth of each link to the flows bypassing it.

Algorithm 5: Algorithm for optimizing coflow scheduling with coflow awareness

Input : Optimized Physical Plans: $G_{phy} = \{G^1_{phy}, ..., G^k_{phy}, ..., G^K_{phy}\}$;
Cluster: $G_{res}(V_{res}, E_{res})$;
Operators Placement: $X^k = \{x^v_i | i \in V^k_{dag}, v \in V_{res}\},\ \forall k$;
Coflow information: $\mathbf{D} = \{D_b | \forall b \text{ are aggregators}\}$;
Output: Flow rate: $R_b = \{r_1, r_2, ..., r_p, ...\}, \forall b$

Initial the link capacity $link(v, v')$ for every link in the cluster;
An unsatisfied coflow set $F = \emptyset$;
Sort $\mathbf{D}$ by the coflow size $\sum_{e_{(p,i)} \in D_b} d_p$ for each coflow $D_b \in \mathbf{D}$;
for *each* D_b *from the smallest coflow* **do**
    Allocate the rate for each $e_{(p,i)}$ as $r_p = e_{(p,i)}$;
    Update the link capacity $link(v, v') = link(v, v') - r_p$, for $x^v_p = 1$, and $x^{v'}_b = 1$;
    if *any of the above* $link(v, v') <= 0$ **then**
        Revoke the above allocation and mark coflow D_b as unsatisfied $F = F \cup \{D_b\}$;
Set the rate for coflows in F by fair sharing the remaining capacity of each link;

The example shown in Figure 3.8 also illustrates the algorithm. There are two applications in which operator placements are done. The link capacities between the gateways and the micro datacenter are both 2 Mbit/s. If the bandwidth is allocated by a fair sharing mechanism as shown in Figure 3.8(b), all

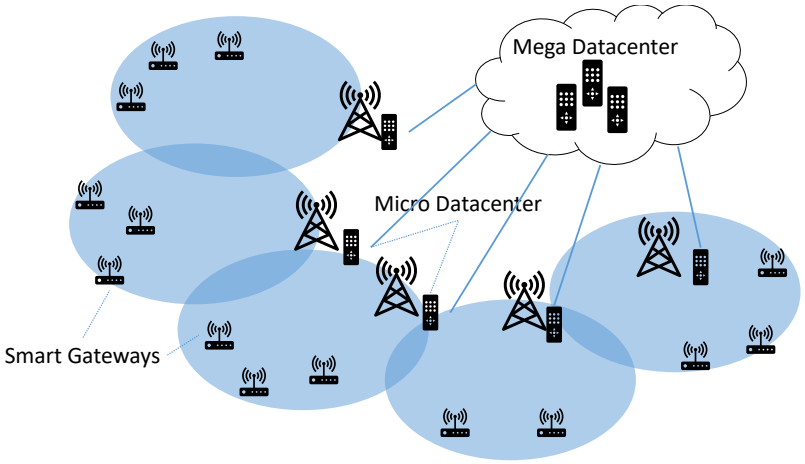


Figure 3.9: Testbed

coflows cannot be completed on time and all of the three applications will be influenced by the back-pressure. However, if we allocate the bandwidth by Algorithm 5, at least the second application can avoid the back-pressure.

3.4 Evaluation

We evaluate Amnis using a set of stream processing applications in an edge computing experimental testbed. The evaluation is designed to measure the performance improvement of Amnis for various stream processing applications and compare its performance with the state-of-the-art.

3.4.1 Implementation and experimental setup

We implement Amnis on top of Apache Storm [135] (v2.0.0). Amnis can also be implemented on other distributed stream processing engines such as the Apache Flink[133], Heron[77], etc. We chose Storm for implementing Amnis due to its widespread use in data science applications [9] and Storm has the lowest overall latency [38] among leading stream processing engines. We extend the DefaultResourceAwareScheduler[1] (DRA) in Storm to implement our algorithms for the physical plan and operator placement optimizations. The coflow optimization is enforced on each operator using an integrated rate controller implemented in the IRichBolt class which is the base class for implementing many operators.

We deploy a testbed on CloudLab [45] with nodes organized in three tiers as shown in Figure 3.9 and

[1] `http://storm.apache.org/releases/2.0.0/Resource_Aware_Scheduler_overview.html`

Table 3.2: Testbed Setup

Nodes	instance	vCPU	RAM(GB)
Mega Datacenter	m1.2xlarge	16	32
Micro Datacenter	m1.xlarge	8	16
Gateway	m1.medium	2	4

Table 3.2. We use the cluster with ten m510 servers in the CloudLab cluster and simulate the three-tier architecture on an Openstack [2] cluster, which is like the multi-tier experiment testbed in related works [141, 42, 10]. The third tier contains fourteen m1.medium instances (2 vCPUs and 4 GB memory) that act as the smart gateways with relatively low computing capacity corresponding to the leaf nodes of the architecture, the second tier has five m1.xlarge instances (8 vCPUs and 16 GB memory), each of them functions as a micro datacenter, and the first tier contains one m1.2xlarge instance (16 vCPUs and 32 GB memory) acting as the computing resource which is available in the mega datacenter. The network bandwidth, latency and topology are configured by dividing virtual *LAN*s (local area networks) between the nodes and adding policies to the ports of each node to enforce, which we apply by the *Neutron* module provided by the OpenStack project and the traffic control (tc) tool in Linux.

The Amnis platform is deployed on the above-described testbed. We deploy the Storm Nimbus service (acting as the master node) on the m1.2xlarge instance and one Storm Supervisor service (acting as the worker node) on each node respectively. In the Storm setup, we increase the default number of slots from 4 to 8 for each Storm supervisor to enable the cluster to run more operators simultaneously on each node so that we can deploy multiple applications on the cluster. The default network is set to be 2 Mbit/s bandwidth in capacity with 20 ms latency between the gateways and micro datacenters, and the bandwidth capacity is 100 Mbit/s with 50 ms latency between the mega datacenter and micro datacenters. In addition, in the default setup of each application, we place a stream generator on each smart gateway to emulate the input stream. The input stream comes from an MQTT service deployed on each smart gateway. The default stream rate is set to be six hundred tuples per source (smart gateway) per second *per application*. We enable the **at-least-once** semantic on the Storm cluster and the MQTT services, so that each tuple will be processed at least once that contributes to the final result. With this feature, we do not take care of the failure during the stream processing, because if a tuple that is being processed or transferred fails, the tuple will be replayed or re-transmitted again.

3.4.2 Applications

To comprehensively evaluate Amnis, the stream processing engine *simultaneously* runs multiple stream applications. We choose an application from vehicular networks using the Linear Road Benchmark [15], one

[2] `https://www.openstack.org/software/`

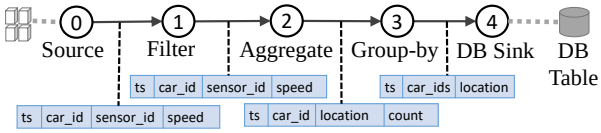


Figure 3.10: $Q1$

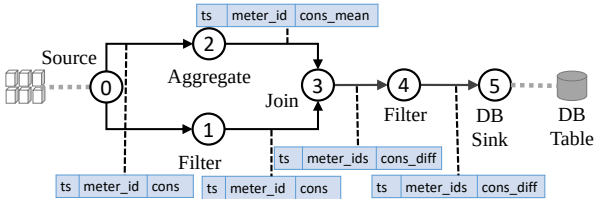


Figure 3.11: $Q2$

application for Smart Grid infrastructure, an application for a VR (virtual reality) Game and a smart city application that calculates the profitability of each area for taxi cars on the road network. Each benchmark contains various kinds of operators (filter, join, aggregate, etc.) with different computational complexities and input/output ratios. The details are described as follows.

($Q1$) **Accident detection**: The first application use case is to detect accidents in linear roads as shown in Figure 3.10. The sensors gather the position and the speed of each car passing it. Then the sensor data is filtered using the condition $speed < \epsilon$, where ϵ is a small number indicating the error range of the sensor, which has around a hundred to one input/output ratio. After that they are aggregated with car ID and time window. When a car is detected to be not moving in a particular continuous time window, it is treated as a broken car and the position will be reported to the next operation. After the detection of broken-down cars, there is a group-by operator that performs group-by operations for the cars based on location which also identifies whether there is more than one car broken in the same position. In the end, the position is reported and stored in a database table.

($Q2$) **Anomaly detection**: The second use case is to detect abnormality in Smart Grids as shown in Figure 3.11. We change the long-term example described in [110] to a short-term one which detects anomaly usage by comparing the windowed average usage and the instantaneous power consumption (which is sampled one from one hundred metrics). When the consumption difference is larger than a threshold, then the difference is reported to be stored into a database table.

($Q3$) **VR game**: The third use case is to simulate a real-time VR game application that gathers the information from multiple VR devices and updates the virtual world states as shown in Figure 3.12. There are multiple players playing a VR game. The VR devices will send its current state (direction and position) one hundred times per second to a filter which compares the current state with the previous state. If the state difference is larger than a threshold, it passes the state to the next operation (the input/output ratio is ten). The states are gathered and mapped to the virtual global coordinate.

($Q4$) **Profitable Areas**: Last one is a taxi trace analysis application based on the dataset and requirement provided in 2015 DEBS Grand Challenge [3]. The goal of the application is to calculate the top ten profitability

[3] `http://www.debs2015.org/call-grand-challenge.html`

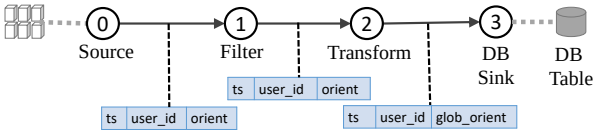


Figure 3.12: $Q3$

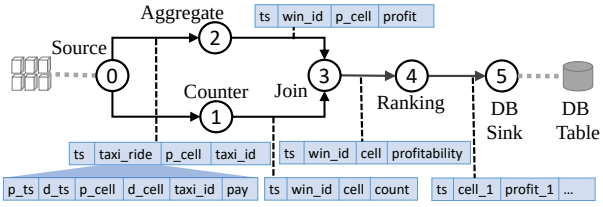


Figure 3.13: $Q4$

areas by processing the taxi rides stream in real-time as shown in Figure 3.13. It first counts the taxi ride and aggregates the profit for each area in each time window (in the default query, the window size is fifteen minutes, which is set to be one second in the experiment). Then, it joins the above two streams to generate profitability for each area and sorts the profitability of each area in real-time. Any time the ranking changes or the time-window reaches, the updated ranking will be sent to the output. The emulated input stream is sampled from the trace provided on the website and replicated corresponding to the input rate setup in the experiment.

3.4.3 Evaluation Results

In our experiment analysis, different mechanisms are measured and compared: (i) **Amnis** which includes all the proposed optimization features, (ii) **SINK**, a heuristic that tries to put all the operators co-located with the *sink* operator, (iii) **DRA**, the original default resource-aware scheduler in Apache Storm that can be seen as a round-robin algorithm which randomly chooses a physical node and puts as many operators as possible until the computational resources are saturated and then, it randomly chooses another physical node, (iv) **Greedy First-Fit (GFF)**, a heuristic approach proposed in [103] that ranks the nodes using a penalty function that considers the network latency and specifications of the node, the heuristic then schedules the operators to the nodes based on the rank in a first-fit manner, and (v) **Local Search (LS)**, an approach proposed in [103] based on the above Greedy First-fit algorithm, it traverses the neighbor solutions to get a local optimal solution, where the neighbor solution refers to the solution in which the difference is only a single operator placement compared to the result of **GFF**. It is worth noting that except for the **SINK** mechanism, all the other techniques assume that the operators' resource requirement can be determined as provided by the user (e.g., **DRA** needs the resource usage information of each operator to be given as parameters) or by profiling. In addition, **Amnis**, **GFF** and **LS** also need the network topology information. For all the mechanisms except **SINK**, the changes in the characteristics of the application or the environment (cluster topology) need to be handled by re-configuring.

To compare the performance of the above algorithms, we set up two different scenarios: (i) we change the input rate on the stream generator for all the applications to test the performance of the algorithms in different workloads and (ii) we modify the last hop bandwidth (from the smart gateways to the Micro

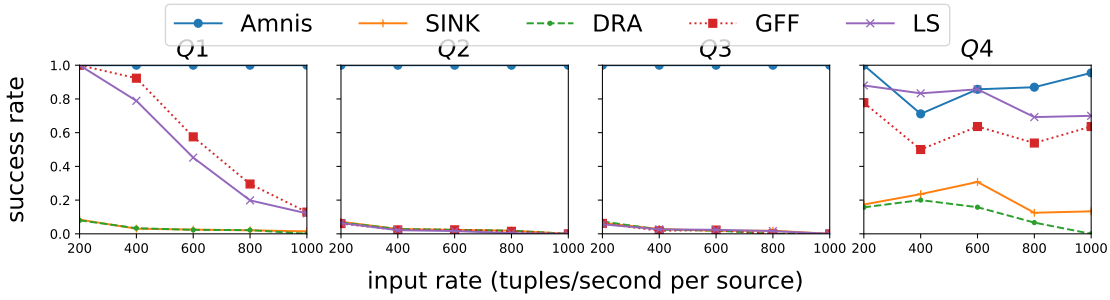


Figure 3.14: Success Rate Comparison with different input rates

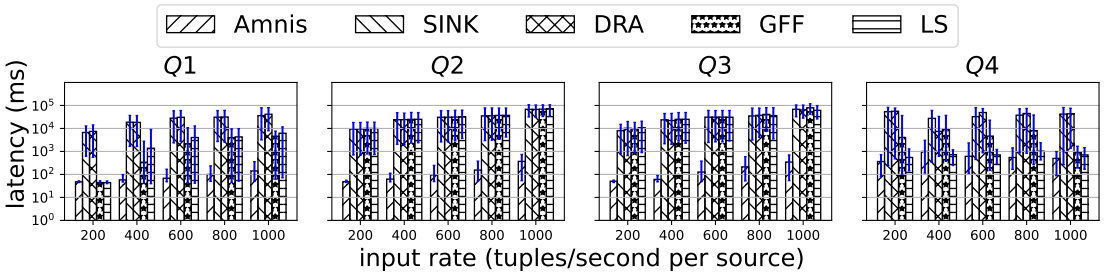


Figure 3.15: End-to-end latency comparison with different input rates

Datacenters) to different values to test the performance under different network constraints. We measured the general critical metrics to the stream processing applications: (i) latency-oriented success rate (maximal 100%, and higher is better), which is calculated based on the number of outputs that can be generated within the deadline (one second in default setting), (ii) end-to-end latency (lower is better), which is calculated by subtracting the output timestamp from the last input tuple which contributes to the output, (iii) bandwidth usage (lower is better), which is the cumulative bandwidth usage during running, (iv) average throughput (higher is better), which is calculated by the overall number of tuples processed dividing the experiment time, and (v) sustainable throughput [70] (higher is better), which is the maximal input rate that the scheduled physical plan can handle without incurring back-pressure. All the results shown in the figures are the average of three runs. For each run, we initialize the four applications at the same time and warm them up for two minutes. Then, we change the workload to the rate indicated in each experiment and gather the results for ten seconds.

As shown in Figure 3.14, we measure the success rate of the stream processing queries. In the figure, we can see that Amnis performs significantly better than the other algorithms for $Q1$ to $Q3$. In most settings, Amnis obtains a nearly 100% success rate within the given deadline. When the input rate increases and when other constraints are kept constant, the performance difference between Amnis and other methods increases substantially. Only for $Q1$, the GFF and LS methods achieve 100% success rate when the input rate is 200 tuples per second per source. For $Q2$ and $Q3$, the success rate of the GFF and LS methods is less than 10%

in the conditions. For the other methods namely SINK and DRA, the performance is even worse and the success rate is less than 10% for $Q1$, $Q2$, $Q3$. We see a different trend for $Q4$ as the aggregation window size is one second and the latency is calculated based on the time interval between the generation time of the last tuple in the time window and the output time of the final result, which causes the average latency to be around one second. Here again, Amnis works better than the other methods and obtains nearly 100% success rate under various conditions for $Q1$ to $Q3$ and obtains higher success rates for all the four queries compared to the other methods.

In Figure 3.15, the latency measurements and their distribution (with 90% confidence interval) are shown for different input rates. The y-axis in the figure is the end-to-end latency in ms and it is in log scale. We can see that for $Q1$ to $Q3$, Amnis achieves an average latency of around 50 ms to 300 ms when the input rate increases. Also, the 95% percentile latency is from 150 ms to 450 ms with increase in input rate. The other methods have the average latency from one second to tens of seconds when the input rate increases from two hundred per second to one thousand per second. GFF and LS get better results for $Q1$ than SINK and DRA as GFF and LS consider the average distance between the nodes to decide the operator placement. The average latency difference is as high as 200X when Amnis is compared with the other four methods for $Q2$ to $Q3$. Even for $Q1$, the difference is significant when *Amnis* is compared with LS when the input rate is one thousand tuples per second. The difference is about 84X (6056ms average latency for LS, and 72ms average latency for *Amnis*). In addition, we can see that the latency distribution is more narrow when Amnis which indicates that latency is more predictable and stable for *Amnis* compared with other methods. Here, we note that the y-axis is in log scale and therefore, the same length in the higher position may indicate several orders of magnitude difference in values. In addition, for $Q4$, SINK, DRA, GFF, and LS generate only three or four outputs when the bandwidth is one Mbit/s because of the bottlenecked network bandwidth, which are less than the normal cases that the number should be more than ten (at least one per second). However, we calculate the success rate only from the output latency distribution which is the portion lower than the threshold (one second in default setting) that we do not assume the number of outputs we should get to calculate the success rate that cause the result here. Also for $Q4$, LS performs similar to Amnis across the setups and GFF performs similar to Amnis when the bandwidth is larger than two Mbit/s.

In Figure 3.16, the results of the success rate for different last hop bandwidth (the bandwidth between the smart gateways and the micro datacenters) is shown. Here, the deadline is set as one second. The result is similar as above and Amnis reaches 100% success rate in almost all of the scenarios for $Q1$ to $Q3$. For $Q1$, we can see that when the bandwidth increases, the success rate increases gradually to 100% for GFF and LS methods. However, for $Q2$ and $Q3$, the success rate does not change significantly as these two applications have higher computational resource requirements and the network is not the bottleneck for these applications. For $Q4$, the success rate of Amnis is also near 100% when the bandwidth is more than 4 Mbit/s but the success rate ranges from 25% to 75% for GFF and LS which is still better than that of SINK and DRA approaches. The latency measurements presented in Figure 3.17 show the same trends as the

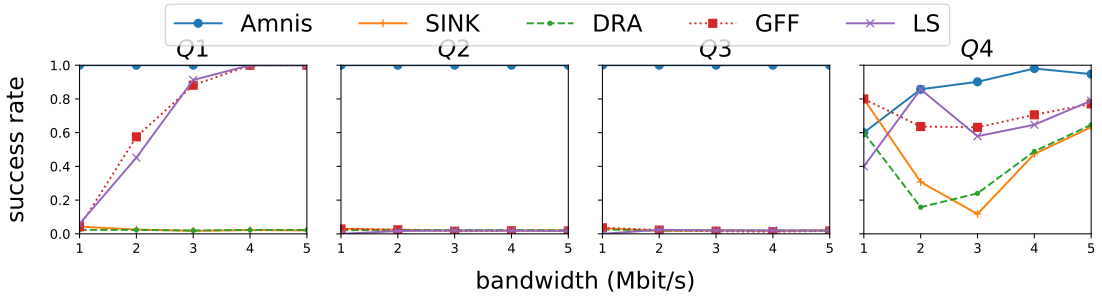


Figure 3.16: Success Rate Comparison with different last hop bandwidth

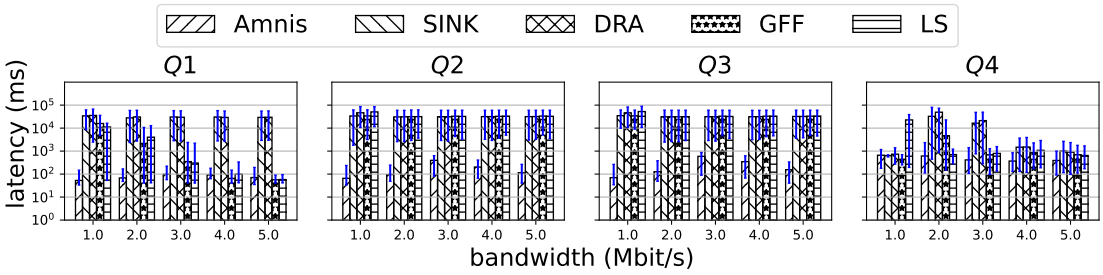


Figure 3.17: End-to-end latency comparison different last hop bandwidth

success rate and Amnis performs better than the other techniques. In the figure, we can see that the latency is around 40 to 80 ms for *Amnis* for $Q1$, $Q2$, and $Q3$. However, for the other four methods, the latency is as high as more than 20 seconds. For $Q4$, the latency is around one second for Amnis, GFF and LS but for SINK and DRA, the latency is as high as more than 10 seconds. Based on the results in Figure 3.18, we can infer the reasons behind these trends. The figure shows the cumulative bandwidth usage (sum of the bandwidth usage during the experiments) for different bandwidth setup. The usage is gathered in each node by a monitor thread in real time (one second per metric) when the experiment is running. The label $1e6$ in the up left corner represents the scale of the y-axis, which also applies to Figure 3.19 and 3.20. We can see that Amnis consumes somewhat similar network resources when the bandwidth is increased from one Mbit/s to five. However, for the other four methods, the bandwidth consumption is about two times when the bandwidth is increased from one Mbit/s to two, which shows that the bandwidth resource is saturated by the other four methods but for Amnis, it is similar for different bandwidth setups. It shows that the overall bandwidth utilization is minimized by the methods in *Amnis*.

Besides the above metrics, we also evaluate the overall throughput under the above two setups (different input rates and last hop bandwidths). As shown in Figure 3.19, we evaluate the throughput with different input rates. We add an *optimal* line in the figure that represents the throughput value which consumes the

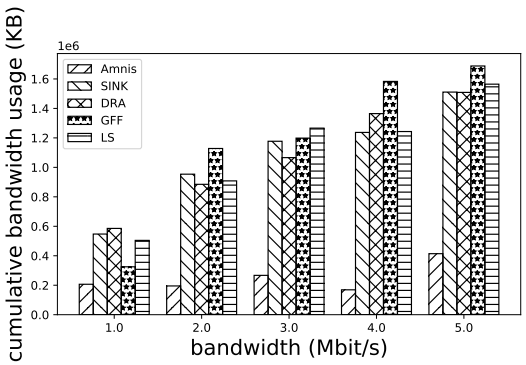


Figure 3.18: Network Usage

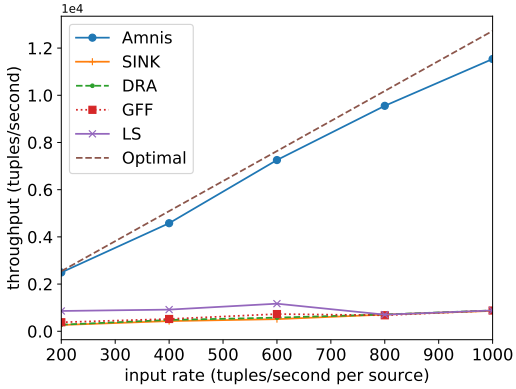


Figure 3.19: Throughput with different input rates

input without delay. In the figure, we can see that Amnis achieves very similar results as the *optimal* line. However, the other four methods only get low throughput which has as much as 10X difference compared with Amnis. In Figure 3.20, the results show the sustainable throughput for different bandwidth setups. The sustainable throughput [70] for each method and each bandwidth setup is obtained by gradually increasing the input rate and analyzing the latency distribution. If the latency is increased continuously during the experiment, the current plan is not sustainable which also means that if the current input rate is kept the same, the current method cannot handle it under the current physical plan and operator placement. For *Amnis*, we reuse the plan in the default setup (the input rate is six hundred tuples per second and the last hop bandwidth is two Mbit/s) to get the sustainable throughput. We can see that Amnis also performs better than the other four methods in different setups. For $Q1$ to $Q3$, Amnis achieves around six thousand tuples per second throughput with the optimized plan. For $Q1$, when the bandwidth is larger than three Mbit/s, GFF and LS can achieve similar throughput as Amnis. For $Q4$, as it aggregates the tuples in a longer time window (the back-pressure is active only when the processing cannot be completed in a time window), the throughput of this application is similar to other mechanisms, which is around three thousand tuples per second. From the above results, *Amnis* can get better results not only when the accurate application information (e.g., how many tuples are coming to the application?) is given but also by employing various *Amnis* methods with estimated information. Thus, the applications using the *Amnis* approach obtain good throughout under various scenarios.

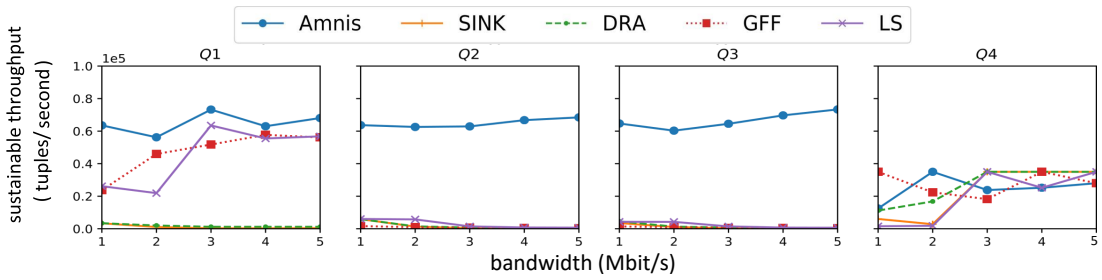


Figure 3.20: Sustainable Throughput with different last hop bandwidths

3.5 Discussion

In this section, we discuss the limitation and future research directions for Amnis.

Generalization and applicability. Our work assumes that the stream processing applications are deployed directly by the distributed stream processing engine on multi-tier edge computing environments. The challenges imposed by the difference in capabilities of resources present in various tiers represent the key difference of edge computing scenarios compared to cloud computing environments [22, 158, 121]. Essentially, our methods can work for many stream processing application scheduling contexts in multi-tier edge computing environments. However, we assume that the control signal in the stream processing application will not be the bottleneck. For example, if the stream processing application is deployed in a geo-distributed edge computing environment, the synchronization of the state of the application (e.g., the back-pressure signal) may become a problem or bottleneck. While this is not handled in our work, we believe that tackling this challenge can be a promising direction for future work.

Profiling and reconfiguration. Our work assumes that the profiling and benchmark results are available for the stream processing applications. Our methods use these basic information of the applications (e.g., the average input rate, the average processing time, the selectivity, and the resource usage) to make informed decisions. While our model and methods can tolerate a certain amount of error, they make require reconfiguration of the application deployment to adapt to long-term changes, especially when there are very significant changes in the environment (e.g., a long-term change in the input rate). Making optimized decisions on when and how to perform the reconfiguration is still an open problem. Future work can focus on developing a framework to automate the reconfiguration process by adaptively learning the environment and application characteristics (e.g. a reinforcement learning-enabled agent to automatically develop reconfiguration strategies).

Data locality. Our work optimizes the data locality primarily by grouping the selective operators with the source operators and by moving them to the data source (Section 3.3.1.1). We assume that the selective operators are stateless so that we can arbitrarily split them with the same number of the corresponding

source operator. However, the selective operator can also be stateful as in the case of windowed maximal and key-by sum operators. The reason we limit the applicability to the stateless operator is that the split stateless operators do not need to consider the shuffling phase which needs to be correct (e.g., for key-by sum, all the tuples belonging to a key needs to be processed by one of the downstream operators) and it can be a bottleneck between the split source operators and the downstream operators. However, it is also possible to automatically split the stateful operator to improve the data locality which we leave it as future work. In addition, Our work does not assume any specific partition function to be used between two connected operators. The partition function can influence the performance in some cases. For example, the partition function can prefer the successor task which is placed locally with the current task to improve the data locality. The optimization of the partition function has been widely studied in the distributed batch processing domain [166], however, applying those methods in a heterogeneous edge computing environment is still a challenge which can be a promising direction for future work.

3.6 Summary

In this chapter, we propose a novel stream query processing framework called Amnis that optimizes the performance of the stream processing applications through a careful allocation of computational and network resources available at the edge. The Amnis approach differentiates from the state-of-the-art through its careful consideration of data locality and resource constraints during physical plan generation and operator placement for the stream queries. Additionally, Amnis considers the coflow dependencies to optimize the network resource allocation through an application-level rate control mechanism. We implement a prototype of Amnis in Apache Storm. Our performance evaluation carried out in a real testbed demonstrates the effectiveness and scalability of the Amnis approach. Our results show that the proposed techniques achieve as much as 200X improvement on the end-to-end latency and 10X improvement on the overall throughput.

The optimizations discussed in this chapter help meet the performance requirements of low-latency stream processing in heterogeneous edge computing environments. However, as edge infrastructures consist of several unreliable devices and components in a highly dynamic environment, failures are more of a norm than exception [81]. Thus, to support reliable delivery of low latency stream processing over edge computing, we need a highly fault-tolerant stream processing solution that understands the properties of the edge computing environment for meeting both the latency and fault tolerance requirements. In the next chapter, we propose a novel resilient stream processing framework that achieves system-wide fault tolerance while meeting the latency requirement for the applications.

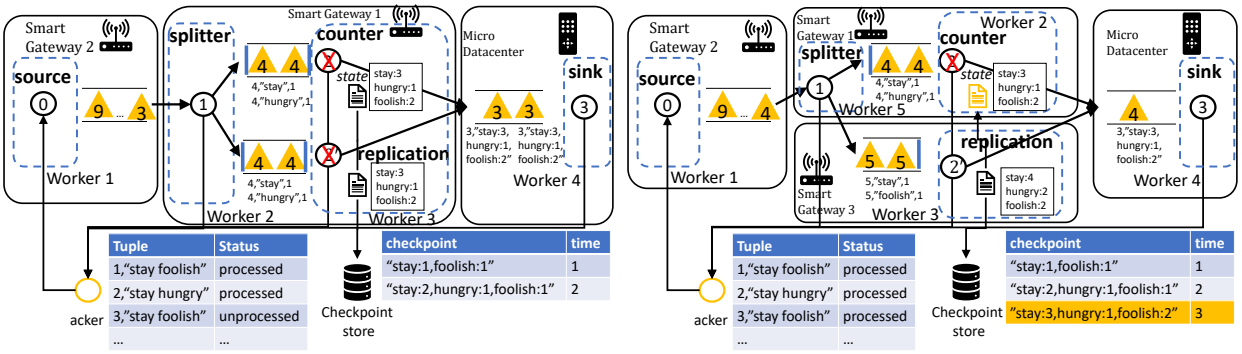
4.0 Resilient Stream Processing in Edge Computing

In this chapter, we present a novel resilient stream processing framework that achieves system-wide fault tolerance while meeting the latency requirement for the applications in the edge computing environment. The proposed approach employs a novel resilient physical plan generation for the stream queries that carefully considers the fault tolerance resource budget and the risk of each operator in the query to partially actively replicate the high-risk operators in order to minimize the recovery time when there is a failure. The proposed techniques also consider the placement of the backup components (e.g. active replication) to further optimize the processing latency during recovery and reduce the overhead of checkpointing delays. We extensively evaluate the performance of our techniques by implementing a prototype on Apache Storm [135] on a cluster test-bed in CloudLab[45]. Our results demonstrate that the proposed approach is highly effective and scalable while ensuring low latency and low-cost recovery for edge-based stream processing applications.

4.1 Background and Motivation

In this section, we discuss the state-of-art stream processing fault tolerance solutions and illustrate the challenges in supporting resilient and fault tolerant stream processing in the edge computing. As the number of IoT devices increases, large amounts of data get generated near the edge of the network in real-time. Traditional cloud solutions for IoT may lead to long processing times and may not be suitable for latency-sensitive IoT applications such as virtual reality (VR) applications (that require less than 16 milliseconds latency to achieve perceptual stability) [120], and applications for smart cities such as connected vehicles (e.g., collision warning, autonomous driving and traffic efficiency with latency requirement around 10 to 100 milliseconds) [14] and intelligent online traffic control systems [22]. Stream processing in an edge computing environment provides a promising approach to meet strict latency requirements while processing huge amounts of data in real-time. Fault tolerance is an important aspect of edge computing as many IoT applications require both high accuracy and timeliness of results. As edge infrastructures consist of several unreliable devices and components in a highly dynamic environment, end-to-end failures are more of a norm than exception [81]. Thus, to support reliable delivery of low latency stream processing over edge computing, we need a highly fault-tolerant stream processing solution that understands the properties of the edge computing environment for meeting both the latency and fault tolerance requirements.

Checkpointing[145] and replication [63] represent two classical techniques for fault tolerant stream processing. The idea behind replication is to withstand the failure by using additional backup resources. Replication approaches include both active replication and standby replication. When there is a failure, the backup resources will be used to handle the workload impacted due to the failure. The difference between active replication and standby replication is that in active replication, both the primary and the replica run



(a) Resilience Unaware

(b) Resilience Aware

Figure 4.1: An example comparing the resilience unaware scheduling and the proposed approach

simultaneously and process the same input and produce the output. However, in standby mechanisms, the standby does not generate outputs. In hot standby mechanisms, the standby processes the tuples simultaneously similar to the primary but with cold standby, the standby only synchronizes the state with the primary operator passively.

The checkpointing mechanisms on the other hand periodically store the state of the operators in persistent storage to create timestamped snapshots of the application. Often, the checkpoint mechanism works with the replay mechanism that caches the input tuples at the *source* operator (e.g. the *source* operator shown in Figure 4.1). When there is a tuple failure or a timeout, the *source* operator re-sends the tuple to the downstream operators for reprocessing. The replay mechanism tracks the processing status of each tuple (e.g., the *acker* shown in Figure 4.1 is used to track this information). When a failure happens, the state of the failed operator is restored and the unprocessed (unacknowledged) tuples are replayed as shown in the example in Figure 4.1. As shown in Figure 4.1, the stream processing application has four operators, a *source* operator which fetches the stream from the data provider outside of the system, a *splitter* that splits the sentence into words, a *counter* that counts the words, and a *sink* that stores the result. It also includes fault tolerance components namely the *acker*, the checkpoint store, and active replication of the *counter* operator.

Hybrid methods employ a combination of both checkpointing and replication. They are referred to as adaptive checkpointing and replication techniques. Adaptive checkpoint and replications schemes have been proposed in several domains [129, 89, 168, 142, 60, 128]. The goal of combining active replication and checkpoint mechanisms is to achieve seamless recovery compared to pure checkpointing. When active replication is applied correctly, the recovery time will be zero (the input of the failed operator is handled by the replica). Combining active replication and checkpointing also decreases the resource overhead as it significantly employs checkpointing that only needs a very limited resource (transferring and storing the checkpoints as shown in Figure 4.1) compared to fully active replication that needs nearly twice the resource usage of the original.

While adaptive checkpointing and replication is a promising approach, its application in edge computing is challenged in several aspects. The heterogeneous nature of both physical nodes and the network components in an edge computing environment significantly challenges the placement of the replicas that directly influences the performance and cost. For example, in Figure 4.1b, if active replication of *counter* is placed in a node far from the *splitter* and *sink* operators assuming a latency of 200ms between the smart gateway 3 and the micro datacenter, the latency to transmit the stream to the *sink* operator from the replica will be very high. We note that it will not influence the performance in the fail-free condition as the application will eliminate the duplicate results at the end and the output from the primary will be always generated earlier than the output from the replica in this condition. However, when the primary *counter* operator fails, the output of the active replication will become the valid output and it will dramatically influence the latency that makes the results not useful as it may contain out of date information. For instance, let us consider that the latency requirement is less than 200ms and the normal processing from the *source* to the *sink* needs 100ms. In the fail-free condition, the latency requirement will be met. However, if the primary operator fails, the output comes only from the replica during the recovery phase and a bad placement of the replica can drastically increase the latency to more than 200ms which may violate the latency requirement and make the results not useful.

Most current state-of-art distributed stream processing systems (e.g. Apache Flink [133], Apache Storm [135]) optimize the performance of an application by placing the operators in a single worker (single process) and in a single physical node or a set of adjacent physical nodes to improve the data locality by reducing the overhead of copying or transmitting the data across the processes or nodes. Therefore, if the placement mechanism is unaware of the fault tolerance mechanisms and requirements, the fault tolerance properties achieved by the mechanisms may be poor. For example, in the worst case as shown in Figure 4.1a, when the active replica is placed along with the primary operator in the same worker 3 (the dotted line in the figure indicates the boundary of a worker), the active replica will also fail because of the influence of the primary failure which makes the active replication scheme ineffective. While optimizing for performance and data locality is important, careful decisions on where to place the backup resources (e.g. the active replication, the checkpoint store) while closely considering their roles in the application is vital to achieving the desired resilience properties. In the example shown in Figure 4.1b, we place the active replication in a different node near the node where the primary is placed so that the failure of either will not influence each other. Here, careful tradeoffs between data locality and minimizing correlated failure probabilities are essential to ensuring both high resiliency and performance in terms of throughput and latency. In the next section, we discuss the system design of our proposed fault tolerant stream processing mechanisms optimized for edge computing environment.

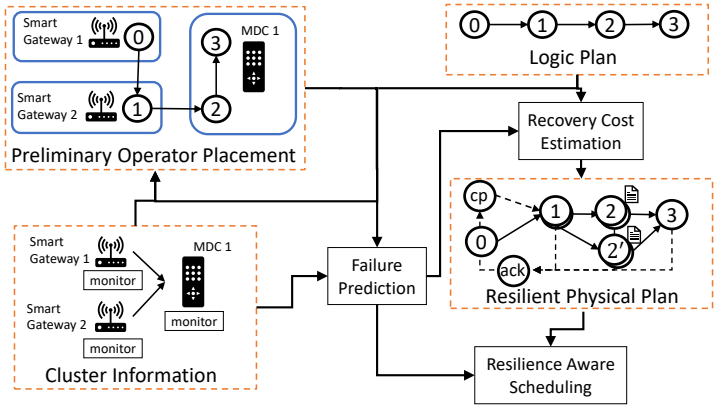


Figure 4.2: System Overview

4.2 System Design

The proposed fault tolerance mechanism for edge computing consists of two phases namely (i) resilient physical plan generation, and (ii) resilience-aware scheduling and failure handling. In the resilient physical plan generation (Figure 4.2), we decide the physical plan of the stream processing application by first translating the user code into a logic plan and then, based on the preliminary operator placement result, we add the necessary backup components (e.g. active replication, and checkpoint store) considering the recovery cost for each operator. Then, when the physical plan is going to be deployed on the cluster, the system needs to decide the placement of both the operators and the backup components and handle the failure when the application fails.

Next, we discuss the details of various components (Figure 4.2) of the proposed resilient stream processing system.

4.2.1 Resilient physical plan generation

A stream processing application can be represented as a Directed Acyclic Graph (DAG) which captures the processing graph provided in the user-defined program. A *logic plan* illustrates the logic of the stream processing application represented by the DAG. Thus, a logic plan consists of vertices and edges where the vertices represent the operators defined by the user and the edges are the streams between the operators as shown in Figure 4.2. We use the notation, $G_{logic}(V_{logic}, E_{logic})$ to represent the logic plan. We note that the logic plan is not the task graph directly deployed onto the cluster. A physical plan extends the logic plan with more detailed configurations including the level of parallelism for each operator, the configuration for the backup operators, etc. The resilient physical plan is represented as a graph $G_{phy}(V_{phy}, E_{phy})$. When

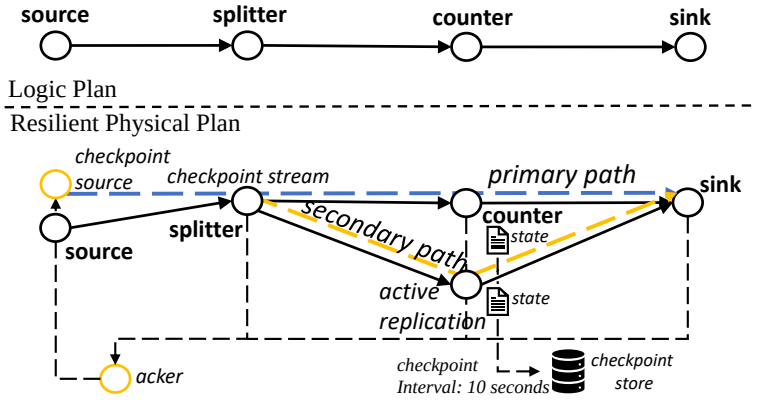


Figure 4.3: Resilient Physical Plan Example

generating the resilient physical plan, the system needs to decide several parameters including the operators which are actively replicated.

We present an example in Figure 4.3, Besides the logic plan, the resilient physical plan generation includes many other components: (i) the *acker* is an operator tracking whether the tuples have been completely processed in the application, (ii) the *checkpoint store* provides the services to store the checkpoint in the volatile memory or in the persistent storage, (iii) the state management for each stateful operator (e.g. the *counter* operator) indicating the checkpoint mechanism and the parameters of the checkpoint mechanism (e.g. the checkpoint interval), (iv) the fault tolerance mechanism for each operator. For example, the *counter* operator is protected by both the active replication, the checkpoint mechanism and the event replaying (the *acker* feedback loop). We denote the fault-tolerant physical plan including the backup components as $V_{phy} = V_{logic} \cup \{i_{ckstore}, i_{cksource}\} \cup V_{acker} \cup V_{active}$, where $i_{acker} \in V_{acker}$ is an *acker* operator in the acker set, $i_{ckstore}$ is the checkpoint store which can be a local database or a memory key-value store. Here $i_{cksource}$ is a *source* operator which is responsible for generating the checkpoint stream (signal) based on the checkpoint interval configuration, which we will discuss the details about the checkpoint stream later, and V_{active} is a copy of the subset of V_{logic} which defines the operator set that actively replicates the operators in the logic plan. The details of how to configure the above components are described in Section 4.3. Also, in the physical plan, the parallelism of each operator i can be decided by the user to determine how many tasks will be run to process the input of an operator which directly influences the decision of active replication. Here we use i^k to indicate the k-th task of an operator i and the parallelism is denoted as K_i, a predefined parameter.

The checkpoint stream is a fault tolerance component which will pass through all the operators in the logic plan in the same sequence defined by the logic plan. When one of the operators receives the checkpoint tuple (barrier), the stateless operator will forward it to the downstream operators, and the stateful operator will

halt the stream processing and perform the checkpointing by committing the current state to a checkpoint store. The checkpoint stream is for synchronizing the state over the whole stream processing application to ensure that the checkpointing is done across the application as an atomic operation. When the checkpoint tuple (barrier) passes all the operators in the logic plan, an acknowledgment will be made by the *acker* to inform the *checkpoint source* that the current checkpointing is done. Next, we discuss the notion of operator and backup component placement and illustrate how the failure is handled by the backup components.

4.2.2 Scheduling and failure handling

After the resilient physical plan is determined, the system needs to decide the placement for each component in the physical plan and handle the application. The scheduling decision can be illustrated as a mapping between the physical plan graph G_{phy} and the cluster graph denoted by $G_{res}(V_{res}, E_{res})$ where the V_{res} indicates the nodes in the cluster and E_{res} indicates the virtual links connecting them. An example is shown in Figure 4.2 in which we have two smart gateways connecting a micro datacenter. Here, we use i to represent the component in the physical plan that $i \in V_{phy}$ and v to indicate the node in the cluster that $v \in V_{res}$. We use c_v to indicate the idle resource capacity of a node which is the full capacity subtracting the resource usage on that. For the mapping between the physical plan and the cluster, we use $X = \{x_v^i | i \in V_{phy}, v \in V_{res}\}$ to indicate if $x_v^i = 1$ then the component i is deployed on node v, and vice versa. After the physical plan is scheduled to the cluster, the stream processing runs continuously until it is shutdown. When there is a failure in the state-of-art stream processing systems (e.g. Apache Flink [133], Apache Storm [135]), the state of the application is typically backed up by checkpointing and the input data is backed up using a replay mechanism. Thus during recovery, a few steps can restore the application to its normal status. However, this process can be time consuming. The recovery time varies from application to application but usually, it can be divided into two parts: (i) the time for detection of the fault and (ii) the time for restoring the computation. For the first component, most distributed stream processing systems use *heartbeat* [4] to detect task failures and node failures. The *heartbeat* is a kind of signal sent between the monitor (e.g. a master node) and the monitored tasks. When there is a timeout of the *heartbeat*, the monitor will assume that the monitored task has failed and will trigger the recovery mechanism (e.g. restarting the fail task). For the second part, the application needs to recover from the failure in order to restore to the state before the failure. In distributed stream processing systems, the state of the operator is restored by the latest checkpoint and the tuples are replayed to gradually recover the state of the operator to the point before the fail happens. Thus, with the above two delays, the application will not produce any output until all the operators are synchronized to the state before the failure which will introduce high recovery latency and processing latency. The latency can cause several issues including violating the user's latency requirement and in some cases, the peak workload during recovery may cause other operators to fail consecutively.

Active replication can be one of the most promising supplementary technique for the regular checkpointing-based fault tolerance. The primary and secondary (replica) will run at the same time to produce results

so that only when both fail simultaneously, it will cause failure of the application, otherwise the failure will be seamlessly covered. There is no *heartbeat* detection latency and no restoring latency during the time of failure and the the total recovery time for this mechanism becomes nearly zero for most conditions. However, if we replicate all operators with active replications, it results in nearly twice the original resource usage cost to handle the workload. Therefore, we need a mechanism to estimate the risk of each operator based on the estimated recovery time and adopt cost-effective approach to selective replication.

4.2.3 Recovery time estimation

If the application is only backed up by the checkpoint mechanism or when the failed operator is not replicated by active replication, the recovery time can be significant. We first estimate the recovery time to obtain the risk of each operator to determine which operators need to be replicated. We note that the recovery time contains two components namely (i) fault detection and (ii) computation restoring.

Fault detection time is determined by the *heartbeat* interval, the latency between the monitor and the fail task and the configuration of the *heartbeat* timeout. We assume that the failed task is i and the monitor task is m which can be either a node manager (e.g. *supervisor* in Apache Storm), a cluster master (e.g. the *nimbus* in Apache Storm), or a cluster coordinator (e.g. a *zookeeper* cluster in Apache Storm). If the monitor is a node manager, the failure of task i can be detected by the timeout of the *heartbeat*, which is denoted as $\tau_{hbtimeout}$ (the timeout can be set by a parameter, for example it can be five seconds). Therefore, in the worst case, the time to detect the fault can be the sum of $\tau_{hbtimeout}$ and the heartbeat interval (the fault happens immediately after acknowledging the last *heartbeat*). As the heartbeat timeout is often significantly larger than the heartbeat interval, we can ignore the heartbeat interval and only consider the influence of the heartbeat timeout, $\tau_{hbtimeout}$, in the recovery. If there is a node v failure, the time to detect the failure needs to include the latency between the failed node and the monitor node which is denoted as $l(v, m)$. The second part namely the time to recover the computation is more challenging to estimate as it related to many aspects including the length of the unacknowledged queue of the tuples, the size of the state, and the average processing time of the fail operator.

When restoring the computation, the recovery time can be divided into three phases namely (i) restarting the task and loading the program into memory, (ii) retrieving of the latest checkpoint from the checkpoint store, and (iii) reprocessing the unacknowledged tuples to restore the computational state before the failure. For the first component namely restarting and loading the program, we assume it is a constant time $\tau_{restart}$. For the second part, the time to retrieve the checkpoint is noted as $\tau_{checkpoint}(size(s_i))$, which can be determined by the size of the state $size(s_i)$ where s_i denotes the latest state of an operator i, and the latency between the operator and the checkpoint store, which is denoted as $l(v, v')$ where v is the node in which operator i is placed, and v' is the node where checkpoint store is placed. For the third part, we need to know how many input tuples of operator i is not acknowledged yet. We assume that it is a function $q_i(\lambda_i)$ denoting the input buffer of the unacknowledged tuples which is related to the input rate λ_i of operator

i. For each tuple, we need d_i to fully process it on average and we can estimate the replaying time as $\tau_{replay} = d_i q_i(\lambda_i)$. With above mentioned steps, we can estimate the overall recovery time if the operator i fails without an active replication by adding the detection time and the restoring time as shown below:

$$\begin{aligned} \tau_i(X) = \quad & \tau_{hbtimeout} + l(v,m) + \tau_{restart} \\ & + \tau_{checkpoint}(size(s_i)) + l(v,v') + \tau_{replay} \end{aligned} \tag{7}$$

Based on the above recovery time estimation, we can estimate the risk of each operator with the operator placement decision and optimize it further to either add more active replications or migrate some of the operators to reduce the risk, which we will discuss the detail in Section 4.3. In the next subsection, we will introduce the method we use to predict the failure, which is an important component to achieve an accurate recovery cost estimation.

4.2.4 Failure prediction

Our method is based on the accurate prediction of the failure. We summarize the failure modes to include: (i) tasks failures in which a specific task fails (e.g., due to memory issues), (ii) node failures in which a node or the supervisor deployed on it fails causing all tasks running on it to fail, (iii) data failures which refer to the loss of data that may occur due to data dropping on a congested network device or a communication timeout due to a high latency network.

For the task and node failures, there are many related works [119] that predict such failures accurately with close to 99% accuracy. There are also some recent efforts in the IoT domain [19] which predicts the failure of the IoT devices. For the data loss, we do not handle it through active replication but through the back-pressure and replaying techniques in stream processing [31] which we discuss in Section 4.3.

With the above observations, we can leverage the failure prediction to help us predict the risk of each operator. We assume the failure probability is $p_i(t)$ for task i and $p_v(t)$ for a node v in a time-slot t. We assume the node failure will cause all the tasks (operators) placed on it to fail. Thus, if a preliminary operator placement $X_0 = \{x_v^i | i \in V_{logic}, v \in V_{res}\}$ is determined as shown in Figure 4.2, we can combine the respective task and node failure probabilities together to form a uniform failure probability $\rho_i(t) = p_i(t) + p_v(t) - p_i(t)p_v(t)$, that $x_i^v = 1$ with the assumption that the task failure and the node failure are independent events. The failure probability may change due to the workload change or environment change but the probability can be updated by the prediction algorithm before each time-slot using the monitors deployed on each node as shown in Figure 4.2. Within each time-slot, we can decide a resilient physical plan based on the risk estimated by the recovery time and the failure probability predicted by the prediction algorithms, which consists of the original operators, their placement, fault tolerance components configuration, and the placement of the components. We discuss its detail in the next section.

4.3 Resilient Stream processing

In this section, we describe the proposed algorithms for resilient stream processing in edge computing that leverage both checkpointing and active replication techniques. We assume that the user can specify a fault tolerance resource budget which can be the amount of the additional computational resources (e.g. CPU, memory). We transform the budget to quantify the extra resources to be used to handle the fault (e.g. the resource amount can be calculated using the resource unit price). Then the multi-dimension resource amount (CPU, Memory, bandwidth, etc.) can be converted into a one-dimension amount using methods such as the dominant resource described in [52]. We use C to denote the additional resources that can be used to run the fault tolerance components (e.g. the checkpoint store and active replication).

With the budget configuration, the resilient physical plan can be generated by considering the risk and failure probability of the operator to selectively replicate some of the operators which have higher recovery costs (e.g. recovery time) and higher failure probabilities. Thus in the physical plan generation, we estimate recovery cost by combining the risk (recovery time) with failure probability: $a_i(t, X) = \tau_i(X)\rho_i(t)$, which can be also seen as the expectation of the recovery cost of the operator i in the time-slot t. For simplicity, we assume that the basic physical plan which is directly generated from the logic plan G_{logic} and the operator placement are already decided by parsing the user's program and calculated by a scheduler. We use X_0 to denote the original operator placement decision generated by the default scheduler (e.g. the default resource aware scheduler in Apache Storm [135]). Our algorithm will use the determined operator placement decision to further optimize the configuration and placement for the fault tolerance components. We note that the placement of the operators can influence the performance of the stream processing application and therefore, jointly optimizing the configuration and placement of both the operators and the fault tolerance components can be an interesting direction of future research. In this chapter, we primarily focus on the fault tolerance aspect and its influence on the applications. We divide the proposed fault tolerance solution into two phases: (i) checkpoint related component configuration (ii) active replication related configuration. For the checkpoint related component, we need to decide: (i) where to place the checkpoint store, (ii) how many *ackers* we need to use to track the completion of each tuple and where to place them. For the active replication, we need to decide: (i) which operators to be actively replicated, (ii) where to place these active replications. In the rest of this section, we illustrate our proposed solutions to achieve fault tolerance in edge computing environments by leveraging both checkpointing and active replication while carefully considering the resource budget and latency requirement.

4.3.1 Checkpoint

The checkpointing mechanism periodically takes snapshots of the state of the whole stream processing application, and when there is a failure, the checkpoints can be used to restore the state of the application to a state before the failure happens. However, merely restoring the state is not sufficient to guarantee

the correctness of the processing as the input data that do not contribute to the restored state should be replayed in order to make sure that they also contribute to the final output. We leverage two techniques here to guarantee the correctness when failure happens: (i) checkpointing which includes the snapshot mechanism, the state committing by the stateful operators, and the checkpoint storage, and (ii) input data replaying which includes the tuple tracking mechanism and the replaying mechanism.

As we want to ensure that all the operators are backed up, we enable the checkpointing mechanism across the application as the basic fault tolerance mechanism before applying the active replication. We take the logic plan of the application $G_{logic}(V_{logic}, E_{logic})$ and add the checkpointing related components into it including a checkpoint store $i_{ckstore}$ which is responsible for storing the checkpoints, a checkpoint source $i_{cksource}$ generating the stream to synchronize the snapshot status across the application, and a set of ackers V_{acker} tracking the accomplishment of all the tuples. Besides, we need to decide the placement for the above-mentioned components. As the checkpoint source only influences the snapshot step by generating synchronization signals and tracking the checkpointing step, the influence of the placement of it is not very significant. Here, we can simply collocate it with one of the *source* operators in the logic plan. For the checkpoint store, we need to consider the network connectivity to all the stateful operators which commit checkpoint information to it. If the operator fails without an active replication, it needs to communicate with the checkpoint store to fetch the latest state promptly. Besides, the checkpoint will be committed to it periodically from the stateful operators and therefore, we need to select a node which is in an appropriate node in which all the stateful operators can commit the checkpoint to it without waiting a long time to get the acknowledgment. In our work, we assume that one-node checkpoint store is capable of handling the checkpoints of the stream processing application with a resource requirement $c_{ckstore}$. For the highly geo-distributed application, we can employ geo-distributed key-value stores [167] to implement a more scalable checkpoint store. We have the initial operator placement decision X_0. We use $V_{stateful}$ to denote the set of all the stateful operators in the application and we can compute the objective function:

$$\min \sum_{i}^{V_{stateful}} l(v, v') \text{ that } x_v^i = 1, x_{v'}^{i_{ckstore}} = 1 \tag{8}$$

The problem can be solved by traversing all the nodes and the computational complexity is $O(|V_{res}|$ $|V_{stateful}|)$.

Next, for the *acker* which tracks the completion of each tuple, we need to determine the following: (i) the minimal number of *ackers* that can handle the tracking of the application, (ii) the placement of the *ackers* to minimize the gap between the accomplishment and the acknowledgment of the tuples, which in turn minimizes the unnecessary replaying of the tuples when there is a failure. We assume that the capacity of an *acker* is o_{acker} with a resource requirement, c_{acker}, which indicates that one *acker* can handle the tracking of at most o_{acker} tuples in a unit time. Therefore, the number of *ackers* can be calculated as follows: $|V_{acker}| = \frac{\sum_i^{V_{logic}} \lambda_i}{o_{acker}}$. The placement can be also decided similar to the checkpoint store that we

need to minimize the weighted distance between the ackers and the operators:

$$\min \sum_{i_{acker}}^{V_{acker}} \sum_{i}^{V_{logic}} \frac{\lambda_i l(v, v')}{|V_{acker}|} \text{ that } x_v^i = 1, x_{v'}^{i_{acker}} = 1 \tag{9}$$

To traverse all the combinations, the computational complexity ranges from $O(|V_{res}||V_{logic}|)$ to $O(\binom{|V_{res}|}{|V_{acker}|} |V_{logic}|)$ which is determined by how many nodes are used for placing the ackers. The result of the above problem composes of a placement decision of the checkpointing components, which we denote as $X_{checkpoint}$ consisting of the placement of the checkpoint source, the checkpoint store, and the *ackers*.

4.3.2 Active Replication

The checkpointing mechanism backs up the application entirely by periodically snapshotting the state of the whole application. However, the recovery time is significant if there is a failure. The restarting of the failed task, the restoring of the state, and the replaying of the unacknowledged tuples incur significant time. Therefore, we add the active replication to the operator which has a higher failure probability and a longer estimated recovery time by considering the fault tolerance budget defined by the user.

Algorithm 6: Select and place active replication

Input : Logic plan: $G_{logic}(V_{logic}, E_{logic})$;
Operator placement: X_0;
Checkpoint Component placement: $X_{checkpoint}$;
User fault tolerance budget for active replications: $C_{active} = C - c_{ckstore} - |V_{acker}|c_{acker}$;
Time-slot: t;
Output: Operator set to be replicated: V_{active};
Active replication placement: X_{active};

```
Initial the operator set V_active = ∅ and placement X_active = ∅;
Sort the operators in V_logic by their risk a_i(t, X) in a descending order;
for each operator i ∈ V_logic do
    if c_i(λ_i) ≤ C_active then
        V_active = V_active ∪ {i};
        Update C_active = C_active − c_i(λ_i);
for each active replication i' ∈ V_active do
    Get the node v which handles the primary operator i of the active replication i';
    Sort the neighbor nodes of v, v' ∈ V_res by their distance l(v, v') in an ascending order;
    for each neighbor node v' ∈ V_res do
        if c_i(λ_i) ≤ c_v' then
            X_active = X_active ∪ {x_i'^v' = 1};
            Update c_v' = c_v' − c_i(λ_i);
            Break;
For the remaining replication which does not find a placement, we use a network-aware mechanism[112] to
  place them;
```

There are two decisions we need to make when dealing with active replication: (i) which operators need to be actively replicated, and (ii) where to place the active replication to minimize the latency when there is a failure. For the selective active replication, we consider a user-defined budget C, which represents the resource that can be used by the fault tolerance components. The resource requirement for each operator

is $c_i(\lambda_i)$, which is related to the input rate. We assume that the resource requirement is a non-decreasing function of the input rate λ_i. We also assume that active replication replicates the method of the primary operator exactly the same way so that the resource requirement of the active replication of an operator i is the same, which is also $c_i(\lambda_i)$. The active replication selection method is shown in Algorithm 6 line 1-6. We can see that the algorithm selects the operators to be replicated based on the initial operator placement decision X_0 and based on their recovery cost until we reach the fault tolerance budget. The output is the operator set which is selected to be actively replicated, V_{active}. The computation complexity is determined by the sorting, which is $O(|V_{logic}| \log |V_{logic}|)$. After the active replicated operators are selected, we need to decide the placement of them to also minimize the latency during failure. Instead of understanding all the operators in the application to estimate the performance and schedule the active replication, we assume that the physical plan and the original placement X_0 already meet the service level agreement (SLA) with the user and thus, we focus on how to minimize the performance (especially latency) gap between the fail-free condition and fault condition. In Algorithm 6 line 7-15, we illustrate the detail of our proposed algorithm to place the active replications. The problem is solved by placing the replication to the node which is the nearest capable neighbor of the primary operator to minimize the influence of the network latency and other impacts on the active replication when there is a failure. The method fetches the placement information of the primary operator and tries to place the replication in one of the neighbors. It is worth noting that the neighbor information can be obtained by clustering the nodes in a latency space [112] or by a predefined cluster architecture as used in our experiments. The algorithm traverses the neighbors in the increasing order of distance (network latency) until there is enough capacity in a node to place the replication. The computation complexity is determined by the outer loop and sorting and the complexity is $O(|V_{active}||V_{res}| \log |V_{res}|)$.

4.4 Evaluation

We evaluate our fault-tolerant stream processing system in an edge computing experimental testbed. The evaluation is designed to measure the performance improvement of our method in comparison with both the baseline and state-of-art solutions. In the evaluation, we study the influence of the fault tolerance component placement and analyze the performance impact of using checkpointing as the only fault tolerance mechanism. Finally, we study the overhead of applying various fault tolerance solutions.

4.4.1 Implementation and experimental setup

We implement the system on top of Apache Storm [135] (v2.0.0). It can also be implemented on other distributed stream processing engines such as the Apache Flink[133]. We extend *DefaultResourceAwareScheduler*(*DRA*) in Storm to implement our algorithms for the physical plan and scheduling optimizations. The

physical plan generation and the scheduling decision is implemented outside of the scheduler. The scheduler takes the physical plan and the scheduling decision as input and based on them to schedule the tasks.

We deploy a testbed on CloudLab [45] with nodes organized in three tiers. We use the cluster with ten m510 servers in the CloudLab cluster and simulate the three-tier architecture on an Openstack [125] cluster. The third tier contains fourteen m1.medium instances (2 vCPUs and 4 GB memory) that act as the smart gateways with relatively low computing capacity corresponding to the leaf nodes of the architecture. The second tier has five m1.xlarge instances (8 vCPUs and 16 GB memory) and each of them functions as a micro datacenter. The first tier contains one m1.2xlarge instance (16 vCPUs and 32 GB memory) acting as the computing resource used in the cloud datacenter. The network bandwidth, latency and topology are configured by dividing virtual *LAN*s (local area networks) between the nodes and adding policies to the ports of each node to enforce. We use the *Neutron* module provided by the OpenStack project and the traffic control (tc) tool in Linux to simulate.

We deploy the Storm Nimbus service (acting as the master node) on the m1.2xlarge instance and one Storm Supervisor service (acting as the slave node) on each node respectively. For the checkpoint store, we use a single node Redis service. The default network is set to be 100 Mbit/s bandwidth in capacity with 20 ms latency between the gateways and micro datacenters, and the bandwidth capacity is 100 Mbit/s with 50 ms latency between the cloud datacenter and micro datacenters. We also place a stream generator on each smart gateway to emulate the input stream. The input stream comes to an MQTT (Message Queuing Telemetry Transport) [62] service deployed on each smart gateway. The default stream rate is set to be 100 tuples per source (smart gateway) per second (which is 1400 tuples per second in total).

4.4.2 Application

The application we use in the experiment is to detect accidents in linear roads as shown in Figure 4.4. The sensors gather the position and the speed of each car passing them. Then the sensor data is filtered using the condition $speed < \epsilon$, where ϵ is a small number indicating the error range of the sensor. After that they are aggregated by the location ID and time window. When a car is detected to be not moving in a particular continuous time window, it is treated as a broken car and the position will be reported to the next operation. In the end, the position is reported and stored in a database table. We implement the application based on the API provided by Apache Storm. For the windowed aggregator in the application, we enable the window persistence so that the tuples in every window will be stored in the checkpoint store periodically. The timeout parameter for tracking the completion of the tuple is set to be 5s. To generally accept the out of order tuples, we enable the lag parameter in the windowed aggregator as one second (the default setting is zero which means the lag tuples will be dropped immediately), which means that the out of order tuple can be accepted in a one second time window, otherwise it will be dropped. We set the window size to one second and the default fault tolerance budget to 10% of the original application. The default parallelism of the filter operator is set to be 14 which is as same as the number of the sources and the parallelism of the

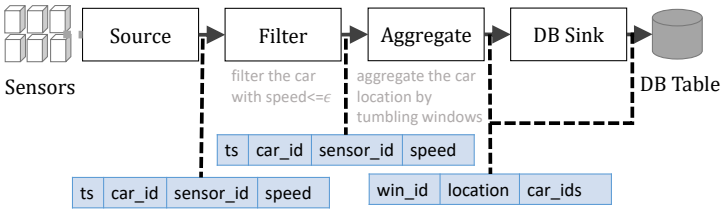


Figure 4.4: Accident Detection Application

aggregator is set to be 2.

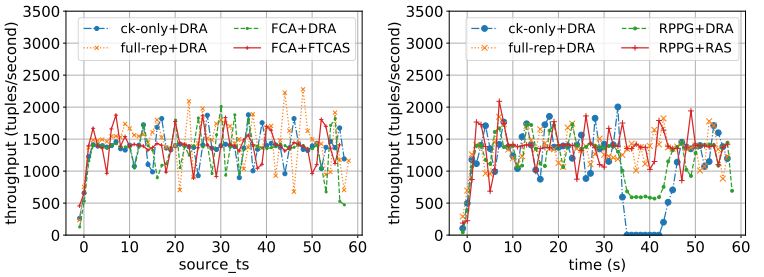
For each experiment, we run the application and the stream generators for one minute and let the application run another thirty seconds to let it fully process the input. The tuples which are not processed in the additional thirty seconds are considered as failed tuples.

4.4.3 Algorithm

We compare our methods in different combinations. We divide the proposed method into two parts: (i) Resilient Physical Plan Generation (*RPPG*) and (ii) Resilience-Aware Scheduling (*RAS*). For the physical plan generation, we compare with: (i) *ck-only*, which only applies the checkpoint to achieve fault tolerance, and (ii) *full-rep*, which applies full active replication to protect all of the operators. For the scheduling optimization, we compare *RAS* with *DRA* which is the default scheduler Apache Storm uses as described above.

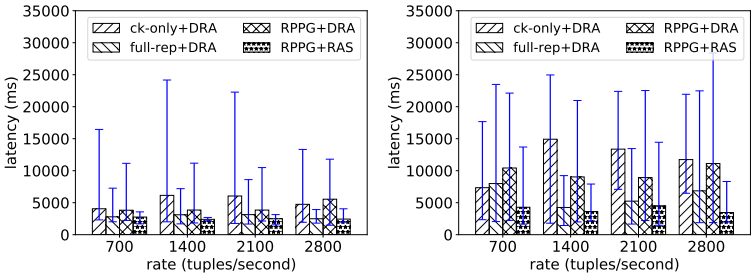
4.4.4 Experiment Results

We first evaluate the performance to compare fail-free and fixed failure conditions as shown in Figure 4.5. We can see that when there is no failure, the throughput fluctuates near the input rate for all the four mechanisms. However, when we inject a failure at the 30s, we can see the difference as shown in Figure 4.5b. The *ck-only+DRA* mechanism has a throughput gap after the failure injection as the primary needs time to recover from the failure. After the recovery is done within about 10 seconds, the throughput gradually becomes normal. For the *RPPG+DRA* mechanism, it uses our proposed mechanism to generate the physical plan but uses the *DRA* to schedule the tasks. Here, we can see that there is also a drop after the failure but the throughput is about half of the input rate unlike *ck-only+DRA* that does not output anything during the recovery. As *RPPG+DRA* replicates some of the operators and the scheduling places the primary and replication on one node, the failure of the primary influences some of the replications (if they are placed in one worker). For *full-rep+DRA* and *RPPG+RAS*, the throughput does not change significantly during the primary recovering from failure. In this experiment, we can see that applying only the adaptive fault



(a) Fail-free condition (b) Fault condition (fail injected at 30s)

Figure 4.5: Throughput



(a) Overall (b) During Recovery

Figure 4.6: Latency

tolerance does not solve the problem entirely. We also need to place the components appropriately to avoid the influence of the correlated failures to further decrease the influence of the failure on the application.

Next, we evaluate the latency of the application by applying the same four mechanisms. We change the input rates in these experiments as shown in Figure 4.6. The bars illustrate the average latency and the ticks represent the 90% confidence interval of the latency. We can see that our method *RPPG+RAS* performs similar to *full-rep+DRA* including the overall runtime which average latency is around 2.5 seconds as shown in Figure 4.6a. However, the *ck-only+DRA* performs similar with *RPPG+DRA* that gets around or higher than 5 seconds latency. When comparing the latency during recovery, the difference becomes larger as shown in Figure 4.6b. We can see that the average latency during recovery is all increased to more than 5 seconds for the mechanisms except *RPPG+RAS* even for the *full-rep+DRA*. The reason is that the bad placement of the replication influences the effectiveness of the application when there is a fail. This experiment shows the priority of our method in the latency metric that our method achieves both lower latency and higher stability comparing with the other three mechanisms whenever considering the latency distribution in overall runtime or only during recovery.

In addition, we compare success rates in Figure 4.7. We observe that the result is similar to the one shown

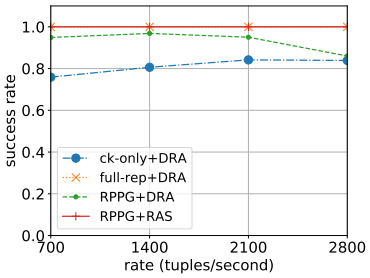


Figure 4.7: Success rate

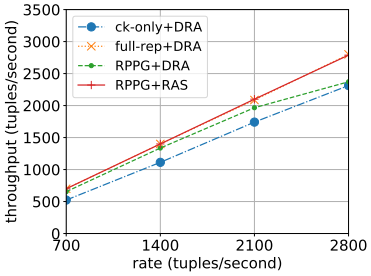
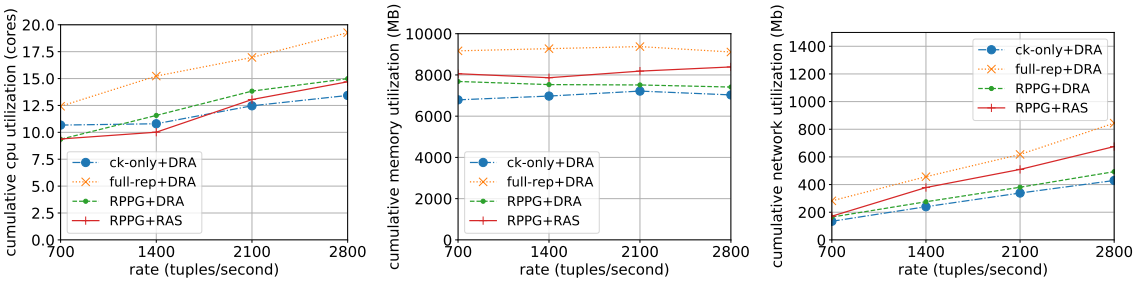


Figure 4.8: Throughput



(a) CPU (b) Memory (c) Network

Figure 4.9: Resource utilization

in Figure 4.5. When there is a failure, our methods *RPPG+RAS* and *full-rep+DRA* are not influenced and hence, they obtain 100% success rate for different input rates. Here, *RPPG+DRA* obtains only around 95% success rate which is higher than the checkpoint only mechanism that has around 80% success rate.

Next, we compare the throughput of the mechanisms in Figure 4.8 when the input rate increases. We can see that *RPPG+RAS* and *full-rep+DRA* achieve similar throughput when input rate increases and it matches the input rate. However, the *RPPG+DRA* and *ck-only+DRA* achieve lower throughput than the input rate which may lead to either loss of data or delayed output.

Finally, we evaluate the resource utilization of the four mechanisms as shown in Figure 4.9. We can see that *full-rep+DRA* uses significantly more resources than the other three techniques. The CPU resource utilization shown in Figure 4.9a increases from 12.5 cores to 19 cores when the input rate increases. The rest of the techniques are similar to each other with CPU utilization ranging from around 10 cores to around 13.5 cores. Overall, the CPU resource usage is as high as 40% more than the other three mechanisms for *full-rep+DRA*. The result is similar in memory and network usage as shown in Figure 4.9b and 4.9c. In this experiment, we can see that full replication uses more resources than our method in all the CPU, memory, and network usages. Comparing our method *RPPG+RAS* to the *ck-only+DRA* and *RPPG+DRA*, the CPU usage is similar but the network usage is higher as *RAS* schedules the active replications to different nodes

which increases the network usage but decreases the influence of correlated failures.

In summary, the proposed method, *RPPG+RAS* combines the consideration of both generating an appropriate resilient physical plan to cover the operators with using less resources than the full replication and also the application's latency requirement to achieve similar latency when recovery from fail with better performance during fail comparing with *RPPG+DRA* which only optimizes the physical plan but lacking the consideration of the optimization of the scheduling.

4.5 Summary and discussion

Edge computing provides a promising approach for efficient processing of low latency stream data generated close to the edge of the network. Although current distributed stream processing systems offer some form of fault tolerance, existing schemes are not optimized for edge computing environments where applications have strict latency and recovery time requirements. In this chapter, we present a novel resilient stream processing framework that achieves system-wide fault tolerance while meeting the latency requirement for edge-based applications. The proposed approach employs a novel resilient physical plan generation for stream queries and optimizes the placement of operators to minimize the processing latency during recovery and reduce the overhead of checkpointing delays. The proposed techniques are evaluated by implementing a prototype in Apache Storm [135] and the results demonstrate the effectiveness and scalability of the approach.

This chapter and the previous chapter discussed techniques to improve the fault tolerance and performance of the stream processing application in edge environments using off-line optimizations for physical plan generation and scheduling of the operators. However, in scenarios where there are changes in the workload distribution or the environment, profiling-based off-line mechanisms may be less efficient. To address this issue, in the next chapter, we proposed a mechanism to dynamically configure the parallelism of the stream processing applications to meet the workload demands based on a model-based reinforcement learning method.

5.0 Elastic Stream Processing in Edge Computing

In this chapter, we proposed a reinforcement learning-based method which achieves elasticity for stream processing applications deployed at the edge by automatically tuning the applications to meet the performance requirements. The proposed approach adopts a learning model to configure the parallelism of the operators in the stream processing application using a reinforcement learning(RL) method. We model the elastic control problem as a Markov Decision Process(MDP) and solve it by reducing it to a contextual Multi-Armed Bandit(MAB) problem. The techniques proposed in our work uses Upper Confidence Bound(UCB)-based methods to improve the sample efficiency in comparison to traditional random exploration methods such as the ϵ-greedy method. It achieves a significantly improved rate of convergence compared to other RL methods through its innovative use of MAB methods to deal with the tradeoff between exploration and exploitation. In addition, the use of model-based pre-training results in substantially improved performance by initializing the model with appropriate and well-tuned parameters. The proposed techniques are evaluated using realistic and synthetic workloads through both simulation and real testbed experiments. The experiment results demonstrate the effectiveness of the proposed approach compared to standard methods in terms of cumulative reward and convergence speed.

5.1 Problem Formulation

DSP applications are often long-running and can experience variable workloads. Additionally, the highly dynamic edge computing environments may change the working conditions of the applications requiring operators to be migrated between nodes with different capacities due to failures or mobility requirements. To bound the performance of the applications within an acceptable range, it is important to design an elastic parallelism configuration algorithm to adapt to the changes in the dynamic edge computing environment. In this section, we first explain the terminologies used in modeling the elastic parallelism configuration problem.
Logic Plan: we assume that there is a stream processing application submitted to the system. The code of the application is translated into a Directed Acyclic Graph (DAG) denoted as $G_{dag}(V_{dag}, E_{dag})$ shown as the logic plan in Figure 5.1. Here, the vertices V_{dag} represent the operators and the edges E_{dag} represent the streams connecting the operators. We use $i \in V_{dag}$ to denote operator i in the application.
Cluster: we assume that the resources of the cluster are also organized as a graph, G_{res}
(V_{res}, E_{res}), where V_{res} denotes nodes in the cluster, and E_{res} indicates the virtual links connecting the nodes. As shown in Figure 5.1, in the edge computing environment, there are multiple tiers of resources such as the micro datacenters (MDCs) and the smart gateways which are deployed near the edge of the network that are used for processing the data locally to provide low latency computing to the applications. Thus, it is

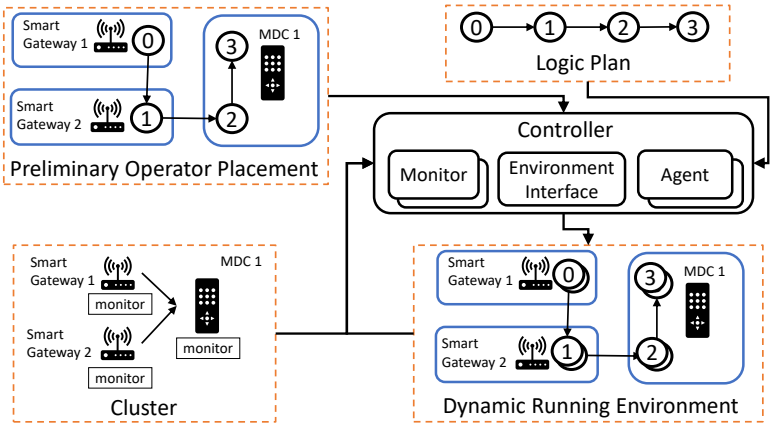


Figure 5.1: Elastic Stream Processing Framework

natural to consider the edge computing as a heterogeneous environment with highly dynamic changes in the execution environment. We simplify the physical resources as virtual nodes in the graph, G_{res}. For example, if a node v represents a micro data center (MDC), we group its resource capacity as C_v by considering all the resources we can use in an MDC as a virtual node.

Preliminary Operator Placement: the operator placement is a map between the operator, $i \in V_{dag}$, and the node, $v \in V_{res}$, in the cluster. We assume that the operator placement for stream processing application is already provided as shown in Figure 5.1. It is denoted as a map $X_0 = \{x_i^v | i \in V_{dag}, v \in V_{res}\}$. For each operator $i \in V_{dag}$, when it is placed on node $v \in V_{res}$, then $x_i^v = 1$. It is worth noting that, in the current state-of-art DSP engines (such as Apache Storm, and Flink), one operator can be replicated to multiple instances (tasks) and the instances (tasks) of the operator can be placed on different nodes. We assume that each operator is placed on one node as it simplifies the representation complexity of the model. Additionally, if the operator placement needs to be reconfigured to fit the working environment changes, we can treat the reconfiguration as a new submission as it does not affect the performance of the parallelism configuration algorithm.

Elastic Parallelism Configuration: the objective of configuring the parallelism is to change the number of instances of the operator to optimize one or multiple QoS requirements of the application. We need to decide the parallelism of each operator i, which is noted as $k_i \in [1, K_{max}]$, where K_{max} is an upper bound for the parallelism. Therefore, the number of possible parallelism configurations is ${K_{max}}^{|V_{dag}|}$ for configuring the whole application, if all the operators have the max parallelism as K_{max}. The parallelism determines the number of threads running for the instances of the operator, which is not directly related to the number of tasks provisioned for an operator. We discuss this in detail in Section 5.3. The configuration can be either static which is fixed when the application is submitted to the engine or dynamic that can be changed when

the application is running. In this work, we deal with the dynamic parallelism configuration problem and we present the detailed solution in Section 5.2.

5.1.1 Quality of Service Metrics

The objective of the parallelism configuration can be decided by the user in terms of QoS requirements. As most of the stream processing applications need to handle the incoming data under acceptable latency, the goal of the parallelism configuration can be to minimize the response time while reducing the resource cost and minimizing the gap between the throughput and the arrival rate to avoid back-pressure[77].

End-to-end latency upper bound: we assume the end-to-end latency of the application is primarily composed of computational or queuing latency. If the network latency or other latency e.g., the I/O latency caused by memory swapping, are significant in an application, we rely on other techniques to optimize the application first before deploying in the edge computing environment [112, 34], [36]. In order to make ensure the end-to-end latency is bounded by a user defined target, we traverse the path in the application's DAG to get the estimated end-to-end latency upper bound. We first define the path as a sequence of operators, starting at a source and ending at a sink, as $p \in P$, where P denotes all the paths in the application. We can estimate the latency upper bound (not tight) of the application as the longest path in the DAG:

$$\bar{l}^{dag} = \max_{p \in P} \sum_{i \in p} \bar{l}_i \tag{10}$$

where $\bar{l}_i$ is the latency upper bound when passing one of the instances of an operator i.

Throughput: in stream processing applications, the throughput requirement is typically defined by matching the processing rate of the application to the arrival rate. If the processing rate is larger than the arrival rate, the application will not incur a back-pressure [77], which will influence the performance of the application and may increase the resource usage (e.g., the memory usage for caching the unprocessed tuples). Therefore, to evaluate the throughput performance, we use the queue length, noted as ω, which is widely used in the queuing model to represent the state of the queue. It also captures the gap between the throughput and the arrival rate in the long run, which is easy to monitor in the stream processing engine.

Resource usage: for the resource usage, we can directly use the parallelism configuration to estimate, which is k_i for the operator i. With an increase in parallelism, there will be more threads allocated to the operator so that the resource usage will increase. Thus, parallelism can be used as the representation of the resource usage.

Reconfiguration cost: as we change the parallelism configuration when the stream processing application is running, it is important to consider the reconfiguration cost if the parallelism is changed (e.g., the operator needs to be restarted to apply the parallelism change). However, most of the previous works assume a static reconfiguration cost [34], which is a constant cost related to the downtime. This kind of measurements is not accurate due to the correlation between the reconfiguration downtime and other metrics such as latency and throughput. The downtime caused by the reconfiguration will lead to a peak latency and throughput after

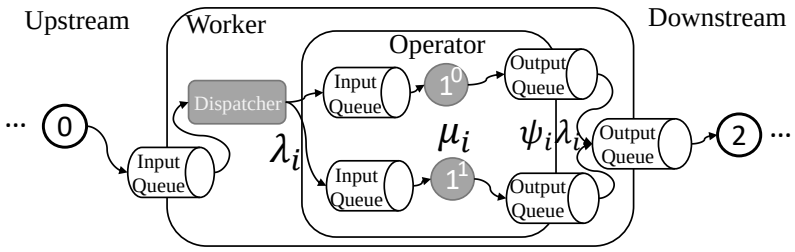


Figure 5.2: Stream Processing Model

the downtime. Therefore in this work, we do not include the reconfiguration cost in the objective. Instead, we include the downtime influence in the other metrics such as the end-to-end latency and throughput.

5.1.2 Stream Processing Model

With the notion of parallelism configuration and the QoS metrics defined above, we represent the stream processing model used to estimate the relationship between the decision (parallelism configuration) and requirements (QoS metrics) based on the queuing model of an operator and the message passing model of the stream processing application. The discussion will guide the later RL method design in Section 5.2.

As discussed in Chapter 1, the highly dynamic workload and the heterogeneous resources make it very difficult to predict the environment dynamics when the stream processing application is deployed in the edge computing environment. However, it is important to extract the invariant from the dynamics for human operators to understand the problem and the condition of the whole system to debug potential problems. Based on the intuition above, we adopt the model from queuing theory and choose the M/M/1 queue (can also be extended to G/G/1 based on distribution information) to model the characteristics of the operator. For each operator i, as shown in Figure 5.2, we assume that the instances of it do not share the input and output queues which can be treated as an M/M/1 queue. An M/M/1 queue can be described as two variables, λ_i, μ_i and one state ω_i, where λ_i is the arrival rate, μ_i is the service rate, and ω_i is the queue length. Based on the theory of M/M/1 queue [127], we can get the response time distribution (which is noted as latency in our work) and the throughput with closed-form formulations. When the queue is stable, which means $\mu_i > \lambda_i$, the queue length will not grow infinitely. Without losing generality, we analyze the latency upper bound here as an example. The 95th percentile of latency can be calculated from the cumulative distribution of an exponential distribution $Exp(\mu_i - \lambda_i)$ as follows:

$$\bar{l}_i = \frac{\ln 20}{\mu_i - \lambda_i} \tag{11}$$

Similarly, the other metrics can be also represented as closed-form formulations. For example, the average latency is $1/(\mu_i - \lambda_i)$. If the arrival rate and service rate distributions (G/G/1 queue) are given, Equation 11

can be modified correspondingly to represent the upper bound (95th percentile) of the latency using the cumulative distribution function. In addition, we added a variable, ψ_i, to enhance the queuing model, which represents the selectivity of the operator i, so that we can get the output rate as $\psi_i \lambda_i$ if $\mu_i > \lambda_i$ as shown in Figure 5.2.

After introducing the queuing model of a single operator, we now move to the model to estimate the performance of an application. As described in the beginning of this section, we assume that the application is organized as a DAG, $G_{dag}(V_{dag}, E_{dag})$, where each vertex $i \in V_{dag}$ represents an operator and each edge $(i, j) \in E_{dag}$ represents a stream. The tuples transmitted between two operators will be partitioned by a default shuffling function, or a user-defined partitioning function, which calculates the index of the downstream instance that the tuple will go to. In the message passing model, instead of composing the overall latency from source to sink as shown in Equation 10, we break down the latency caused on one operator and based on that, we set the target latency from the overall latency requirement. With the split objective, for each operator, the performance can be tuned without taking into consideration the other operators or the overall application. Here, we just use a simple heuristic to decide the maximum latency target of each operator proportional to its contribution to the overall latency:

$$\bar{l}_i^{max} = \frac{\bar{l}_i}{\bar{l}^{dag}} \bar{l}^{max} \tag{12}$$

where $\bar{l}^{max}$ is the upper bound latency of the overall application set by the user. If the profiling information is not available or not possible to obtain, we can use other heuristics such as evenly dividing the latency upper bound into the sub-objective of each operator with the given number of stages in the DAG. We leave the dynamic orchestration of the sub-objectives of the application's objective by gathering more information from executing the application as one of our future works. For the other metrics, such as throughput, we can monitor the input rate and processing rate for an operator and the throughput sub-objective can be directly obtained from the local information (e.g., queue length) of a particular operator so that we can rely on the local information to optimize the throughput. Therefore in the RL algorithm, we only need to focus on tuning the parallelism for one operator with the given sub-objective. The usage of sub-objective can decrease the complexity of the parallelism configuration problem, which we will discuss in details in Section 5.2.2.

In the next section, we present the details of the proposed model-based RL method based on the above model to automatically decide the parallelism in a dynamic and heterogeneous edge computing environment.

5.2 Reinforcement Learning For Elastic Stream Processing

We structure the elastic parallelism configuration as a Markov Decision Process (MDP) that represents an RL agent's decision-making process when performing the parallelism decision. We then reduce the MDP to a contextual MAB problem and apply LinUCB with a model-based pre-training.

5.2.1 A Markov Decision Process Formulation

The problem of continuously configuring the parallelism of the stream processing applications in an edge computing environment can be naturally modelled as an MDP. Formally, an MDP algorithm proceeds in discrete time steps, $t = 1, 2, 3, ...$:

(i) The algorithm observes the current DSP application state s_t, which is a set of metrics gathered from any monitor threads running out of the system (e.g., node utilization, network usage) or the metrics reported by the application itself (e.g., latency, throughput, queue length as described in Section 5.1.1).

(ii) Based on observed reward in the previous steps, the algorithm chooses an action $\mathbf{k}_t \in \mathcal{A}$, where $\mathcal{A}$ is the overall action space, and receives reward $r_{\mathbf{k}_t}$, whose expectation depends on both the state s_t and the action $\mathbf{k}_t$. In the parallelism configuration process, each action is composed by the parallelism configuration of all the operators, which can be noted as $\mathbf{k}_t = \{k_{t,i} | i \in V_{dag}\}$.

(iii) The algorithm improves its state-action-selection strategy with the new observations, $(s_t, \mathbf{k}_t, r_{\mathbf{k}_t}, s_{t+1})$.

We choose the *Finite-horizon undiscounted return* [37] as the objective of the MDP, which can be noted as:

$$\sum_{t=0}^{T} r_{\mathbf{k}_t}(s_t, s_{t+1}) \tag{13}$$

where, T is the number of continuous time steps considered in the objective. It is a cumulative measure of the undiscounted rewards in a predefined T time steps. As shown in the equation, compared to the infinite discounted reward, the finite undiscounted reward treats each time step equally. This fits the objective of the parallelism configuration that aims to maximize the utility uniformly among time steps. It also fits well into the contextual MAB problem which we discuss in the next subsection.

5.2.2 Model-based Reinforcement Learning

As discussed in Chapter 1, the traditional RL methods based on q-value tables or other methods need a large number of data points to converge. The deep reinforcement learning methods use DNN to improve the convergence rate but they also need a lot of efforts either to tune the hyperparameters to tradeoff between expressivity and the convergence rate or to gather enough data to feed into the neural networks, which may be costly or even not possible in some conditions. In addition, the incomprehensible and nonadjustable deep neural model is the major barrier for those kinds of models to be practical in system operations [96]. In our work, we use LinUCB [80] that assumes a linear relationship between the state and the reward, and is proved to be effective under the contextual MAB assumptions even when the process is non-stationary. The LinUCB method fits into the parallelism configuration problem well based on our two observations: (i) the parallelism configuration MDP (defined in Section 5.2.1) can be reduced to a contextual MAB, and (ii) we can define the reward function with a linear relationship between the reward and the parallelism configuration or most of the common objectives (e.g., latency, throughput) are linear (or can be transformed to be linear) to the parallelism configuration. Next, we discuss the above two observations in detail.

The major difference between the MDP and the contextual MAB is based on whether the agent considers the state transitions to make the decision. From the theory of M/M/1 queue [127], we can see that for each time step, the state transition is only dependent on the arrival rate λ, the service rate μ, and the initial state of the time step, ω, which is the initial length of the queue. Therefore, if we have the above variables in a particular state, we can get the state transition probability for any possible states in the next time steps. If the distributions of the arrival process and service process are stationary, the reward (determined by any QoS metrics) can be determined by the current state and action regardless of the trajectory of the previous states. It also means that the decision of the action can be made based on the current state instead of the trajectory. The above observation is intuitive when there is only one operator. If there are multiple connected operators organized as a DAG, the problem is significantly more complex. However, instead of connecting the queuing model of each operator to build a queuing network, we can break the objective (reward) function of the overall application using a heuristic (as discussed in Section 5.1.2) based on the message passing model to the individual objective (reward) for each operator so that for each operator, we can safely use the LinUCB algorithm to fit the queuing model with the given objectives and also reduce the possible action space for the RL method (from exponential to linear).

For the second observation namely, the linear relationship between the state and the reward, we begin analyzing it using a single queuing model. To keep it simple, we omit the time step notation t in the following discussion. If we have an operator i, it has only one instance. We then have the arrival rate λ_i, the service rate μ_i (for one instance), and the queue length ω_i. We assume the relationship between the parallelism setup and the speedup of the operator by comparing a single parallelism condition that obeys Gustafson's law[58] with a parameter ρ_i that defines the portion of the operation that can benefit from increasing the resource usage. Here $\mu_i(k_i, \rho_i)$ is the estimated service rate when the parallelism is k_i and the parallel portion is ρ_i, which can be estimated by:

$$\mu_i(k_i, \rho_i) = (1 - \rho_i + \rho_i k_i)\mu_i \tag{14}$$

Without losing generality, we estimate the latency (response time) distribution in the time step as an example, which can be an exponential distribution of $Exp(\mu_i(k_i, \rho_i) - \lambda_i)$ plus an estimated upper bound of the processing time of the queuing tuples $\omega_i Exp(\mu_i(k_i, \rho_i))$ (not tight). Therefore, the latency upper bound can be estimated as combining Equation 11:

$$\bar{l}_i(\lambda_i, \mu_i, \omega_i) = \frac{\ln 20}{\mu_i(k_i, \rho_i) - \lambda_i} + \omega_i \frac{\ln 20}{\mu_i(k_i, \rho_i)} \tag{15}$$

With the given parallel portion ρ_i and the average processing rate μ_i, the overall processing rate of the operator i with k_i instances is proportional to the number of instances, k_i. If throughput is part of the reward, it will have a linear relationship with the parallelism. For latency, in Equation 15, the operator will start at a state when the queue length is zero, $\omega_i = 0$, and the first part of the equation is inversely proportional to the processing rate if the arrival rate λ_i is fixed. Therefore, through a simple transformation (e.g., set $x_1 = 1/(\mu_i(k_i, \rho_i) - \lambda_i)$), we can refer to a linear relationship between the latency upper bound

and the parallelism. For the other metrics such as throughput, queue length, and resource utilization, we can also analyze the relationship between the parallelism and obtain similar results for a particular operator. Through similar simple transformations, the linear relationship between the metrics (which represent the states in RL methods) and the parallelism (number of instances) can be obtained.

Based on the two observations, we apply LinUCB as an RL agent to decide the parallelism configuration for an operator and pass the messages between the connected operators in the DAG. Using the notation of Section 5.2.1, we assume that the expected reward of an action (parallelism configuration) is linear in its d-dimensional state s_{t,k_i} with some unknown coefficient vector $\theta^*_{k_i}$. Therefore, the linear relationship between the reward and the state can be described as:

$$\mathbf{E}[r_{t,k_i}|s_{t,k_i}] = s^\top_{t,k_i}\theta^*_a \tag{16}$$

As described in LinUCB[80], it uses a ridge regression to fit the linear model with the training data to get an estimate of the coefficients $\hat{\theta}_{k_i}$ for each action k_i of each time step t. We omit the detailed steps of the LinUCB algorithm and we refer the interested readers to the original paper [80]. Here, we only discuss the action selection policy of LinUCB, which can be represented as:

$$k_{t,i} = \arg\max_{k_i \in [1, K_{max}]} \left(s^\top_{t,k_i}\hat{\theta}_a + \alpha\sqrt{s^\top_{t,k_i}(\mathbf{D}^\top_{k_i}\mathbf{D}_{k_i} + \mathbf{I}_d)^{-1}s_{t,k_i}} \right) \tag{17}$$

where α is a constant, $\mathbf{D}_{k_i}$ is a design matrix of dimension $m \times d$ at time step t, whose rows correspond to m training inputs (states), and $\mathbf{I}_d$ is a $d \times d$ identity matrix. From the above equation, we can see that the action selection of LinUCB considers both the current knowledge we obtained from the previous trials in $s^\top_{t,k_i}\hat{\theta}_a$ and the uncertainty (UCB) of the action-reward distribution in the second part of Equation 17. This is the reason why LinUCB has the ability to tradeoff between the exploration and exploitation.

With the above analysis, we can see that if LinUCB is directly used to set the parallelism for one operator, it can be efficient as the uncertainty of the operator can be captured by the linear model (e.g., the parallel portion, the base processing rate). However, on one hand, it has a cold start phase which needs multiple rounds to get enough data for each possible action to reach a reasonable performance level. On the other hand, in a stream processing DAG, the operators are connected to each other and the overall performance of the application may vary due to different bottlenecks. Given the DAG, it is a challenging problem to determine how to relate the overall performance of the application to the metrics of every operator in it. Therefore, instead of directly optimizing the overall application, we use the objective function split as discussed in Section 5.1.2 to only deal with the optimization for one operator for each RL agent. In addition, we use the queuing model-based simulation to validate the configuration to set the initial parameters for the LinUCB model. The simulation also gives additional benefits. On one hand, we can assume different distributions for the arrival rate and service rate that can support arbitrary G/G/1 queuing models, which is evaluated in the experiment in Section 5.4. On the other hand, the simulator can work in different modes to either generate a lot of synthetic data to directly feed into the model or interact with the RL agent as a simulation environment, which can fit into more RL algorithms. In the simulation, instead of trying all

Algorithm 7: Model-based LinUCB pre-train

Procedure `pretrain`$(G_{dag}) \rightarrow \Theta$
  q is initialed as an empty queue ;
  O are sources of G_{dag} (in-degrees are zero) ;
  for $o \in$ **O do**
    $q.append((o, \lambda_o))$;
  while *q is not empty* **do**
    $i, \lambda_i = q.pop()$;
    $\theta_i = train(i, \lambda_i)$ add trained parameters θ_i to output Θ ;
    for *all downstream operators i' of i* **do**
      $\lambda_{i'} = \lambda_{i'} + \psi\lambda_i$;
      remove edge (i, i') from G_{dag} ;
      if $indegree(i') == 0$ **then**
        $q.append((i', \lambda_i'))$;
Procedure `train`$(i, \lambda_i) \rightarrow \theta_i$
  initial the model parameters θ_i ;
  while *not terminate and not converge* **do**
    $k_{t,i} = selection(\theta_i, s_{t-1,k_i})$ by Equation 17 ;
    $r_{s_t,k_{t,i}}, s_{t,k_i} = simulate(\lambda_i, \mu_i, k_{t,i})$;
    $\theta_i = updateLinUCB(\theta_i, r_{s_t,k_i}, s_{t,k_i}, k_{t,i})$;
    reset operator i's state to initial state ;

the combinations of the parallelism configuration of the operators at the same time, we gradually train the model for each operator by a topological order [67] of the DAG to ensure that the upstream operators' configuration is fixed before the downstream operator's model is trained. The simulation process is shown in Algorithm 7. The reward function for each operator i is defined by using the Simple Additive Weighting (SAW) technique [162]:

$$r_i(s_{t,k_i}) = w_{lat} r_i^{lat} + w_{que} r_i^{que} + w_{res} r_i^{res} \tag{18}$$

where s_{t,k_i} is the state of time slot t with parallelism as k_i, and $r_i^{lat}, r_i^{que}, r_i^{res}$ are the reward function for latency, throughput, and resource usage based on the application's requirements. To balance the optimization for latency, queue length (gaps between throughput and input rate) and resource usage, we add $w_{lat}, w_{que}, w_{res}$ as the weights for each component and $w_{lat} + w_{que} + w_{res} = 1$. For different requirements, the reward function can be set in different forms. For example, if the application requires deadline-awareness and has a strict latency bound, we can set $r_i^{lat} = -1$ when $\bar{l}_i \geq l_i^{max}$, otherwise, it is zero. If the application's utility is linear to the latency, we can set $r_i^{lat} = -\frac{\bar{l}_i}{l_i^{max}}$, which decreases when the latency is increasing. Without losing generality, we define the reward function by setting $r_i^{lat} = -1$ if $\bar{l}_i \geq l_i^{max}$ else zero, $r_i^{que} = -1$ if $\omega_i \geq \omega_i^{max}$ else zero, and $r_i^{res} = -\frac{k_i}{k_i^{max}}$. In the definition above, both the latency and throughput penalties have a bounded reward. The reward of resource usage is linear in terms of the number of instances running. To eliminate the impact of the state transition that the model has experienced through a previous bad selected action (e.g., a bad parallelism configuration may put too many tuples waiting for processing and hence, the continuous states will be influenced), we reset the state of the simulation each

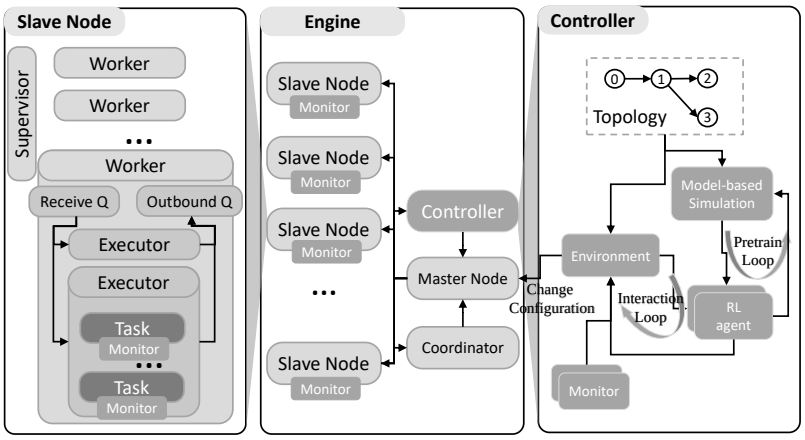


Figure 5.3: System Architecture Overview

time when we update the model with one parallelism configuration in line 20 of Algorithm 7. Therefore, for each sample of the model, the state of the operator will start from the same initial state so that each sample will not be influenced by the previous sample's state.

In the real environment, we first apply the trained parameters to each model as shown in Algorithm 8. Then, for each operator, the controller decides the parallelism from the metrics gathered from the system similar to the steps in the pre-train iteration. The heterogeneity is captured by the linear model in LinUCB. If the operator is migrated from one node to another, it only influences the processing rate distribution μ_i (when the other latencies are already appropriately optimized). We evaluate this experimentally in Section 5.4.

In the next section, the implementation is discussed for the above methods in a real-world DSP engine.

Algorithm 8: MBLinUCB

```
1  Procedure MBLinUCB(G_dag)
2      initial the model parameters for each operator i ∈ V_dag, θ_i ;
3      Θ = pretrain(G_dag) ;
4      decide initial k_0 = {k_{0,i} | i ∈ V_dag} from Θ ;
5      Submit G_dag with k_0 to stream processing engine ;
6      while G_dag not terminate do
7          gather metrics in the time slot t as s_t = {s_{t,k_i} | i ∈ Vdag} ;
8          for each operator i do
9              θ_i = updateLinUCB(θ_i, r_{s_{t,k_i}}, s_{t,k_i}, k_{t-1,i}) ;
10             k_{t,i} = selection(θ_i, s_{t,k_i}) by Equation 17 ;
11             if k_{t,i} ≠ k_{t-1,i} then
12                 Submit parallelism change k_{i,t} to stream processing engine ;
```

5.3 Implementation

We implement a prototype of the proposed method using Apache Storm (v2.0.0) [135]. Though the proposed algorithms can be implemented on other DSP engines such as Apache Flink[133], we chose Apache Storm for implementation due to its widespread use in data science applications [9] and Storm has the lowest overall latency [38] among the leading stream processing engines. We use Apache Storm to deploy the DSP application which runs on the distributed worker nodes managed by the Storm framework. In Storm, the DSP application can be represented as a DAG topology that is used to schedule and optimize the application. However, when we actually deploy the application, it has an execution plan which can be seen as an extension of the DAG topology. The execution plan replaces each operator with its tasks. A task represents an instance of an operator and is in charge of a partition of the incoming tuples of the operator. In addition, one or more tasks are grouped into executors, implemented as threads as shown in Figure 5.3. Storm can process a large amount of tuples in parallel by running multiple executors. The executors are handled by the worker process in Storm, which is a Java process acting as a container, which configures a number of parameters including the maximum heap memory that can be used. The parallelism is configured by the number of executors allocated to an operator. When the number of the executors reaches the number of tasks, the operator gets its maximum parallelism. The number of executors can be re-configured without restarting the application (but the executors need to be restarted to redistribute the tasks) by the re-balancing tool provided by Storm.

The implementation of our algorithms in Storm is straightforward. As illustrated in Figure 5.3, we implement a centralized controller of the application in python. The controller is implemented using the gym environment [26] interface which can be directly used on most of the RL libraries. The interface requires the environment to provide several functionalities, which include `step()`, `reset()`, `close()`, etc. Here, the most important interface is the `step()` interface, which takes in the action for the time step and returns a four-tuple including the observation (state), the immediate reward, the end of episode signal, and the auxiliary diagnostic information. Based on the above interface, we implement the DSP controller to control the parallelism configuration based on the action the algorithms calculated in each time step. Additionally, the controller also takes the responsibility of monitoring the status of the DSP application by capturing the metrics from the output of the application, each physical node, and each instance of the operators.

By wrapping the controller of the application, we extended the algorithm (LinUCB) in RLlib[83] to implement the proposed algorithms, which can directly use the gym environment to interact with the DSP application (shown as *interaction loop* in Figure 5.3). Therefore, when the DSP application is submitted, a controller is created and based on the algorithm chosen for configuring the parallelism, an RL agent (or multiple RL agents) is created and attached to the controller. Additionally, the pre-training also implemented the same gym environment interface, which can directly interact with the RL agent. Using our MBLinUCB method, we tune the parallelism for each operator using a specific RL agent and hence, it is possible to

Table 5.1: Default Simulation Parameter Setup

K_{max}	64	$w_{lat}, w_{que}, w_{res}$	$\frac{1}{3}$
Average input rate	100 tuples/s	$\bar{l}^{max}$	1000 ms
μ_i	10 tuples/s	Time step interval	10 s

distribute the agent to be attached with the operator to make the decision. In that way, the agent does not need to be placed together with the controller and can be distributed to anywhere near the operator to make the decision without significantly degrading performance.

5.4 Evaluation

We evaluate the proposed techniques compared to several state-of-art RL methods. We use both simulation and real testbed experiments to study the behavior of the RL algorithms when they are used in optimizing the parallelism configuration of DSP applications.

5.4.1 Experimental setup

We describe the experimental setup for the simulation environment and real test-bed environment separately.

For the simulation environment, we implement a DSP application simulator by extending the queue and load balancing environments provided in Park project [88] and make it compatible with the gym environment as discussed in Section 5.3. The default setup of the simulation is shown in Table 5.1. In the simulation experiment, we test three different datasets: (i) a synthetic Poisson distribution dataset with default arrival rate $\lambda = 100/s$, (ii) a synthetic Pareto distribution dataset with the shape parameter $\alpha = 2.0$ and the scale parameter $x_m = 50$ (so that the average input rate is also 100 tuples/s in default from the Pareto distribution), and (iii) a trace-driven dataset, which is made available by Chris Whong [149] that contains information about the activity of the New York City taxis. Each data point in the dataset corresponds to a taxi trip including the timestamp and location for both the pick-up and drop-off events. As the data is too sparse (around three hundred tuples per minute) to be used in stream processing experiments, we speed up the input rate of the dataset by sixty times, which means that the tuples arriving in one minute in the original dataset will arrive in one second in the experiment.

In the real testbed experiments, we implement the DSP application for the 2015 DEBS Grand Challenge (`http://www.debs2015.org/call-grand-challenge.html`) to calculate the most profitable areas for each time window. The dataset used is the New York City taxis mentioned above and the data rate is also sped up by sixty times. We deploy a testbed on CloudLab [45] with nodes organized in three tiers. We use the cluster with ten xl170 servers in the CloudLab cluster and simulate the three-tier architecture on an

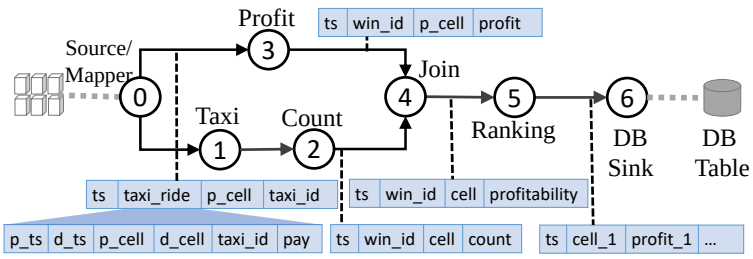


Figure 5.4: NY Taxi Profitable Area Application

Table 5.2: Default Parameter Setup for Real Testbed

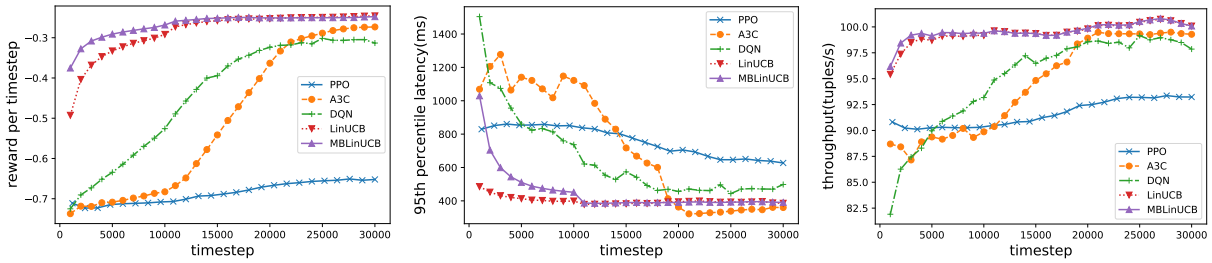
K_{max}	8	$w_{lat}, w_{que}, w_{res}$	$\frac{1}{3}$
Average input rate	4500 tuples/s	$\overline{l}^{max}$	2000 ms
Time step interval	60 s		

Openstack cluster. The third tier contains fourteen m1.medium instances (2 vCPUs and 4 GB memory) that act as the smart gateways with relatively low computing capacity corresponding to the leaf nodes of the architecture. The second tier has five m1.xlarge instances (8 vCPUs and 16 GB memory) and each of them functions as a micro datacenter. The first tier contains one m1.2xlarge instance (16 vCPUs and 32 GB memory) acting as the computing resource used in the cloud datacenter. The network bandwidth, latency and topology are configured by dividing virtual *LAN*s (local area networks) between the nodes and adding policies to the ports of each node to enforce using the *Neutron* module of OpenStack and the traffic control (tc) tool in Linux to simulate.

We deploy the Storm Nimbus service (acting as the master node) on a m1.2xlarge instance and one Storm Supervisor service (acting as the slave node) on each node respectively. For the checkpoint store, we use a single node Redis service placed on the master node. The default network is set to be 100 Mb bandwidth in capacity with 20 ms latency between the gateways and micro datacenters, and the bandwidth capacity is 100 Mb with 50 ms latency between the cloud datacenter and micro datacenters. We also place a stream generator on each smart gateway to emulate the input stream. The input stream comes to an MQTT (Message Queuing Telemetry Transport) service deployed on each smart gateway. The dataset is replicated and replayed on each smart gateway and the average input rate is around 4500 tuples/s overall. The default parameters used in the real testbed experiments are listed in Table 5.2.

5.4.2 Application and Operator Placement

To comprehensively evaluate the proposed algorithm, we choose a smart city application that ranks the profitability of the areas for taxis in New York city. As shown in Figure 5.4, there are seven operators: (i)



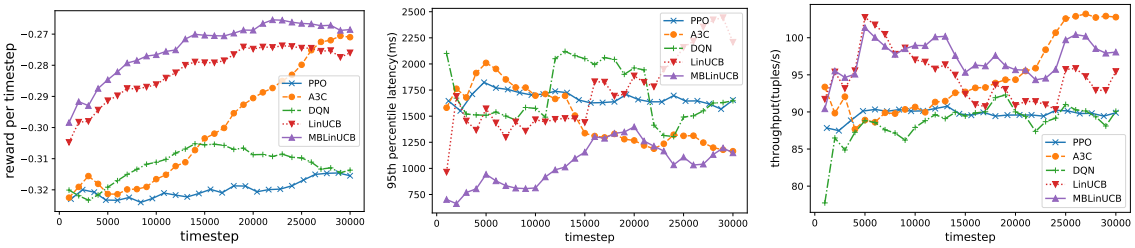
(a) Reward (b) 95th percentile Latency (c) Throughput

Figure 5.5: Results of simulation with Synthetic Dataset (Poisson distribution)

source and mapper, which consume the input stream from the MQTT service and transform the raw tuple to the data type that can be understood by the system, (ii) taxi aggregator, which aggregates the trips by the taxi IDs in time windows, (iii) taxi counter, which counts the number of taxis in a particular area in time windows, (iv) profit aggregator, which aggregates the profits by the pickup area in time windows, (v) joiner, which joins the profit and number of taxis to calculate the profitability of a particular area, (vi) ranking, which sorts the profitability of the area, (vii) sink, which stores the results of the most profitable areas into a database for further usage. We have optimized the placement of the application by placing each operator to one of the three tiers based on its selectivity. The data source which consumes the input tuples from the MQTT services are placed at the same node (one of the gateways) where the MQTT service is placed. The heavy load aggregators are placed in the micro data centers. The join, ranking and sink operators are placed in the mega datacenter. It is worth noting that because of the windowed aggregators (taxi and profit aggregators) handling most of the workloads, only those two operators are possible to be the bottlenecks in the overall stream processing application. So in the real testbed experiments, we only consider the scale up of those two operators.

5.4.3 Algorithms

In our experiment evaluation, different mechanisms are measured and compared: (i) **PPO**, which is a policy gradient method for RL[123], (ii) **A3C**, which is the asynchronous version of the actor-critic methods [98], (iii) **DQN**, which is a method based on DNN to learn the policy by Q-learning [99], (iv) **LinUCB**, which is a MAB method based on a linear model to approximate the reward distribution and it uses UCB to select the action [80], and (v) **MBLinUCB**, which is the method proposed in this work. All the methods use the default hyper-parameters configured in RLlib. For the proposed method, we generate ten thousand data points to initialize the parameters in the linear models in the MBLinUCB method as described in Section 5.2.



(a) Reward (b) 95th percentile Latency (c) Throughput

Figure 5.6: Results of simulation with Synthetic Dataset (Pareto distribution ($\alpha = 2.0$))

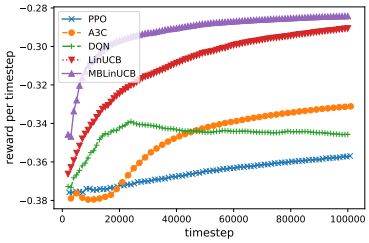
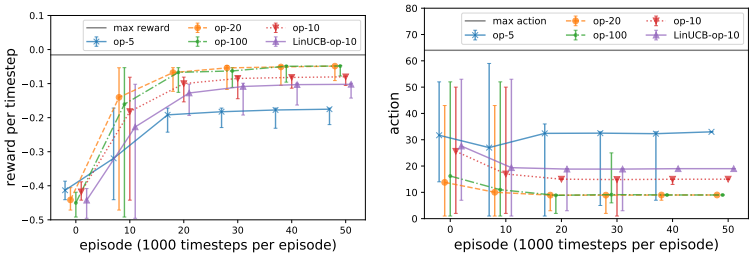


Figure 5.7: Rewards of simulation on the New York taxi trace

5.4.4 Simulation Experiment Results

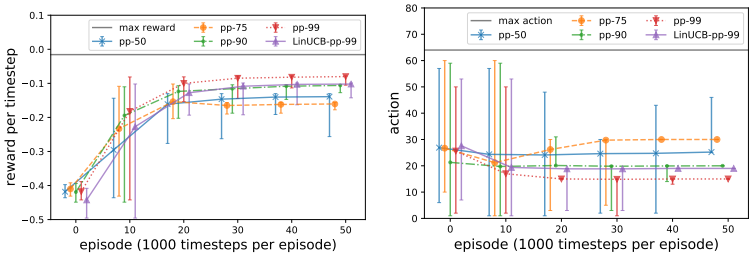
We first evaluate the performance of the algorithms in the simulation environment. As shown in Figure 5.5, we compare the algorithms with a synthetic Poisson distribution workload. We can see that our method converges faster than the other methods and it only needs three thousand time-steps to reach an average reward of -0.3. It also starts from a relatively good initial position above -0.4 compared to -0.5 in the LinUCB method. With respect to latency and throughput metrics, our method and LinUCB perform better than the others. However, the latency performance of MBLinUCB converges from one thousand milliseconds, which is much higher than the LinUCB method. As the MBLinUCB initializes its linear model with the data generated from the environment model, it starts from a configuration which just meets the upper bound latency requirement (1000ms) with the minimum parallelism needed.

As shown in Figure 5.6, we compare the algorithms with another synthetic workload from a Pareto distribution (for each time slot, the input rate is sampled from a Pareto distribution). The workload is used to test the performance of the algorithms in conditions when the workload has significant fluctuations while executing the application. In Figure 5.6, we can see similar results as in Figure 5.5. The proposed method performs better than the other methods. We note that even LinUCB does not converge to a good result as MBLinUCB does but with enough iterations (around thirty thousand timesteps), A3C can get similar results



(a) Reward (b) Action (parallelism)

Figure 5.8: Evaluation of applicability for heterogeneous resources (Poisson distribution)



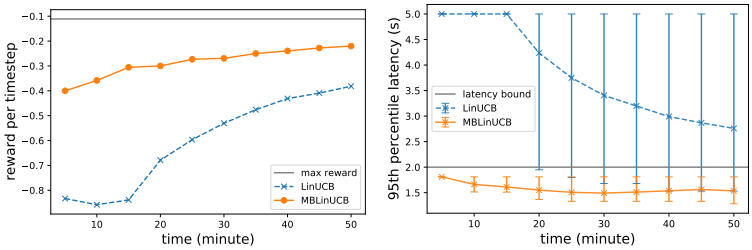
(a) Reward (b) Action (parallelism)

Figure 5.9: Evaluation of applicability for heterogeneous operators (Poisson distribution)

as MBLinUCB. This is expected as A3C uses the actor-critic method to improve the sample efficiency so that it performs better than the other RL methods that also rely on DNN.

In the next set of experiments, we study the performance of the algorithms using a real trace as shown in Figure 5.7. We can see that our method converges to an average reward greater than -0.3 in less than twenty thousand time steps. However, LinUCB needs more than sixty thousand time steps, and the other methods require even more time steps.

In the next two sets of experiments shown in Figure 5.8 and Figure 5.9, we evaluate the impact of heterogeneity in the available resources and operators. As we can see in Figure 5.8a, for different operator processing rates which may get influenced by the characteristics of the operator or the power of the underline server, MBLinUCB can converge to a good reward range within a few episodes. We can see that LinUCB converges to a lower reward when the service rate is ten compared to when MBLinUCB is used (noted as *op-10* in the figure). The above results can be explained by comparing the results in Figure 5.8b. We can see that all the conditions converge to a small range of actions at the end of 50 episodes. However, compared to MBLinUCB, LinUCB converges to a larger number of parallelism so that it has a higher resource usage penalty so as a lower reward compared to our method. As shown in Figure 5.9a, we can see similar results



(a) Reward (b) 95th percentile latency

Figure 5.10: Real Testbed Results

that within different operator parallelism portions (i.e., how many percentiles of the operator's processing can be parallelized), our mechanism can converge within a limited number of episodes. We also note that LinUCB converges to a lower reward with larger parallelism as shown in Figure 5.9b.

5.4.5 Real Testbed Experiment Results

We evaluate the performance of LinUCB and our method in the real testbed with a real stream processing application and a real dataset. As shown in Figure 5.10a, we can see that the proposed method converges faster as it has a better initial configuration. It only takes fifteen time steps (each one minute) to reach a reward more than -0.3. For the latency analysis, we illustrate the latency upper bound (95th percentile of the latency distribution) in Figure 5.10b. As shown in the results, we can see that our method starts from a latency which already meets the requirements (the latency upper bound is less than two seconds) and is stable during the experiments. However, the original LinUCB method starts from a very high latency (more than five seconds and we cut the latency metric to five seconds if it is larger than that) and then it gradually improves the average latency upper bound to three seconds. Our MBLinUCB already meets the latency bound requirements at the initial state and tries to improve it by exploring the possible parallelism configurations in the real environment.

5.4.6 Summary and discussion

In this chapter, we proposed a learning framework achieves elasticity for stream processing applications deployed at the edge by automatically tuning the application to meet the Quality of Service requirements. The method adopts a reinforcement learning (RL) method to configure the parallelism of the operators in the stream processing application. We model the elastic parallelism configuration for stream processing in edge computing as a Markov Decision Process (MDP), which is then reduced to a contextual Multi-Armed Bandit (MAB) problem. Using the Upper Confidence Bound(UCB)-based RL method, the proposed

approach significantly improves the sample efficiency and the convergence rate compared to traditional random exploration methods. In addition, the use of model-based pre-training in the proposed approach results in substantially improved performance by initializing the model with appropriate and well-tuned parameters. The proposed techniques are evaluated using realistic workloads through both simulation and real testbed experiments. The experiment results demonstrate the effectiveness of the proposed approach in terms of cumulative reward and convergence speed.

In the next chapter, we will present the resource allocation and management mechanism for geo-distributed resources.

6.0 Latency-aware resource allocation and management mechanism for geo-distributed edge resources

In the previous three chapters, we have presented three aspects of optimization for stream processing applications deployed in edge computing environments in the platform layer. As we discussed before, it is also important to manage the geo-distributed resources provided in the infrastructure layer to cooperate with the platform layer (e.g., stream processing engines) to achieve low-latency stream processing. In this chapter and the next two chapters, we focus on resource allocation and management aspects of geo-distributed edge and cloud computing resources (Figure 6.1).

In this chapter, we propose *Zenith*, a new resource allocation model for allocating computing resources in an edge computing platform that allows edge service providers to establish resource sharing contracts with edge infrastructure providers *apriori*.

Concretely, this chapter makes the following contributions: first, we propose *Zenith*, a decoupled resource allocation model that manages the allocation of computing resources distributed at the edges independent of the service provisioning management performed at the service provider end. Second based on the model, we develop an auction-based resource sharing contract establishment and allocation mechanism that ensures truthfulness and utility-maximization for both the EIPs (Edge Infrastructure Providers) and SPs (Service Providers). Third, we develop a latency-aware task scheduling mechanism that allocates the resources committed in the contracts to specific jobs in the workloads. Finally, we evaluate the proposed techniques through extensive experiments that demonstrate the effectiveness, scalability, and performance of the proposed model.

The remainder of this chapter is organized as follows. Section 6.1 provides a background of various existing edge computing solutions and motivates the proposed resource allocation model. In Section 6.2, we present the *Zenith* architecture for decoupled resource management and introduce the system model. In Section 6.3, we present the proposed resource allocation framework that comprises of the contract establishment process and the task scheduling mechanism. Section 6.4 presents the performance evaluation of *Zenith* through extensive experiments. We conclude in Section 6.5.

6.1 Background & Motivation

There has been an increasing growth in modern low-latency computing applications using wearables and IoT technologies that include (i) augmented reality applications [107], (ii) real-time traffic control systems [72] that require low-latency responses to avoid potential collisions, (iii) real-time smart grid management systems [126] that aggregate data from geo-distributed sensors and control the grid in real time. Though

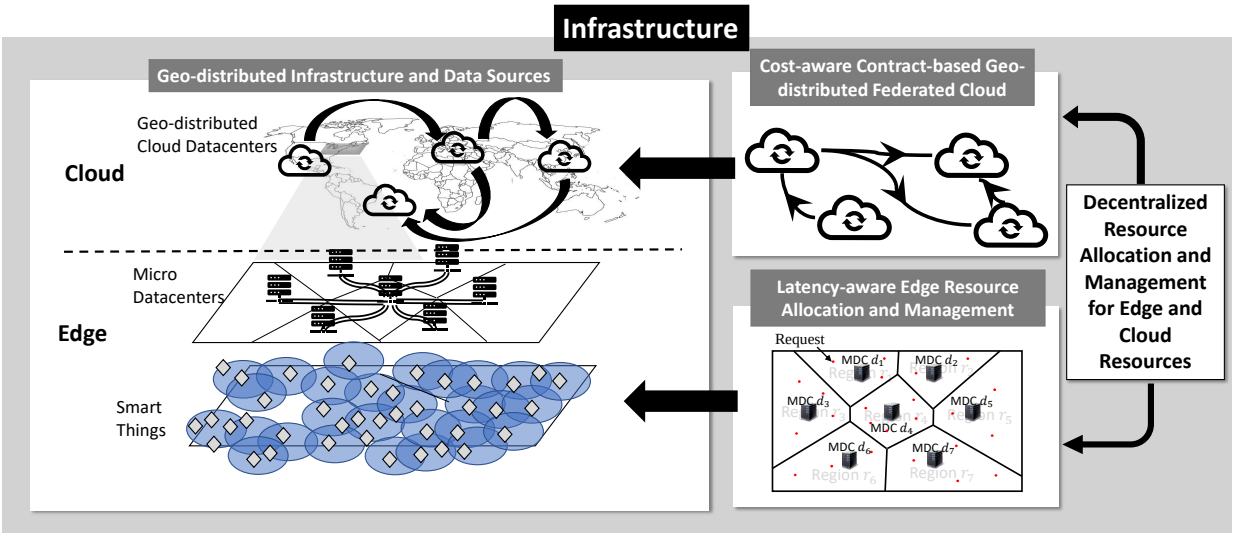


Figure 6.1: Resource allocation and management of Geo-distributed edge and cloud resources

Cloud Computing has been a very cost-effective solution[76][74] to several computing needs, clouds fail to meet low latency requirements of modern computing applications that demand strict guarantees on response times. Edge Computing [87, 18, 6, 50] complements the backend computing provided by clouds to fill the critical latency gaps between the endpoints and the Cloud.

To achieve efficient processing at the edge, smart gateways [2] and Micro DataCenters(MDCs) [55] are two key methods proposed in the literature. A smart gateway is a device which is placed at the edge of the network near the sensors. It provides a platform for the edge applications to intermediately operate the data from the endpoints to the Cloud or directly respond to the requests from the endpoint applications. A Micro Datacenter (MDC) is a data center which has a small number of resources and located close to the edge of the network to support Edge Computing services. MDCs are densely geo-distributed to provide a low and predictable latency infrastructure to the end-point applications. Compared to smart gateways, MDCs are obviously more powerful and contain more number of servers and they possess higher computing capacity than smart gateways.

In this chapter, we consider MDCs as the main source of computational resources in the edge computing platform and the goal of the MDCs is to support low latency applications at the edge, enabling them to meet stronger guarantees on response time. To effectively manage and leverage MDCs in an edge computing platform, there are several key challenges that need to be addressed. For instance, an effective resource management of MDCs should address (i) how to provision the application containers[109, 66] to serve jobs to maximize the utility of the services and (ii) how to schedule workloads on the application containers to both cover the demands and satisfy the latency constraints. While edge computing as a research area is

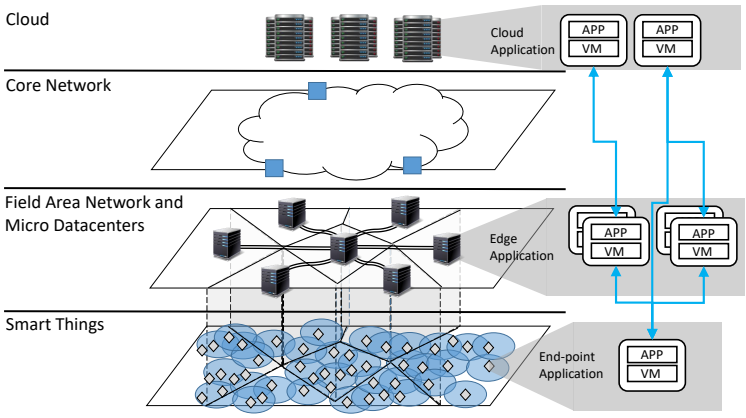


Figure 6.2: Edge Computing Architecture

emerging fast, there are a few prior efforts that discuss the above challenges[44, 1]. A fundamental assumption in these solutions includes a tight coupling of the management of the Edge Computing Infrastructures(ECIs) with that of service management by Service Providers(SPs), which means that the computational resources present at the edge MDCs are coupled and controlled directly by edge Service Providers(SPs). We argue that such a coupled model for management of Edge Computing Infrastructures (ECIs) by Service Providers (SPs) significantly limits the cost-effectiveness and the opportunities for latency-optimized provisioning of edge infrastructure resource to applications.

In contrast to existing solutions, our proposed model, *Zenith* decouples the infrastructure management from service management, enabling the ECIs to be managed by EIPs independently of the service provisioning and service management at the SPs. Such a decoupled model enables EIPs to join up to establish an Edge Computing Infrastructure Federation(ECIF) to provide resources to the Edge Computing applications provisioned and managed by the SPs. In addition, the model provides increased opportunities for resource consolidation and utilization as the geo-distributed ECIs can be jointly managed and allocated to maximize application utility and minimize cost. In the next section, we introduce the architectural details of *Zenith* and present its system model.

6.2 Zenith: System Architecture and Model

We introduce the system architecture and describe the individual components of the *Zenith* system model.

6.2.1 System Architecture

As shown in Figure 6.2, the proposed system uses a layered architecture [22]. In the bottom layer, the smart things represent the end devices (e.g. sensors, smart phones) that act as the endpoints in the Edge

Computing platform. The field area network layer is the layer where MDCs and the Edge Computing services are placed. The core network layer provides the back bone of the wide area network connecting the field area networks at the edge with the cloud's large-scale datacenters that may be located at a farther distance from the local field area network. In the resource allocation model of *Zenith*, the service management and the infrastructure management are decoupled. In other words, the service management is handled by the Service Provider(SP) that determines the provisioning decisions such as (i) where to place the containers to meet the latency requirements of the services, (ii) how many tasks in the workload are scheduled to a single container and (iii) increasing the number of containers to support the oncoming workloads. The infrastructure management is performed by the Edge Infrastructure Provider(EIP) which invests and operates the infrastructures for supporting the services placed at the edges. The EIPs are federated to set up an Edge Computing Infrastructure Federation(ECIF) which provides a resource market for the SPs wanting to deploy edge computing services at the edge. Each EIP manages several adhoc MDCs which can be densely geo-distributed. The resources of MDCs are leased or provisioned on-demand to SPs by agreeing on a contract agreement between the EIPs and SPs. The contract may include the duties and rights between the EIP managing the resource and the SP that uses the resources.

6.2.2 System Model

We next describe the system model for *Zenith* in five steps: first, we describe the features of the Service Providers(SPs) that provide Edge Computing services. Next, we represent the features of Edge Infrastructure Providers(EIPs) which provide infrastructure services for Edge Computing. We then illustrate the region division process for simplifying the resource discovery problems. After that, the agreements and the responsibilities of the coordinator are presented and finally, we discuss the role of the contract manager which is part of the *Zenith* model to manage the resource sharing contracts that are agreed between the EIPs and SPs.

6.2.2.1 Service Provider

We consider there are N SPs that require edge computing infrastructure to support their services. For simplicity, we assume that for each SP $i \in [1, N]$, it only runs one service. This model can be easily extended to one SP running multiple services with additional small changes. For each service, there is a quantifiable service demand of the SPs in each geographic region for every discrete time slot of a day.

The application container[109, 66](a configured VM integrated with the service software) has several requirements such as CPU consumption, memory size, network bandwidth and latency requirement. We use the workload demand to estimate the required number of the application containers to service the workload. Therefore, when the demand increases, the SPs begin to start adding more containers to serve the workload.

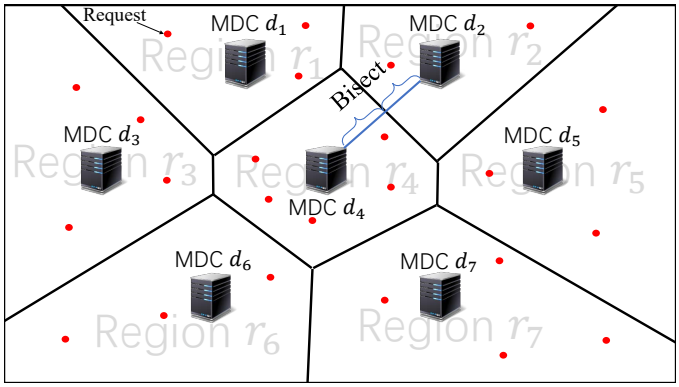


Figure 6.3: An illustration of a WVD in Zenith with seven MDCs

6.2.2.2 Edge Infrastructure Provider

Each EIP handles a large number of highly geo-distributed Micro DataCenters (MDCs) and each MDC is operated by one EIP. We assume each MDC $d \in [1, R]$ has several servers for the infrastructure service. It has a server list M_d which contains all the servers controlled by the MDC d. The capacity of one server $m \in M_d(\tau)$ is C_d^m. For simplicity, we assume that every container consumes equal resources for running the application service. The capacity for one server, C_d^m can be also represented as the number of containers which can be run on the server.

6.2.2.3 Regions Division

The problem of choosing the right MDCs to minimize the latency for every end-point and every Edge Computing service in the geographic map is intractable:

Theorem 1. The placement decision of which MDCs should host which edge computing application in order to maximize utility in terms of response time and bandwidth is NP-hard.

Proof. In order to show that the problem is NP-Hard, we first simplify the problem by assuming that each MDC can only run one container and each SP only needs to place several containers to some of the MDCs and we show that the simplified version is NP- Hard. Here, the optimization problem is to minimize the response time (latency) between the edge containers to the end-users. We can map the containers and users as the facilities and the MDCs as the locations in a Quadratic Assignment Problem (QAP)[86] which is shown to be an NP-Hard problem. As the simplified problem is a QAP problem which is intractable, the original container placement decision problem is also NP-Hard. □

For solving the problem, we use Weighted Voronoi Diagrams (WVD)[65] a technique widely used in GIS, sensor networks and wireless networks[93] for making placement decisions to maximize the utility function.

The use of Weighted Voronoi Diagram (WVD) simplifies the latency minimizing problem to a map division problem which can be solved by buiding the WVD in a polynomial time. In the example shown in Figure 6.3, the geo-location is divided into seven regions by the WVD generating algorithm with the seven MDCs as the sites in WVD. With default condition that all the sites have equal weights, the polygon of each region divides the map and all the positions in the polygon are close to the site of the region. Therefore, for each smart thing, the nearest MDC which can serve its request at the edge is located in the region where the smart thing belongs to. This method simplifies the model, especially the process of estimating the latency to the end users by dividing the map into several regions and registering an area to one region. The Voronoi Diagrams are predetermined by considering the location of the MDCs as the sites and the expected workload of the services. The weight for each MDC can be calculated as the ratio of the capacity of the MDC to the historical workload amount in the nearby area (e.g., an area within 30-mile radius). Thus, a micro datacenter which has a higher workload pressure in the adjacent area will handle a smaller region in the WVD.

We assume that the predetermined WVD divides the map into R sub-regions. Each region $r \in [1, R]$ only contains one MDC and the nearest MDC for every position in region r is the MDC $d = r$ in that region. We also assume that the expected workload distribution for each time slot τ is $\lambda_i^r(\tau)$ for SP i in the region r. The workload distribution contains all the workloads coming from the region. We use $\lambda_i^p(\tau) \in \lambda_i^r(\tau)$ to represent the workload coming from a particular position p in region r. As we primarily consider the workload which needs real-time serving, $\lambda_i^p(\tau)$ is often the upper bound of the workload during the time slot τ from position p.

6.2.2.4 Coordinator

The coordinator is a third-party service which is trusted by the EIPs and SPs in the system and it is responsible for providing a platform for the EIPs to trade resources with the SPs. There is an agreement which is committed with the coordinator before EIPs and SPs join the federation. The agreement stipulates the rights and duties of the three parties, the coordinator, the SPs and the EIPs.

6.2.2.5 Contract Manager

The contract manager is a component associated with both the SP and EIP to manage the resource sharing contracts agreed by them. Its responsibility is to manage the resource sharing contracts and observe the contracts' status. For the SP which buys resources, it must pay for the contract and has the right to observe the performance of the resource that is allocated to it. For the EIP which sells resources, it must guarantee the performance of the resources which are leased to the buyers. It collects the payments from the buyers. The contract is an agreement between the SP and the EIP to lease resources from the EIP's MDC which is effective in a particular time period with a particular constraint such as latency and availability.

6.3 Zenith: Resource Allocation

In this section, we present the proposed resource allocation techniques for EIPs and SPs to establish relationships(contracts) with each other and discuss the job scheduling technique employed in *Zenith*.

6.3.1 Contracts Establishment

The key idea behind the contract establishment process is to match the demands (e.g. workload and revenue) of the SPs and the supplies(e.g. capacity and operating cost) of the MDCs. The SPs want to maximize their utility of serving the customer with a better quality of service to potentially gain more profits. The MDCs want to maximize their utility (revenue) by renting their servers to more SPs and the SPs who can pay more.

For the sake of model simplicity, in this subsection, we only model the resource sharing problem for one MDC though the model is generic to be extended to the scenarios where there are multiple MDCs. As the WVD algorithm divides the map into several regions, the problem of finding the MDC with the lowest latency is transformed into a problem of determining which region a smart thing belongs to. In each region, every SP estimates the workload in that region based on historical workload information and statistical prediction. It then bids for the resource for running the application containers for serving the workload in that region. The bid is decided by the workload demand, $\lambda_i^r(\tau)$, the latency requirement and the estimated utility that the SP can gain from running the service in the MDC for serving the customers.

6.3.1.1 Utility of SPs

First, we model the utility of the SP to run the service on the edge. Here, we consider services having higher requirements for latency such as location-based augmented reality games[107] and intelligent traffic light control [164]. The utility of the SP can be expressed by the gain in changing the execution of the real-time service from the cloud to the edge, which we represent by the function:

$$u_i^p(\tau) = f(l_{pd}(\tau)) - f(l_{pi}(\tau)) \tag{19}$$

where $f(x)$ is a function which estimates the utility that can be obtained by providing the service with a latency x. It can be approximated by an affine utility function which translates the user-perceived criterion(latency) into utility(e.g., revenue), $f(x) = -ax + b$ where a and b are the parameters in the affine utility function and $l_{pd}(\tau)$ represents the latency between the position p where the workload comes from and the MDC d in time slot τ. Here $l_{pi}(\tau)$ represents the latency between the mega datacenter of SP i and the position p.

6.3.1.2 Utility of EIPs

For the EFIP, its objective is to earn higher revenue by providing the infrastructure to SPs. So the utility for the EFIP is obviously the profit that it can obtain by renting the resource to the SPs. For each MDC, the utility function $u_d(\tau)$ can be defined as the profit of selling the resource:

$$u_d(\tau) = \sum_{m}^{M_d} (C_d^m * \pi_d^s(\tau) - Cost_d^m(\tau)) \tag{20}$$

where $\pi_d^s(\tau)$ is the sell price in time slot τ for MDC d, $Cost_d^m(\tau)$ is the fluctuating operating cost of server m in MDC d in time slot τ.

6.3.1.3 Bidding Strategy

For SP, the bidding strategy is to bid by the true value that the SP believes for the resources, which is represented by the utility function we discussed above. The bid can be represented as $< b_i^p(\tau), \lambda_i^p(\tau) >$, where $b_i^p(\tau)$ represents the bid price for each position the workload comes from. The bid price $b_i^p(\tau)$ can be estimated by the utility of SP i in time slot τ for running the service on the edge instead of on the cloud. So the bid price for each position p can be calculated as $b_i^p(\tau) = u_i^p(\tau)$.

For MDC, the sell bid is set to the operating cost, which means if the bid wins, the MDC can at least break even the cost. The sell bid can be represented as $s_d(\tau) = Cost_d(\tau)$, where $Cost_d(\tau)$ represents the operating cost of running one application container for MDC d in time slot τ.

6.3.2 Determining Winning Bids

After designing the bidding strategy of the SPs and MDCs, we next design our algorithm for determining the winning bids as shown in Algorithm 9. The winning bids decision algorithm is based on the McAfee mechanism[90]. It guarantees *truthfulness* (Definition 2) and *budget balance* for the auction. *Truthfulness* provides a huge benefit for designing the auction which simplifies the bidding strategies for all the participants. If the auction mechanism satisfies *truthfulness*, it ensures that the strategy which bids with the true value is the dominant strategy among all the other strategies. The *budget balance* is a feature which guarantees that the auctioneer will not subsidize for the auction, which means the payment from the buyers is always more than the payment to the sellers. The decision of the auction is indicated by a set of indicators, $X^r(\tau)$. The buying bid $b_i^p(\tau)$'s indicator is set to be $x_i^p(\tau) = 1$ if bid $b_i^p(\tau)$ wins. The time complexity of the winner deciding algorithm can be shown as $O(|B(\tau)| \log |B(\tau)|)$ as the computation complexity is determined by the initial sorting of the bids, which is heavier than the computation deciding the winning bids which has the computation complexity $O(|B(\tau))$.

Next, we present the proposed contracts establishing algorithm based on the winning bids decision (Algorithm 9). The basic idea behind the resource sharing auction framework is to maximize the utility for the

Algorithm 9: Algorithm for winners selection

Input : MDC #: d;
Buy bids: $B(\tau) = \{< b_1^{p_1}(\tau), \lambda_1^{p_1}(\tau) >, < b_1^{p_2}(\tau), \lambda_1^{p_2}(\tau) >, ..., < b_2^{p_1}(\tau), \lambda_2^{p_1}(\tau) >, ...\}$;
Operating Cost: $Cost_d(\tau)$
Output: Clearing Buying Price: $\pi_d^b(\tau)$ Clearing Selling Price: $\pi_d^s(\tau)$;
Auction decision: $X^r(\tau) = \{< x_1^{p_1}(\tau) >, < x_1^{p_2}(\tau) >, ..., < x_2^{p_1}(\tau) >, ...\}$;
Sort $B(\tau)$ in descending order by the bid price per container: $\bar{b}_i^p(\tau) = b_i^p(\tau)/\lambda_i^p(\tau)$;
Initially, set current buy price b as $\bar{b}_i^r(\tau)$ as the first bid (highest price) in $B(\tau)$. number of trading containers $h = 0$, bid index $i = 1$;
while $b \geq Cost_d(\tau)$ **do**
    if $h + \lambda_i^p(\tau)$ is larger than the capacity of MDC, $\sum_m^{M_d} C_d^m$, or $i + 1$ is equal to the size of the number of buy bids: break;
    $b = b_i^p(\tau)$;
    $h+ = \lambda_i^p(\tau)$;
    $i++$;
$\rho = (b_{i+1}^p(\tau) + Cost_d(\tau))/2$;
if $b_i^p(\tau) \geq \rho \geq Cost_d(\tau)$ **then**
    All the first i buyers win with price per container:;
    $\pi_s^d(\tau) = \pi_b^d(\tau) = \rho$;
else
    All the first $i - 1$ buyers win with buy price per container: $\pi_b^d(\tau) = b_i^d(\tau)$;
    The sell price per container is $\pi_s^d(\tau) = Cost_d(\tau)$;

MDCs as well as for the EIPs and the SPs in a fair manner. As discussed earlier, the objective of SPs is to increase their profit by maximizing the utility of serving the customer with better service. The EIPs want to provide more edge computing resources to the SPs to increase the revenue. The auction process for a given slot τ can be presented as three steps. In Step 1, The SPs estimate the workload for each position in every region r to get the estimated workload, $\lambda_i^p(\tau)$. They send buy bids, $< b_i^p(\tau), \lambda_i^p(\tau) >$, where the bid price $b_i^p(\tau)$ is estimated from the utility it can gain from providing the service on the edge in region r, to the coordinator. In Step 2, the coordinator decides the winning bids using the Algorithm 9 for every region r. Each winner establishes a resource sharing contract with the owner of the MDC d. Finally in Step 3, if the auction is cleared without the MDC d selling all the resources, the MDC d will attend the next round of the auction. In the next round of the auction, if SP i has the workload that is not satisfied, it sends the buy bids $< b_i^p(\tau), \lambda_i^p(\tau) >$ where $p \in r$ to randomly choose an adjacent region $r\prime$ of region r. For the MDCs which have remaining available resources, the process operates the next round of auction from *Step 1* until all the buy bids are satisfied or all the resources of MDCs are allocated.

The above sequence of steps is operated for each time slot $\tau = 1$ to $\tau = T$ where T is the maximum time slot for consideration. The result of the auction establishes the utility-maximizing contracts between the SPs and MDCs for the effective time slot from $\tau = 1$ to $\tau = T$. After the previous steps, each SP has a set of contracts established with the EIPs. The contracts are denoted by $Contract_i^r(\tau) = \{Contract_i^{d_1}(\tau), Contract_i^{d_2}(\tau), ...\}$ for serving the workload $\lambda_i^p(\tau) \in \lambda_i^r(\tau)$, where $d_1, d_2, ...$ are the index of the MDCs, $Contract_i^{d_1}(\tau) =< \pi_b^d(\tau), \pi_s^d(\tau), C_i^d(\tau) >$ is the contract which is established by SP i with MDC

d for effective in time slot τ, where $\pi_b^d(\tau)$ is the clear buying price for the contract, $\pi_b^d(\tau)$ is the clear selling price for the contract and $C_i^d(\tau)$ is the capacity of the resources in the contract.

6.3.3 Provisioning

The contracts establishment algorithm solves the problem of allocating the resources for each SPs in a utility-maximizing manner. After the establishment of the contracts, the SPs hold resources which are densely geo-distributed at the edge of the network. The provisioning process is required for each SP to schedule the tasks to the containers on the MDCs to handle the requests of its services at the edge. The provisioning algorithm here decides how to place tasks on the containers in the MDCs of the established contracts in a contracts-aware manner.

In the provisioning process, the SP needs to react for the workload changes in a real-time manner. A reactive provisioning and task scheduling algorithm is needed to place the tasks on the application containers and decide the placement of the application containers to the right MDCs which can both optimize the cost of using the resource and the service performance that can potentially increase the application experience to the user. Our task scheduling algorithm aims at minimizing the network latency between the end nodes and the MDCs where the application container is hosted.

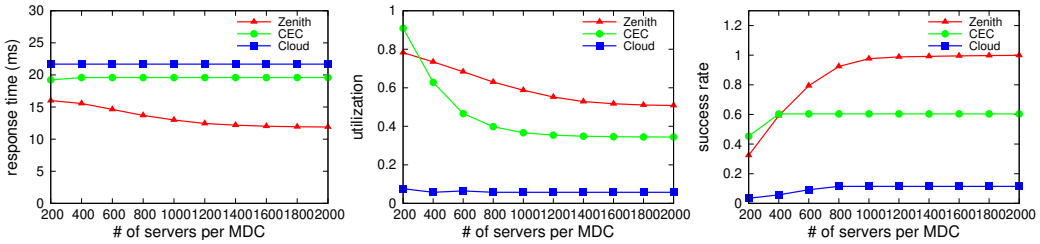
6.4 Evaluation

We experimentally evaluate the effectiveness of *Zenith* in terms of job response times, success rate of meeting response time guarantees and resource utilization levels at the edge micro datacenters.

6.4.1 Setup

The simulation uses a geographic map of 3000 miles *3000 miles size and randomly chooses locations from the map to place the MDCs. The WVD (Weighted Voronoi Diagram) is generated from the map with the locations of the MDCs as the sites. The latency used in the experiments is estimated using the distance-based model presented in [54]. This linear model estimates the latency based on the distance between the two points.

We consider that each region contains one micro data center and each MDC has 1000 servers in the default setting. The server has the same performance as that of the IBM server x3550 (2 x [Xeon X5675 3067 MHz, 6 cores], 16GB). Each server hosts up to 5 application containers at a given time. The location of the MDC is randomly chosen, and the timezone of the MDC is determined by the location. The geographic map area is divided into four time zones evenly to simulate the time-varying aspect of the dynamic electricity pricing. The electricity price is generated based on the hourly real-time electricity price from [139]. We use



(a) Response time (b) Utilization (c) Success Rate

Figure 6.4: Impact of number of Servers per MDC

the distribution of the data in 2015 from NationalGrid's hourly electricity price to simulate the fluctuation of the real electricity market.

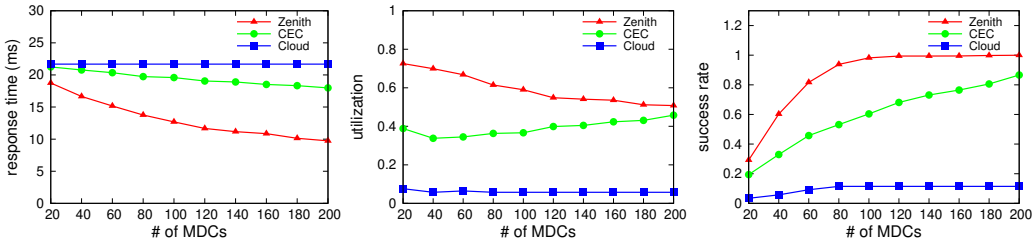
The default workload generates job requests of low-latency data processing tasks (of 100 bytes in size) and the container running at the MDC processes the request. The distribution of the workload is uniform throughout the map for the default setting. We consider that all workloads are response time sensitive, which means that if the response time exceeds the constraint, the task is considered to be a failed task. The default response time constraint is 30ms. We compare *Zenith* with two candidate mechanisms: (i) Coupled Edge Computing (CEC) mechanism, in which each MDC is owned by one of the SPs and the workload to the MDCs comes only from the SP that owns the MDC; (ii) conventional cloud-based (Cloud) solution in which the workload is processed at the large-scale datacenters placed on the left and right ends of the map.

6.4.2 Experiment Results

To evaluate the performance efficiency of *Zenith*, we perform three sets of experiments: first, we study the impact of the number of servers in MDCs on the average response time of tasks, the average utilization at the MDCs and the overall success rate of the tasks. Second, we study the impact of the number of MDCs in the geographic map. Finally, we analyze the impact of different response time constraints on the perceived performance efficiency.

6.4.2.1 Impact of No. of servers in MDCs

In this experiment, we compare the performance of the mechanisms with different number of servers in each MDC. The number of servers per MDC is increased from 200 to 2000 in the evaluation. For the Cloud-based mechanism, the total number of servers present in the two large-scale datacenters are increased in such a way that they have the same number of total servers as the total number of servers in all MDCs. As shown in Figure 6.4a, the y-axis is the average response time of the tasks. The x-axis is the number of servers per MDC. We find that with the increased ability to share resources among MDCs, *Zenith* achieves



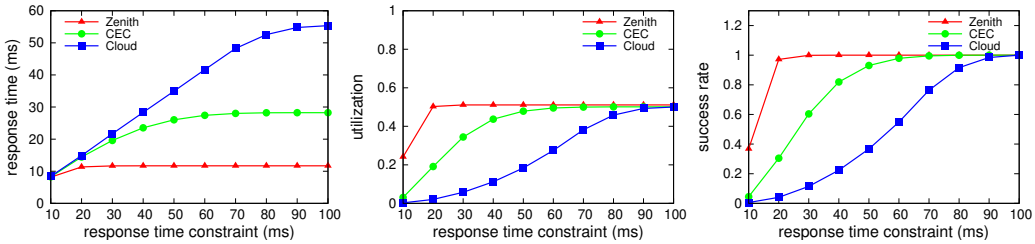
(a) Response time (b) Utilization (c) Success Rate

Figure 6.5: Impact of number of MDCs

the best result compared to CEC and Cloud mechanisms even when the number of servers is low. As shown in Figure 6.4b, *Zenith*, in general, can achieve higher utilization of the MDCs except when the resources are scarce such as when one MDC only handles 200 servers. In Figure 6.4c, we observe that the success rate of the tasks increases with increasing the number of servers. In addition, the success rates of CEC and Cloud schemes do not reach 100%. This is due to the fact that even when the resources are available, these schemes suffer from reduced proximity between the tasks and the MDCs assigned to them. Here, the response time constraints cannot be met with the nearest datacenters. From the above experiments, we can see that *Zenith* performs significantly better than CEC and Cloud mechanisms with respect to response time, resource utilization and success rate.

6.4.2.2 Impact of No. of MDCs

We next study the performance of *Zenith* with different number of MDCs present in the geographic map. The number of MDCs is increased from 20 to 200. For the CEC mechanism, the MDCs are divided into 10 even groups such that each group is owned by one SP. As shown in Figure 6.5a, the x-axis is the number of MDCs. We find that the response time decreases from 20ms to about 10ms for *Zenith* as increasing the number of MDCs decrease the average distance between the endpoints and the MDCs which results in a decrease in response time. For the CEC mechanism also, the response time decreases from around 21ms to about 18ms. Here, we note that the Cloud-based mechanism is not influenced by the number of MDCs as the placement of the large-scale datacenters are fixed. In Figure 6.5b, we compare the resource utilization levels at the MDCs and we find that *Zenith* achieves higher utilization than CEC and Cloud mechanisms. As shown in Figure 6.5c, the success rate increases with increasing the number of MDCs. Here, *Zenith* performs significantly better than the two other mechanisms.



(a) Response time (b) Utilization (c) Success Rate

Figure 6.6: Impact of latency constraints

6.4.2.3 Impact of Response Time Constraints

Finally, we study the performance of *Zenith* to analyze the impact of different response time constraints. In this experiment, we increase the mean response constraint from 10ms to 100ms. As shown in Figure 6.6a, the x-axis is the response time constraint. The obtained response time increases for all the three mechanisms as increasing the response time constraint provides additional flexibility for task scheduling to satisfy more workloads with longer distance which results in an increase in the obtained response time. In Figure 6.6b, the utilization of all the three mechanisms are compared and we find that it increases with an easier response time constraint. For the Cloud mechanism, when the response time constraint increases significantly, it also achieves similar performance as *Zenith* and CEC. This is due to the fact that when the response time constraint is relaxed, a cloud solution allows the tasks to be transferred and executed in remotely located large-scale datacenters leading to a higher success rate. Figure 6.6c shows that the success rate increases with extending the limitations of the response time constraints. Here the Cloud solution also attains 100% success rate as *Zenith* and CEC. From the above experiments, we observe that *Zenith* performs significantly better than CEC and Cloud when the response time constraints are significant.

6.5 Summary and discussion

In this chapter, we propose *Zenith*, a resource allocation model for allocating computing resources in an edge computing platform. In contrast to conventional solutions, *Zenith* employs a new decoupled architecture in which the infrastructure management at the Edge Computing Infrastructures (ECIs) is performed independent of the service provisioning and service management performed by the service providers (SPs). Based on the proposed model, we present an auction-based mechanism for resource contract establishment and a latency-aware scheduling technique that maximizes the utility for both EIPs and SPs. The proposed techniques are evaluated through extensive experiments that demonstrate the effectiveness, scalability and performance efficiency of the proposed model.

In this chapter, we addressed the problem of allocating and managing edge computing resources for edge infrastructure providers to enable them provide resources to the service providers in a latency-aware manner. Similar to edge computing environments, geo-distributed clouds can also benefit from resource sharing among providers to increase utility. In the next chapter, we propose a resource allocation and management mechanism for geo-distributed clouds that allows cloud service providers to trade computing resources in geo-distributed datacenters in a cost-aware manner.

7.0 Cost-aware resource allocation and management mechanism for geo-distributed cloud resources

In this chapter, we propose a contracts-based resource sharing model for federated geo-distributed clouds that allows Cloud Service Providers (CSPs) to establish resource sharing contracts with individual datacenters *apriori* for defined time intervals during a 24 hour time period. Concretely, this chapter makes the following contributions: first, we develop the proposed contracts-based resource sharing model and present an optimal contract establishment algorithm that produces the optimal design of resource sharing contracts considering the size and type of resources in each resource sharing contract. Second, we develop an auction-based contract allocation mechanism that ensures both fairness and revenue maximization for the individual datacenter providers. Third, we develop a suite of job scheduling and contracts-based resource provisioning algorithms that leverage the established contracts for each CSP and minimizes the resource usage cost of individual CSPs. We evaluate the proposed techniques through extensive experiments using realistic workloads generated using the SHARCNET cluster trace. The experiments demonstrate the effectiveness, scalability and resource sharing fairness of the proposed model.

The remainder of this chapter is organized as follows. Section 7.1 provides a background of various resource sharing models for geo-distributed clouds and motivates the proposed contracts-based model. In Section 7.2, we develop the proposed contracts-based resource allocation system model. In Section 7.3, we present new techniques for optimal contracts designing and allocation. In Section 7.4, we present our proposed contracts-based job scheduling techniques. Section 7.5 evaluates the performance of the contracts-based resource allocation mechanisms in comparison with conventional complete cooperation geo-distributed clouds using real-world datacenter workload traces. We conclude in Section 7.6.

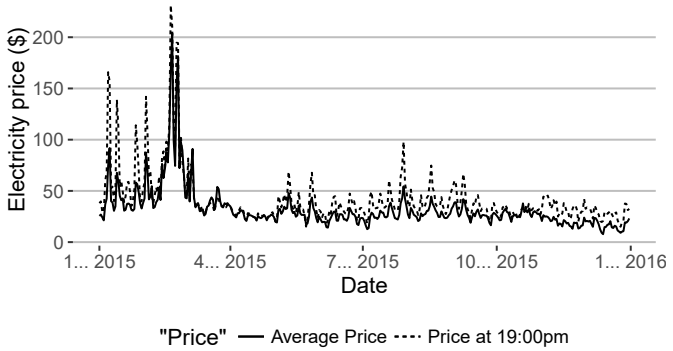
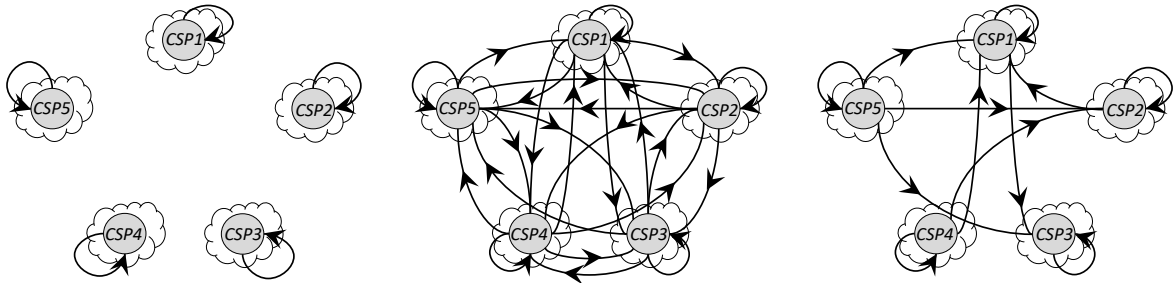


Figure 7.1: Electricity price trends of NationalGrid in 2015



(a) No resource sharing between the CSPs

(b) Completely cooperating resource sharing

(c) Contracts-based resource sharing

Figure 7.2: Resource sharing mechanisms comparison

7.1 Background & Motivation

In this section, we briefly review the background concepts related to various models of operating a geo-distributed cloud and discuss their merits and demerits.

7.1.1 Stand-alone Clouds

Conventional cloud computing models (e.g., Amazon EC2[12] and Google Cloud[137]) use a single datacenter or a set of datacenters jointly managed by a single CSP. Thus, the CSPs do not cooperate with each other and do not aim at optimizing resource allocation and cost across multiple CSPs (Figure 7.2a). Despite resulting in sub-optimal resource allocation and resource management, this centralized single-site resource management model has the benefit of easier resource management as each datacenter is managed independently of each other, providing higher autonomy and control for individual datacenters. Even though this "stand-alone" datacenter management may result in locally optimized resource management at individual datacenters, such an approach can be largely sub-optimal with respect to global resource management considering all datacenter resources jointly in a federated geo-distributed cloud environment. As an example, Figure 7.1 shows the dynamic electricity pricing from the NationalGrid[139] data in 2015. We observe that besides the notable long-term (e.g., one year) fluctuations, there are significant short-term price variations even on a single day: the highest per-day pricing on one given day can be as much as six times the lowest price observed on the same day. Thus, "stand-alone" clouds that have neither complete nor partial co-operation with each other can operate very sub-optimally forcing individual datacenters to run workloads locally at higher electricity prices even though resources for which may be available at remote datacenters at a possibly lower electricity cost.

7.1.2 Federated Clouds with Complete Cooperation

In the literature, several techniques for global management geo-distributed datacenters have been proposed. These mechanisms can be classified into two broad categories:

Virtual Geo-distributed Clusters: This class of techniques builds Virtual Machines (VMs) for users to use computing resources across the geo-distributed datacenters as a single virtual cluster. There are several works focusing on optimizing the performance in the geo-distributed environment [3, 114, 154, 163, 117, 57, 169]. Here, the datacenters are treated as one single virtual entity and having a single centralized cloud manager makes it easier to schedule the jobs and place data to achieve the overall goal. The cloud manager obtains the global information of the jobs and the individual workload requirements of each datacenter to balance the load and schedule the jobs.

Federated Cloud: Federated clouds provide a platform for the CSPs to share computing resources with each other. Each CSP is assumed to manage its datacenters autonomously. There is a centralized Cloud Exchange Institution that obtains all infrastructure information from the datacenters and provides the platform for the CSPs to discover the resources from the members of the federated cloud [35] [27]. The key objective for the CSPs is to share their resources on the federated cloud platform to maximize their resource utilization and increase the success rate of meeting the SLAs for the jobs.

We illustrate these two types of global resource management mechanisms in Figure 7.2b and we refer to this model as federated clouds with complete cooperation. This model enables the free use of the resources through a centralized broker such that all the resources in the geo-distributed datacenters can be used by all the other members participating in the system. However, this model suffers from a few key drawbacks, which include (i) lack of fairness in revenue earned by competing CSPs, i.e., since the global resource optimization objective of this approach does not lead to locally optimized profits for individual datacenters, the individual profit of each datacenter may be even lower than the profits they can get by operating standalone and (ii) limited scalability - as it is difficult for all the geo-distributed datacenters to globally synchronize the information necessary for sharing, provisioning and allocating resources in a real-time manner for job scheduling can be a significant challenge.

7.1.3 Contracts-based Resource Sharing

In this chapter, we propose a new contracts-based resource sharing architecture for the CSPs to share resources across globally geo-distributed datacenters. The demerits of the complete cooperation model lead us to a more flexible and limited sharing mechanism that provides a controlled cost-aware resource sharing opportunity. Thus, the contracts-based resource sharing model finds suitable tradeoffs between traditional clouds without federation and that with complete cooperation as illustrated in Figure 7.2. The figure shows three different architectures for a geo-distributed cloud of five CSPs. Here, the edges represent the usage of resources among the CSPs. As illustrated in Figure 7.2a, none of the five CSPs can use others' resources

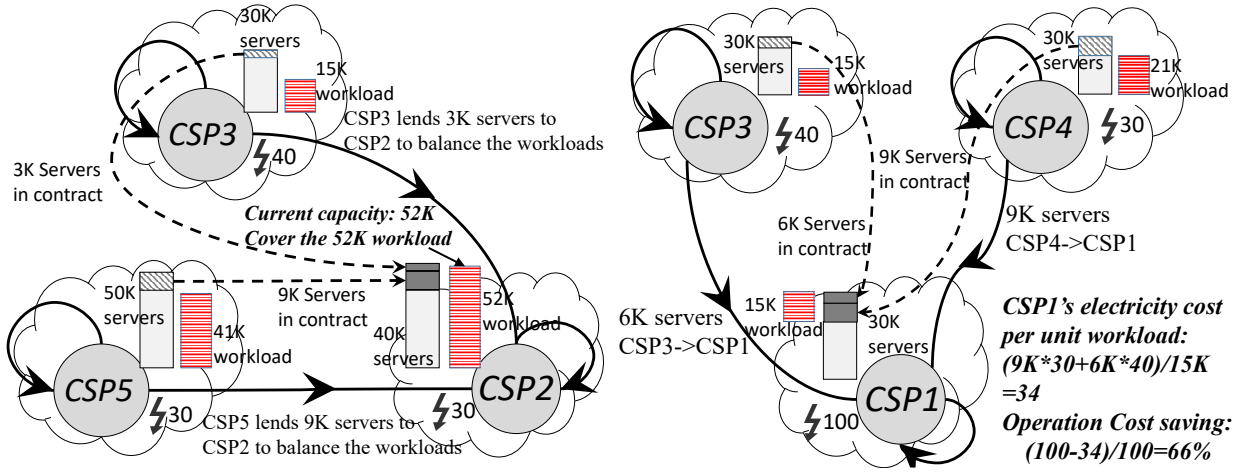


Figure 7.3: Contracts-based cloud federation example of saving electricity cost and balance the workload

when there is no federation. However, in Figure 7.2b, we find that there is a complete graph showing that every CSP can use every other's resources with complete co-operation. In Figure 7.2c, there are only six edges between the CSPs representing a partial graph. Here, each CSP does not share resources with every other CSP in the federation. Each edge represents a contract between the CSPs to share resources.

The proposed contracts-based resource sharing mechanism is based on resource sharing contracts that are established between the CSPs after negotiations. The resource sharing contracts could be signed by the CSPs stipulating the rights and duties of the CSPs to share the committed resources during the time duration and the negotiated price in the contract. For the contracts-based resource sharing mechanism, the CSPs design and trade the resource sharing contracts with each other. Thus, the contract may be predetermined and established apriori before the effective time. The establishment of contracts involves two key challenges namely (i) how to design and build contracts that can maximize individual profit of the CSP and (ii) how to schedule jobs to maximize the utility of using the contracts.

Figure 7.3 and Table 7.1 present an example scenario to illustrate the key benefits of using a contracts-based resource sharing mechanism namely (i) balancing the workload, (ii) minimizing the operating cost and (iii) increased resource utilization:

Balancing the workload: In the example shown in Figure 7.3a, CSP2 has overcapacity workload and needs 52K servers to meet the workload requirements. However, it has only 40K servers. Therefore, under normal operations, it has to either delay some jobs in the workload or drop them entirely. Alternately, in the contracts-based federated cloud model, CSP2 borrows 9K servers from CSP5 and 3K servers from CSP3 to meet workload requirements of 52K servers. As we can see, this not only increases the revenue for CSP2 but also for the other CSPs participating in the contracts-based resource sharing.

Table 7.1: The status of the five providers in the contracts-based example

	CSP1	CSP2	CSP3	CSP4	CSP5
Electricity Price	100	30	40	30	30
# of Servers	30,000	40,000	30,000	30,000	50,000
Require # of Servers	15,000	52,000	15,000	21,000	41,000

Minimizing operating cost: In Figure 7.3b, CSP1 experiences an increased electricity cost requiring to spend $100 per megawatt per hour. Even though it has the similar workload amount as CSP4 and CSP3, it uses contractual relationships to borrow 9K servers from CSP4 and 6K servers from CSP3 respectively. This minimizes the operating cost and saves up to 66% in operating cost for CSP1.

Increased resource utilization: Figure 7.3 also illustrates that some CSPs that have idle resources share their resources with other CSPs (e.g., CSP 3,4,5). Thus, contracts-based resource sharing results in an increased utilization of the computing resources in the datacenter infrastructures.

As discussed above, we find that contracts-based resource sharing has additional potential and flexibility to achieve a more efficient resource allocation while increasing the profit and minimizing the cost for each individual datacenter. In this chapter, we model the problem formally, analyze and develop algorithms for contract establishment and job scheduling to efficiently and profitably share resources between CSPs.

7.2 System Model

In this section, we describe the system model for the proposed contracts-based federated geo-distributed cloud model. We discuss it in three steps: first, we describe the features of the CSPs that participate in the cloud federation process. We then discuss the agreements and the responsibilities of the federation coordinator and finally, we discuss the role of the contract manager that manages the resource sharing contracts agreed between the CSPs.

7.2.1 Cloud Service Provider

CSPs offer a variety of cloud computing services to the customers. We primarily consider CSPs offering Infrastructure as a Service (IaaS) [94] that provide customers with various computing resources such as VMs and virtual disk space to store and process their data. The providers may offer different types of VMs with different Quality of Service (QoS) guarantees and the VMs may be priced differently. The QoS provided by the VMs may depend on how many CPU cores are present in the VMs, memory, network and other resources that are guaranteed in the period of time when the resources are provided to the user. The price is set by the CSP which provides the service based on the QoS provided by the VM type, the Service Level Agreement (SLA) and the market demand and supply.

The provider charges the customers *on-demand* based on the length of the running time and the price of the VM type. The profit of the CSP is determined by the charges provided by the customers, the operating cost and the penalty for violating the SLA. The operating cost which varies with the time includes the electricity cost, management cost and cost for maintenance. The penalty is paid by the CSP to the customers to compensate their loss in case of violating the SLA. For example, in Amazon Elastic Compute Cloud(Amazon EC2)[12], the SLA stipulates that if the monthly uptime of the service is less than 99.95% and greater than 99.0%, Amazon EC2 will pay 10% of the charge of using the service back to the users' account and 30% if the uptime is fallen to less than 99.0% [12].

Every CSP has limited resources to serve the users.To handle the overcapacity workload that cannot be serviced within the CSP's own datacenter, the CSPs can engage in a federation process to share idle resources and handle overcapacity requests. The negotiating steps are done by a trusted third party which we refer to as the Federation Coordinator.

7.2.2 Federation Coordinator

The federation coordinator is a third-party service which is trusted by the CSPs in the federated cloud and it is responsible for providing a platform for the CSPs to trade computing resources with each other. There is an agreement signed with the coordinator before the CSP joins the federation. The agreement stipulates the rights and duties of the coordinator and the CSP. The coordinator follows the optimized contract establishment process proposed and discussed in Section 7.3 to establish the contracts between the CSPs.

When building the contracts, each CSP sends its demand and supply to the coordinator to compare with the demands and supplies from others. The demand and supply information may have private information of the CSPs and hence the agreement also stipulates the privacy policy for the coordinator which determines to which degree the coordinator can publish or share the information submitted by the CSPs. All the CSPs are autonomous and have their own customers. Each CSP has its own utility (which can be estimated approximately by the profit) and each CSP wants to increase the utility after participating in the federation. Under this condition, the problem contains both the cooperating and competing aspects with multiple participants which cannot be solved by the methods that assume that all resource allocation decisions are handled with a central objective of global resource optimization. So in this scenario, auction mechanisms that are widely studied in Game Theory are most suitable. Auctions allow the participants to both cooperate and compete[30]. The essence of the auction is to match the supply and demand at both sides which fit the characters of the problem intuitively. The coordinator uses an auction-based mechanism to match the demands and supplies of the CSPs. The auction ends with a set of results which contains the winning decisions and the market clearance prices. Based on the results, the coordinator establishes the contracts.

7.2.3 Contract Manager

The contract manager manages the resource sharing contracts agreed by the CSPs. The contract is an agreement between the CSPs which stipulates the rights and duties of both sides, the buyer and the seller in the contract.

An actual resource sharing contract contains the following four information: (i) the buyer and the seller of the resource in the contract, (ii) effective time of the contract which controls the starting and ending time of the contract, (iii) the resource type and quantity in the contract and (iv) agreed resource price. In our work, we use dedicated resources as the unit of trading in the contracts. A dedicated resource is a collection of servers which is hardware isolated from the other resources in the datacenter. Examples of such resources include IBM Bluemix Dedicated Cloud [64] and Amazon EC2 Dedicated Instances[11]. The pricing model used for the contracts consists of two components namely (i) the reservation price that is paid when the two sides establish the contract and (ii) the usage price which is paid when the resources in the contract are actually used. The usage price is paid according to the usage amount and time of the resource. The contract can also include the SLA which provides more guarantees for the performance of the resource included in the contract. Other constraints that can be included in the contract include the location constraints of placing the jobs, the business policies and security requirements.

In the next section, we present the proposed mechanisms for establishing resource sharing contracts.

7.3 Resource Sharing Contracts Establishment

We design an auction-based mechanism for establishing resource sharing contracts as the nature of the contract establishment problems naturally fits the auction mechanism. Some key features such as *truthfulness*(Definition 2) and *budget balance* of the auctioning protocol are highly desirable and essential for solving the contract establishment problem. *Truthfulness* or "strategy-proofness" is a feature provided by many auction mechanisms such as VCG mechanism[106] and McAfee mechanism[90] and it ensures that the participants of the auction can maximize its utility only by bidding with the true value which he/she values the goods in the auction. This feature simplifies the problem by narrowing down the choices of the participants. The *budget balance* feature guarantees that the payment from the buyers is equal to or more than the payment to the sellers. This feature guarantees that the coordinator will not subsidize in the auction.

As discussed above, we choose to design the proposed auction mechanism based on McAfee mechanism as it inherently guarantees both *truthfulness* and *budget-balance*. In the McAfee mechanism, the selling price and the buying price are determined separately which helps to keep the auction truthful and budget-balanced. Even though having separate selling and buying prices makes the trade efficiency sub-optimal, it is necessary to design a truthful auction mechanism. As the uniqueness-of-prices theorem [105] implies, this subsidy problem (the auctioneer need to subsidize the auction) is inevitable - any truthful mechanism that optimizes

the social welfare will have the same prices (up to a function independent of the bid prices of each bidder). If we want to keep the mechanism truthful while not having to subsidize the trade, we must compromise on efficiency and implement a less-than-optimal social welfare function [150]. The McAfee mechanism is designed based on the above theorem which has a bounded trade efficiency loss, $1/\min(|B|, |S|)$ where B is the set of buy bids and S is the set of sell bids, but maintains both *truthfulness* and *budget-balance*.

We design the framework of the contracts establishment process in three parts: first we model the problem of establishing the resource sharing contracts into an auction; second we develop the strategy for the CSP to bid in the auction; third we design the auction algorithm which determines winning bids and the market clearance prices. Finally, we design the iterative process of building the contracts, which is based on the proposed auction algorithm.

7.3.1 Problem Description

We first model the contracts establishment problem as a sealed-bid double auction problem. In a sealed-bid double auction[106], there are three kinds of participants: first are the buyers who have the demands for the goods; second are the sellers that can supply the goods; the third is the auctioneer which is responsible for conducting the auction. In the contracts establishment problem, the CSPs can act as both buyers and sellers based on their demands and supplies. The coordinator of the federated cloud, a trusted third party acts as the auctioneer. The traded goods in the auction are the rights to use a certain amount of cloud resource in a certain period of time (time slot). We use dedicated resource types [11] [64] to represent a cloud resource. The dedicated resources can be considered as bundles of servers isolated from other resources in the data center. It is defined by $k \in \{1, 2, ..., K\}$. Each type-k resource may contain several servers which can be represented by a list D_k and each server $d \in D_k$ has a capacity V_k^d and the overall capacity of a type-k resource is $V_k = \sum_{d \in D_k} V_k^d$. We note that the resource types are sorted by the resource capacity which means that if $k_1 > k_2$, $V_{k_1} > V_{k_2}$.

We consider a federated cloud with N individual CSPs. The contracts establishing problem is formulated using discrete time slots τ. The cloud datacenters are located in geo-distributed locations and each of them is controlled by one CSP. We assume each CSP $i \in [1, N]$ has only one datacenter for simplicity. We assume that each datacenter has several types of servers. It has a server list $M_i(\tau)$ which contains all the servers controlled by the CSP i in time slot τ. The server list can be modified in each time slot τ by adding or removing the servers which are controlled by the cloud manager of the CSP. These operations simplify the representation of the resource which is changed every time slot with different contracts signed in each time slot. The capacity of each server $m \in M_i(\tau)$ is C_i^m. Therefore, the capacity of CSP i in time slot τ can be represented by $C_i(\tau) = \sum_{m \in M_i(\tau)} C_i^m$. Each CSP serves its customers by providing resources for running their computationally-intensive jobs. The job requests are sent to the CSP, which are pushed into a job queue. The jobs in the queue are processed according to a FCFS (First Come First Served) service policy. We assume that the demand for each time slot τ is $\lambda_i(\tau)$ for CSP i which can be determined by predicting

the upcoming workloads through mechanisms such as ARIMA [29] or Hidden Markov Modeling (HMM) [71]. The profit earned by the CSPs is determined by the difference between payments from the users and the operating cost and the penalty. The payments are related to the demand $\lambda_i(\tau)$ and we use ϱ_i to denote the unit price for one unit resource (for example, one VM with one EC2 Compute Unit (ECU) and one-gigabyte memory in Amazon EC2[136] can be a unit resource). Therefore, the capacity for each server C_i^m and V_k^d can also represent the number of unit resources that can be run on the server. We use $Cost_i^k(\tau)$ to denote the operating cost of CSP i for the type-k dedicated resource.

The problem of establishing the contracts in each time slot τ for each type-k dedicated resource can be represented as a double auction in which each CSP decides the sell bid (ask prices), $s_i^k(\tau)$, for each type-k dedicated resource in time slot τ and buy bid, $b_i^k(\tau)$, based on the valuation of the resources and the expected utility. For simplicity, in the rest of the chapter, we use the term sell bid to indicate the minimal price that the sellers expect in order to sell their resources. The coordinator base on the bids to decide the pairs of winning bids. Here $X_b(\tau) = \{x_{b_1}(\tau), x_{b_2}(\tau), ...\}^k$ (if $x_{b_i} = 1$ means b_i wins and vice versa) denotes the buy bids result and $X_s(\tau) = \{x_{s_1}(\tau), x_{s_2}(\tau), ...\}^k$ denotes the sell bids result. The resource sharing contracts establishment problem is solved by the auction. The resource sharing contract represents the following information: (i) the effective time of the contract determined by τ, (ii) the two sides of the contract determined by matching the bidders of the winning bids. We use index i to denote the seller's index and j to denote the buyer's index, (iii) the selling and buying price of the contract represented by $\pi_s^k(\tau)$ and $\pi_b^k(\tau)$, (iv) the resource type and quantity represented by k and D_k. The contract is denoted by $Contr_{ij}^k(\tau) =< \pi_s^k(\tau), \pi_b^k(\tau), D_k >$. The optimal solution for establishing the contracts maximize every CSP's utility after attending the auction. We first propose the suggested bidding strategy and we discuss the utility function of the CSPs participating in the auction.

7.3.2 Proposed Bidding Strategy

As discussed previously, in our model, each provider participates in the federated cloud to potentially increase its profit. We design the bidding strategy for the CSPs to ensure that the CSPs increase their profits through their participation in the federated cloud. Before we design the bidding strategy for the provider, we first discuss the utility function of the providers. The utility of participating in the auction is defined based on the profit a CSP can gain from running the jobs on the resources in the contracts and the profit it can gain from selling its local resources to other CSPs in the contract.

First, we define the utility function using the profit a provider i can get from renting type-k dedicated resources from others:

$$u_i^k(\tau) = \varrho_i \max\{\min\{Res(\lambda_i(\tau)) - C_i(\tau), V_k\}, 0\} - \pi_b^k(\tau) \tag{21}$$

where $Res()$ is a function that calculates the resource from the service demand.

Then, if the demand is under capacity but the operating cost is higher, the CSP can also participate in the auction to increase the utility from running jobs on other CSPs' resources. In this condition, the utility function is:

$$u_i^k(\tau) = Cost_i^k(\tau) - \pi_b^k(\tau) \tag{22}$$

There is only one condition in which the CSP wants to sell their resource to others: the demand is notably less than the capacity of the available resources. In this condition, the utility function of the CSP which wants to provide type-k dedicated resources to others can be represented by:

$$u_i^k(\tau) = \pi_s^k(\tau) - Cost_i^k(\tau) \tag{23}$$

When participating in the auction, provider i needs to consider the utility it can gain from the auction. For the potential seller who has idle resources, it wants to increase its profit by increasing the utilization of the idle resources. For the potential buyer, it also wants to increase its profit by either serving the demand which is over the capacity of the local resource or decreasing the operating cost by outsourcing the jobs to the other lower cost resource in the contracts.

From the above discussion, we can get the bidding strategy for the CSPs. For the potential seller, it only needs to estimate the usage of the resource and match the idle resource into one type of dedicated resource and set the bid price by the operating cost. For the potential buyer, it has a mixed strategy: if the predicted service demand is over the capacity of the current servers in the server list, it bids by the profit it can get from serving the overcapacity demand; otherwise, it bids by the operating cost instead.

From the above bidding strategies' discussion, provider i that has idle servers matching type-k dedicated resources can set the selling price by the operating cost:

$$s_i^k(\tau) = \begin{cases} Cost_i^k(\tau) & \text{if } Res(\lambda_i(\tau)) > C_i(\tau) - V_k \text{and } D_k \subset M_i(\tau) \\ \text{NULL} & \text{otherwise} \end{cases} \tag{24}$$

where "NULL" represents a null bid. Here, we note that the condition, $D_k \subset M_i(\tau)$, checks whether the available server list, $M_i(\tau)$, contains the type-k resource, D_k, or not.

The buyer who wants to buy type-k dedicated resource will bid in two conditions: first, the predicted demand is above the current capacity; second, the operating cost is relatively high in the time slot τ. So the bidding strategy is a combination of two separate strategies:

$$b_i^k(\tau) = \begin{cases} \varrho_i \min\{Res(\lambda_i(\tau)) - C_i(\tau), V_k\} & \text{if } Res(\lambda_i(\tau)) > C_i(\tau) \\ Cost_i^k(\tau) & \text{otherwise} \end{cases} \tag{25}$$

It is worth noting that as shown in the above strategies, when there are idle resources, the CSP will set the buy bid with its operating cost regardless of whether it needs the resources or not. We can understand this condition from two aspects: if the CSP does not bid when it has idle resources, it will always get zero utility in this round of auction; instead, if the CSP bids with the operating cost, it will at least get zero utility, which will become the dominant strategy for the CSP.

While above suggested bidding strategy provides a basic methodology to estimate the benefits CSPs can obtain from participating in the auction, we note that the bidding strategy can be extended with additional requirements for CSPs (e.g., reliability requirements, scheduling policy constraints, data locality constraints, etc.,) to further customize the bidding process.

7.3.3 Winning Bids Decision

In this subsection, we present the proposed algorithm for determining the winning bids. As mentioned earlier, the proposed auction algorithm (Algorithm 10.) guarantees both truthfulness and budget balance properties.

The winner decision algorithm for the auction is based on the McAfee mechanism which both guarantees truthfulness and budget balance. The decision of the auction is indicated by two sets of indicators, $X_b(\tau)$ and $X_s(\tau)$. The buying bid b_h's indicator is set to be $x_{s_h} = 1$ if bid b_h wins. For the selling bid, it is the same as for the buying bid. We note that the time complexity of Algorithm 10 is $O(N \log N)$. Here the key

Algorithm 10: Algorithm for the double auction to choose winners for one type of dedicated resource

Input : Type of the dedicated resource : k;
Buy bids: $B^k(\tau) = \{b_1(\tau), b_2(\tau), ...\}^k$;
Sell bids: $S^k(\tau) = \{s_1(\tau), s_2(\tau), ...\}^k$;
Output: Clearing Buying Price: $\pi_s^k(\tau)$ Clearing Selling Price: $\pi_b^k(\tau)$;
Auction decision: $X_b(\tau) = \{x_{b_1}(\tau), x_{b_2}(\tau), ...\}^k$;
$X_s(\tau) = \{x_{s_1}(\tau), x_{s_2}(\tau), ...\}^k$

Sort $B(\tau)$ in descending order by $b_i(\tau)$ and $S(\tau)$ in ascending order by $s_i(\tau)$;
Initially, set current buying price b as b_k as the first bid (highest price) in $B_k(t)$ and current selling price s as s_k as the first bid (lowest price) in $S_k(t)$. current bid indicator $h = 0$;
while $b \geq s$ **do**
    $s = s_i(\tau)$;
    $b = b_i(\tau)$;
    $h = h + 1$;
    if h is larger than the size of $B^k(\tau)$ or $S^k(\tau)$ break;
$\rho = (b_{h+1} + s_{h+1})/2$;
if $b_h \geq \rho \geq s_h$ **then**
    All the first h buyers and first h sellers win with price:;
    $\pi_s^k(\tau) = \pi_b^k(\tau) = \rho$;
else
    All the first $h - 1$ sellers win with selling price: $\pi_s^k(\tau) = s_h$;
    All the first $h - 1$ buyers win with buying price: $\pi_b^k(\tau) = b_h$;

time-consuming operation is the initial sorting operation.

7.3.4 Contracts Establishment Process

In this subsection, we discuss the overall process for building the contracts based on the auction algorithm. We illustrate it as a sequence of procedural steps:

Step.1 Begin the auction from $k = K$ that represents the CSP with the largest amount of resources. The auction begins at time slot $\tau = 1$.

Step.2 CSPs send the bids to the coordinator of the federation using the strategy described above.

Step.3 The coordinator decides the winners by the algorithm as shown in Algorithm 10.

Step.4 The winning bids build the contracts one to one in the order of the sell and buy bids. Each winning buyer adds the servers in the dedicated resource into the server list $M_i(\tau)$ in time slot τ. Each winning seller also updates the server list $M_i(\tau)$ by removing the servers from the list.

Step.5 The losing bids of the type-k dedicated resource are sent back to the CSP. The CSP will separate it into several bids which may be used in the auctions for the smaller types of dedicated resource. These bids also obey the strategies defined in Subsection 7.3.2.

Step.6 For the next smaller type-$\{k-1\}$ dedicated resource, the CSPs execute the above steps from Step.2 until the smallest type dedicated resource is reached.

Step.7 The CSPs execute the above steps from Step.1 for the next time slot $\tau+1$ until the last contract time slot $\tau = T$ is reached;

We note that the optimized contract provides the CSPs with a set of available remote resources in a cost-efficient manner. However, the individual CSP needs to employ intelligent job scheduling techniques that understand the cost implications of the underlying contract structure to leverage the remote resources available to the CSPs effectively. We discuss them in the next section.

7.4 Contracts-based Job Scheduling

In this section, we first formulate the contracts-based job scheduling problem and then propose our mechanisms to schedule jobs using the extended resources provided to the CSPs through the resource sharing contracts.

7.4.1 Job Scheduling Problem Model

We model the scheduling problem for each CSP that participates in the federated cloud. We note that in the scheduling model, the time slot indicated by t can be a very short time interval. It can be several orders of magnitude shorter than the time slot for establishing the contracts which is represented by τ. We introduce two new terms, t_τ^{begin} and t_τ^{end}, to indicate the beginning and end of the interval of the contract time slot τ during job scheduling. All the jobs come to a CSP enter into an FIFO queue. The arriving rate in time slot t is denoted as $r_i(t)$ for provider i. The queue is updated every time slot:

$$Q_i(t+1) = \max\{Q_i(t) - U_i(t) + r_i(t) - A_i(t), 0\} \tag{26}$$

where $U_i(t)$ is the set of scheduled jobs in time slot t, $A_i(t)$ is the set of the jobs which are dropped because of violating the SLA or other failures in time slot t. The queue is only updated at the beginning of the time slot.

Table 7.2: IBM server x3550 Xeon X5675 power consumption with different workload

Workload	0%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
Power Consumption(Watts)	58.4	98	109	118	128	140	153	170	189	205	222

In addition, there is a constraint that the amount of the used resources of the running jobs cannot exceed the current capacity of the CSPs. We use the notations below to represent the running jobs.

$$Z_i(t+1) = Z_i(t) + U_i(t) - F_i(t) \tag{27}$$

where $Z_i(t)$ is the running jobs' set at time slot t, $U_i(t)$ is the scheduled jobs in time slot t, $F_i(t)$ is the finished or failed jobs in the running jobs set in time slot t. Here the capacity constraint is:

$$Res(Z_i(t)) \leq \sum_{m}^{M_i(\tau)} C_i^m, \ \forall t \in [t_\tau^{begin}, t_\tau^{end}] \tag{28}$$

where $Res(Z_i(t))$ represents the estimated resources that all the running jobs need for CSP i in time slot t. The resource list $M_i(\tau)$ is updated at the beginning of every contract time slot τ.

The actual benefits a CSP i can get from the contract is obtained as:

$$Contract_i(t) = (\sum_{k}^{K} \pi_s^k(\tau) x_{s_i}^k(\tau) - \sum_{k}^{K} \pi_b^k(\tau) x_{b_i}^k(\tau))/(t_\tau^{end} - t_\tau^{begin} + 1), \ \forall t \in [t_\tau^{begin}, t_\tau^{end}] \tag{29}$$

where $\pi_s^k(t)$ is the clearing selling price for the contracts in time slot $t \in [t_\tau^{begin}, t_\tau^{end}]$ for the type-k dedicated resource.

The actual cost of the electricity consumption is calculated by the consumption of each server in time slot t. For each server, the electricity consumption can be approximated by a linear model[20] as illustrated in Table 7.2. For each server $m \in [1, M_i]$, the electricity consumption in time slot t is calculated as:

$$E_i^m(t) = \begin{cases} \xi_i^m + \frac{\psi_i^m * u_i^m(t)}{C_i^m} \text{ if server } m \text{ is on} \\ 0 \text{ otherwise} \end{cases} \tag{30}$$

where ξ_i^m and ψ_i^m are the parameters in the linear model for estimating the electricity cost of the server, u_i^m is the utilization of server m in time slot t. Therefore the actual electricity cost for CSP i can be calculated as:

$$Electricity_i(t) = PUE_i * \sum_{m}^{M_i} E_i^m(t) * \delta_i(t) \tag{31}$$

where PUE_i is the Power Usage Effectiveness (PUE) of CSP i, $\delta_i(t)$ is the electricity price for the time slot t for CSP i.

We note that the profit is approximately proportional to the resource consumption of running jobs. Therefore this is another objective for the provider to maximize:

$$Income_i(t) = \varrho_i * Res(Z_i(t)) \tag{32}$$

Finally, we note that there is a penalty of violating the SLA of the jobs. The penalty can be estimated as:

$$Penalty_i(t) = \sum_{p \in Q_i(t)} \vartheta_i * \varrho_i * \gamma_p \quad \text{if violates } p\text{'s SLA} \tag{33}$$

where ϑ_i is the parameter of the penalty which is determined in the SLA for violating the SLA, γ_p is the resource usage of job p.

Considering all the objectives and constraints together, we can obtain the objective for the CSP i as:

$$\begin{aligned} \max \lim_{T \to \infty} \sum_{t=0}^{T-1} & Income_i(t) - Electricity_i(t) - Penalty_i(t) + Contract_i(t), \\ & \forall i \in [1, N] \\ & s.t.\ Constrains\ (26) - (28) \end{aligned} \tag{34}$$

We note that the above-mentioned scheduling problem can be reduced to a bin packing problem. However, bin packing problems are shown to be NP-hard [51]. Thus, we need to resort to heuristic techniques to achieve a scalable solution. We describe them in the next subsection.

7.4.2 Contracts-based Job Scheduling Mechanisms

We propose a set of heuristic scheduling mechanisms to schedule jobs across the geo-distributed clouds based on the resource sharing contracts. We develop two mechanisms to optimize the scheduling decision based on the contracts: one is to use the contracts in a cost-aware manner; another is to schedule the job with minimal live migrations by understanding the duration of the contracts.

The objective of the CSPs in the job scheduling technique is to schedule the jobs to maximize the utility of using the contracts. The utility of using the contracts consists of two parts namely the payment for successfully completing the jobs and the cost of using the contracts. As optimizing the number of completed jobs is an NP-hard problem, we use the basic real-time FCFS service policy and first-fit scheduling algorithm as the preliminary approach. We then extend the basic scheduling algorithm to optimize the cost of using the contracts. We note that the cost of using the contracts can be separated into two components namely (i) the cost to use the contracts, which is decided by the price and (ii) the additional cost which occurs when using the contracts such as the job migrating cost and the penalty of violating the contracts.

Concretely, we propose three schemes. While the contracts cost-aware scheduling (ConBCA) adopts a lowest cost resource (resources in the contract or local datacenter) first policy to minimize the cost of using the contracts, the contracts duration-aware scheduling (ConBConA) considers the duration of the contracts in order to minimize the number and cost of live migrations. The contracts duration-aware and cost-aware scheduling (ConBCAConA) simultaneously aims to minimize both the cost of using the contracts as well as the number of live migrations based on contracts duration.

7.4.2.1 Contracts cost-aware scheduling

In the contracts cost-aware scheduling approach, the CSPs cooperate with each other based on the established contracts to maximize their profit. In an intuitive scheduling policy, the providers may choose only to use the contracts when the local resource is not sufficient to meet the workload demands, which we refer to as contracts-based local first (ConBLF) scheduling mechanism. The disadvantage of this mechanism is that if the local operating cost is higher than some of the negotiated price in the contracts, the contracts become poorly utilized. An alternate intuitive scheduling policy may approach the scheduling problem in the opposite manner which is the contracts first contracts-based scheduling mechanism. This scheme also has shortcomings, the cost of the contracts may be higher than the local resource in some scenarios, in which cases, the contracts-based remote resource may only be used when the local resources are exhausted.

The above discussion provides the intuition behind the contracts-based cost-aware (ConBCA) scheduling algorithm. The provider can estimate the unit cost compared with the local unit operating cost. It may use the contract which has a lower cost than the local operating cost first. It may then use the local resources. The contracts which have a higher cost than the local operating cost are used only when the previous two kinds of resources are exhausted. The detailed algorithm is illustrated in Algorithm 11.

7.4.2.2 Contracts duration-aware scheduling

For maximizing the utility of using the contracts, another aspect to consider is how to increase the utilization of the contracts and avoid violating the contracts. An example of contract violation includes using the resources in the contracts beyond the effective time. When the contract is near the end of effective time, the jobs which are not already finished should be moved to other places that have resources to continue the jobs. Therefore, we have another cost which occurs in this condition. This cost represents the cost of migrating the jobs when the contract ends but the jobs are not finished already. Furthermore, for providing a more continuous service, most of the jobs should be moved using a seamless live migration method [40] that incurs minimal or no impact on the performance of the job. Live migration is a way of migrating the VMs which minimize the down-time of the VM. This mechanism iteratively copies the dirty memory to the remote server and moves the job to the server in a short down-time. However, the migrating operation is expensive and may consume network bandwidth between the two geo-distributed datacenters and the CPU resources on both servers. Therefore, the provider should avoid the migrating process by carefully understanding the duration of the contracts. We call this scheme as the contracts-based contracts duration-aware mechanism (ConBConA).

The provider needs to minimize the probability of live migrating the jobs when the contract is near expiration time. If the expected finish time of the job is beyond the expiration time of the contract, the job should not be scheduled to the datacenter unless the benefit of running outside is more than the migration cost.

We model the probability of migrating the job as follows. The remaining time from the expected finish time of each job p in time slot t can be calculated as:

$$t_p^{remain} = t_\tau^{end} - (t + l_p) \tag{35}$$

Where t_τ^{end} represents the end time of the contract, t is the starting time of job p which is indicated by the current time slot t, l_p is the expected length of job p.

We assume that the probability of migrating is inversely proportional to the remaining time from the expected end time to the expiration time of current effective contract. The possibility of migrating can be estimated by the above remaining time of the job p:

$$Pr(z_p = 1) = \alpha * \frac{1}{t_p^{remain}} \tag{36}$$

where $z_p = 1$ is an indicator that indicates whether the job needs to be live migrated, $Pr(z_p = 1)$ is the probability that the job needs to be live migrated, α is a parameter. The probability that the job needs to be migrated live is inversely proportional to the remaining time from end of the job to the contract expiration time.

The migration cost can be estimated by a function which is related to the dirty rate and the size of memory. We use the equation in [85] to estimate the migration cost:

$$Migration_p(t) = \begin{cases} 0, \ p \text{ is scheduled locally} \\ (\eta Size(v_p) + \iota) Pr(z_p = 1), \ otherwise \end{cases} \tag{37}$$

where η, ι is the parameter in the live migrating cost estimating model, $Size(v_p)$ is the estimated live migration size of the job p which can be calculated by the algorithm in [85]. The actual size of migrating is related to the kind of jobs running on the VM. This function calculates the migrating cost if the job is scheduled in time slot t to the contract.

As shown above, we use a simplified live migration model to calculate the migration cost. For a more accurate estimation of the migration cost, the migration model may be replaced with other sophisticated models such as [8] that considers the migration bandwidth and the page dirty rate, [23] which considers the bandwidth waste of in-band migration and the downtime of the job or [152] which optimizes the live migration cost on a Wide Area Network (WAN). Thus, the key idea behind the contracts-based contracts duration-aware scheduling algorithm is to minimize the possible cost of having to migrate the tasks.

7.4.2.3 Contracts duration-aware and cost-aware scheduling

Considering the two mechanisms discussed above, we develop a new mechanism for scheduling the jobs with the features of both contracts duration-aware and cost-aware scheduling with using the contracts (ConBCAConA).

In this approach, the provider first sorts all the contracts by their unit buying price and separates them into two sets: lower cost contracts which has lower cost than the local resource; higher cost contracts which has higher cost than the local resource which is only used when the workload is beyond the capacity of the previous two sets of resources.

In every time slot t, the provider schedules the tasks by considering the utility of finishing the task p:

$$u_i^p(t) = \varrho_i Res(p) * l_p - Migration_p(t) - Cost_i^p \tag{38}$$

where $Res(p)$ is the resource task p needs, $Cost_i^p$ is the operating cost depending on whether the job runs locally (calculated by the local operating cost) or on the remote contract resources (calculated by the buying price).

The scheduler needs to maximize the outsourcing profit and minimize the cost of using the contracts and live migrating the jobs as shown in Algorithm 11. For each time slot t, the time complexity of the scheduling

Algorithm 11: Algorithm of Scheduling the jobs based on contracts

```
1 Separate the contracts in time slot τ into two set: Contracts_low and Contracts_high;
2 foreach Scheduling time slot t ∈ [t_τ^begin, t_τ^end] do
3     if job p can be scheduled to Contr_j ∈ Contracts_low then
4         Schedule the job p which Pr(z_p = 1) ≤ ϵ with Contr_j;
5     else
6         if Job p can be scheduled to local resource then
7             Schedule the job p to the local resource
8         else
9             Schedule the job p which Pr(z_p = 1) ≤ ϵ with Contr_j ∈ Contracts_high;
```

Algorithm 11 is determined by the number of jobs $|P|$ so that the time complexity is $O(|P|)$.

7.5 Evaluation

In this section, we present our experimental study on the performance of the proposed contracts-based resource management techniques.

7.5.1 Setup

We implement the simulator and the proposed algorithms in JAVA and the simulator runs with a global virtual clock. We log the status of all the servers in the CSPs for each virtual second which includes the available resources, the job status (submitted, running or finished) information, and the performance metrics of the jobs. We run all the experiments on an Intel i5-3210M Machine with 8GB memory.

Table 7.3: Datacenters' Default Configuration

# of providers	**25**	# of servers / provider	**300**
Cores per server	**6**	MIPS per Core	**3067**
Memory per Server	**16 GB**	Bandwidth per Server	**1 GB**
PUE	**1.2**	Contract Interval	**4 hours**
Prepaid ratio	**0.5**	Maximal response time	**3600**

7.5.1.1 Datacenters

We consider that each provider has one datacenter in our evaluation and we use the default configuration shown in Table 7.3 for each datacenter. The default server has the same performance as the IBM server x3550 (2 x [Xeon X5675 3067 MHz, 6 cores], 16GB). The different power consumption of the server from [20] is shown in Table 7.2. The locations of the datacenters are chosen from Amazon's AWS datacenters' locations with the timezone and location from [136]. The price model uses the AWS EC2 On-Demand price model [13]. The actually model is given by: $\beta_0 + \beta_1 * ECU + \beta_2 * Memory$, and from the linear regression, we get $\beta_0 = 0.0005884, \beta_1 = 0.0093460, \beta_2 = 0.0076067$. Here, one ECU equals 1000 MIPS (Million Instructions Per Second) in our definition.

7.5.1.2 Real electricity price

The electricity price is generated based on the hourly real-time electricity price from [139]. We obtained the distribution of the data in 2015 from NationalGrid's hourly electricity price and the distribution includes two type of features: one is auto-correlation which means that the electricity price's trend has very high possibility to be similar in a period of observation; another is the burstiness, which indicates that the electricity price can fluctuate significantly in a short period. In our simulation, we use the distribution of the electricity price and randomly choose each day's price from it by shifting the time based on the datacenters' time zones.

7.5.1.3 Workload

We conduct the experiments using a real-world workload trace from the online Parallel Workload Archive (PWA) repository [46]. We choose the SHARCNET clusters' trace from the archive as it is a computational-intensive High-performance Computing (HPC) workload trace logged in real clusters. The trace is for a duration of thirteen months (From Dec. 2005 to Jan. 2007) with 1,195,242 independent jobs[73, 116]. As suggested by the publisher of PWA, we do not use the entire SHARCNET trace as the configuration of the clusters has changed during the duration of the trace. Instead, as recommended, we extract a two-day (From Dec. 1st to Dec. 2ed) period that does not contain cluster configuration changes. For our simulation, we have extracted the following information from the trace: the job submitted time, the job running time, the requested number of CPUs, the requested size of memory and the utilization of CPUs. In order to keep

Table 7.4: Compared Algorithms

Algorithm	Description	Use Contracts	Cost-aware	Contracts duration-aware
NF	No Federation			
ConBLF	Contracts-based Local First	√		
ConBCA	Contracts-based Cost-aware	√	√	
ConBConA	Contracts-based Contracts duration-aware	√		√
ConBCAConA	Contracts-based Contracts cost-aware and duration-aware	√	√	√
RT	Real Time complete cooperation			

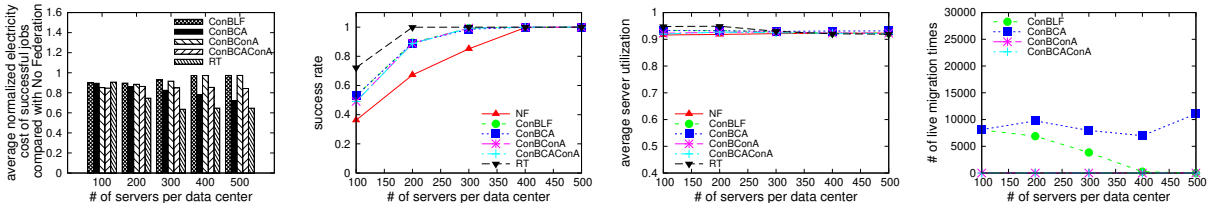
the workload same in every sub-experiment, we have used two different methods to generate the workloads. Except for the evaluation of testing the impact of number of providers in Section 7.5.2.2, in other scenarios, all the jobs in the workload are replicated in each provider so as to keep the same amount of jobs in each sub-evaluation. For the evaluation of the different number of providers, we have replicated the extracted trace 25 times that corresponds to half of the maximum number of providers in the evaluation and we have randomly assigned the jobs to the CSPs so as to maintain the same amount of jobs in each sub-evaluation. We set the dedicated resources in the simulation using a set of hierarchical categories having 256, 128, 64, 32, 16, 8, 4 servers respectively. By default, we assume an accurate prediction of the workload at each CSPs. For evaluating under conditions of erroneous predictions, we dedicate a separate set of experiments to test the performance of our algorithms under various levels of errors in workload prediction.

7.5.1.4 Algorithms

The reference algorithms include: (i) no federation (NF), which does not share any resources and workload with others and the scheduler tries the best effort to minimize the electricity cost; (ii) contracts-based scheduling algorithm using the local resource first (ConBLF); (iii) contracts-based scheduling algorithm with contracts cost-aware (ConBCA) scheduling; (iv) contracts-based contracts duration-aware scheduling algorithm that avoids live migration (ConBConA); (v) contracts-based contracts cost-aware and duration-aware scheduling (ConBCAConA); (vi) the last candidate approach for comparison is the unrealistic method which uses a greedy algorithm (filling the jobs to the datacenter with lowest operating cost by the FCFS policy) to optimize the operating cost across all the datacenters without considering the local datacenter's profit. We refer to it as real-time complete cooperation scheduling (RT). The summary of the algorithms is shown in Table 7.4.

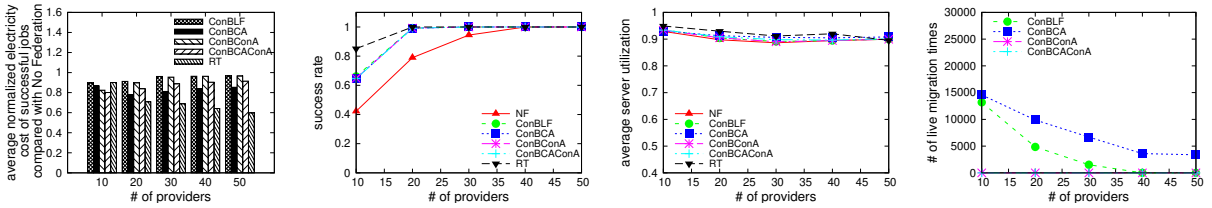
7.5.2 Experimental Results

For illustrating the performance of our contracts-based algorithms, we complete five sets of experiments: first, we study the impact of increasing the number of servers in the datacenters; second, our experiment



(a) Electricity Cost (b) Success Rate (c) Average Server Utilization (d) Number of Live Migrations

Figure 7.4: Evaluation results for a different number of servers per datacenter



(a) Electricity Cost (b) Success Rate (c) Average Server Utilization (d) Number of Live Migrations

Figure 7.5: Evaluation results for a different number of datacenters

analyses the impact when the number of datacenters is increased; third, we add different amount of errors to the prediction of the workloads and study its impact; fourth, we test the influence of different contract intervals on the performance; finally, we evaluate the fairness of our algorithm compared to global optimization approaches that do not consider local profits of the individual datacenters. For each set of the first four experiments, we measure the electricity cost per successful job, the success rate, the average server utilization and the number of job live migrations.

7.5.2.1 Impact of Number of Servers

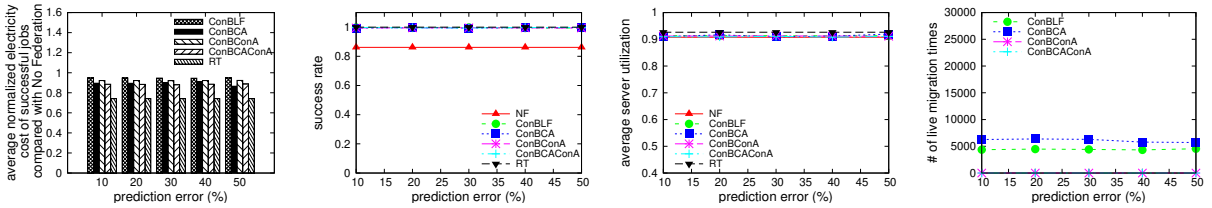
We first test the performance of our mechanisms using different number of servers per datacenter. The number of servers increases from 100 to 500 per datacenter in the evaluation. As shown in Figure 7.4a, the y-axis is the normalized electricity cost per successful job compared with NF. The x-axis is the number of servers per datacenter. We observe that with the ability to share resources in the cloud federation, the electricity cost compared with no federation has been optimized from about 10% to 40% as the number of servers increase. ConBCA achieves the best result and it is close to that of RT. This is due to the fact that if the provider does not need to consider violating the contracts or the cost of live migrations, it can send all the jobs to the lower cost contracts it holds. As this will increase the utilization of the low cost contracts, it can potentially decrease the operating cost per successful job. In Figure 7.4b, we observe that the success rate increases with increase in the number of servers. The priority of sharing the resources

are mainly reflected when the resources are scarce. Thus, if the resource is scarcer, the difference can be larger. In Figure 7.4c, we can observe that our contracts-based mechanism and the RT mechanism obtain better results (around 2%) as the sharing mechanism makes the workload more balanced in the providers which increases the utilization of the servers. In Figure 7.4d, we observe that the techniques that avoid live migration (ConACB and ConACAConB) achieve the best result. The cost-aware algorithm performs poorly as it uses the contracts regardless of the effective time of the contracts. ConBLF which is the local resource first mechanism also does not perform well (near 8K live migrations when the number of servers is 100) as it ignores the length of the contract. ConBLF performs better than the ConBCA when the number of servers is increased as ConBLF uses local resource first strategy. Therefore, if the resource is available, the usage of the contracts will decrease and the number of live migrations will be decreased as well.

Overall, we can deduce that contracts-based algorithm performs significantly better than the NF scheme with respect to operating cost and success rate. Live migration is avoided when using contracts duration-aware mechanisms (ConBConA and ConBCAConA) and the performance of our contracts-based mechanisms is close to that of the RT method in most of the measurements.

7.5.2.2 Impact of Number of Providers

We next evaluate the performance of our mechanisms to study the impact of different number of service providers in the federated cloud. The number of providers is increased from 10 to 50. As shown in Figure 7.5a, the y-axis is the normalized electricity cost per successful job compared with no federation. The x-axis is the number of providers which remains unchanged in this set of evaluation. We can observe that the electricity cost compared with no federation has been optimized from about 10% to 20% in the proposed schemes. This result is similar to the previous experiment. Here, ConBCA optimizes the electricity cost very significantly compared to the other contracts-based algorithms when the resources are sufficiently available in the federation. As shown in Figure 7.5b, the success rate also increases with increase in the number of providers. Thus, all contracts-based algorithms perform better than NF with more than 20% improvements. Also, the contracts-based algorithms perform similarly to RT when the number of providers is more than 20. The utilization levels measured in Figure 7.5c also show a similar trend as previous experiments. We can see that when the number of providers is increased, the number of live migrations of ConBLF and ConBCA are decreased. The reason is that when the resources are available and when the number of jobs submitted to each datacenter is decreased, the number of live migrations also decrease. *From the above experiments, we can conclude that contracts-based algorithm performs better than the NF in operating cost and success rate. Live migration is avoided when using contracts duration-aware mechanisms (ConBConA and ConBCAConA). Most of the measurements of our contracts-based mechanisms are close to the RT.*



(a) Electricity Cost (b) Success Rate (c) Average Server Utilization (d) Number of Live Migrations

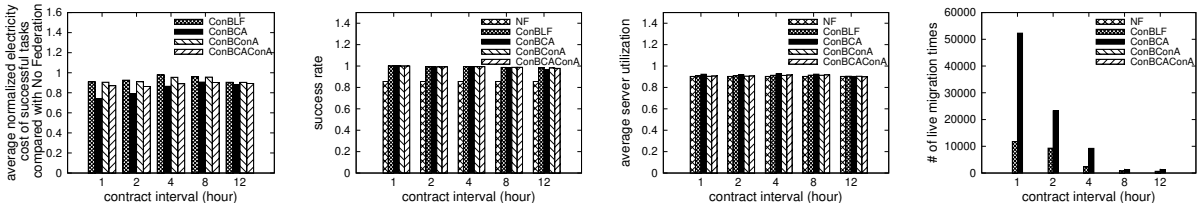
Figure 7.6: Evaluation results for different average errors of the workload predictions

7.5.2.3 Impact of Prediction Errors

We next evaluate the performance of our mechanisms using different proportions of prediction error added to the demand and workload prediction. We use a white noise[153] to introduce the error in the predicted values. The amount of error introduced is increased from 10% to 50% in the experiment. All the other settings are kept as shown in the default configuration Table 7.3. As shown in Figure 7.6a, the y-axis is the same as the previous evaluations. The x-axis is the average percentage of prediction errors. The electricity cost compared with no federation has been optimized from about 10% to 15% regardless of the prediction errors. The result shows that the prediction errors do not influence the result significantly(2% with 50% added error) as the error only influences the contract trading volume. The resources are traded between the providers with the true value. As only the volume is influenced insignificantly by the added error, it does not have a significant impact on the outcome. In Figure 7.6b, we also observe that the success rate is not influenced much (less than 1% variance) by the prediction error. We find a similar trend with the measurements on utilization and live migrations in Figure 7.6c and Figure 7.6d respectively. *From the above experiments we can deduce that the prediction error does not influence our algorithm significantly and hence the proposed techniques are robust under a wide range of errors in the workload prediction.*

7.5.2.4 Impact of Contract Intervals

Next, we test the performance of our mechanisms with different duration of contract intervals. The contract interval is set to five values (1, 2, 4, 8 and 12 hours). As shown in Figure 7.7a, the x-axis is the contract interval. We can see that the contract interval does not influence the operating cost significantly. But when the contract interval increases, the normalized electricity price is also increased considerably between 3% to 5% except for ConBCA. ConBCA is influenced more and has an increase of 16%. This is because with a longer interval, the evaluation accuracy of the true values for each time slot will decrease which will influence the contracts establishment process. However, the influence is not very significant. In Figure 7.7b, we observe that the success rate is not influenced significantly by the contract intervals and in Figure 7.7c, we note that the average utilization of the running servers also does not change significantly. As shown in



(a) Electricity Cost (b) Success Rate (c) Average Server Utilization (d) Number of Live Migrations

Figure 7.7: Evaluation results for different inner intervals of the contracts

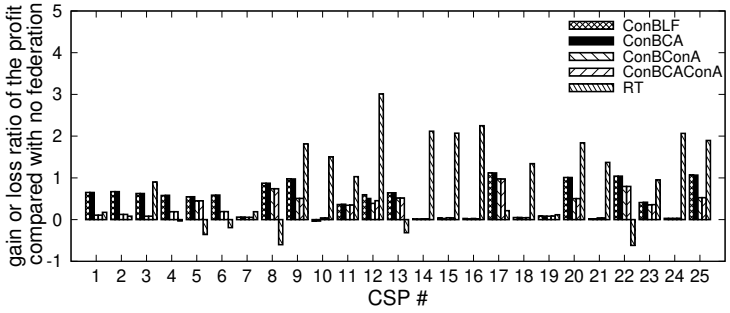


Figure 7.8: The gain or loss ratio of the profit for each individual CSP

Figure 7.7d, the number of live migrations decreases when the contract interval increases(for LFConB, from 12K to 0.6K; for ConBCA, from 52K to 1K) as the longer effective time decreases the probability of live migrations. The contracts duration-aware mechanisms (ConBConA and ConBCAConA) also perform better here.

Overall, from the above experiment, we can see that the contract interval influences the number of live migration and electricity cost. With an increase in the contract interval, the normalized electricity cost is slightly increased and the number of live migrations is decreased.

7.5.2.5 Fairness

In this set of experiment, we evaluate the fairness of the proposed schemes by comparing the individual profit of each CSP. The result is observed with the setup of 100 servers per datacenters. We use the default setting for the other experiment parameters. As shown in Figure 7.8, the y-axis is the gain or loss ratio of the normalized profit which is the difference between the profit that can be earned using the federated cloud and the profit that can be earned otherwise when operating alone. When the number is larger than 0, it means that the provider earns more using the federated cloud than when it operates alone and vice versa. The x-axis represents the index for each CSP. From the figure, we can see that, when the federated cloud is operated using the real-time complete cooperation mechanism, there are six CSPs (CSP4, 5, 6, 8,

13, 22) of the total 25 CSPs losing profits compared with the profits they can earn when operating alone. The contracts-based mechanisms perform significantly better than operating alone except for CSP10 which gets a very minimal decrease (less than 4%) compared to the profit that it can earn from operating alone.

From the observations above, we can see that the real-time complete cooperation mechanism globally optimizes the operating cost but results in several CSPs losing profits. In contrast, the proposed contracts-based mechanisms perform better than the real-time complete cooperation mechanism and achieve higher fairness with most of the CSPs obtaining higher profits when participating in the federation.

7.6 Summary and discussion

In this chapter, we proposed a contracts-based mechanism for resource sharing between CSPs in a federated cloud. Compared with previous work in this area, our proposed approach considers both the global cost minimization as well as the local profit maximization of each individual datacenters participating in the federation process. We developed an auction-based mechanism for contract establishment and a suite of contracts cost-aware and duration-aware scheduling techniques that maximize the local profits of the CSPs while meeting the individual job requirements. We evaluated the performance of the proposed approach using a trace-driven simulation study with realistic workload traces and electricity pricing. The contracts-based solution achieves good performance and performs significantly better than the traditional model in terms of fairness in local profits while achieving similar operational costs and success rate properties as existing methods.

While the techniques presented in this chapter assume a centralized approach, in the next chapter, a decentralized implementation of allocating resources in geo-distributed environments is presented.

8.0 Decentralized resource allocation and management mechanism for geo-distributed edge and cloud resources

In this chapter, we propose, design and implement a decentralized platform for allocating geo-distributed edge resources. We first propose a decoupled decentralized resource allocation model that manages the allocation of computing resources distributed at the edges to the service providers that have application demand to use those resources. Then, we propose a sealed bid double auction protocol based on decentralized smart contracts with a two-phase sealed bidding and revealing mechanism. Next, we develop a decentralized auction-based resource sharing contract establishment and allocation mechanism that ensures truthfulness and utility-maximization for the providers. Finally, we implement a prototype of the proposed model on a real blockchain test bed and our extensive experiments demonstrate the effectiveness, scalability and performance efficiency of the proposed approach.

8.1 Background and Motivation

In the Internet of Things (IoT) era, the demands for low-latency computing for latency-sensitive applications (e.g., location-based augmented reality games, real-time smart grid management, real-time navigation) has been growing rapidly. Edge Computing provides an additional layer of infrastructure to fill latency gaps between the IoT devices and the back-end cloud computing infrastructure.

In current edge computing models, we have service providers that provide geo-distributed cloud infrastructure or edge computing infrastructure as a service. For example, Google Espresso [161] supports low latency content delivery around the world with more than one hundred points-of-presence (PoPs). Microsoft deployed more than one hundred data centers (DCs) that can serve 140 countries around the world [97]. Current models also include cloud services that are extended to support edge computing. For instance, Amazon Greengrass[131] supports Amazon Lambda [130] and other AWS services on the edge devices and Azure IoT Edge [17] extends the Azure cloud services to the edge devices. Current models operate by restricting each service provider to only utilize their own edge infrastructure resources for providing service. It creates a strong barrier between the providers and makes the infrastructure investment not only inefficient but also redundant.

Sharing edge infrastructure has significant benefits to optimizing resource usage cost and meeting strict latency requirements. Geo-distributed edge infrastructures that cooperate together to optimize resource allocation can build a seamless geo-distributed edge federation platform to provide infrastructure to a wide range of edge computing applications (e.g. virtual reality, smart city, big data analytic) that have strict latency and bandwidth requirements.

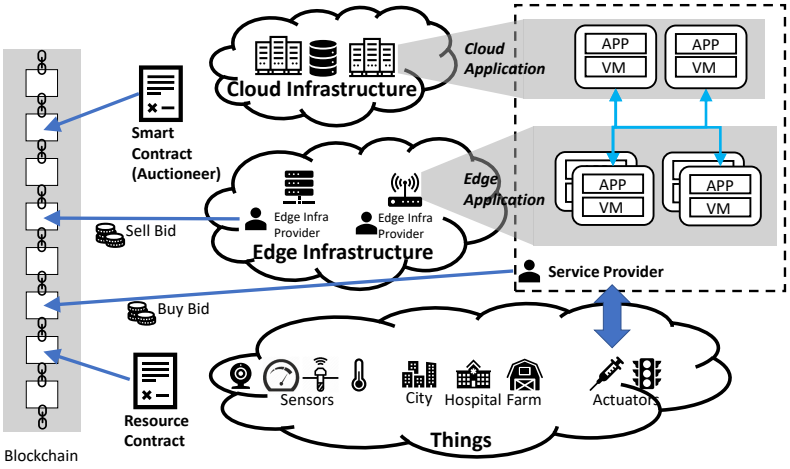


Figure 8.1: Overview

In this work, we design a decentralized mechanism which enables resource sharing among the service providers and the edge infrastructure providers (Figure 8.1). The proposed approach implements a long-term persistent resource sharing scheme at the edge using decentralized blockchain networks. In the remaining part of this section, we describe the background of edge resource sharing and blockchain-based smart contracts.

8.1.1 Edge Resource Sharing

We assume that the resources are organized in a layered architecture [22] where the edge infrastructure acts as the middle layer between the cloud infrastructure and the smart things as shown in Figure 8.1. The dense geo-distributed edge infrastructure includes the micro datacenters (MDCs) and the smart gateways placed at the edge of the network which are located one hop from the end devices. We model the edge resource sharing platform similar to the model described in [158] (Chapter 6). The platform includes (i) potential buyers namely service providers (SPs) who are responsible to provide services directly to the end users and want to lease edge resources on demand to increase their revenues, and (ii) potential sellers namely edge infrastructure providers (EIP) who operate either the MDCs or the smart gateways that support multi-tenant resource allocation by running containers.

To model the resource sharing problem at the edge, we assume that there are $|N|$ SPs that require edge computing infrastructure to support their services. For simplicity, we assume that for each SP $i \in N$, where N is the set of the SPs, there is a quantifiable service demand of the SPs in each geographic region for every discrete time slot of a day. The resource requirement is represented by the application container [109, 66] (a configured virtual machine integrated with the service software), which has several requirements such as CPU consumption, memory size, network bandwidth and latency requirement. We use $\lambda_i^p(\tau)$ to represent the workload coming from a particular location p in time slot τ for SP i as shown in Figure 8.3. We assume

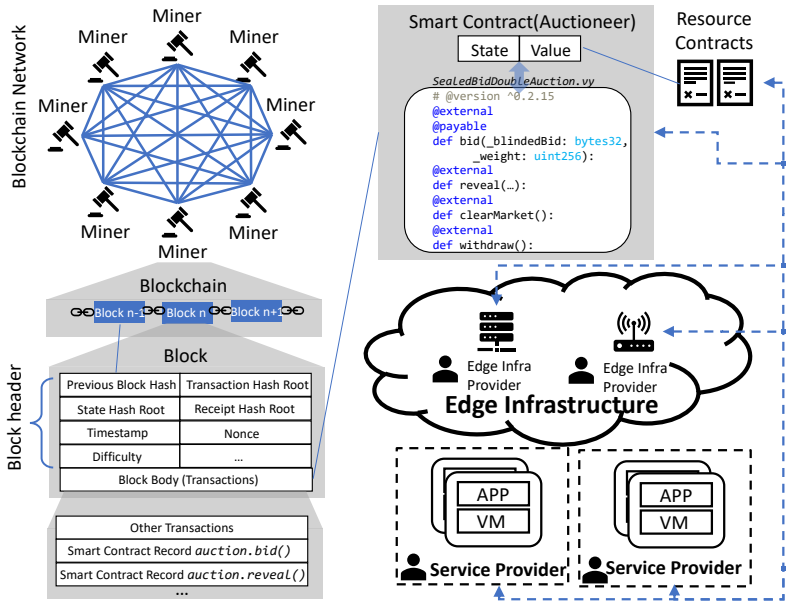


Figure 8.2: Blockchain-based Edge Resource Sharing

that there are several EIPs and each of them handles a large number of highly geo-distributed MDCs and smart gateways that represent the edge computing resources. We assume that each MDC or smart gateway can act as a virtual node $d \in D$, which can support the deployment of several containers. For simplicity, we assume that every container consumes equal amount of resources for running the application service. The capacity of the overall node is denoted as C_d and it represents the number of containers that can be run on the node. The final resource allocation decision can be simplified as a mapping between the edge resources $C_d, \forall d \in D$, and the workloads, $\lambda_i^p(\tau), \forall i \in N$, which decides the actual node that run the containers to serve the corresponding workloads from a location p. The decision problem is NP-hard [158] (Chapter 6) and in this work, we simplify the problem by first delineating non-overlapping regions and then dividing the continuous sharing time into non-overlapping time slots (e.g., one hour). Thus, each trade (auction) is handled for a particular region $r \in R$ and for a particular time slot τ, which allows the problem to be solved using the proposed auction mechanism (Section 8.2.3).

The key challenges of the edge resource sharing problem are two folds: (i) the geo-distributed nature and the distributed ownership of the edge infrastructure make it hard to centralize the resource allocation decision and (ii) the competitive relationship between the EIPs and the SPs make it challenging to guarantee efficiency and fairness. To tackle these challenges, we employ blockchain-based smart contracts to deploy a truthful auction that automatically processes the bids from the potential buyers and sellers to generate resource contracts between them guaranteeing both efficiency, fairness, and decentralization at the same time.

8.1.2 Blockchains and Smart Contracts

Blockchain is a distributed ledger that stores transaction records as a chain of blocks maintained by a set of miners in the decentralized blockchain network (Figure 8.2). The miners mine the blocks to include the transactions to form the blockchain for the state in which all miners reach an agreement through a consensus protocol (e.g., proof-of-work (PoW), or proof-of-stake (PoS)). The architecture of blockchain makes it possible to achieve decentralization, integrity, auditablility, transparency and high availability at the same time. Smart contracts are built on top of blockchain and they allow user-defined programs to be executed on the blockchain. More specifically, the smart contract can be treated as a program deployed on the blockchain network, which resides at a specific address (generated when deploying) on the blockchain, including algorithms (functions within a contract) and data (the state of the smart contract) as shown in Figure 8.2. To interact with smart contracts, there are two ways: (i) retrieving the state or data from the smart contracts which can be directly restored from the blockchain data without sending transactions, and (ii) change the state of the smart contracts which require calling the functions of the smart contract by sending transactions and the execution is completed after the transaction is included in the blockchain network. Ethereum is well-known blockchain network that supports smart contracts. Ether is the cryptocurrency used on Ethereum. It is held in and can be transferred between accounts including externally owned accounts (EOAs) and contract accounts (CAs). An EOA is determined by a unique public-private key pair owned by an individual who can use the private key to sign the transactions sent from the account. CAs are different than EOAs. Each CA does not have a key pair and is associated with a deployed smart contract that is activated (deployed) by an EOA. To execute the smart contract, the EOA that deploys a new smart contract or calls a function of a deployed smart contract needs to pay Gas [151] included in the transaction. Gas can be exchanged from Ether. A transaction in Ethereum is a build-in instruction signed by an EOA. Each transaction specifies several information including the sender's address, the receiver's address and *data* (e.g., smart contract bytecode, and a function call with the arguments). Each function call and its input are included in the transaction so that the output can be verified by multiple miners with the same input and the given program. The correctness can be guaranteed by the miners and the consensus protocol of the blockchain.

In this work, the proposed resource sharing (auction) protocol is enforced without trusted third-parties using smart contracts. Additionally, the proposed truthful auction is designed to guarantee that the bidders will bid with their true valuation on the goods and thus, it will reduce the complexity of the auction algorithm. The resource contracts that are persistent and distributed on the blockchain can be used for assessment, billing, and auditing. To cooperate with the geo-distributed utility-aware task scheduling, the edge resource sharing platform can be seamlessly integrated with the existing utility-aware cloud platform to handle a wide range of applications with different requirements.

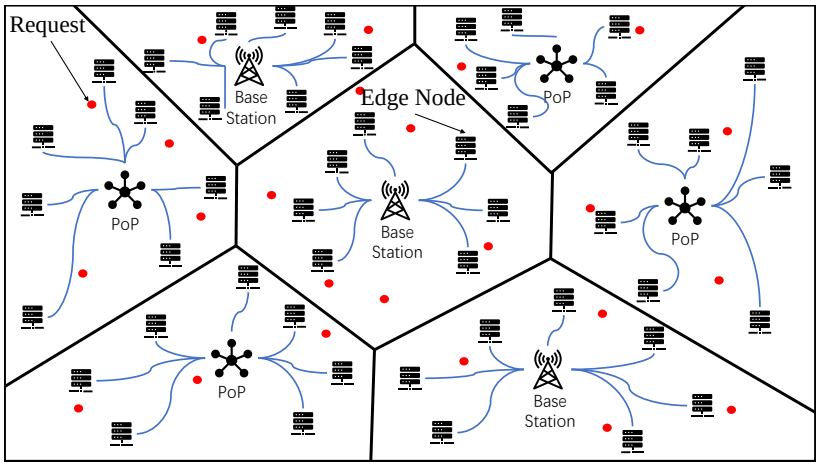


Figure 8.3: Regions

8.2 Smart Contract-based Edge Resource Allocation

We design the proposed edge resource sharing platform using blockchain-based smart contracts. The blockchain acts as the decentralized ledger that stores the trade information (e.g., when and where the resource is shared), and the procedure (smart contracts) of the trade between the buyers and sellers transparently for all the participants.

8.2.1 System Architecture

The architecture of the proposed edge resource sharing platform is shown in Figure 8.2. The basic functionality of the platform is to match the supply and demand of the edge resources based on the auction algorithm deployed in the smart contracts. To make the process decentralized, we employ blockchain-based smart contracts to act as the auctioneer.

As shown in Figure 8.2, we can see that the decentralized consensus and mining of the new blocks are controlled by the miners connected to the blockchain network. Any node can download the client of the blockchain network and be a miner of the network. The public blockchain can be audited by anyone who participates in the blockchain network and accepts the broadcast. The blockchain is established by a sequence of blocks, each of them is generated by the consensus mechanism and connected to its parent block by its hash value. The block stores all the information related to the state changes of the blockchain (e.g., transactions between accounts, and modifications of values in the smart contracts) in their block body as records of transactions. The smart contracts deployed on the blockchain can achieve automatic execution of the algorithms and guarantee the correctness of transactions. In the proposed system, we assume that there are four types of entities namely the service providers (SPs), Edge Infrastructure Providers (EIPs) discussed in Section 8.1, Region Coordinators, and smart contracts as shown in Figure 8.4.

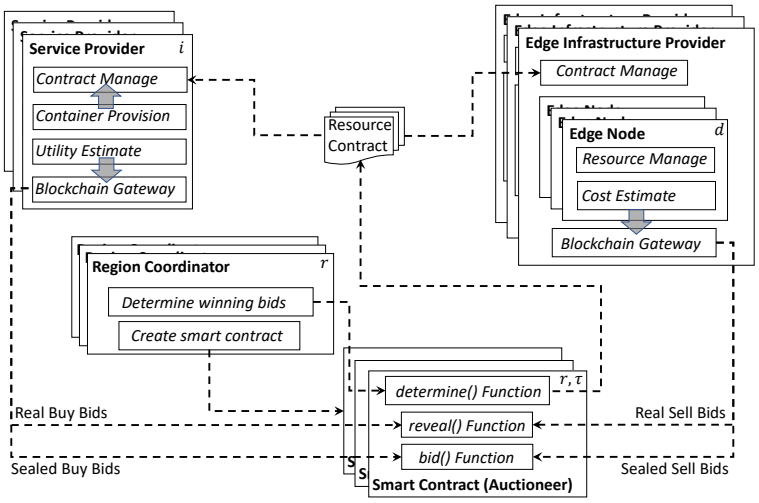


Figure 8.4: Edge Resource Sharing Framework

We use the notion of divided regions to reduce the complexity of matching the latency or location requirement of the edge application to the resources available in a certain area. We assume that the map corresponding to the geographic area is divided into $|R|$ sub-regions, where R is the set of all the regions. The region division can be generated by negotiation between the EIPs and SPs, which can either be based on the location of base stations, Point of presences (PoPs) of Internet Service Providers (ISPs) or based on administrative divisions (e.g., counties) as shown in Figure 8.3. Each region $r \in R$ only includes a specific area and there is no overlap between the regions. We also assume that the expected workload distribution for each time slot τ is $\lambda_i^r(\tau)$ for SP i in the region r. The workload distribution contains all the workloads coming from the region. We use $\lambda_i^p(\tau)$ to represent the workload coming from a particular position p in region r. As we primarily consider the workload which needs real-time service, $\lambda_i^p(\tau)$ is often the upper bound of the workload during the time slot τ from position p. In each region, there can be multiple nodes that serve as the infrastructure. We use E_r to represent the list of nodes which serves in region r. In addition, the nodes that serve in one region can guarantee the lowest possible latency of placing the services of the SPs with an average latency, l_r, as they either directly connect to the PoP or base station of the region, which is one hop from the end devices that send the requests. With the above assumptions, the edge nodes can be either micro datacenters (MDCs), smart gateways, or mega datacenters that satisfy the placement requirement for a certain region.

We use smart contracts to run the resource trading between the SPs and the EIPs in each region. The smart contract acts as the trusted third party that uses the predefined auction algorithm to decide the winning bids and establish the resource contracts. However, as we need both the resource buyers (SPs) and the sellers (EIPs) to bid in the (double) auction, both of them are not suitable to create or handle the smart contract as the owner. Therefore, we assume that there is a region coordinator for each region.

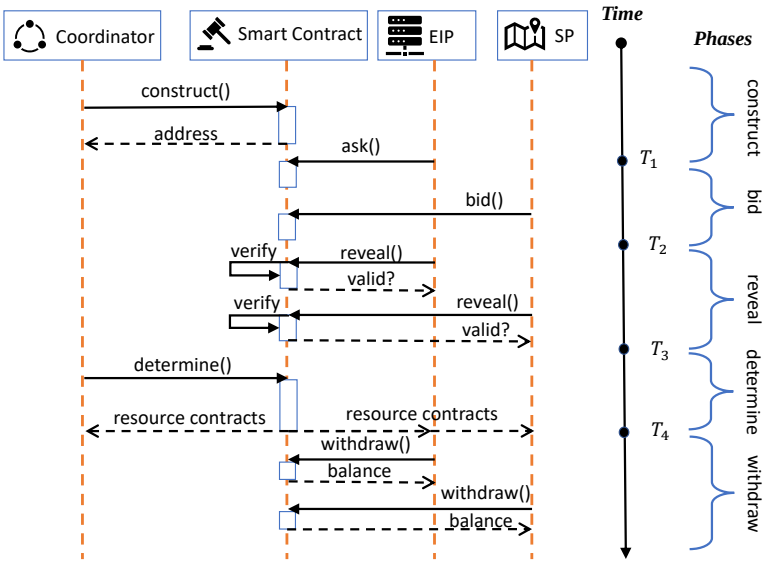


Figure 8.5: Decentralized Sealed Bid Double Auction Procedure

The coordinator creates the smart contract for the auctions based on the rules (e.g., when the participants can bid, and what is the time duration for the resource contracts for a particular auction). It notifies the participants the information of the smart contract (e.g., address, and interfaces), monitors the smart contract and calls the function of the smart contract to determine the winning bids. The coordinator is the owner of the smart contract and the cost of running the smart contract is paid either by registering the auction or from the subsidy of the auction.

The auction algorithm and the status is published in the smart contract. The participants and the coordinator can audit the status of the smart contract from the blockchain data. The smart contract acts as the auctioneer and runs the auction automatically from the predefined auction algorithm and decides the winning bids. Then, based on the policy, the resource contracts are established and recorded in the blockchain for further accounting and auditing when the resources are used.

8.2.2 Decentralized Sealed Bid Double Auction Protocol

In this section, we present the proposed resource allocation techniques for EIPs and SPs to establish relationships (resource contracts) with each other and explain how smart contracts help in this process.

We design a sealed bid double auction using smart contracts on the blockchain that enables the EIPs and the SPs to bid in the auction with their sealed bids. We assume that for each region r and each time slot τ, there is an auction that decides which EIP and SP pairs trade with each other. We note that all participants and the coordinator can interact with the blockchain network using blockchain gateway services such as

```
# @version ^0.2.15
# SealedBidDoubleAuction.vy
struct Bid:
    bidder: address
    blindedBid: bytes32
    deposit: uint256
    weight: uint256
    value: uint256
...
# Auction parameters
coordinator: public(address)
biddingEnd: public(uint256)
revealEnd: public(uint256)
...
# State of the bids
asks: public(HashMap[address, Bid[MAX_BIDS]])
bids: public(HashMap[address, Bid[MAX_BIDS]])
askCounts: public(HashMap[address, uint256])
bidCounts: public(HashMap[address, uint256])
validAsks: public(Bid[MAX_CONTRACTS])
validAskCount: public(uint256)
validBids: public(Bid[MAX_CONTRACTS])
validBidCount: public(uint256)
# Allowed refund map (withdraw or payment)
pendingReturns: public(HashMap[address, uint256])
```

Figure 8.6: Smart Contract Initial Parameters

Infura [1]. We assume that each EIP and SP have at least a client that is responsible to interact with the region coordinators, resource management, and the smart contracts to control the process of evaluating the resource valuation. They interact with the smart contract (auctioneer) and record and report the established resource contracts to the internal resource management for resource allocation and task scheduling.

The auction procedure consists of five phases as shown in Figure 8.5:

Smart Contract Creation

Input: region id r, resource sharing time slot τ, region coordinator account r^{addr}
Output: smart contract address z
Before time T_1, for each region r and each time slot τ:

1. The region coordinator creates the smart contract by calling $z = r^{addr}.deploy(Z, T_1, T_2, T_3, r, \tau)$, where Z is the class definition of the resource sharing auction.
2. The region coordinator waits for the transaction to be included in the blockchain network and gathers the address z for the smart contract.
3. The region coordinator broadcasts the smart contract tuple $< z, r, \tau >$ to all the participants registered.

Phase 1. Construct: includes the registration and smart contract creation. Before the auction smart contract is created, all the EIPs and SPs that want to participate need to register their accounts to the region coordinator through the steps shown below. The region coordinator will also record the account information. We use the notations T_1, T_2, T_3, T_4 to represent the end time of the first four phases (construct, bid, reveal, and determine) as shown in Figure 8.5. When handling the creation of the smart contract,

[1] https://infura.io/

the region coordinator will obey the rules of phase periods. For example, for the construct phase, the region coordinator will call the construct function of the smart contract before T_1. It is worth noting that T_1, T_4 are enforced by the auction protocol which needs to obey the region coordinator but T_2, T_3 can be enforced directly by the smart contract which is included in the program of the smart contract. Based on the negotiated time slot length of the edge resource sharing period (for example, in an one hour period), the region coordinator creates a smart contract for each resource sharing period. Several parameters are initialed together with the smart contract as shown in Figure 8.6. It written in Vyper [68] to include the details. We define a `Bid` structure at the beginning of the smart contract and we initialize the map between the sellers and the sell bids (`asks`), the map between the buyers and the buy bids (`bids`) and the counts of them. After the smart contract for the auction is created, the region coordinator will get an address for the smart contract. Then, both the address and other related information (e.g., region id, and time slot) are broadcasted to all the participants.

Bid Procedure

Input: smart contract address z, bid value b, weight c, SP account i^{addr}
After time T_1, before time T_2:

1. SP i generates the blinded bid $\beta = hash(b, c, \alpha)$ with a randomly generated secret α.
2. SP i decides a deposit value, which is $\gamma = b * c + random(b * c)$.
3. SP i uses its registered account i^{addr} to send the bids by interacting with the smart contract z by calling $z.bid(\beta, c, \{"from" : i^{addr}, "value" : \gamma\})$, where the entries in the brackets describe the corresponding entries included in the transaction.
4. The smart contract z records the blinded bid information as a tuple, $< \beta, c, i^{addr}, \gamma >$.
5. SP i waits for the transactions to be verified and included in the blockchain network. Then, it record the bid in the local bid array $\mathbf{B}_i$ including tuples of the true bids, $< b, c, \alpha >$.

Phase 2. Bid: After the auction smart contract is broadcasted to all the participants, the participant can fetch the auction calendar (e.g., when the participants can bid, and reveal) from the smart contract. It is as mentioned above as noted by T_1, T_2, T_3. When the bidding period is started, all the participants who registered can bid to the smart contract. For each bid, the EIP or SP sends the bid by interacting with the smart contract by sending a transaction including the function for bidding, the blinded bid (hash value of the entire bid with a randomly generated secret), the number of containers requested or available, and the deposit to the smart contract. We omit the procedure for the seller (EIPs) to place their sell bids (asks) as it is similar to the bid procedure shown below. The deposit of the sell bid (ask) will be used to guarantee that the resources are preserved for the resource contracts during the effective time slot.

Phase 3. Reveal: in the reveal phase, the participant needs to interact with the smart contract to reveal the bids they sent by sending the real bid to be verified by the smart contract. The bid value, weight and secret will be sent to the smart contract for verification. If the stored hash value matches the hash value of the above tuple, the bid is valid and will be reserved for the auction. Otherwise the bid will be removed and the corresponding deposit will be appended to the withdraw fund list. We omit the procedure for the seller (EIPs) to reveal their sell bids (asks) as it is similar to the reveal procedure of the buyers (SPs) shown

below. It is worth noting that, in the reveal phase, it is possible for the bidders to cancel the previous bids by sending a wrong bid value to the smart contract in the corresponding bid entry. Therefore, we do not implement the cancellation functionality in the smart contract and we leave it to the client to implement it. As only the bidder has the private key to sign its own bids, the authentication of the blockchain network can make sure that the denial of service attack (e.g., send the wrong bids to the smart contract to cancel others bids) is hard to conduct.

Reveal Procedure

Input: smart contract address z, bid array $\mathbf{B}_i$, SP account i^{addr},
After time T_2, before time T_3:

1. SP i sends the bid array $\mathbf{B}_i$ where each entry includes the bid value b, weight c, secret α of the bid, to the smart contract for verification by calling $z.reveal(\mathbf{B}_i, \{"from" : i^{addr}\})$.
2. Smart contract z verifies each bid by the blinded bid hash β placed in the bid phase. If the buy bid matches the hash value and the deposit is larger than the total bid value ($\gamma \geq b * c$), it will be appended to the valid buy bid array, B, stored in the smart contract shown as an array `validBids` in Figure 8.6.

Phase 4. Determine: In the winner determination phase, the auctioneer (smart contract) decides the winning bids. For simplicity, we omit the region id r and time slot τ in the following discussion. We use a binary notation x_b to denote whether the buy bid b wins or not ($x_b = 1$ wins, and vice versa). Similarly, x_s denotes whether the sell bid (ask), s, wins or not. We denote the buy price as π_b and sell price as π_s. Similarly, we denote the auction result as two sets, $X_s = \{x_s | s \in S\}$ and $X_b = \{x_b | b \in B\}$. Each entry of the set determines the decision of one sell or buy bid in the auction. Besides the winners, the algorithm also sets the map (shown as `pendingReturns` in Figure 8.6) between the bidders and the pending withdraw amounts that determines how much fund can be withdrawn for each bidder including the bid deposit of the fail bids and the overvalued deposit of the winning bids. The resource contracts are also established in this phase based on the results of the auction. Each of them can be denoted as a tuple $< i, \mathbf{D}, \mathbf{C}, \tau, \pi_b, \pi_s >$, where $\mathbf{D}$ is the list of sellers (edge nodes) that provides the resources to SP i in the resource contract and $\mathbf{C}$ is the array of the number of containers provided by each seller.

Winner Determination Procedure

Input: smart contract address z
After time T_3, before time T_4:

1. The region coordinator calls $z.determine()$, the predefined auction algorithm, in the smart contract to decide the winning bids, which is discussed in Section 8.2.3. The decision X_b and X_s are stored in the smart contract along with the initial resource contracts.

Phase 5. Withdraw: After the auction is closed and the resource contracts are established, the participants can withdraw their remaining funds (e.g., the excess value deposit and the deposit of the fail bids) from the smart contract.

Withdraw Procedure

Input: smart contract address z, bidder's account a
For each bidder (either EIP or SP):

1. The bidder calls $z.withdraw()$ with its account a.
2. Smart Contract z verifies the available withdraw amount `pendingReturns[a]` defined in Figure 8.6. If `pendingReturns[a]>0`, the fund will be sent back to account a and set `pendingReturns[a]=0` to avoid double withdrawal.

8.2.3 Auction Algorithm

From the Myerson–Satterthwaite theorem [102], we can see that there are no auction algorithms that can satisfy all of the four auction properties at the same time namely: (i) Individual Rationality (Definition 3), which means that no participants should lose from bidding in the auction, (ii) Weak Balanced Budget (Definition 5), which means that the auctioneer will not subsidize the auction, (iii) Truthfulness (Definition 2) that ensures that bidding with true valuation is the dominant strategy of all the bidders and (iv) Economic efficiency (Definition 6) ensures that the good should be finally allocated to the bidder who values it the most. However, there are auction algorithms that can satisfy three of the properties with a bounded loss on the remaining one. McAfee mechanism[90] can satisfy individual rationality, weak balanced budget, truthfulness with a bounded loss of economic efficiency ($1/\min(|B|, |S|)$ in our problem). In our work, we use the McAfee mechanism in the auction design. The truthfulness property guarantees that for the participants whose objectives are to maximize their utilities, the dominant bidding strategy is to bid by their true valuation. Based on the above assumption, we first define the utility of the SP and EIP.

Utility of Service Provider: We model the utility of the SP to run the service at the edge. Here, we consider services having higher requirements for latency such as location-based augmented reality games[107] and intelligent traffic light control [164]. The utility gain of the SP can be expressed by the gain in changing the execution of the real-time service from the cloud to the edge which can be represented by the function:

$$v_i^p(\tau) = f(l_r) - f(l_{pi}(\tau)) \tag{39}$$

where $f(l)$ is a function which estimates the utility that can be obtained by providing the service with a latency l. We assume that the function is a non-increasing function related to the latency which means that when the latency is increased, the utility will decrease. Here $l_{pi}(\tau)$ represents the latency between the mega datacenter of SP i and the position p. We note that the utility gain can be also modeled by other criteria such as the bandwidth cost (e.g., when moving an aggregator operator to the edge to reduce the overall bandwidth cost of moving the data to the cloud).

Utility of Edge Infrastructure Provider For the EIP, its objective is to earn higher revenue by providing the infrastructure to SPs. Therefore, the utility for the EIP is obviously the profit that it can obtain by renting the resource to the SPs. The true valuation of the resource for the EIP can be defined using the

Table 8.1: Auction Schedule Example for the resource usage in 15:00 - 16:00 9/30

Phase	Time	Participants/Executor
Auction creation	0:00 9/29	coordinator
Bid	0:00-16:00 9/29	EIPs and SPs
Reveal	16:00-23:00 9/29	EIPs and SPs
Determine	23:00-23:59 9/29	coordinator
Withdraw	after 23:59 9/29	Everyone

operating cost of the resources. For each node, the unit operating cost function $Cost_d(\tau)$ can be defined as the ratio of the sum of the operating cost of each server and the capacity of the node:

$$Cost_d(\tau) = \frac{\sum_m^{M_d} Cost_d^m(\tau)}{C_d} \tag{40}$$

where $Cost_d^m(\tau)$ is the fluctuating operating cost of server m in node d in time slot τ. To determine the winning bids, we extend the McAfee mechanism [90] by allowing each bid to contain both the unit price and the number of containers at the same time. The group bidding method can save a significant amount of cost when the auction is running on the smart contract.

The auction algorithm is shown in Algorithm 12. As we can see, the time complexity is $O(max(|B|log|B|, |S|log|S|))$. We omit the region id r and the time slot τ in the algorithm definition. In the algorithm, we can see that the buy bids and sell bids (asks) are sorted in their natural ordering (ascending order for sell bids and descending order for buy bids). Then, we find the break-even index by accumulating either from the buy bids or sell bids (asks) by counting the number of containers. When the break-even index is found (the next buy bid price is lower than the next sell bid price), the supply is filled (all the possible containers are sold) or we meet the end of the bid array. Then based on the McAfee mechanism, we decide the winning bids and the final selling and buying prices.

8.2.4 Smart Contract Implementation

We implement the smart contract by Vyper [68], which can run on any blockchains that support Ethereum Virtual Machine (EVM) (such as Ethereum, Hyperledger Fabric, etc.). Our choice of using Vyper is due to its security, auditability and being predictable by implicitly limiting the features of the language such as recursive function calls, which may lead to unpredictable results when interacting with the smart contract. As Vyper does not support the dynamic array, we set the size of all the arrays that appear in the smart contract as 64. Limiting the number of bids can decrease the cost of establishing the smart contract and running the functions.

As our resource contracts can be established before the Service Providers actually use the resources, it provides adequate time for the coordinator to conduct the auction. As shown in Table 8.1, the auction can be handled a day ahead and the two phases can be scheduled with in certain time ranges. As the time

Algorithm 12: Algorithm for determining the winning bids

Procedure `determine`$(B, S) \rightarrow \pi_s, \pi_b, X_s, X_b$
  sort B in descending order by the bid price ;
  re-index B as $B = \{b_i, c_i | i \in [1, |B|]\}$;
  sort S in ascending order by the bid price ;
  re-index S as $S = \{s_j, c_j | j \in [1, |S|]\}$;
  set the overall supply $C_r = \sum_{d \in E_r} C_d$;
  set current buy price b as the first bid (highest price) in B ;
  set the sell price s as the first sell bid (lowest ask) in S ;
  set number of buying containers $h = 0$, and selling containers $k = 0$;
  set current index $i = 1, j = 1$;
  while $True$ **do**
    if $h \geq C_r$ **then**
      $h = C_r$;
      break ;
    else if $i + 1 > |B|$ *or* $j + 1 > |S|$ *or* $b_{i+1} < s_{j+1}$ **then**
      break ;
    if $h > k$ **then**
      $s = s_j$, $k+ = c_j$, $x_j = 1$;
      $j++$;
    else
      $b = b_i$, $h+ = c_i$, $x_i = 1$;
      $i++$;
  $\rho = (b_{i+1} + s_{j+1})/2$;
  if $b \geq \rho \geq s$ **then**
    $\pi_s = \pi_b = \rho$;
  else
    $x_i = 0, x_j = 0$;
    $\pi_b = b_i$;
    $\pi_s = s_j$;

range is sufficient for each participant to interact with the smart contract and to wait for the transaction to be included in a block, and verify the transaction, the gas price can be set with a low priority fee or even without setting the priority fee to save the overall cost.

8.2.5 Resource Contract

After the auction is cleared, the resource contracts for time slot τ are established. The length of the time slot τ can be negotiated by the EIPs and SPs to determine an appropriate granularity for the resource allocation by considering both low-cost (e.g., the cost of running the auction on the smart contract) and efficiency for placing services (e.g., minimizing the migration when the resource contracts are expired). We assume the length of the time slot is one hour, and the auction will be handled on the previous day when the resource will be used. We present an example in Table 8.1. After the winning bids are determined, the resource contracts are built between the EIP and SP pairs one by one, and the buyer i, the sellers $\mathbf{D}$ (the nodes provide the resources), the buying price π_b, selling price π_s, the array of the number of containers $\mathbf{C}$ (each determines the resources provided by an edge node), and effective time slot τ are recorded either in the auction smart contract (e.g., by a smart contract event) or in a new smart contract that can track the

Table 8.2: Default Experiment Setting for each auction

# of EIPs	6	# of bids (total)	30
# of SPs	6	# of containers per bid	100
PUE	1.2	electricity cost/operating cost	10%
gas price	20 gwei	gas limit (per block)	30M

resource usage on the run. The client of each EIP and SP will monitor the resource contract record in the smart contract to negotiate the resource allocation and gather the payment or provision the tasks.

8.3 Evaluation

In this section, we present the experimental evaluation of the proposed smart contract-based resource allocation implemented and deployed on the real testbed, Rinkeby[140].

8.3.1 Setup

In the experiment evaluation, we focus on testing the performance of the algorithms when they are implemented in a smart contract and deployed on the real testbed, Rinkeby. We assume that in each region, multiple EIPs participate as sellers, multiple SPs participate as buyers and a coordinator handles the coordination. For each auction, we keep the default setting as shown in Table 8.2. We estimate the power consumption using a real server model that has the same performance as that of the IBM server x3550 (2 x [Xeon X5675 3067 MHz, 6 cores], 16GB) [20]. Each server hosts up to 5 service containers at a given time. The electricity price is generated based on the hourly real-time electricity price from NationalGrid's dataset [139]. We use the distribution of the data in 2015 from NationalGrid's hourly electricity price to simulate the fluctuation of the real electricity market. We also set the energy cost to 10% of the overall operating cost [75] and the Power Usage Effectiveness (PUE) is 1.2. For the utility model of the service provider, we choose a base rate similar to that of the a1.large instance (2 vCPUs and 4 GB memory) of Amazon EC2, which is $0.05 per hour usage. The linear growth of utility is based on the utility gain of the latency improvement which is in the 1-100 range.

8.3.2 Methodology

In our experiments, we evaluate two kinds of performance: (i) the performance of the smart contract which includes the gas cost of different scenarios and participants and function calls, (ii) the performance of the auction algorithms in comparison to other baselines. The evaluation metrics are defined as follows:
Gas cost: is the measurement of the cost of smart contracts. It is measured by the EVM which executes the

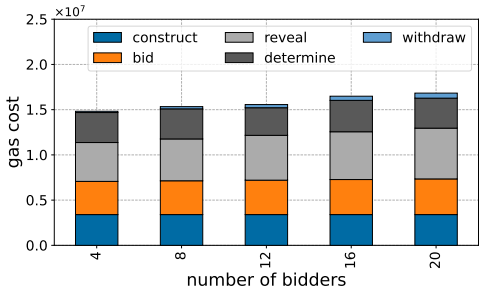


Figure 8.7: Gas cost of different number of bidders

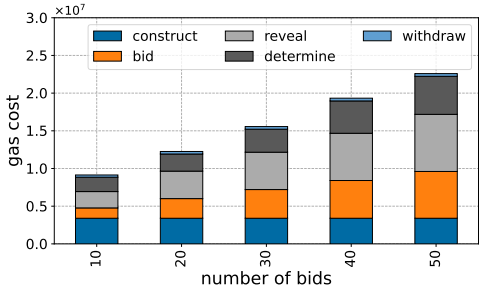


Figure 8.8: Gas cost of different number of bids

function and each assembly operation (opcode) has a fixed gas cost based on its expected execution time.

Social Welfare: is a metric used to evaluate the performance of the auction. The social welfare can be calculated as the sum of the true valuation of the winners. The social welfare measures the efficiency of the auction. It is maximized if the goods are allocated to the buyers who value them the most.

Subsidy: is the difference between the payment from the buyers and the payment given to the sellers. The subsidy is generated based on the auction algorithms. As discussed in the definition, when it is negative, the auctioneer can gather fees from the auction, and when it is positive, the auctioneer needs to subsidize the trade to make up the difference.

8.3.3 Smart Contract Performance

As shown in Figure 8.7, we evaluate the gas cost with different number of bidders (sellers and buyers). The default number of bids is 30 as shown in Table 8.2 and the bids are evenly distributed to each bidder using a simple round robin algorithm. We illustrate the breakdown of the gas cost in five phases: (i) **construct**, in which the coordinator creates the smart contract, (ii) **bid**, in which the participants bid, (iii) **reveal**, in which the participants reveal the sealed bid, (iv) **determine**, in which the smart contract determines the winning bids by the auction algorithm, and (v) **withdraw**, in which the coordinator and participants

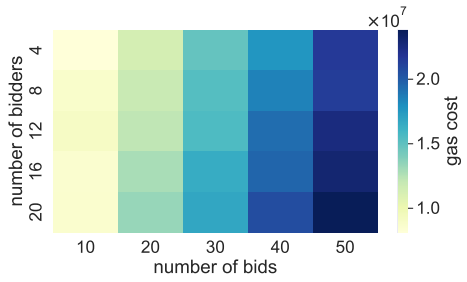


Figure 8.9: Total gas cost comparison of different number of bids and different number of bidders

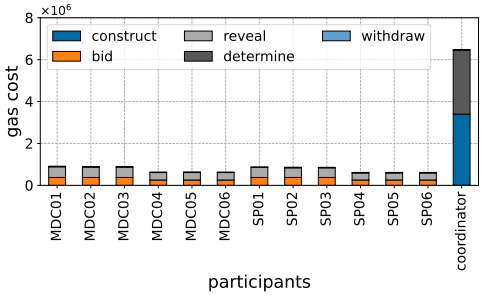


Figure 8.10: Gas cost of different participants

withdraw their funds. As shown in the results, we can see that when there are more participants, the overall gas cost has only a small increase and it only influences the reveal and withdraw phases as it increases the number of reveal and withdraw function calls. In Figure 8.8, the impact of the number of bids is evaluated. The setup is similar to the above experiment and we fixed the number of bidders to 12. We can see that with increasing the number of bids, the gas cost increases significantly especially for the bid, reveal and determine phases. The reason is that the number of bids influence the time complexity of each function call in the three phases. The influence of the number of bidders and bids are illustrated in Figure 8.9. We can get similar conclusion that the overall number of bids influences the gas cost much more than the number of bidders.

We also evaluate the gas cost for different participants and function calls in Figure 8.10 and 8.11. As shown in Figure 8.10, we can see that most of the gas cost is paid by the coordinator and the participants only pay gas cost when they bid or reveal the bids. As we design the auction mechanism based on McAfee mechanism, the coordinator has the opportunity to get payment from the auction and the participants can also pay management fees to the coordinator to cover such cost. In Figure 8.11, we observe similar results that show that construct and determine phases cost most of the gas. As shown in Table 8.3, we convert the gas cost to real ether cost using the price listed in July 2021 (1 ether=$1787). As we can see, the coordinator

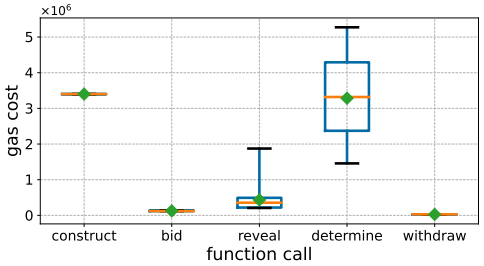


Figure 8.11: Gas cost distribution of each function call

Table 8.3: A breakdown of the gas costs in $ of the function calls using the conversion rate of 1 ether=$1787 and the gas price of 20 gwei (1 gwei = 10^{-9} ether) as listed in July 2021

function	gas cost						cost in $
	mean	min	25%	50%	75%	max	mean
construct	3400175	3400175	3400175	3400175	3400175	3400175	121.52
bid	126469	120002	120026	120056	137126	137156	4.52
reveal	433973	207976	218673	355149	491625	1873449	15.51
determine	3284284	1457251	2371536	3318685	4291935	5273596	117.38
withdraw	26899	23458	23458	28465	28465	28465	0.96

needs to pay nearly 240 dollars to complete one auction, which is relatively high for the current market but there are many alternatives that can decrease the cost. For example, the coordinators can build a private blockchain (e.g., by using Hyperledger Fabric or Ethereum 2.0) to decrease the operating cost of running the smart contract. Each bidder may need to pay $4.5 for one bid and $15.5 for revealing all the bids (linear to the number of bids) on average. The withdrawal only needs less than $1.

8.3.4 Auction Performance

In this experiment, we test the performance of different auction algorithms, in terms of social welfare and subsidy. We compare the following auction algorithms:

McAfee [90]: is the auction mechanism we use in our method. It guarantees both truthfulness and weak balanced budget at the same time.

OPT: is a straightforward auction mechanism that always chooses the highest buy bids and lowest sell bids to trade. It is named as optimal single price omniscient (OPT) [53]. It can always guarantee balanced budge but not truthfulness.

VCG: is a well-known auction mechanism [144, 41, 56] which guarantees truthfulness but not balanced budget.

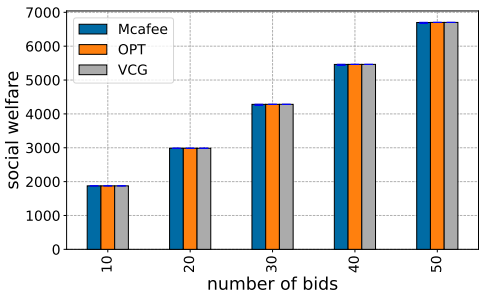


Figure 8.12: Social welfare of different methods with different number of bids

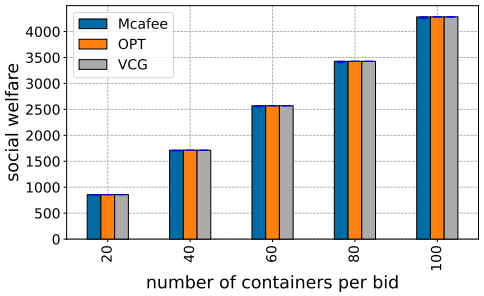


Figure 8.13: Social welfare of different methods with different number of containers per bid

In Figure 8.12, we evaluate the social welfare of the above three auction algorithms with different number of bids. We can see that all of the three methods have similar social welfare in different setups. From the theoretic aspect, only Mcafee may lose social welfare in the second condition (as shown in Algorithm 12) and because the bids are evenly distributed, the probability of the occurrence of the second condition is low. The result is similar when we increase the number of containers being traded in each bid as shown in Figure 8.13. When the subsidy is considered in the evaluation (Figure 8.14 and 8.15), we can see that VCG will suffer from positive subsidy and the coordinator needs to subsidize the trade. However, Mcafee mechanism has weak balanced budget and the subsidy can be only negative. It means that this can be one possible way for the coordinator to gather fees and mitigate the operating cost of running the smart contracts. OPT always has balanced budget and the subsidy is always zero.

8.4 Summary and discussion

In this chapter, we propose a blockchain-based auction for allocating computing resources in an edge computing platform that allows service providers to establish resource sharing contracts with edge infras-

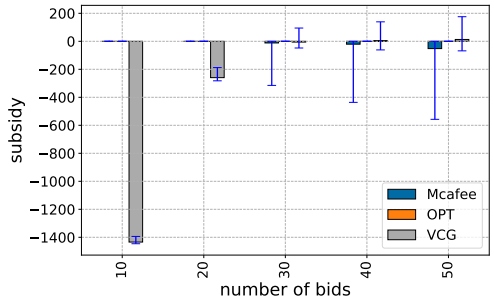


Figure 8.14: Subsidy of different methods with different number of bids

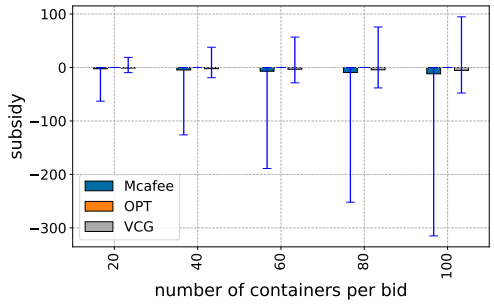


Figure 8.15: Subsidy of different methods with different number of containers per bid

tructure providers apriori using smart contracts in Ethereum. The decentralized auction protocol makes the trust decentralized on the blockchain network, which reliefs the concern on both the centralized auction and the single point of failure. We implement a prototype of the proposed model on a real blockchain test bed and our extensive experiments demonstrate the effectiveness, scalability and performance efficiency of the proposed approach.

While the decentralized resource sharing mechanism is designed for the geo-distributed edge environment, we note that the idea can be applied in geo-distributed clouds as well. Based on the geo-distributed cloud resource sharing model discussed in Chapter 7, we can modify the smart contract defined in Section 8.2 to implement the auction procedure for geo-distributed clouds designed in Chapter 7 so that all cloud service providers can place buy bids and sell bids (asks) and the smart contract can act as the decentralized auctioneer to establish the resource contracts.

9.0 Conclusion and Future Work

9.1 Conclusion

In this dissertation, we study the optimization of stream processing applications in geo-distributed edge and cloud environments by considering key challenges in both the data processing (platform) layer and the infrastructure management layer.

In the first part of the dissertation (Chapter 3, 4, and 5), we present a distributed stream processing platform that optimizes the resource allocation of heterogeneous edge computing resources along three dimensions. Specifically, in Chapter 3, we present Amnis, a distributed stream processing platform that optimizes the resource allocation for stream queries by carefully considering the data locality and resource constraints during physical plan generation and operator placement in edge computing environments. Amnis closely considers physical plan generation while optimizing distributed stream processing. The physical plan generation in combination with data locality optimization in Amnis enables data locality-aware operator placement that optimizes the placement of operators near the data sources. The coflow optimization feature in Amnis further increases the utilization of the network by scheduling the flows by considering the dependencies that exist between them. The results demonstrate the performance of Amnis in terms of both end-to-end latency and throughput compared to previous techniques. In Chapter 4, we extend Amnis to address fault tolerance in edge-based stream processing. We present a novel resilient stream processing framework to achieve system-wide fault tolerance while meeting the latency requirement for the applications in the edge computing environment. The proposed approach employs both checkpointing-based data replication and active replication techniques to seamlessly handle the failure while considering the heterogeneous nature of hardware and software components in edge-based stream processing applications. The heterogeneity challenges the placement and execution of operators that have higher recovery cost expectations than the others. We use the recovery cost estimate to place the active replication in order to reduce both the latency during failures and the fault tolerance cost (e.g., adding duplicate operators). The results demonstrate that our methods find an effective tradeoff between full replication and checkpointing-only mechanisms to achieve low recovery cost (latency) with a bounded fault tolerance budget. In Chapter 5, we further extend the stream processing optimization to incorporate elastic scaling using a reinforcement learning-based method. The proposed method automatically tunes the edge-based stream processing applications to meet the performance requirements. The model-based reinforcement learning method handles the uncertainty of the stream processing application by considering the variation in the workload and processing rate dynamics. The dynamic parallelism configuration problem is modeled as a Markov Decision Process (MDP) and it is solved by reducing it to a contextual Multi-Armed Bandit (MAB) problem using the well-studied LinUCB method. The model-based LinUCB further enhances the pre-train performance by adapting arbitrary G/G/1 queues

with the distribution information gathered from the historical information and improves the initial states of the reinforcement learning model. The experiment results demonstrate the effectiveness of the proposed approach compared to standard methods in terms of cumulative reward and convergence speed. Thus, in the first part of this dissertation summarized above, we extend the current state-of-the-art distributed stream processing engines to adapt to heterogeneous edge environments and the proposed techniques achieve low-latency fault-tolerant elastic stream processing by intelligently optimizing the physical plan, the placement of the operators and automatically deciding the parallelism of the operators.

In the second part of this dissertation Chapter 6, 7, and 8, we propose a set of resource management and resource sharing mechanisms to efficiently allocate edge computing resources in a geo-distributed environment. Specifically, in Chapter 6, we propose *Zenith*, a new resource allocation model for allocating computing resources in an edge computing platform that allows edge service providers to establish resource sharing contracts with edge infrastructure providers *apriori*. The edge resource sharing platform is implemented based on the McAfee mechanism to establish resource contracts between the service providers (SPs) and the edge infrastructure providers (EIPs). Truthfulness, individual rationality and balanced budget properties of McAfee mechanism simplify the design of the auction and the proposed bidding strategies (utility optimization) for the SPs and EIPs. In Chapter 7, we extend the auction model to geo-distributed cloud environments through a contracts-based resource sharing model that allows Cloud Service Providers (CSPs) to establish resource sharing contracts with individual datacenters *apriori* for defined time intervals during a 24 hour time period. In Chapter 8, a decentralized platform for sharing geo-distributed edge and cloud resources among multiple entities is proposed. It employs an auction mechanism that is designed and implemented using blockchain-based smart contracts to carry out a decentralized sealed bid double auction between the infrastructure providers and service providers. Thus, the second part of the dissertation focuses on optimizing the utility of the providers by establishing a resource sharing platform to increase the efficiency of resource usage by either sharing the infrastructure or shifting the peak loads. The proposed techniques based on the theory of mechanism design make it possible for competitors to cooperate and increase the overall social welfare of the system in a decentralized manner.

9.2 Discussion and Future Work

We believe that the outcome of this dissertation would contribute to extending the scope of stream processing applications in geo-distributed edge and cloud computing environments. We briefly discuss a list of possible future directions of our work.

- Automating the optimization for stream processing applications deployed in edge computing environments is more challenging than doing it in cloud environments. This is due to not only the nature of heterogeneous edge resources (computing resources, network resources and topologies, etc.) but also the

nature of the requirements in edge-based stream processing applications (e.g., different requirements, different memory or CPU utilization features, and different hardware requirements). In this dissertation, we deal with some of the key challenges including data locality, fault tolerance and elasticity. There are several promising directions to explore in stream processing using heterogeneous resources for edge-based applications. We summarize them below:

- In our work, we assume that the control signal in the stream processing application will not be the bottleneck. For example, if the stream processing application is deployed in a geo-distributed edge computing environment, the synchronization of the state of the application (e.g., the back-pressure signal) may become a bottleneck. While this is not handled in our work, we believe that tackling this challenge can be a promising direction for future work.
- Our elastic stream processing work (Chapter 5) only deals with the parallelism reconfiguration in an online manner. However, there are many other decisions that can be made within an online algorithm including operator placement, network flows and scheduling in a heterogeneous edge environment. Future work can focus on developing a framework to automate the reconfiguration process by adaptively learning the environment and application characteristics.
- Our work (Chapter 3) optimizes the data locality primarily by grouping the selective operators with the source operators and by moving them to the data source (Section 3.3.1.1). We assume that the selective operators are stateless so that we can arbitrarily split them with the same number of the corresponding source operator. We note that the selective operator can also be stateful as in the case of windowed maximal and key-by sum operators. The reason we limit the applicability to the stateless operator is that the split stateless operators do not need to consider the shuffling phase which needs to be correct (e.g., for key-by sum, all the tuples belonging to a key needs to be processed by one of the instances of the downstream operators) and it can be a bottleneck between the split source operators and the downstream operators. However, it is possible to automatically split the stateful operator to improve the data locality which we leave it as future work.
- Our work does not assume any specific partition function to be used between two connected operators. The partition function can influence the performance in some cases. For example, the partition function can prefer the successor task which is placed locally with the current task to improve the data locality. The optimization of the partition function has been widely studied in the distributed batch processing domain [166], however, applying those methods in a heterogeneous edge computing environment is still a challenge which can be an interesting direction of future work.
- In Chapter 4, we deal with the fault tolerance problem for edge-based stream processing applications using a hybrid recovery cost-aware mechanism. The work considers both the failure rate of the physical nodes and the recovery cost of the operators to decide the active replications. The mechanism acts in a proactive way which predicts the failure and adds the replication beforehand. However, it works better if the prediction is accurate. Maximizing the accuracy of predicting failures

in heterogeneous edge environments is still an open problem and the techniques need to consider both the computing and network heterogeneity at the same time.

- For geo-distributed resource management and resource sharing, we discussed how to fairly share the resources between different entities in a utility-aware manner. Both the centralized (Chapter 6 and 7) and the decentralized approaches (Chapter 8) are based on several key assumptions. We identify the following future directions of work to extend it further:
  - We assume that participants will participate the auction by obeying the rules. If the assumption does not hold (e.g., an attacker can disrupt the auction procedures by making false bids or the participants may collude with each other), the fairness or truthfulness properties of the auction may not be valid. Designing a collusion-resistant auction protocol for sharing resources can be a promising direction in which any out-of-order behaviors can be detected or prevented and can be penalized by the system.
  - In the trading procedure, we assume that all the participants obey the resource contracts established in the auction. It may be violated. The utility-aware penalty can be used to mitigate the loss when the contractor violates the resource contracts. For example, the infrastructure provider does not reserve enough resources for the resource contracts and hence, the penalty can be paid to mitigate the loss of the buyer. It is also a promising direction to automate and enforce the resource contracts on the blockchain network and use the off-chain verification techniques (e.g., zero-knowledge proof) to ensure that the resource contracts are executed correctly.
  - In the auction mechanism design, we assume that there are sufficient number of participants in each auction (i.e., no monopoly) to ensure an effective trading. In real environments, it is possible that in an area, there is only one edge infrastructure provider or service provider who controls most of the market. In such cases, we can rely on traditional auction mechanisms in which only one side, either the buyers or the sellers, bid in the auction. Vickrey auction mechanism is an example of this kind of auction [144]. We note that if both the service and infrastructure markets are a monopoly, our mechanism may not work. To avoid such situations, it is important to maintain an active market that encourages adequate competition and avoids monopoly. Addressing market environments with monopolies is a limitation of the proposed approach and it is an interesting future direction of work.
  - In the utility estimation during resource allocation, we assume that the migration cost is not significant in Chapter 7 and we do not consider it in Chapter 6 and 8. However, the migration cost may be significant in some cases with data-intensive batch workloads. We note that the migration cost can be included in the utility estimation in the model proposed in Section 6.3 and Section 8.2. As the service providers know which region the application will migrate to, it is possible to include the migration cost in the utility estimation when determining the bid value. For the geo-distributed cloud environments (Chapter 7), it is relatively hard to accurately estimate the migration cost as the service may be migrated to any geo-distributed datacenters. We leave this as future work.

– For the techniques presented in Chapter 6, 7, and 8, we assume that all the participants have the ability to estimate the utility (e.g., operating cost) to make sure that the trade is efficient. However, the estimation may not be accurate and may have some errors. We have conducted an experiment in Section 7.5.2.3 to evaluate the influence of the prediction errors. It demonstrates that the resource contracts-based resource sharing mechanism can tolerate some prediction error. However, if the utility estimation is not accurate, it may lead to a loss of revenue for the participants. Future work may address how to accurately estimate the utility with zero or very low prediction error.

Appendix A Mechanism Design

In this chapter, we first introduce the basic notations used to discuss the mechanism design, which is primarily used to design the auction algorithms. We assume in an auction, there are n bidders that want to bid in the auction and each bidder $i \in N$ bids with their bid value, b_i. Here N denotes the set of bidders $N = \{1, 2, ..., n\}$ The bids are represented by a vector $\vec{b} = \{b_1, b_2, ..., b_n\}$. Each bidder has a true valuation of the good, which is private to the bidder, represented by $\vec{v} = \{v_1, v_2, ..., v_n\}$. Depending on the bidding strategy, the bid may be equal or not equal to the real valuation of the good for the bidder. The outcome of the auction is determined by the auction mechanism which can be represented by a vector $\vec{x} = \{x_1, x_2, ..., x_n\}$ where x_i is a binary indicator that indicates whether the bid b_i wins or not. The payments are represented by $\vec{p} = \{p_1, p_2, ..., p_n\}$ where p_i is the payment of bidder i to buy the good in the auction. The objective of each bidder is to maximize the per-bidder utility which can be represented by using the following utility function:

$$u_i = (v_i - p_i)x_i \tag{41}$$

where u_i is the utility of bidder i.

We next introduce Dominant Strategy [108] from game theory that forms a fundamental solution concept for auction mechanism designs.

Definition 1. (Dominant Strategy) Strategy s_i is a bidder i's dominant strategy in a game, if for any strategy $s'_i \neq s_i$ and any other bidders' strategy profile s_{-i},

$$u_i(s_i, s_{-i}) \geq u_i(s'_i, s_{-i}). \tag{42}$$

The concept of dominant strategy is related to truthfulness. In an auction, truthfulness means that revealing truthful information is the dominant strategy for every bidder.

Definition 2. (Truthfulness) "Truthfulness" is also called as strategy-proof or incentive compatibility in auction literature. In game theory, an asymmetric game where players have private information is said to be strategy-proof (SP) if it is a weakly-dominant strategy for every player to reveal his/her private information.

If an auction mechanism is truthful, then the bidders will tend to bid with their true valuation of the products. This is a powerful feature for auction mechanism design as it ensures that both the buyers and sellers can get maximum utility from the auction without cheating.

Formally, we can define the truthfulness property as

$$E[u_i(s_i, s_{-i})] \geq E[u_i(s'_i, s_{-i})]$$

where the u_i is the utility of bidder i, s_i is the strategy that bidder i bids with the true value of the product. Here s_{-i} represents the strategies for the bidders other than bidder i and s'_i represents a strategy

other than s_i. The function illustrates that the strategy that bids with the true value will give the bidder the highest utility compared to any other strategies. If this function is true for all the bidders, it ensures that the auction mechanism is truthful.

In auction mechanism design, there is another property called Individual Rationality guaranteeing that every bidder will not lose utility in the auction. It is defined as below:

Definition 3. (Individual Rationality) An auction is individual rational if and only if $u_i \geq 0$ holds for every bidder $i \in N$.

There are also two other properties of the mechanism design called Balanced Budget and Economic Efficiency defined as follows:

Definition 4. (Strong Balanced Budget) An auction is Strong Balanced Budget if and only if $\sum_{p_i \in \vec{p}} p_i = \sum_{\rho_j \in \vec{\rho}} \rho_j$

where ρ_j is the payment paid to the seller j. The definition defined above shows that the strong balanced budget guarantees that all the payments from the buyers go to the sellers (which is often discussed in double auction where both the buyers and sellers can bid) and nothing is left for the auctioneer. There is another property called weak balanced budget which is defined as follows:

Definition 5. (Weak Balanced Budget) An auction is Weak Balanced Budget if and only if $\sum_{p_i \in \vec{p}} p_i \geq \sum_{\rho_j \in \vec{\rho}} \rho_j$

which means that the auctioneer may gain from the auction by gathering fees from the difference between the payments from the buyers and the payments paid to the sellers.

The economic efficiency is also an important property in the mechanism design:

Definition 6. (Economic efficiency) An auction has Economic efficiency if and only if $\sum_{v_i \in \vec{v}} v_i x_i$ is maximized.

which means that the social welfare is maximized such that the goods of the auction go to the bidder who value them most.

Appendix B Publication list

Papers contributing to this dissertation:

- Jinlai Xu, and Balaji Palanisamy. "Cost-aware resource management for federated clouds using resource sharing contracts." In 2017 IEEE 10th International Conference on Cloud Computing (CLOUD), IEEE, 2017.
- Jinlai Xu, Balaji Palanisamy, Heiko Ludwig, and Qingyang Wang. "Zenith: Utility-aware resource allocation for edge computing." In 2017 IEEE international conference on edge computing (EDGE), IEEE, 2017.
- Jinlai Xu, and Balaji Palanisamy. "Optimized contract-based model for resource allocation in federated geo-distributed clouds." IEEE Transactions on Services Computing (TSC), IEEE, 2021.
- Jinlai Xu, Balaji Palanisamy, Qingyang Wang. "Resilient Stream Processing in Edge Computing." In 21st IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGrid), IEEE, 2021.
- Jinlai Xu, Balaji Palanisamy, "Model-based reinforcement learning for elastic stream processing in edge computing", In 28th IEEE International Conference on High Performance Computing, Data, and Analytics (HiPC) (accepted), IEEE, 2021.
- Jinlai Xu, Balaji Palanisamy, Qingyang Wang, Heiko Ludwig, and Sandeep Gopisetty. "Amnis: Optimized Stream Processing for Edge Computing." Journal of Parallel and Distributed Computing (JPDC), 2022.

Other papers during my PhD study:

- Jinlai Xu, Balaji Palanisamy, Yuzhe Tang, and SD Madhu Kumar. "PADS: Privacy-preserving auction design for allocating dynamically priced cloud resources." In 2017 IEEE 3rd International Conference on Collaboration and Internet Computing (CIC), IEEE, 2017.
- Jingzhe Wang, Balaji Palanisamy, and Jinlai Xu. "Sustainability-aware Resource Provisioning in Data Centers." In 2020 IEEE 6th International Conference on Collaboration and Internet Computing (CIC), IEEE, 2020.
- Chao Li, Balaji Palanisamy, Runhua Xu, Jinlai Xu and Jingzhe Wang. "SteemOps: Extracting and Analyzing Key Operations in Steemit Blockchain-based Social Media Platform." In Proc. of 11th ACM Conference on Data and Application Security and Privacy (CODASPY), 2021.

Bibliography

[1] Mohammad Aazam and Eui-Nam Huh. Fog computing micro datacenter based dynamic resource estimation and pricing model for iot. In *Advanced Information Networking and Applications (AINA), 2015 IEEE 29th International Conference on*, pages 687–694. IEEE, 2015.

[2] Mohammad Aazam, Pham Phuoc Hung, and Eui-Nam Huh. Smart gateway based communication for cloud of things. In *Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), 2014 IEEE Ninth International Conference on*, pages 1–6. IEEE, 2014.

[3] Sharad Agarwal, John Dunagan, Navendu Jain, Stefan Saroiu, Alec Wolman, and Harbinder Bhogan. Volley: Automated data placement for geo-distributed cloud services. In *NSDI*, pages 17–32, 2010.

[4] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Heartbeat: A timeout-free failure detector for quiescent reliable communication. In *International Workshop on Distributed Algorithms*, pages 126–140. Springer, 1997.

[5] Abdul Ahad, Mohammad Tahir, and Kok-Lim Alvin Yau. 5g-based smart healthcare network: Architecture, taxonomy, challenges and future research directions. *IEEE Access*, 7:100747–100762, 2019.

[6] Arif Ahmed and Ejaz Ahmed. A survey on mobile edge computing. In *Intelligent Systems and Control (ISCO), 2016 10th International Conference on*, pages 1–8. IEEE, 2016.

[7] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013.

[8] Sherif Akoush, Ripduman Sohan, Andrew Rice, Andrew W Moore, and Andy Hopper. Predicting the performance of virtual machine migration. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on*, pages 37–46. IEEE, 2010.

[9] Rachel Allen and Michael Li. Ranking popular distributed computing packages for data science. Accessed November. 2, 2020.

[10] Gayashan Amarasinghe, Marcos D de Assunçao, Aaron Harwood, and Shanika Karunasekera. A data stream processing optimisation framework for edge computing applications. In *2018 IEEE 21st International Symposium on Real-Time Distributed Computing (ISORC)*, pages 91–98. IEEE, 2018.

[11] Amazon. Amazon EC2 Dedicated Instances. https://aws.amazon.com/ec2/purchasing-options/dedicated-instances/. Accessed Nov. 11, 2016.

[12] Amazon. AWS Amazon EC2. https://aws.amazon.com. Accessed Jan. 4, 2016.

[13] Amazon. AWS Amazon EC2 On-demand Pricing. Accessed Jan. 4, 2016.

[14] Zubair Amjad, Axel Sikora, Benoit Hilt, and Jean-Philippe Lauffenburger. Low latency v2x applications and network requirements: Performance evaluation. In *2018 IEEE Intelligent Vehicles Symposium (IV)*, pages 220–225. IEEE, 2018.

[15] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear road: a stream data management benchmark. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 480–491. VLDB Endowment, 2004.

[16] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1383–1394, 2015.

[17] Azure. Azure iot edge. https://azure.microsoft.com/en-us/services/iot-edge/. Accessed Oct. 14, 2021.

[18] Michael Till Beck, Martin Werner, Sebastian Feld, and S Schimper. Mobile edge computing: A taxonomy. In *Proc. of the Sixth International Conference on Advances in Future Internet*. Citeseer, 2014.

[19] Viacheslav Belenko, Valery Chernenko, Vasiliy Krundyshev, and Maxim Kalinin. Data-driven failure analysis for the cyber physical infrastructures. In *2019 IEEE International Conference on Industrial Cyber Physical Systems (ICPS)*, pages 1–5. IEEE, 2019.

[20] Anton Beloglazov and Rajkumar Buyya. Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in cloud data centers. *Concurrency and Computation: Practice and Experience*, 24(13):1397–1420, 2012.

[21] Flavio Bonomi, Rodolfo Milito, Preethi Natarajan, and Jiang Zhu. Fog computing: A platform for internet of things and analytics. In *Big Data and Internet of Things: A Roadmap for Smart Environments*, pages 169–186. Springer, 2014.

[22] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh 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*, pages 13–16. ACM, 2012.

[23] David Breitgand, Gilad Kutiel, and Danny Raz. Cost-aware live migration of services in the cloud. In *SYSTOR*, 2010.

[24] David Breitgand, A Marashini, and Johan Tordsson. Policy-driven service placement optimization in federated clouds. *IBM Research Division, Tech. Rep*, 9:11–15, 2011.

[25] Andrey Brito, Christof Fetzer, and Pascal Felber. Multithreading-enabled active replication for event stream processing operators. In *2009 28th IEEE International Symposium on Reliable Distributed Systems*, pages 22–31. IEEE, 2009.

[26] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[27] Rajkumar Buyya, Rajiv Ranjan, and Rodrigo N Calheiros. Intercloud: Utility-oriented federation of cloud computing environments for scaling of application services. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 13–31. Springer, 2010.

[28] Miguel Caballer, Carlos De Alfonso, Fernando Alvarruiz, and GermáN Moltó. Ec3: Elastic cloud computing cluster. *Journal of Computer and System Sciences*, 79(8):1341–1351, 2013.

[29] Rodrigo N Calheiros, Enayat Masoumi, Rajiv Ranjan, and Rajkumar Buyya. Workload prediction using arima model and its impact on cloud applications' qos. *IEEE Transactions on Cloud Computing*, 3(4):449–458, 2015.

[30] Colin Camerer. *Behavioral game theory: Experiments in strategic interaction*. Princeton University Press, 2003.

[31] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[32] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. Optimal operator placement for distributed stream processing applications. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, pages 69–80. ACM, 2016.

[33] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. Optimal operator replication and placement for distributed stream processing systems. *ACM SIGMETRICS Performance Evaluation Review*, 44(4):11–22, 2017.

[34] Valeria Cardellini, Francesco Lo Presti, Matteo Nardelli, and Gabriele Russo Russo. Optimal operator deployment and replication for elastic distributed data stream processing. *Concurrency and Computation: Practice and Experience*, 30(9):e4334, 2018.

[35] Emanuele Carlini, Massimo Coppola, Patrizio Dazzi, Laura Ricci, and Giacomo Righetti. Cloud federations in contrail. In *European Conference on Parallel Processing*, pages 159–168. Springer, 2011.

[36] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*, pages 725–736, 2013.

[37] Hyeong Soo Chang, Michael C Fu, Jiaqiao Hu, and Steven I Marcus. An adaptive sampling algorithm for solving markov decision processes. *Operations Research*, 53(1):126–139, 2005.

[38] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, et al. Benchmarking streaming computation engines: Storm, flink and spark streaming. In *2016 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*, pages 1789–1792. IEEE, 2016.

[39] Mosharaf Chowdhury and Ion Stoica. Coflow: a networking abstraction for cluster applications. In *HotNets*, pages 31–36, 2012.

[40] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286. USENIX Association, 2005.

[41] Edward H Clarke. Multipart pricing of public goods. *Public choice*, 11(1):17–33, 1971.

[42] Alexandre da Silva Veith, Marcos Dias de Assuncao, and Laurent Lefevre. Latency-aware placement of data stream analytics on edge computing. In *International Conference on Service-Oriented Computing*, pages 215–229. Springer, 2018.

[43] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[44] Cuong T Do, Nguyen H Tran, Chuan Pham, Md Golam Rabiul Alam, Jae Hyeok Son, and Choong Seon Hong. A proximal algorithm for joint resource allocation and minimizing carbon footprint in geo-distributed fog computing. In *Information Networking (ICOIN), 2015 International Conference on*, pages 324–329. IEEE, 2015.

[45] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[46] Dror Feitelson. Logs of real parallel workloads. http://www.cs.huji.ac.il/labs/parallel/workload/. Accessed Aug. 29, 2017.

[47] Ana Juan Ferrer, Francisco HernáNdez, Johan Tordsson, Erik Elmroth, Ahmed Ali-Eldin, Csilla Zsigri, RaüL Sirvent, Jordi Guitart, Rosa M Badia, Karim Djemame, et al. Optimis: A holistic approach to cloud service provisioning. *Future Generation Computer Systems*, 28(1):66–77, 2012.

[48] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. Dhalion: self-regulating stream processing in heron. *Proceedings of the VLDB Endowment*, 10(12):1825–1836, 2017.

[49] Xinwei Fu, Talha Ghaffar, James C Davis, and Dongyoon Lee. Edgewise: a better stream processing engine for the edge. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, pages 929–945, 2019.

[50] Pedro Garcia Lopez, Alberto Montresor, Dick Epema, Anwitaman Datta, Teruo Higashino, Adriana Iamnitchi, Marinho Barcellos, Pascal Felber, and Etienne Riviere. Edge-centric computing: Vision and challenges. *ACM SIGCOMM Computer Communication Review*, 45(5):37–42, 2015.

[51] Michael R Gary and David S Johnson. Computers and intractability: A guide to the theory of np-completeness, 1979.

[52] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, page 323–336, USA, 2011. USENIX Association.

[53] Andrew V Goldberg and Jason D Hartline. Competitiveness via consensus. In *SODA*, volume 3, pages 215–222, 2003.

[54] Rohitha Goonatilake and Rafic A Bachnak. Modeling latency in a network distribution. *Network and Communication Technologies*, 1(2):1, 2012.

[55] Albert Greenberg, James Hamilton, David A Maltz, and Parveen Patel. The cost of a cloud: research problems in data center networks. *ACM SIGCOMM computer communication review*, 39(1):68–73, 2008.

[56] Theodore Groves. Incentives in teams. *Econometrica: Journal of the Econometric Society*, pages 617–631, 1973.

[57] Lin Gu, Deze Zeng, Peng Li, and Song Guo. Cost minimization for big data processing in geo-distributed data centers. *IEEE Transactions on Emerging Topics in Computing*, 2(3):314–323, 2014.

[58] John L Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[59] Sijin He, Li Guo, and Yike Guo. Real time elastic cloud management for limited resources. In *2011 IEEE 4th International Conference on Cloud Computing*, pages 622–629. IEEE, 2011.

[60] Thomas Heinze, Mariam Zia, Robert Krahn, Zbigniew Jerzak, and Christof Fetzer. An adaptive replication scheme for elastic data stream processing systems. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, pages 150–161, 2015.

[61] Thomas Hiessl, Vasileios Karagiannis, Christoph Hochreiner, Stefan Schulte, and Matteo Nardelli. Optimal placement of stream processing operators in the fog. In *2019 IEEE 3rd International Conference on Fog and Edge Computing (ICFEC)*, pages 1–10. IEEE, 2019.

[62] Urs Hunkeler, Hong Linh Truong, and Andy Stanford-Clark. Mqtt-s—a publish/subscribe protocol for wireless sensor networks. In *2008 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE'08)*, pages 791–798. IEEE, 2008.

[63] Jeong-Hyon Hwang, Sanghoon Cha, Uğur Cetintemel, and Stan Zdonik. Borealis-r: a replication-transparent stream processing system for wide-area monitoring applications. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1303–1306, 2008.

[64] IBM. Bluemix Dedicated. Accessed Nov. 7, 2016.

[65] Mary Inaba, Naoki Katoh, and Hiroshi Imai. Applications of weighted voronoi diagrams and randomization to variance-based k-clustering. In *Proceedings of the tenth annual symposium on Computational geometry*, pages 332–339. ACM, 1994.

[66] Bukhary Ikhwan Ismail, Ehsan Mostajeran Goortani, Mohd Bazli Ab Karim, Wong Ming Tat, Sharipah Setapa, Jing Yuan Luke, and Ong Hong Hoe. Evaluation of docker as edge computing platform. In *Open Systems (ICOS), 2015 IEEE Confernece on*, pages 130–135. IEEE, 2015.

[67] Arthur B Kahn. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562, 1962.

[68] Mudabbir Kaleem, Anastasia Mavridou, and Aron Laszka. Vyper: A security comparison with solidity based on common vulnerabilities. In *2020 2nd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*, pages 107–111. IEEE, 2020.

[69] Chuanqi Kan. Docloud: An elastic cloud platform for web applications based on docker. In *2016 18th international conference on advanced communication technology (ICACT)*, pages 478–483. IEEE, 2016.

[70] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. Benchmarking distributed stream data processing systems. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 1507–1518. IEEE, 2018.

[71] Arijit Khan, Xifeng Yan, Shu Tao, and Nikos Anerousis. Workload characterization and prediction in the cloud: A multiple time series approach. In *Network Operations and Management Symposium (NOMS), 2012 IEEE*, pages 1287–1294. IEEE, 2012.

[72] Rob Kitchin. The real-time city? big data and smart urbanism. *GeoJournal*, 79(1):1–14, 2014.

[73] Dalibor Klusáček and Hana Rudová. Alea 2: job scheduling simulator. In *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*, page 61. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2010.

[74] Derrick Kondo, Bahman Javadi, Paul Malecot, Franck Cappello, and David P Anderson. Cost-benefit analysis of cloud computing versus desktop grids. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12. IEEE, 2009.

[75] Jonathan Koomey, Kenneth Brill, Pitt Turner, John Stanley, and Bruce Taylor. A simple model for determining true total cost of ownership for data centers. *Uptime Institute White Paper, Version*, 2:2007, 2007.

[76] Parul Kudtarkar, Todd F DeLuca, Vincent A Fusaro, Peter J Tonellato, and Dennis P Wall. Cost-effective cloud computing: a case study using the comparative genomics tool, roundup. *Evolutionary Bioinformatics*, 6:197, 2010.

[77] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 239–250, 2015.

[78] Karthik Kumar and Yung-Hsiang Lu. Cloud computing for mobile users: Can offloading computation save energy? *Computer*, 43(4):51–56, 2010.

[79] Hongxing Li, Chuan Wu, Zongpeng Li, and Francis CM Lau. Profit-maximizing virtual machine trading in a federation of selfish clouds. In *2013 Proceedings IEEE INFOCOM*, pages 25–29. IEEE, 2013.

[80] Lihong Li, Wei Chu, John Langford, and Robert E Schapire. A contextual-bandit approach to personalized news article recommendation. In *Proceedings of the 19th international conference on World wide web*, pages 661–670, 2010.

[81] Lixing Li, Zhi Jin, Ge Li, Liwei Zheng, and Qiang Wei. Modeling and analyzing the reliability and cost of service composition in the iot: A probabilistic approach. In *2012 IEEE 19th International Conference on Web Services*, pages 584–591. IEEE, 2012.

[82] Teng Li, Zhiyuan Xu, Jian Tang, and Yanzhi Wang. Model-free control for distributed stream data processing using deep reinforcement learning. *Proceedings of the VLDB Endowment*, 11(6):705–718, 2018.

[83] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael Jordan, and Ion Stoica. Rllib: Abstractions for distributed reinforcement learning. In *International Conference on Machine Learning*, pages 3053–3062. PMLR, 2018.

[84] Hui Lin, Zetao Yang, Zicong Hong, Shenghui Li, and Wuhui Chen. Smart contract-based hierarchical auction mechanism for edge computing in blockchain-empowered iot. In *2020 IEEE 21st International Symposium on" A World of Wireless, Mobile and Multimedia Networks"(WoWMoM)*, pages 147–156. IEEE, 2020.

[85] Haikun Liu, Hai Jin, Cheng-Zhong Xu, and Xiaofei Liao. Performance and energy modeling for live migration of virtual machines. *Cluster computing*, 16(2):249–264, 2013.

[86] Eliane Maria Loiola, Nair Maria Maia de Abreu, Paulo Oswaldo Boaventura-Netto, Peter Hahn, and Tania Querido. A survey for the quadratic assignment problem. *European journal of operational research*, 176(2):657–690, 2007.

[87] Tom H Luan, Longxiang Gao, Zhi Li, Yang Xiang, Guiyi Wei, and Limin Sun. Fog computing: Focusing on mobile users at the edge. *arXiv preprint arXiv:1502.01815*, 2015.

[88] Hongzi Mao, Parimarjan Negi, Akshay Narayan, Hanrui Wang, Jiacheng Yang, Haonan Wang, Ryan Marcus, Ravichandra Addanki, Mehrdad Khani Shirkoohi, Songtao He, et al. Park: An open platform for learning-augmented computer systems. *Advances in Neural Information Processing Systems 32 (NIPS 2019)*, 2019.

[89] André Martin, Tiaraju Smaneoto, Tobias Dietze, Andrey Brito, and Christof Fetzer. User-constraint and self-adaptive fault tolerance for event stream processing systems. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 462–473. IEEE, 2015.

[90] R Preston McAfee. A dominant strategy double auction. *Journal of economic Theory*, 56(2):434–450, 1992.

[91] A Stephen McGough, Clive Gerrard, Paul Haldane, Dave Sharples, Dan Swan, Paul Robinson, Sindre Hamlander, and Stuart Wheater. Intelligent power management over large clusters. In *Proceedings of the 2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, pages 88–95. IEEE Computer Society, 2010.

[92] A Stephen McGough, Clive Gerrard, Jonathan Noble, Paul Robinson, and Stuart Wheater. Analysis of power-saving techniques over a large multi-use cluster. In *Dependable, Autonomic and Secure Computing (DASC), 2011 IEEE Ninth International Conference on*, pages 364–371. IEEE, 2011.

[93] Seapahn Meguerdichian, Farinaz Koushanfar, Gang Qu, and Miodrag Potkonjak. Exposure in wireless ad-hoc sensor networks. In *Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 139–150. ACM, 2001.

[94] Peter Mell and Tim Grance. The nist definition of cloud computing. *National Institute of Science and Technology, Special Publication*, 800:145, 2011.

[95] Gabriele Mencagli, Patrizio Dazzi, and Nicolò Tonci. Spinstreams: a static optimization tool for data stream processing applications. In *Proceedings of the 19th International Middleware Conference*, pages 66–79, 2018.

[96] Zili Meng, Minhu Wang, Jiasong Bai, Mingwei Xu, Hongzi Mao, and Hongxin Hu. Interpreting deep learning-based networking systems. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 154–171, 2020.

[97] Microsoft. Azure regions. Accessed Oct. 23, 2017.

[98] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.

[99] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[100] Rafael Moreno-Vozmediano, Rubén S Montero, and Ignacio M Llorente. Iaas cloud architecture: From virtualized datacenters to federated cloud infrastructures. *Computer*, 45(12):65–72, 2012.

[101] Thaha Muhammed, Rashid Mehmood, Aiiad Albeshri, and Iyad Katib. Ubehealth: a personalized ubiquitous cloud and edge-enabled networked healthcare system for smart cities. *IEEE Access*, 6:32258–32285, 2018.

[102] Roger B Myerson and Mark A Satterthwaite. Efficient mechanisms for bilateral trading. *Journal of economic theory*, 29(2):265–281, 1983.

[103] Matteo Nardelli, Valeria Cardellini, Vincenzo Grassi, and Francesco LO PRESTI. Efficient operator placement for distributed data stream processing applications. *IEEE Transactions on Parallel and Distributed Systems*, 2019.

[104] Xiang Ni, Jing Li, Mo Yu, Wang Zhou, and Kun-Lung Wu. Generalizable resource allocation in stream processing via deep reinforcement learning. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 857–864. AAAI Press, 2020.

[105] Noam Nisan et al. Introduction to mechanism design (for computer scientists). *Algorithmic game theory*, 9:209–242, 2007.

[106] Noam Nisan, Tim Roughgarden, Eva Tardos, and Vijay V Vazirani. *Algorithmic game theory*, volume 1. Cambridge University Press Cambridge, 2007.

[107] Soh K Ong and Andrew Yeh Chris Nee. *Virtual and augmented reality applications in manufacturing.* Springer Science & Business Media, 2013.

[108] Martin J Osborne and Ariel Rubinstein. *A course in game theory.* MIT press, 1994.

[109] Claus Pahl and Brian Lee. Containers and clusters for edge cloud architectures–a technology review. In *Future Internet of Things and Cloud (FiCloud), 2015 3rd International Conference on*, pages 379–386. IEEE, 2015.

[110] Dimitris Palyvos-Giannas, Vincenzo Gulisano, and Marina Papatriantafilou. Genealog: Fine-grained data streaming provenance at the edge. In *Proceedings of the 19th International Middleware Conference*, pages 227–238. ACM, 2018.

[111] Milan Patel, B Naughton, C Chan, N Sprecher, S Abeta, A Neal, et al. Mobile-edge computing introductory technical white paper. *White Paper, Mobile-edge Computing (MEC) industry initiative*, 2014.

[112] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. Network-aware operator placement for stream-processing systems. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 49–49. IEEE, 2006.

[113] F. Pisani, J. R. Brunetta, V. M. d. Rosario, and E. Borin. Beyond the fog: Bringing cross-platform code execution to constrained iot devices. In *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 17–24, Oct 2017.

[114] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Paramvir Bahl, and Ion Stoica. Low latency geo-distributed data analytics. *ACM SIGCOMM Computer Communication Review*, 45(4):421–434, 2015.

[115] Benny Rochwerger, David Breitgand, Eliezer Levy, Alex Galis, Kenneth Nagin, Ignacio M Llorente, Ruben Montero, Yaron Wolfsthal, Erik Elmroth, Juan Caceres, et al. The reservoir model and architecture for open federated cloud computing. *IBM Journal of Research and Development*, 53(4):4–1, 2009.

[116] Ivan Rodero, Francesc Guim, and Julita Corbalan. Evaluation of coordinated grid scheduling strategies. In *High Performance Computing and Communications, 2009. HPCC'09. 11th IEEE International Conference on*, pages 1–10. IEEE, 2009.

[117] Heejun Roh, Cheoulhoon Jung, Wonjun Lee, and Ding-Zhu Du. Resource pricing game in geo-distributed clouds. In *INFOCOM, 2013 Proceedings IEEE*, pages 1519–1527. IEEE, 2013.

[118] Gabriele Russo Russo, Valeria Cardellini, and Francesco Lo Presti. Reinforcement learning based policies for elastic stream processing on heterogeneous resources. In *Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems*, pages 31–42, 2019.

[119] Felix Salfner, Maren Lenk, and Miroslaw Malek. A survey of online failure prediction methods. *ACM Computing Surveys (CSUR)*, 42(3):1–42, 2010.

[120] Mahadev Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, 2017.

[121] Mahadev Satyanarayanan, Zhuo Chen, Kiryong Ha, Wenlu Hu, Wolfgang Richter, and Padmanabhan Pillai. Cloudlets: at the leading edge of mobile-cloud convergence. In *Mobile Computing, Applications and Services (MobiCASE), 2014 6th International Conference on*, pages 1–9. IEEE, 2014.

[122] Mahadev Satyanarayanan, Pieter Simoens, Yu Xiao, Padmanabhan Pillai, Zhuo Chen, Kiryong Ha, Wenlu Hu, and Brandon Amos. Edge analytics in the internet of things. *IEEE Pervasive Computing*, 14(2):24–31, 2015.

[123] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[124] Zoe Sebepou and Kostas Magoutis. Cec: Continuous eventual checkpointing for data stream processing operators. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, pages 145–156. IEEE, 2011.

[125] Omar Sefraoui, Mohammed Aissaoui, and Mohsine Eleuldj. Openstack: toward an open-source solution for cloud computing. *International Journal of Computer Applications*, 55(3):38–42, 2012.

[126] Pierluigi Siano. Demand response and smart grids—a survey. *Renewable and Sustainable Energy Reviews*, 30:461–478, 2014.

[127] William J Stewart. *Probability, Markov chains, queues, and simulation*. Princeton university press, 2009.

[128] Li Su and Yongluan Zhou. Passive and partially active fault tolerance for massively parallel stream processing engines. *IEEE Transactions on Knowledge and Data Engineering*, 31(1):32–45, 2017.

[129] Omer Subasi, Osman Unsal, and Sriram Krishnamoorthy. Automatic risk-based selective redundancy for fault-tolerant task-parallel hpc applications. In *Proceedings of the Third International Workshop on Extreme Scale Programming Models and Middleware*, pages 1–8, 2017.

[130] Amazon. AWS Lambda. https://aws.amazon.com/lambda/. Accessed Oct. 14, 2021.

[131] Amazon AWS. AWS IoT Greengrass. https://aws.amazon.com/greengrass/. Accessed Oct. 14, 2021.

[132] Apache Edgent. Apache edgent. http://edgent.apache.org/. Accessed April. 12, 2018.

[133] Apache Flink. Apache flink. Accessed August 24, 2020.

[134] Apache Kafka. Apache kafka. https://kafka.apache.org/. Accessed April. 12, 2018.

[135] Apache Storm. Apache storm. Accessed August 24, 2020.

[136] AWS. AWS Regional Data Centers mapping. https://github.com/turnkeylinux/aws-datacenters. Accessed Jan. 4, 2016.

[137] Google Cloud. Google Cloud. Accessed Oct. 27, 2021.

[138] IHS Markit. The internet of things: A movement, not a market. https://ihsmarkit.com/Info/1017/Internet-of-things.html. Accessed November. 2, 2020.

[139] National Grid. Large General TOU. Accessed Jan. 4, 2016.

[140] Rinkeby Ethereum Testnet. Rinkeby: Ethereum Testnet. https://www.rinkeby.io/#stats. Accessed Nov. 14, 2021.

[141] Liang Tong, Yong Li, and Wei Gao. A hierarchical edge cloud architecture for mobile computing. In *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9. IEEE, 2016.

[142] Prasang Upadhyaya, YongChul Kwon, and Magdalena Balazinska. A latency and fault-tolerance optimizer for online parallel query plans. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 241–252, 2011.

[143] Luis M Vaquero and Luis Rodero-Merino. Finding your way in the fog: Towards a comprehensive definition of fog computing. *ACM SIGCOMM Computer Communication Review*, 44(5):27–32, 2014.

[144] William Vickrey. Counterspeculation, auctions, and competitive sealed tenders. *The Journal of finance*, 16(1):8–37, 1961.

[145] Huayong Wang, Li-Shiuan Peh, Emmanouil Koukoumidis, Shao Tao, and Mun Choon Chan. Meteor shower: A reliable stream processing system for commodity data centers. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 1180–1191. IEEE, 2012.

[146] Lin Wang, Lei Jiao, Ting He, Jun Li, and Max Mühlhäuser. Service entity placement for social virtual reality applications in edge computing. In *Proc. INFOCOM*, 2018.

[147] Shangguang Wang, Yali Zhao, Jinlinag Xu, Jie Yuan, and Ching-Hsien Hsu. Edge server placement in mobile edge computing. *Journal of Parallel and Distributed Computing*, 2018.

[148] Roy Want, Bill N Schilit, and Scott Jenson. Enabling the internet of things. *Computer*, 48(1):28–35, 2015.

[149] Chris Whong. Foiling nyc's taxi trip data. *FOILing NYCs Taxi Trip Data. Np*, 18, 2014.

[150] Wikipedia. Double auction. https://en.wikipedia.org/wiki/Double_auction. Accessed Aug. 29, 2017.

[151] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[152] Timothy Wood, K. K. Ramakrishnan, Prashant Shenoy, Jacobus Van der Merwe, Jinho Hwang, Guyue Liu, and Lucas Chaufournier. Cloudnet: Dynamic pooling of cloud resources by live wan migration of virtual machines. *IEEE/ACM Transactions on Networking*, 23(5):1568–1583, 2015.

[153] Yongwei Wu, Kai Hwang, Yulai Yuan, and Weiming Zheng. Adaptive workload prediction of grid performance in confidence windows. *IEEE Transactions on Parallel and Distributed Systems*, 21(7):925–938, 2010.

[154] Hong Xu and Baochun Li. Joint request mapping and response routing for geo-distributed cloud services. In *INFOCOM, 2013 Proceedings IEEE*, pages 854–862. IEEE, 2013.

[155] Jinlai Xu and Balaji Palanisamy. Cost-aware resource management for federated clouds using resource sharing contracts. In *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, pages 238–245. IEEE, 2017.

[156] Jinlai Xu and Balaji Palanisamy. Model-based reinforcement learning for elastic stream processing in edge computing. *to appear in 28th IEEE International Conference on High Performance Computing, Data, and Analytics (HiPC)*, 2021.

[157] Jinlai Xu and Balaji Palanisamy. Optimized contract-based model for resource allocation in federated geo-distributed clouds. *IEEE Transactions on Services Computing*, 14(2):530–543, 2021.

[158] Jinlai Xu, Balaji Palanisamy, Heiko Ludwig, and Qingyang Wang. Zenith: Utility-aware resource allocation for edge computing. In *2017 IEEE international conference on edge computing (EDGE)*, pages 47–54. IEEE, 2017.

[159] Jinlai Xu, Balaji Palanisamy, and Qingyang Wang. Resilient stream processing in edge computing. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 504–513. IEEE, 2021.

[160] Jinlai Xu, Balaji Palanisamy, Qingyang Wang, Heiko Ludwig, and Sandeep Gopisetty. Amnis: Optimized stream processing for edge computing. *Journal of Parallel and Distributed Computing*, 160:49–64, 2022.

[161] Kok-Kiong Yap, Murtaza Motiwala, Jeremy Rahe, Steve Padgett, Matthew Holliman, Gary Baldus, Marcus Hines, Taeeun Kim, Ashok Narayanan, Ankur Jain, et al. Taking the edge off with espresso: Scale, reliability and programmability for global internet peering. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 432–445. ACM, 2017.

[162] K Paul Yoon and Ching-Lai Hwang. *Multiple attribute decision making: an introduction*. Sage publications, 1995.

[163] Boyang Yu and Jianping Pan. Location-aware associated data placement for geo-distributed data-intensive applications. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 603–611. IEEE, 2015.

[164] Andrea Zanella, Nicola Bui, Angelo Castellani, Lorenzo Vangelista, and Michele Zorzi. Internet of things for smart cities. *IEEE Internet of Things journal*, 1(1):22–32, 2014.

[165] Aleksandr Zavodovski, Suzan Bayhan, Nitinder Mohan, Pengyuan Zhou, Walter Wong, and Jussi Kangasharju. Decloud: Truthful decentralized double auction for edge clouds. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 2157–2167. IEEE, 2019.

[166] Wanxin Zhang, Dongsheng Li, Ying Xu, and Yiming Zhang. Shuffle-efficient distributed locality sensitive hashing on spark. In *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 766–767. IEEE, 2016.

[167] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K Aguilera, and Jinyang Li. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 276–291, 2013.

[168] Zhe Zhang, Yu Gu, Fan Ye, Hao Yang, Minkyong Kim, Hui Lei, and Zhen Liu. A hybrid approach to high availability in stream processing systems. In *2010 IEEE 30th International Conference on Distributed Computing Systems*, pages 138–148. IEEE, 2010.

[169] Zhi Zhou, Fangming Liu, Yong Xu, Ruolan Zou, Hong Xu, John CS Lui, and Hai Jin. Carbon-aware load balancing for geo-distributed cloud services. In *2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 232–241. IEEE, 2013.