JANUS: Latency-Aware Traffic Scheduling for IoT Data Streaming in Edge Environments

Zhenyu Wen, *Senior Member, IEEE*, Renyu Yang, *Member, IEEE*, Bin Qian, Yubo Xuan, Lingling Lu, Zheng Wang, Hao Peng, Jie Xu, *Member, IEEE*, Albert Y. Zomaya, *Fellow, IEEE*, Rajiv Ranjan, *Senior Member, IEEE*

Abstract—This paper focuses on a simple, yet fundamental question of distributed edge computing: "how to handle IoT traffic with different levels of sensitivity and criticality by satisfying the application-specific latency constraints?" This question arises in the practical deployment of edge computing, where user data can arrive at a much faster rate than that they can be processed by an edge node. Addressing this question is critical for meeting the latency requirement for latency-sensitive applications, but existing approaches are inadequate to the problem. We present JANUS, a multi-level traffic scheduling system for managing multiple data streams with various degrees of latency constraints. At the edge node level, JANUS uses multi-level queues to manage data streams with different latency constraints. It then allocates the output bandwidth of the edge node according to the requirements of applications in different priority queues, aiming to reduce the queuing and processing delay of latency-sensitive streams while maximizing the edge-node throughput. At the network level, JANUS actively redirects incoming data streams to the less-loaded ones to achieve better network-wide load balance and improve the overall throughput. Experiments show that JANUS reduces the latency to only 16.6% of a non-priority based solution and improves the throughput by 1.7x of a state-of-the-art priority-aware data stream scheduling approach.

Index Terms—data streaming, latency, QoS, edge computing

1 INTRODUCTION

The rise of the Internet of Things (IoT) is making computing an integrated and ubiquitous part of society, enabling data to be collected, correlated and analyzed at an unprecedented scale. Concurrent to this development is the quick adoption of edge computing by deploying computing sources closer to end-users and data sources. Edge computing paradigm is highly attractive because it offers a cost-effective and scalable capability of aggregating and processing data of connected IoT devices and sensors before sending the data streams to remote clouds [1]. Edge nodes (aka. edge servers, IoT Edge gateways, etc.) operate as gateways and provide a smooth connection between such devices on the network and the cloud.

While promising, edge nodes with heterogeneous computing power and diverse network bandwidths [2] tend to cause a fundamental problem in IoT streaming management known as *throughput mismatch*. Incoming data are generated at a much faster pace than that can be processed by an edge node. This mismatch is more frequently manifested due to high data volume or data spike (e.g., crowd gathering) of data-intensive IoT applications, and unstable network connectivity (e.g., cellular network) between edge nodes and the cloud [3], [4], [5]. Consequently, data are massively buffered on the edge nodes and devastatingly increases the *end-to-end (E2E) latency* of data transfer – typically including transmission time over network links from data sources to cloud servers and waiting time (queuing and processing) on the edge nodes – in many real-life edge deployments.

Meanwhile, latency requirements vary substantially among different applications. For example, intelligent transport systems [6] aggregate traffic information from each road by means of CCTV, Piezoelectric Sensor and Radar Microwave Sensors to support urban road safety warning and traffic efficiency improvement. While the E2E latency for urban road safety is usually less than 100 ms, it can be relaxed to 100-500 ms in some IoT scenarios [7]. Particularly for a latency-sensitive application with *co-flow* [8] – a collection of parallel data flows to process and transmit – each edge node is responsible for transferring numerous co-flows of multiple applications generated from different IoT sensors. The E2E latency is determined by the last flow to complete and thus extremely susceptible to any delays within an edge node and over the network.

Congestion control approaches in traditional computer networks [9], [10], [11] require either heavy support from switches or modification of transmission protocols and OS kernel/application modules. Such drawbacks make it im-

- *Z. Wen and Yubo Xuan are with the School of Cyberspace Security, and the School of Information Engineering, Zhejiang University of Technology, Hangzhou, Zhejiang 310023 China. Email: {zhenyuwen,221122120292}@zjut.edu.cn*
- *R. Yang is with the School of Software, Beihang University, Beijing China. Email: renyu.yang@buaa.edu.cn.*
- *B. Qian and R. Ranjan are with the Computing Science and Internet of Things, Newcastle University, Newcastle, UK, E-mail: {b.qian3, raj.ranjan}@newcastle.ac.uk.*
- *L. Lu is with the College of Computer Science and Technology, Zhejiang University, Hangzhou, China, E-mail: lulingling@email.cufe.edu.cn*
- *Z. Wang and J. Xu are with the School of Computing, University of Leeds, Leeds LS2 9JT, UK. Email: {z.wang5, j.xu}@leeds.ac.uk.*
- *H. Peng is with School of Cyber Science and Technology, Beihang University, Beijing, China. E-mail: penghao@buaa.edu.cn*
- *A.Y.Zomaya is with the University of Sydney, Australia. E-mail: albert.zomaya@sydney.edu.au.*

This work was supported in part by MIIT of China (2105-370171-07-02-860873), S&T Program of Hebei Grant (20310101D) and UK EPSRC (EP/T01461X/1)
Manuscript received Jan 2023, revised July 2023, accepted xx 2023. (Corresponding author: Renyu Yang)

possible to deploy upon commodity hardware and to provide guaranteed bound on the latency and flow deadlines. QJUMP [12] was among the first attempts to tackle the package queuing delay and control network interference at the switch level. It prioritizes latency-sensitive applications by allowing data from higher-priority applications to jump the queue over packets from lower-priority ones. Although QJUMP is a good fit for traditional datacenters, it is unsuited for edge environments. QJUMP is a network-layer solution that requires full control over key network infrastructures such as switches that can be hardly fulfilled in edge environments. QJUMP also assumes a homogeneous computing and networking environment where all network devices have the same computational resources and capabilities. However, such an assumption does not hold in real-world edge environments. Furthermore, congestion control mechanisms in sensor networks [13] drop new arrival data packets when the buffer of an edge node is full or the network bandwidth is oversubscribed. However, such a passive strategy cannot actively prevent buffer overflow or bandwidth oversubscription by simply coordinating the resources used across a distributed network, inevitably leading to long-standing backlogging and information loss. Other network-layer load balancing algorithms [14], [15] lack the application-level communication semantics between groups of devices and hence fail to customize users' requirements and make the best use of edge nodes. Therefore, we are in great need of a lightweight, easy-to-deploy yet flexible streaming traffic management system that can tackle diverse requirements of latency and throughput sensitivity.

We present JANUS, a distributed and QoS-centric streaming traffic scheduling system to make the best utilization of bandwidth resources on edge nodes. It aims to reduce the queuing delay and maximize the QoS assurance of latency-sensitive applications without compromising the overall system throughput. To do so, we formulate two distinct yet interconnected optimization problems and tackle them in JANUS based on greedy-based heuristics for their speed and ability to adapt quickly to changing conditions at runtime: i) *At edge node level*, JANUS queue manager differentiates the traffic from different streams and manages them separately through a multiple-level priority queuing mechanism. JANUS then dynamically adjusts the allocated bandwidth to each queue if the estimated queue delay surpasses a configurable threshold. At the core of bandwidth allocation is to ensure the allocated bandwidth be large enough to clear both the backlogged records that have been awaiting delivery and the new records accumulated within a time interval within a predefined latency requirements. ii) *Above multiple edge nodes*, JANUS employs a global coordination mechanism for flow redirection. It detects bandwidth shortage due to traffic spiking by monitoring traffic and bandwidth usage at application level, and make a best-fit mapping between the bandwidth shortage and available bandwidth on idle nodes. Requests from high-priority queue with a larger amount of bandwidth shortage will be prioritized in the bandwidth redirection and edge nodes with larger available bandwidth will be first considered as the forwarding destination. Such an elastic and timely forwarding mechanism facilitates to break the barrier of local bandwidth capacity and mitigate

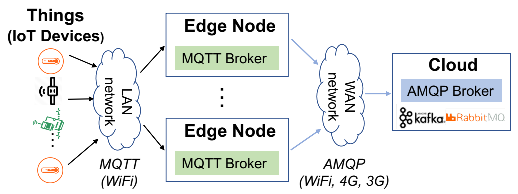


Fig. 1. Streaming pipeline in Cloud-edge computing paradigm

the mismatch issue, with the overall throughput improved.

JANUS is easy to deploy as it is agnostic to heterogeneous computing devices without the need of control access to internal resources of the network infrastructure. We evaluate JANUS in simulation settings and an edge testbed built upon real-world hardware. Experiments show that JANUS reduces the E2E latency by up to 5 and improves the system throughput by up to 1:5 over the state-of-the-art priority-aware data stream scheduling (e.g., QJUMP) in an emulated network environment. In a 4G network testbed, JANUS reduces the latency to only 16.6% of a native approach without priority-based queue management and improves the network-wide throughput by 1:7 against the existing priority-aware approach. We showcase that JANUS is lightweight because it uses only 9:5% RAM and 4% CPU time on a low-cost Raspberry Pi 3 development board.

This paper makes the following contributions:

- We formulate the streaming delivery in the edge environment as optimization problems to make the best utilization of edge nodes (§4).
- A novel bandwidth scheduling mechanism for adaptively allocating *output bandwidth* for individual edge nodes based on the node's forwarding capacity and the requirements of input data streams (§5.1).
- A traffic orchestrator to avoid oversubscribing edge nodes and improve the overall system throughput (§5.2).

Organization. §2 outlines the research background and motivation. §3 describes the system architecture and overview. §4 presents the formalized problems. Key techniques and system implementation of JANUS are discussed in §5 and §6, respectively. We evaluate the system in §7. §8 presents related works and we conclude the paper in §9.

2 BACKGROUND AND MOTIVATION

2.1 Streaming in Edge-Cloud Paradigm

Streaming Pipeline. Our work specifically targets the typical streaming management for IoT applications in edge environments. As depicted in Fig. 1, IoT devices (e.g., smartphones, sensors, actuators, etc.) connected to and exchange information with a proximate edge node (e.g., edge servers, IoT edge gateways) that serves as intermediary between IoT devices and cloud. Streaming messages are sent to edge nodes through the lightweight Message Queuing Telemetry Transport (MQTT) protocol [16] and then forwarded to remote Cloud servers through Advanced Message Queuing Protocol (AMQP) protocol [17] that can provide enhanced reliability. Edge nodes can be deployed in a variety of environments via wireless or cellular networks, particularly when the deployment involves remote or mobile IoT applications, including remote monitoring in such areas as natural reserves, oil rigs, rural farms, and smart transportation where edge nodes are usually deployed in a vehicle or robot.

TABLE 1
Network Latency Requirement of IoT Applications [7]

Applications	Tolerable Latency	Categories
Factory automation	0-50ms	Latency-sensitive (ls)
Smart grids	50-100ms	Normal (nm)
Road safety urban	50-100ms	
Traffic efficiency	100-500ms	
Cooling system	500-1000ms	Latency-tolerant (lt)

Latency Criticality. Some studies [7], [18], [19] exemplify a variety of IoT applications and Table 1 briefly summarizes their latency requirements. For example, factory automation applications are referred to as the real-time control of machines or systems in production lines and are generally considered to be highly latency-sensitive. By comparison, smart grids can allow longer delay to obtain the required data, while cooling and heating systems in smart buildings can tolerate much longer response time. According to the sensitivity of applications to E2E latency, in the context of this paper, we roughly categorize applications into three classes: latency-sensitive *(ls)*, latency-tolerant *(lt)* and normal *(nm)* applications. As will be discussed in §6.1, one can flexibly customize the threshold of each class and define fine-grained categorization.

2.2 Terminologies

We now summarize the key concepts used in this paper in terms of streaming applications and the network traffic generated from stream transmission.

Stream of records. In this paper, a data record is referred to as a key-value pair. One producer writes records and they can be read by one or more consumers. A stream is an ordered, unbounded and continuously-updating sequence of records.

Traffic. The amount of streams transmitted over the network during a given time period. In $[T; T + t]$, the traffic is measured by the number of records ($\#records$).

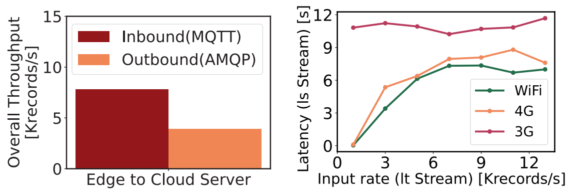
Bandwidth. This refers to the maximal allowed data transfer rate for a fixed period of time and can be practically measured by $\#records=s$. Bandwidth is physically used by all co-existing traffic streams. There are two types of bandwidth: *inbound* bandwidth is consumed when data streams come into the device/server while *outbound* bandwidth is referred to as the rate limit when the device/server sends out data records. In this paper, we mainly target the outbound bandwidth allocation among different streams to coordinate their flow rates.

Throughput and input/output rate. Throughput is the actual traffic rate, practically measured by the actual $\#records=s$. The *input rate* of a given stream literally depicts the inbound throughput, while *output rate* is referred to as the outbound throughput or equivalently throughput, which is most notably an indispensable performance indicator. The effect of bandwidth adjustment is to throttle the runtime traffic throughput.

2.3 Throughput Mismatch

As a motivating example, consider latency-sensitive (*ls*) and latency-tolerant (*lt*) data streams from the edge to the cloud.

Setup. This experiment is conducted on a micro-testbed consisting of three Raspberry Pi devices and a multi-core server. We use two Raspberry Pis to generate sensor data and another to act as an edge node to receive and forward the sensor data to the server. All the computing devices are connected through a dedicated WiFi enabled switch with 200 mbps bandwidth; we control the latency and bandwidth of the network to emulate the typical 3G, 4G and WiFi environments. More details of our testbed can be found in §7.1. We use a large-scale real-world smart building dataset [20] consisting of CO_2, occupancy and temperature data samples, to generate the data. We measure the end-to-end *throughput* of an edge node when the data are forwarded to the server via the edge node.



(a) End to end throughput (b) Message delivery latency

Fig. 2. Throughput mismatch (WiFi) and the latency of *ls* streams

Motivation Results. Fig. 2(a) shows a substantial throughput mismatch between the inbound and outbound on a given edge node in a WiFi environment. The data arrive 2 faster than the amount that can be sent to the remote server. This is unsurprising because AMQP is, in general, slower than MQTT due to the overhead associated with its reliability guarantee. The delay is also because the incoming messages incur a significant CPU and memory footprint, utilizing 60% of the CPU and occupying over 70% of the RAM, which further limits the processing capability and responsiveness of the edge node. Fig. 2(b) shows that when the input rate of lt streams climbs up, the ls streams generally experience an increase in the latency. In reality, the network bandwidth could be increasingly occupied by the throughput of lt streams during peak time, which will in turn increase the forwarding latency of ls streams.

Negative Impact on Latency-sensitive Streams. Observably, throughput mismatch can severely slow down the responsiveness of latency-sensitive streams. This becomes even worse in a low-speed network (e.g., 3G cellular network in this experiment), as records coming into an edge node have to share bandwidth and await forwarding in the same queue. Latency-sensitive applications will suffer from a long-standing queue delay if all incoming data streams are treated equally without considering the application-specific latency criticality. To tackle this, we need a scheme to differentiate data streams and prioritize time-critical streams for reduced latency, and to coordinate available bandwidth across edge nodes to mitigate the inherent mismatch issue. JANUS is designed to offer such capabilities.

3 JANUS OVERVIEW

3.1 Requirements

This paper focuses on streaming data management for satisfying diverse latency requirements of different applications. Hardware or OS kernel based queuing management approaches [12] are inadequate for this problem due to the strong dependencies on switch-level support and intrinsic difficulties of development and deployment on commodity hardware[21]. We turn to look at application layer solutions

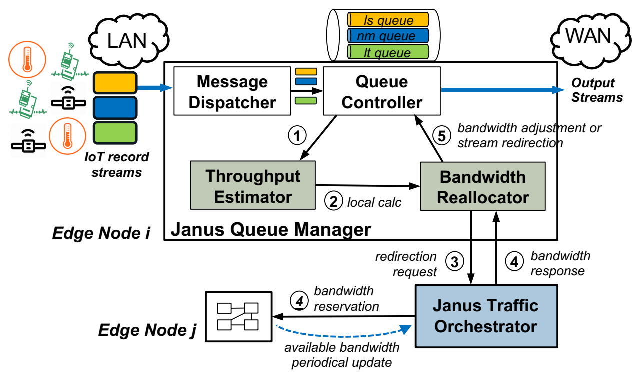


Fig. 3. JANUS System Architecture

based on throughput control in the publish-subscribe subsystem. We aim to achieve three research objectives:

[R1] We need to differentiate and prioritize data streams, i.e., labelling the data records and prioritizing latency-sensitive applications over other workloads, to allow diverse application-specific latency requirements.

[R2] We need a traffic-aware and multi-level queue management mechanism for enabling individual management of a specific type of data records and for harnessing bandwidth allocation. Bandwidth of each queue can be dynamically throttled and flexibly allocated at runtime according to the varying traffic condition.

[R3] We need to harvest idle bandwidth resources to achieve a holistic network-level balance in response to the varying load across edge nodes. This can enable the re-distribution of overflowed streams, from a saturated edge node to light-loaded edge nodes, to mitigate the congestion and the consequential latency increase.

3.2 System Architecture

JANUS is a loosely-coupled streaming management system that aims to assure the QoS both locally within an edge node and globally across edge nodes. Fig. 3 depicts the system architecture consisting of a *Queue Manager* within each edge node and a global *Traffic Orchestrator* at the network level.

To satisfy **[R1]** and **[R2]**, JANUS primarily attempts to resolve the potential queuing delay by using multi-queue management within an edge node. JANUS categorizes streams into *three* distinct classes and assigns differential priorities according to the particular type of traffic and the associated QoS requirements. For example, ls and lt streams are given the highest and lowest priority, respectively. It is worth noting that this multi-priority queue model can be extended to underpin any number of priorities if the applications have additional fine-grained QoS requirements.

Queue Manager comprises three key components: *Message Dispatcher*, *Throughput Estimator*, and *Bandwidth Reallocator*. Specifically, *Message Dispatcher* sends off the arrived stream records into different queues. The records are then enqueued and buffered, awaiting forwarding to the endpoints of cloud servers. *Throughput Estimator* exploits runtime metrics of both throughput and bandwidth (either consumed or available) to estimate the flow status (step ①, §4.1). If a QoS violation is detected – the estimated delayed time surpasses the pre-defined threshold – bandwidth reallocation for each queue will be triggered (step ②). *Bandwidth*

TABLE 2
Symbol Notations

Symbols	Descriptions
E_i, I	edge node i and the index set of all edge nodes
$q^i_{ls}, q^i_{nm}, q^i_{lt}$	*ls*, *normal* and *lt* queue in E_i
$^{i}_{j}$	Individual input rate of stream j in E_i
Λ_i	Total input rate of E_i
$\Lambda^i_{ls}, \Lambda^i_{nm}, \Lambda^i_{lt}$	Total input rate of $q^i_{ls}, q^i_{nm}, q^i_{lt}$
$\mathcal{V}^i$	Total output rate of E_i
$\mathcal{V}^i_{ls}, \mathcal{V}^i_{nm}, \mathcal{V}^i_{lt}$	Total output rate of $q^i_{ls}, q^i_{nm}, q^i_{lt}$
C^i_{in}	Maximal inbound bandwidth of E_i
C^i_{out}	Maximal outbound bandwidth of E_i
$''$	Processing delay introduced by E_i
$tp^i_{ls}, tp^i_{nm}, tp^i_{lt}$	The throughput of queue $q^i_{ls}, q^i_{nm}, q^i_{lt}$
tp^i	The throughput of edge node i
w^i_{nm}, w^i_{lt}	The weights of the penalty function
$^{i}, '^{i}$	The variables for replacing max penalties
$\Phi(q^i_{lt}; p)$	The volume of the data from q^i_{lt} to E_p.
$d^i_{ls}, d^i_{nm}, d^i_{lt}$	Estimated queuing delay of $q^i_{ls}, q^i_{nm}, q^i_{lt}$
$s^i_{ls}, s^i_{nm}, s^i_{lt}$	# of existing records in $q^i_{ls}, q^i_{nm}, q^i_{lt}$
$^i_{ls}, ^i_{nm}, ^i_{lt}$	Maximal tolerable latency of $q^i_{ls}, q^i_{nm}, q^i_{lt}$
$bw^i_{ls}, bw^i_{nm}, bw^i_{lt}$	Inferred bandwidth requirement of $q^i_{ls}, q^i_{nm}, q^i_{lt}$
P_i	Available nodes to receive the redirected traffic

Reallocator is responsible for calculating the bandwidth adjustment among different types of queues to better utilize local bandwidth resources to guarantee the QoS of high-priority streams such as ls streams. This is achieved by theoretically formulating an Integer Linear Programming (ILP) problem (§4.2.1) and practically using a heuristic solution to find a proper amount of bandwidth that is enough to clear the backlogged records in the queue and satisfy the queue latency requirement (§5.1).

To meet **[R3]**, we formulate the global traffic scheduling as another ILP problem (§4.2.2). A global orchestrator is devised to harness all available bandwidth resources across the system in response to traffic redirection requests. If an edge node is saturated, *Bandwidth Reallocator* proactively petitions *Traffic Orchestrator* by submitting a redirection request. The request encompasses the amount of bandwidth that is still required by the sourcing queue to mitigate the long data backlogging (step ③). *Traffic Orchestrator* is a key component holistically responsible for making high-level coordination over edge nodes. Three replicas of Traffic Orchestrator are deployed in edge nodes for high availability, with a primary component and others on hot standby. It globally hunts for a suitable bandwidth matching based on available bandwidth periodically reported by all running edge nodes or piggybacked in the heartbeat messages between edge nodes and the Orchestrator. The results will be returned to the corresponding *Bandwidth Reallocator* (step ④) and *Queue Controller* then tweaks the bandwidth among queues in the edge node or to establish a connection to migrate the pending records to other edge nodes (step⑤). More details will be presented in §5.2.

4 PROBLEM FORMULATION

In this section, we firstly demonstrate how to model and estimate the runtime queuing delay of each queue based on the data streams on the fly and the real-time output rate. We then progressively introduce how we formulate the *two-level* optimization problem at the local edge node level and at the global orchestrator level to maximize the throughput of the entire system whilst ensuring proper QoS of latency-sensitive streams. To improve readability, we summarize detailed notations used in this paper in Table 2.

4.1 Runtime Queue Delay and Throughput Estimation

JANUS works on the basis of estimating queue delay and throughput at runtime. We partition all data records into separate queues according to their sensitivity to latency. As an example, we use three queues for *latency-sensitive*, *normal* and *latency-tolerant* streams. We assume that an edge node E_i receives a collection of streams with an *input rate* λ^i_j for the stream j. The total *input rate* of an edge node E_i is $\lambda^i = \sum_{j \in N} \lambda^i_j$, where N is the total number of the streams. The *input rate* of each queue is defined by: $\lambda^i_{ls} = \sum_{j \in N_{ls}} (\lambda^i_j)$, $\lambda^i_{nm} = \sum_{h \in N_{nm}} (\lambda^i_h)$ and $\lambda^i_{lt} = \sum_{k \in N_{lt}} (\lambda^i_k)$, separately, where N_{ls}, N_{nm} and N_{lt} represent the collection of ls, nm and lt streams respectively. Accordingly, the *output rates* of these three queues is defined as V^i_{ls} V^i_{nm} and V^i_{lt}. *Output rate* in total is $V^i = V^i_{ls} + V^i_{nm} + V^i_{lt}$. In reality, the maximum of *input rate* and *output rate* are constrained by the processing capacity of the edge node and the available network bandwidth between the edge node and the cloud. Hence, *input rate* λ^i and *output rate* V^i constraints can be formulated as $\lambda^i \le C^i_{in}$ and $V^i \le C^i_{out}$, where C^i_{in} and C^i_{out} represent the maximum inbound and outbound bandwidth of E_i. For example, for a *latency-sensitive* queue, with a time window Δt, $\Delta t \cdot \lambda^i_{ls}$ data records will be fed into q^i_{ls}. $\mathrm{Size}(q^i_{ls}) \in \mathbb{Z}^+$ is the queue length of q^i_{ls}. The total number of data passing through q^i_{ls} will be $s^i_{ls} = \Delta t \cdot \lambda^i_{ls} + \mathrm{Size}(q^i_{ls})$. As a result, the estimated queuing delay of the latency-sensitive queue d^i_{ls} can be calculated by:

$$d^i_{ls} = \frac{s^i_{ls}}{V^i_{ls}} + \varepsilon, \tag{1}$$

where ε is a bias variable which can be instantiated as the processing delay introduced by E_i. Since the actual throughput is co-restrained by the data amount passing through (s^i_{ls}) within the time window Δt and the inherent restriction of the queue itself (V^i_{ls}), tp^i_{ls} can be then expressed as $\min\{\frac{s^i_{ls}}{\Delta t}, V^i_{ls}\}$. Similarly, the overall throughput of a given node E_i can be formulated as Eq. 2.

$$\mathrm{tp}^i = \mathrm{tp}^i_{ls} + \mathrm{tp}^i_{nm} + \mathrm{tp}^i_{lt} \tag{2}$$

$$\begin{aligned} \mathrm{tp}^i_{ls} &= \min\{\frac{s^i_{ls}}{\Delta t}, V^i_{ls}\} \\ \mathrm{tp}^i_{lt} &= \min\{\frac{s^i_{lt}}{\Delta t}, V^i_{lt}\} \\ \mathrm{tp}^i_{nm} &= \min\{\frac{s^i_{nm}}{\Delta t}, V^i_{nm}\} \end{aligned} \tag{3}$$

4.2 Two-Level Optimization

4.2.1 Optimizing Throughput for a Local Edge Node

In a given edge node E_i, the optimization model aims to maximize the throughput of E_i whilst meeting a set of constraints:

$$\max \quad \mathrm{tp}^i \tag{4}$$
$$\text{s.t.} \quad d^i_{ls} \le \theta^i_{ls}, d^i_{nm} \le \theta^i_{nm}, d^i_{lt} \le \theta^i_{lt} \tag{5}$$
$$V^i \le C^i_{out} \tag{6}$$

As shown in Eq. (4), the optimization goal is to allocate the most suitable output bandwidth for each queue $\{V^i_{ls}, V^i_{lt}, V^i_{nm}\}$ to maximize the node-level throughput. Constraint (5) indicates the maximal delay allowed by each specific queue while satisfying the capacity constraint in Constraint (6). However, §2 shows that low computing capacity or low network bandwidth may cause the violation of constraint (5), resulting in a non-existing solution.

During bandwidth insufficiency, to allow soft constraints in lower-priority queues (*nm* and *lt* queues), we can practically relax Constraint (5) by incorporating the following penalty terms into the objective function:

$$\max \quad \mathrm{tp}^i - w^i_{nm}\max\{d^i_{nm} - \theta^i_{nm}, 0\} - w^i_{lt}\max\{d^i_{lt} - \theta^i_{lt}, 0\} \tag{7}$$
$$\text{s.t.} \quad \text{constraints}(3) \tag{8}$$
$$d^i_{ls} \le \theta^i_{ls} \tag{9}$$
$$w^i_{nm} \ge w^i_{lt} \tag{10}$$

Constraint (9) indicates a stringent latency guarantee for the *latency-sensitive* queue, while *normal* and *latency-tolerant* queues could tolerate a certain degree of constraint violation. w^i_{nm} and w^i_{lt} are the weights in the penalty function, and the values in constraint (10) indicate the partial order of the queue priorities and the relative importance of its constraints, e.g., it is of more importance to allocate available bandwidth to *nm* queue than *lt* queue. We then introduce two more variables ψ^i and φ^i to overcome the non-smoothness of the objective function revealed from the $\max$ penalties. Based on the bounds and direction of optimization, the penalties in the function can be relaxed further as follows:

$$\max \quad \mathrm{tp}^i - w_{nm}\psi^i - w_{lt}\varphi^i \tag{11}$$
$$\text{s.t.} \quad \text{constraints}\,(3), \text{and}\,(8) \tag{12}$$
$$\psi^i \ge d^i_{nm} - \theta^i_{nm}, \psi^i \ge 0 \tag{13}$$
$$\varphi^i \ge d^i_{lt} - \theta^i_{lt}, \varphi^i \ge 0 \tag{14}$$

4.2.2 Optimizing Global Throughput across Edge Nodes

We then formulate the procedure of data stream redirection at the overall network level. In the event of overflowed streams, the records in an existing data queue q^i_{ls} in E_i, for example, can be redirected to a set of destination nodes P_i that have available resources for the time being.

We make the following assumption.

Assumption 1:. The queuing data streams can be flexibly split and sent out to any edge nodes at a specific rate.

Assumption 2:. The edge nodes are connected with the local network which has a much smaller latency than the delay between edge nodes and the cloud.

We use $\zeta(q^i_{ls}, p) \in \mathbb{Z}^+$ to represent the volume of the data stream sent from q^i_{ls} to a destination edge node E_p where $p \in P_i$. Particularly, if p is equal to i, the portion of the data stream remains in the queue of the source node. The total throughput of q^i_{ls} is therefore $tp^i_{ls} = \frac{\sum_{p \in P_i} \zeta(q^i_{ls}, p)}{\Delta t}$. Correspondingly, the overall system throughput – considering all types of queues and all edge nodes in the system – can be calculated as follows:

$$\mathrm{TP} = \sum_{i 2 I}\ \sum_{p 2 P_i} \frac{(q^i_{ls}; p)}{t} + \frac{(q^i_{nm}; p)}{t} + \frac{(q^i_{lt}; p)}{t}$$
$$= \sum_{i 2 I} tp^i_{ls} + tp^i_{nm} + tp^i_{lt} \tag{15}$$

Hence, the final optimization problem will be:

$$\max \quad \mathrm{TP} \tag{16}$$
$$s{:}t{:}\quad d^i_{ls} \quad {}^i_{ls};\ d^i_{nm} \quad {}^i_{nm};\ d^i_{lt} \quad {}^i_{lt};\ 8i\ 2\ I \tag{17}$$
$$V^i \quad C^i_{out};\ 8i\ 2\ I \tag{18}$$
$$\sum_{p 2 P;8i 2 I} (q^i_{ls}; p) \quad s^i_{ls}; \tag{19}$$
$$\sum_{p 2 P;8i 2 I} (q^i_{nm}; p) \quad s^i_{nm}; \tag{20}$$
$$\sum_{p 2 P;8i 2 I} (q^i_{lt}; p) \quad s^i_{lt} \tag{21}$$

This paper assumes the available resources are sufficient to transfer the data streams to the cloud within a pre-defined domain of latency tolerance as indicated in constraint (17). Otherwise, we can relax the constraint by using the technique aforementioned in the local node's optimization. A solution to the objective (16) necessitates the specific value assignment to $\ 2\ \mathbb{Z}^+$ and $\ 2\ f0; 1g$.

This is an integer linear programming (ILP) problem and proved to be NP-hard [22] – Capital Budgeting problem [23], Knapsack problem [24], Traveling Salesperson problem [25]. While the strategies such as Branch-and-Bound, cutting plane can obtain an optimal solution for these problems, the time complexity of these algorithms is polynomial time. ILP solvers such as Gurobi [26] and CPLEX [27] are extremely time-consuming – even taking a few hours for small instances – and thus cannot be applied to satisfy the requirements of real-time streaming systems. We turn to heuristics design in a runtime traffic management system. Such algorithms are well-suited for problems particularly in the time-critical domains where real-time decisions are required to be made with low latency.

5 Key Techniques

To fulfill the aforementioned optimization objectives, Janus leverages two greedy-based heuristic algorithms. Once any queue's delay is detected exceeding a defined threshold, the throughput will be adjusted at each edge node level to achieve the objective (7). If any latency constraint of the queues is breached, *Traffic Orchestrator* will carry out a plan to achieve the objective (16).

5.1 Bandwidth Reallocation in an Edge Node

The key procedure of dynamic bandwidth allocation is to ascertain and diminish the discrepancy between the pre-defined latency threshold and the runtime queuing delay within a time window. To meet the constraints (8) and (9), we leverage a priority-based bandwidth reallocation mechanism for prioritizing latency-sensitive queue over other queues. At the core of the reallocation is to determine the minimal required bandwidth that is believed to satisfy the latency constraint of each type of stream, while diminishing the existing queue backlogging delay. The intuition behind this design is to leverage as less bandwidth as possible to fulfill a given latency requirement on a per queue basis (e.g., ${}^i_{ls}$), and thus to spare unused bandwidth on an edge node to handle bandwidth shortage on other busy neighbors.

Alg. 1 shows the detailed procedure: initializing from the time T, we evenly assign bandwidth to three queues. `Counter()` reads the monitored *input rate* of each queue during a time interval t (Line 3). Afterwards, we calculate the estimated queuing delay before comparing against the pre-defined threshold. If any expected queue delay is detected to surpass the threshold , bandwidth reallocation will be triggered, and `BWallocator` will rapidly alleviate throughput discrepancy between current allocation and the

Algorithm 1: Priority Based Throughput Throttle

```
PBTT()
    // Count the input rate of each queue within the time interval t
    $\Lambda^i_{ls}; \Lambda^i_{nm}; \Lambda^i_{lt}$   Counter($t$)
    // Compute the queuing delay for three queues
    $d^i_{ls}; d^i_{nm}; d^i_{lt}$   $\frac{s^i_{ls}}{\mathcal{V}^i_{ls}} + ";\ \frac{s^i_{nm}}{\mathcal{V}^i_{nm}} + ";\ \frac{s^i_{lt}}{\mathcal{V}^i_{lt}} + "$
    if $d^i_{ls} >\ {}^i_{ls}$ or $d^i_{nm} >\ {}^i_{nm}$ or $d^i_{lt} >\ {}^i_{lt}$ then
        // Reallocate the bandwidth for each queue
        $V^i_{ls}, V^i_{nm}, V^i_{lt}$   BWAllocator($\Lambda^i_{ls}; \Lambda^i_{nm}; \Lambda^i_{lt}$)
        return $V^i_{ls}, V^i_{nm}, V^i_{lt}$
    end
    else
        return NULL // Keep current bandwidth for each queue
    end
```

Algorithm 2: Bandwidth reallocation (E_i)

Input: $\Lambda^i_{ls}; \Lambda^i_{nm}; \Lambda^i_{lt}$: current input rate of each queue
$bw^i_{ls}, bw^i_{nm}, bw^i_{lt}$: current bandwidth allocation
$freebw^i$: available bandwidth

```
BWAllocator()
    // Predict the output rate requirement of each queue
    $bw^i_{ls}; bw^i_{nm}; bw^i_{lt}$   using Eq. 22
    if $bw^i_{ls}$   $C^i_{out}$ then
        $V^i_{ls}$   $bw^i_{ls}$
        if $bw^i_{nm}$   $C^i_{out}$   $V^i_{ls}$ then
            $V^i_{nm}$   $bw^i_{nm}$
            if $bw^i_{lt}$   $C^i_{out}$   $V^i_{ls}$   $V^i_{nm}$ then
                $V^i_{lt}$   $bw^i_{lt}$
                $freebw^i$   $C^i_{out}$   $V^i_{ls}$   $V^i_{nm}$   $V^i_{lt}$
                $V^i_{ls}$   $V^i_{ls} + freebw^i$
            end
            else
                BWRequest(0, 0, $V^i_{lt}$   $bw^i_{lt}$)
            end
        end
        else
            $V^i_{nm}$   $C^i_{out}$   $V^i_{ls}; V^i_{lt}$   0
            BWRequest(0, $V^i_{nm}$   $bw^i_{nm}$,   $bw^i_{lt}$)
        end
    end
    else
        $V^i_{ls}$   $C^i_{out}; V^i_{nm}$   0; $V^i_{lt}$   0
        BWRequest($V^i_{ls}$   $bw^i_{ls}$,   $bw^i_{nm}$,   $bw^i_{lt}$)
    end
    return $V^i_{ls}, V^i_{nm}, V^i_{lt}$
```

wanted allocation (Lines 6 - 10).

As shown in Alg. 2, `BWallocator` aims to allocate the finite amount of bandwidth to three prioritized queues. The decision depends upon both the existing and the estimated queuing length for the next time interval t. Ideally, the allocated bandwidth is desired to be large enough to clear both the backlogged records that have been awaiting delivery and the new records accumulating during the time interval within a predefined latency requirements.

$$bw^i_{ls} = \frac{s^i_{ls} + (\ ^i_{ls} \quad V^i_{ls}) \quad t}{_{ls}} + V^i_{ls} \tag{22}$$

As shown in Eq. 22, the estimated queuing accumulation is calculated by the difference between the *input rate* $^i_{ls}$ and *output rate* V^i_{ls} times t. After the ideal amount of bandwidth of each queue is calculated, we proceed to the realistic bandwidth re-allocation. The *ls* queue has the highest priority to obtain all possibly available bandwidth C^i_{out}, followed by the *nm* and *lt* queue, respectively. This is aligned with the weight associated with each soft constraint defined in constraint (10). In the event of bandwidth shortage, bandwidth request will be generated by `BWallocator` via `Update()` and bandwidth allocation will take effect on each queue.

5.2 Global Traffic Coordination across Edge Nodes

Traffic Orchestrator coordinates the imbalanced data streams among edge nodes and maps the redirection request from a saturated edge node onto other edge nodes that currently have sufficient bandwidth resources. Its core responsibility is to find the most suitable match between the waiting requests and available bandwidth resources that can achieve the objective (16).

5.2.1 Bandwidth Request and Response

To simplify the bandwidth requirement of each edge node in the scheduling model, we leverage the uniformed *3-attributes tuple* $(b_{ls}; b_{nm}; b_{lt})$ to represent the following bandwidth operations for different queues:

Available/Allocated Bandwidth: the amount of available bandwidth that can be provisioned by different queues in an edge node, and the current bandwidth allocation among different queues in an edge node;

Bandwidth Request/Response: the amount of requested bandwidth (equivalent to the traffic shortage or traffic for redirection) and the amount of bandwidth granted to an edge node for redirecting pending stream records.

We use a positive value to indicate the available resource and a negative value to imply the requested resource to resolve the current bandwidth shortage. For instance, one could request $[0; \quad 5; 0]$, indicating 5 units bandwidth shortage in the second-level queue. The pertaining traffic would be, ideally, redirected to other edge nodes to avoid the latency violation. An edge node labelled $[0; 0; 10]$ can lend out 10 units bandwidth to rescue the traffic delay.

Note that, by design, the available bandwidth tuple (the unallocated bandwidth pertaining to each queue) is collected and reported periodically from each edge node to Orchestrator through an independent thread sitting in the edge node, or piggybacked in the heartbeat message between edge nodes and Orchestrator.

Algorithm 3: PMRFS-based Traffic Redirection

```
1  redirection(bw_requests, bw_availables)
2      // Descend Sort of required bandwidth and available bandwidth
3      Receivers   PriorityFirstSort(bw_requests)
4      Providers   Sort(bw_availables)
5      forall $P_j$ 2 Providers do
6          forall $R_i$ 2 Receivers do
7              if $freebw_{P_j}$   $reqbw_{R_i}$ then
8                  tmpbw   $reqbw_{R_i}$
9                  BWRespond($R_i$, $P_j$, tmpbw)
10                 BWReserve($P_j$, tmpbw)
11                 Receivers   Receivers n f $R_i$ g
12                 $freebw_{P_j}$   $freebw_{P_j}$   tmpbw
13             end
14             else
15                 tmpbw   $freebw_{P_j}$
16                 BWRespond($R_i$, $P_j$, tmpbw)
17                 BWReserve($P_j$, tmpbw)
18                 Providers   Providers n f $P_j$ g;
19                 $reqbw_{R_i}$   $reqbw_{R_i}$   tmpbw
20                 break
21             end
22         end
23     end
```

5.2.2 Prioritized Max Request First Served Heuristic

Key Idea. We categorize edge nodes into bandwidth provider and bandwidth receivers, according to the provision/shortage role, i.e., negative/positive value within the attribute tuple. Our previous algorithmic study on bin packing [28] concluded that Given that node-centric algorithms can achieve a good trade-off between running time and solution quality. Based on this, we propose an approximation *Redirection* mechanism, Prioritized Max-Request First Served (PMRFS) algorithm, to i) prioritize the bandwidth request from the high-priority queue with a larger amount of bandwidth shortage; and ii) consider edge nodes that has larger available bandwidth as the bandwidth providers. The intuition is that the proposed scheme can mitigate bandwidth shortage as soon as possible to reduce the redirection times whilst increasing the success possibility of bandwidth allocation, and thus maximize the overall throughput.

As shown in Alg. 3, we first sort both bandwidth request and available bandwidth in a descending order. Most notably, we employ a priority first sort (Line 3) to ensure that requests from higher priority queue can be ranked on top of requests of other queues, even if they may have smaller request amount. To achieve PMRFS, we firstly pick the head request from the sorted *Receivers* and pick the edge node with the largest available amount of bandwidth from the *Providers* set. If a single provider cannot completely satisfy a request, the satisfied fraction will be held by the receiver; meanwhile the receiver awaits next available providers until all its requested bandwidth is satisfied. It is worth noting that the held fraction will be incrementally assigned and Traffic Orchestrator will notify the edge node to redirect the given amount of traffic to the designated edge node (Lines 14-21). Likewise, if one provider from *Providers* has more bandwidth than the requested value of the current receiver, the remaining fraction can be further allocated to the next receiver (Lines 7-13). The iterative allocation will not cease until all receivers get the required bandwidth. Once Traffic

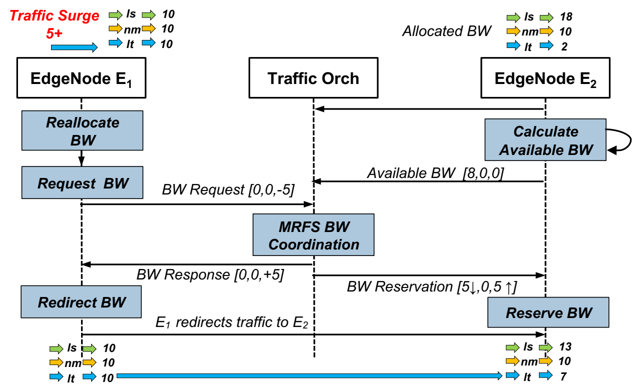


Fig. 4. Bandwidth request, response and reservation

Orchestrator makes a decision, the beneficiary node will forward a specific number of records from its queue to the destination node according to the allocated bandwidth. The destination node will reserve the bandwidth (Line 10) and prepare for receiving the forthcoming record streams.

Working Example. We use the following numerical example to showcase the algorithm procedure. To simplify the explanation, we selectively depict the interactions between two edge nodes (E_1 and E_2) and the centralized Traffic Orchestrator. Fig. 4 shows the details of bandwidth request and response among these components. More specifically, the current bandwidth allocation for E_1 and E_2 are [10,10,10] and [18,10,2], respectively. At time T, E_1 experiences a bursting traffic into the *latency-tolerant* queue and thus requires extra 5 units bandwidth to resolve the transient latency issue. The bandwidth request [0,0,-5] will be sent to the orchestrator. Orchestrator will gather the available bandwidth from each edge node and calculate which running edge node can provide a spare resource. Given 8 units free resources can be provided by E_2 from its latency-sensitive queue, the available bandwidth tuple [8, 0, 0] will be known by the orchestrator via periodic resource updating and synchronization between edge nodes and orchestrator.

The orchestrator will then carry out Alg. 3 and confirm that E_1 should redirect its overflowed traffic to E_2 and E_2 has to reserve the corresponding bandwidth temporarily for the redirection. To this end, Orchestrator will respond to all the involved nodes, notifying the satisfied amount [0, 0, 5] to the requesting edge node. Within the designated edge node, an internal bandwidth shift [5#, 0, 5"] to the destination edge node for the bandwidth reservation. Once upon E_2 receiving the instruction, 5 units of available bandwidth will be secured to its latency-tolerant queue, i.e., the bandwidth allocation will shifted from [18, 10, 2] to [13, 10, 7]. Upon receiving the request response, E_1 redirects its traffic to E_2.

Complexity Analysis. JANUS is a simple and efficient stream processing system with low complexity. Alg.2 only has $O(n)$ time complexity, where n denotes the types of the queues. The time complexity of Alg.3 is $O(p \quad r)$ where p is the number of bandwidth provider, and r represents the edge nodes that redirect their streams to the providers.

5.2.3 Other Comparative Heuristics

We are aware of many other counterpart heuristics that can serve the requests of traffic redirection. Specifically, our previous study [28] presented comprehensive experimental comparisons among node-centric, application (request)-centric and multi-node approaches. As will be shown in §7.1, we select the following representative winning algorithms as baselines to compare their runtime performance.

6 SYSTEM IMPLEMENTATION

6.1 Implementation Details

Message Exchange Protocols. We firstly present the selection of message exchange protocols to underpin streaming flows: *1) IoT devices to edge nodes:* We leverage MQTT protocol to fulfill lightweight record transmission due to its dedicated design for IoT messaging following the Pub/Sub model. Eclipse Mosquitto [29] is a lightweight implementation of MQTT protocol and well-suited for diverse devices stretching from low-power sensors to high-performance computer servers. We launch an instance of Eclipse Mosquitto server on each edge node and other modules can use a Mosquitto client to subscribe the pre-defined topics through the Pub/Sub. *2) edge nodes to cloud servers:* We choose AMQP as the application-layer protocol, for it provides resilient flow-controlled communication with message-delivery guarantees. We use RabbitMQ [30] to underpin the fundamental message delivery in WAN environments, building up the streaming tunnels between edge nodes and cloud servers. RabbitMQ is a lightweight and easy-to-deploy messaging system software that has been widely used as Pub/Sub system by industries for almost a decade. Owing to the in-memory operations without permanently storing to intermediate disks, RabbitMQ has far lower latency against Kafka when transferring messages. Additionally, RabbitMQ offers more flexibility of developing bespoken redirection algorithms due to its intrinsic nature of supporting Pub/Sub messaging.

Streams Categorization. Table 1 presented an example of representative value range that approximates the typical latency requirement of the listed applications. One can flexibly determine the number of categories and define the corresponding thresholds for each of them according to their latency requirement. Our solution is adaptive to, and working correctly in extreme cases. For instance, if all streams belong to the same category, JANUS will perform in the first come first served manner; if all tasks are completely different in the latency requirement, one can simply partition the tasks, putting one task into one category.

Scalability. Due to the loosely-coupled system design, additional edge nodes can be simply added into the system in the event of increasing demands or insufficient forwarding capability. Our redirection approach that adopts greedy heuristic can underpin thousands of edge nodes even on a low-power computational device. However, collecting information from many edge nodes to a centralized edge node may become a system bottleneck. We plan to employ decentralized methods [31], [32], [33] to tackle this communication scalability in the future work.

Fault Tolerance. In an attempt to reduce single point of failure, we set up hot-standby replications for key components – such as Queue Manager, Edge Agent, and Traffic Orchestrator – to enable their automatic failover in the face

TABLE 3
Emulated network environment.

Network	Latency	Throughput
WiFi	5ms	200mbps
4G	50ms	18mbps
3G	100ms	780kbps

of any crash-stop faults. Each edge node periodically checkpoints the arrived streams within a given time frame. We use sequence numbers to track the last consumed records of all streams. Once an edge node recovers from software component or hardware crashes, we simply restore the most recently saved checkpoints to continue the execution: if the faulty node has pending data to transmit, Queue Manager will retry the transfer to the corresponding cloud endpoints, and the PMRFS algorithm will be called, where necessary, to re-transmit the recovered data to other peer edge nodes.

Fairness. Our current implementation does not consider fairness scheduling among different queues in a single edge node. However, we found that it is rare to starve *lt* or *nm* queues. This is because the arrival rate of *ls* streaming is typically small and the available bandwidth on idle edge nodes are thus sufficient for our redirection mechanism to tackle the overflow in *lt* or *nm* queues. Prior work [34], [35], [36], [37] considered the fair bandwidth allocation in the context of cloud computing. These techniques can be integrated with JANUS to address fairness scheduling problem and will be left for future study.

6.2 Discussion

JANUS makes its best effort to target QoS requirements of data streaming delivery theoretically based on manageable throughput measurement. This, however, is not an easy task; practically numerous uncertainties may influence its effectiveness: i) *resource interference*: stream processing is memory and CPU intensive, and the throughput (input/output) significantly depends on the amount of available resource and the degree of resource interference in the node, which is however in short supply and sometimes difficult to predict. ii) *network variability*: our real-world evaluation in §7.3 indicates that bandwidth of the IoT network (e.g., cellular network) fluctuates drastically over time, and is extremely vulnerable to factors such as weather, shared connections and network coverage, etc. These variables complicate the design of the robust and optimal mechanism for throughput throttle and traffic redirection. JANUS uses a simple and efficient means particularly for low-power and low-cost computational devices. New algorithms can be easily plugged into JANUS owing to the loosely-coupled component-based architecture.

Additionally, JANUS focuses on effective IoT streaming management, without a particular consideration of computation latency. The computational latency assurance of IoT streaming processing investigated in many existing works [38], [39], [40] could be easily integrated with JANUS.

7 EVALUATION

7.1 Experimental Setup

Environments. We evaluate JANUS in two environments:

Lab Testbed: We evaluate JANUS on a testbed consisting of real edge hardware including Raspberry Pis and servers. In terms of IoT networking, we consider an emulated network and real-world 4G: In the emulated network, we run micro-benchmarks over 10 Raspberry Pi 3 model B+ (with 1.4GhZ 4 cores and 1GB of RAM) and a bare metal Ubuntu machine, with 20 cores (Intel(R) Xeon(R) Silver 4114 CPU @ 2.20GHz) and 64 GB memory. The network is emulated by using Linux traffic control (TC) [41]. Table 3 shows the network configurations for the experiments where the configuration parameters are based on real world measurement [42].

Real-world Testbed: We connect the edge nodes (Raspberry Pis) with an amazon EC2 cloud server (4 vCPU and 8 GB memory, Vodafone 4G network through 4G USB Dongles). In the micro-benchmarks, 4 Pis are set as edge nodes which consume sensor data sending from sensors. All Pis are connected through local WiFi network through Netgear switch with 200 Mbps bandwidth.

Dataset and Workloads. To generate realistic workloads, we use a real-world smart building dataset [20]. It contains data samples collected from $4,000+$ sensors such as occupancy, CO_2 and temperature etc. We categorize the input data from different sensors and analyze the distribution of their arrival rates. Input rate of each edge node is controlled and tuned by changing the sensor's forwarding rate to the pertaining edge nodes. According to the empirical profiling, we set the input rate 30 records/s for all three types of streams on on idle edge nodes. The input rate of busy nodes is configured at 30 records/s for latency-sensitive streams and 300 records/s for normal streams. Since JANUS focuses on the throughput optimization in streaming data delivery, we simply run a data query program that includes aggregate functions, e.g., sum/avg operation in a time window, as the representative workload in the cloud.

Methodology and Metrics. We vary the input rate of throughput-tolerant streams (from 1K records/s to 5K records/s) and network environment (3G, 4G and WiFi), and measure their impact on the system effectiveness. We compare JANUS with three baselines:

Native scheme: No queue management mechanism is enabled and inbound streams will be directly forwarded from sensors to cloud without any interventions by priority queues.

QM-Only scheme: This is a comparable strategy adopted in QJUMP that only encompasses the local queue manager with the traffic orchestration disabled.

We also compare JANUS that adopts PMRFS for global traffic coordination with several baseline heuristics:

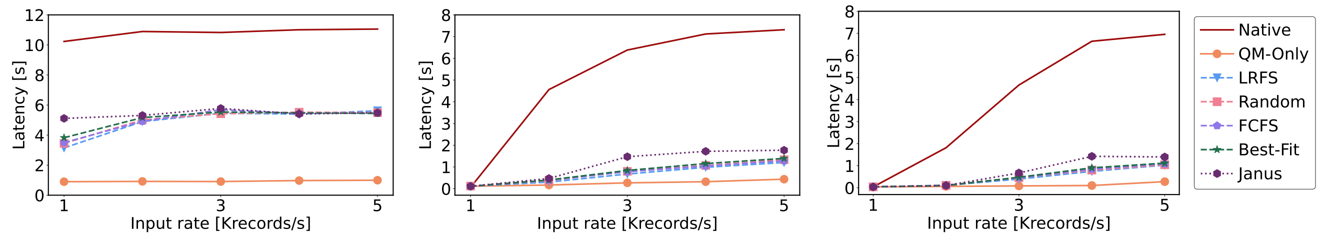
Random scheme randomly picks requests from multiple queues among different overloaded edge nodes.

First Come First Served (FCFS) scheme picks the request that asks for additional bandwidth first.

Least Request First Served (LRFS) scheme chooses the request with minimal bandwidth request.

Best Fit (BF) scheme traverses the list of available bandwidth providers and pick the bandwidth request that is closest in size to the current available provider.

For accurate results, we repeat each experiment 5 times independently and calculate the average. We consider three metrics in our evaluation:



(a) 3G network (b) 4G network (c) WiFi network

Fig. 5. The end-to-end latency in various network environment.

TABLE 4
The end-to-end latency of each types of queues under various network environment

Network	3G			4G			WiFi		
Approaches	*ls* (s)	*nm* (s)	*lt* (s)	*ls* (s)	*nm* (s)	*lt* (s)	*ls* (s)	*nm* (s)	*lt* (s)
Native		10.86±0.12			7.52±0.08			7.26±0.09	
QM-Only	1.13±0.18	8.51±0.19	13.22±0.14	0.53±0.07	4.51±0.08	9.69±0.09	0.41±0.07	3.57±0.06	8.72±0.07
FCFS	4.75±0.16	6.95±0.14	7.69±0.07	1.52±0.04	2.29±0.06	5.19±0.11	1.29±0.03	2.24±0.05	4.63±0.08
LRFS	4.73±0.07	6.89±0.11	7.77±0.04	1.48±0.06	2.26±0.04	5.68±0.13	1.26±0.03	2.19±0.08	4.81±0.09
Best-Fit	4.68±0.12	6.83±0.11	7.70±0.03	1.55±0.04	2.33±0.16	5.04±0.06	1.30±0.03	2.29±0.12	4.08±0.13
Random	4.81±0.14	6.93±0.07	7.66±0.13	1.49±0.04	2.30±0.08	5.26±0.17	1.27±0.03	2.21±0.04	4.41±0.08
JANUS	4.71±0.11	6.54±0.09	8.04±0.05	1.58±0.08	2.41±0.11	3.53±0.06	1.32±0.04	2.39±0.03	3.06±0.05

End-to-end Latency: This mainly refers to the turnaround time of latency-sensitive *ls* workloads that includes the data transmission latency between data sources and the cloud and the execution time of workloads in the cloud. For better evaluating the impact on latency of the neighboring workloads and their compromise on the performance, we also measure the end-to-end latency of other standard *nm* and latency-tolerant *lt* workloads.

System Throughput: This is defined as the total number of data records forwarded from sensor nodes to the cloud. This is particularly important for those latency-tolerant yet throughput-intensive workloads.

Resource Consumption: We measure the resource overhead incurred by running JANUS components to examine the runtime resource cost to the native system.

7.2 Effectiveness Evaluation

7.2.1 Impact on Latency of Latency-Sensitive Streams

We first evaluate the end-to-end latency in various network environment. The comparisons can be carried out from the following three perspectives:

Comparison by approaches. As shown in Fig. 5, the latency can be significantly reduced in both *QM-Only* and JANUS compared to the *Native* approach under all network conditions. For example, in 3G network condition, the latency of *Native* is roughly 10.2x times higher than *QM-Only* and JANUS. This is simply because the multi-level queue management will prioritize latency-sensitive streams and secure its first order in the record forwarding, thereby assuring a low latency. Due to the intrinsic resource consumption within edge node during the global traffic coordination, the forwarding capability for latency-sensitive stream will be dropped, resulting in a reduced turnover rate for such stream records and accordingly a higher latency of JANUS against *QM-Only*.

Comparison by network conditions. The overall latency of all the approaches will decrease when the output bandwidth increase (i.e., from 3G to WiFi). For instance, the average latency of JANUS under WiFi can be reduced by roughly 85% compared against 3G environment. Obviously, an improved network indicates larger outbound bandwidth that will accelerate the transmission of all sorts of streams. Hence, the latency will be inherently decreased.

Comparison by different heuristics. Compared with other methods, Janus can process multiple requests simultaneously based on streams' priority, thereby making better use of bandwidth on idle nodes. As shown in Fig. 5, JANUS exhibits slightly higher latency for latency-sensitive data streams compared with other baselines. This derives from requiring additional resource and computation cost when conducting PMRFS algorithm and involving an increased number of stream redirection than other baselines.

Impact of input rate. In 4G and WiFi network, the latency of *Native* dramatically ramps up with the increment of input rate, mainly due to the queuing backlog introduced by the throughput-intensive records. When the input rate reaches 5k records/s, the latency of *Native* even peaks approximate 20x and 6x times that of *QM-Only* and JANUS, respectively. By contrast, the latency of both *QM-Only* and JANUS remains stable without noticeable increase when input rate increases. Overall, *QM-Only* in most cases has the lowest latency – the most significant effect in latency management – indicating that administrators may switch off the global coordination for sake of a constantly stringent latency control. Notably, when the input rate is lower than $2k$ records/s under 4G and WiFi network, JANUS is in very close proximity to *QM-only* because the functionality of *traffic redirection* has not been triggered.

7.2.2 Impact on Latency of Other Streams

Apart from examining the impact of different approaches and different network conditions on the latency of latency-sensitive streams, we investigate how the latency of *lt* and *nm* streams will be affected so as to evaluate how JANUS trade their performance for prioritizing the latency criticality of *ls* streams. Table 4 demonstrates the end-to-end latency of each type of the queues under different network conditions where the input rate is kept to 5k records/s.

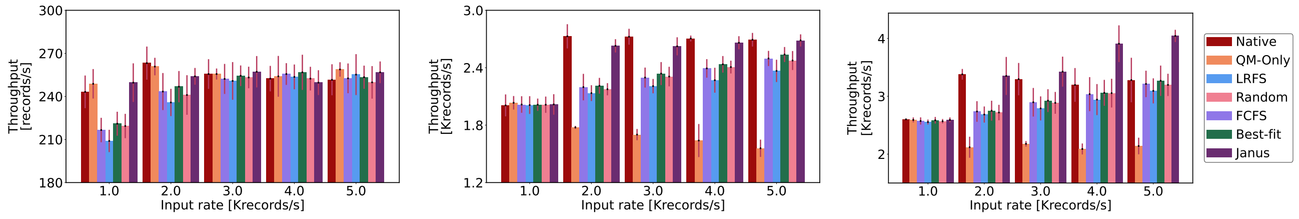

(a) 3G network (b) 4G network (c) WiFi network

Fig. 6. Overall throughput in various network environment.

TABLE 5
Latency violation comparison under 4G network

Approach	*ls*	*nm*	*lt*
Native	17.5%	4.2%	5.7%
QM-Only	0	9.8%	15.5%
JANUS	0	6.7%	9.4%

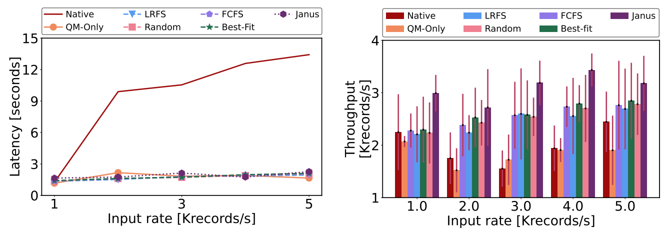

(a) Latency (b) Throughput

Fig. 7. e2e latency and throughput in real 4G network

Comparison by approaches and heuristics. Overall, JANUS outperforms other approaches in balancing bandwidth among queues and thus well guarantee the low latency of the latency-sensitive workloads without involving too much starvation to the standard or latency tolerant workloads. For example, in the 4G environment, compared with native scheme where no exclusive queues are differentiated – as shown in Table 4 all queues share the same latency result – JANUS can reduce the latency of *ls* from 7.52 seconds to merely 1.58 seconds on average. Despite the fact that *QM-Only* can even reduce *ls*'s latency to 0.53 seconds on average, the resultant latency of *lt* will ramp up to 9.69 seconds, much higher than its performance in the native case. This is because, when compared with *QM-Only* without system-level traffic orchestration, JANUS can better utilize the bandwidth provided by idle nodes to mitigate the impact of latency on the inferior queues. Different Heuristics do not show obvious disparities. Table 5 also reveals the latency violation ratio of each queue under 4G network. Observably, thanks to the mechanism of adaptive bandwidth reallocation among queues, JANUS and QM-only can guarantee no violation of latency-sensitive streams, with some violations in the inferior queues. Compared with QM-only approach, JANUS can use the idle edge nodes to mitigate the violations of both *nm* and *lt* streams. While *native* approach has much lower violation ratio of *nm* and *lt* streams, it suffers from high violation of *ls* streams, which is unacceptable.

Comparison by network conditions. It is also observable that network conditions affect the latency results. Arguably, 4G and wireless connections can provision much more sufficient network resource within the entire Edge system than the 3G environment. The advantage of traffic orchestrator in JANUS can be thus better leveraged to alleviate the degree of latency increase of *nm* and *lt* streams.

7.2.3 Impact on Throughput

Similarly, we measure the overall throughput under the same experimental environment.

Comparison by approaches. As illustrated in Fig. 6, JANUS outperforms other approaches, particularly when the network condition improves to WiFi environments. It is observable that, in most cases, *Native* and JANUS have a substantial throughput increase compared against *QM-Only*. For instance, JANUS achieves similar throughput as *Native* and 1.7 times more than that of *QM-Only* in 4G network. This improvement increases as the network bandwidth grows. In WiFi network, JANUS achieves higher throughput than that of *Native* and more than 2 times of *QM-Only*. In fact, *QM-Only* blocks the traffic redirection for latency-tolerant streams due to the lack of elastic forwarding mechanism, even if other edge nodes have sufficient bandwidth capability. By contrast, *Native* can directly forward throughput-intensive records without distinguishing the record types whilst JANUS can rapidly bypass the congesting edge node other than using the local capacity, resulting in the highest system throughput constantly.

Comparison by network conditions. Intuitively, the growth of outbound bandwidth gives rise to the enlarged capability of record digestion and transmission. For instance, WiFi and 4G network experience far better throughput against the 3G scenario. In reality, in 3G network, there are marginal disparities among three approaches purely because of the limited network bandwidth.

Comparison by different heuristics. Janus can yield higher throughput than other baselines across various network conditions and input rates, and the disparity is more noticeable in conditions with more bandwidth capacity. For example, In Fig. 6, JANUS achieves 1.3x higher throughput than other heuristics under WiFi network. This is because Alg. 3 can consider various bandwidth requests simultaneously and thus outperform others which can merely handle a single bandwidth request a time.

Impact of input rate. Increased input rate indicates a growing crowd of records feeding into the system. In 4G and WiFi network, the throughput of *QM-Only* decreases when the number of input records ramps up. This is because local processing and queuing ability of an individual edge node cannot afford timely forwarding for the growing input, thereby slightly reducing the overall throughput. However, the throughput of JANUS will, on the other hand, steadily increase, owing to the redirection mechanism for rapid forwarding to other nodes. Since the *Native* approach directly forwards all records upon their arrival, its upper throughput boundary will be confined by the local processing capacity. JANUS can reuse capacity from both local and neighbor edge nodes; hence the highest average throughput.

7.2.4 Overhead

We mainly measure the resource (CPU and memory) usages of each component at runtime as the primary system overhead. The experiment result shows that JANUS's overhead is negligible – *Edge Agent* only consumes approximate 4:5% memory and 2:5% CPU; while *Traffic Orchestrator* uses 1:5% CPU and 5% memory. This indicates it is worth enabling the light-weight agent for latency guarantee and throughput improvement at the cost of marginal system cost.

7.3 Real-world Evaluation

We evaluate JANUS using real-world 4G network. As presented in Fig. 7(a), JANUS and *QM-Only* scheme have far lower latency than the *Native* baseline. In particular, latency of *Native* is more than 5 times that of others when the input rate is 3k records/s. The difference is further amplified with the increase of input rate. For example, JANUS is able to reduce the latency to only 16.6% against *Native* when the input rate is 5k records/s. Interestingly, we observe that in a dynamic mobile network condition, the latency disparity between JANUS and *QM-Only* is negligible. In fact, the available bandwidth between an edge node and cloud is dramatically volatile that hugely increases the difficulty in finding the optimal solution to the bandwidth allocation for different types of queues. As a result, *bandwidth reallocation* in Alg. 2 may not be able to allocate an optimal bandwidth for low latency queue and thus declines the effect of redirection and paving ways for latency-sensitive records.

As shown in Fig. 7(b), the throughput of JANUS can be improved by 1.56 and 1.7 against the *Native* baseline and *QM-Only*, respectively. The proposed redirection mechanism greatly facilitates to overcome the dynamicity issue that is ubiquitously manifested in real-world 4G network. The elasticity provided by traffic redirection advances the throughput maintenance particularly when available bandwidth of an edge node fluctuates sharply. It is nontrivial to note that real world environments bring numerous uncertainties. For example, when the input rate reaches 3k records/s, throughput of the *Native* scheme is lower than *QM-Only*, noticeably because of a network bandwidth drop during the *Native* experiments.

8 RELATED WORK

Network management in Datacenters. Congestion control is a common practice in network community, typically by effectively limiting transmission rate and forwarding network packets to their destination. [10] extends the window adjustment algorithm adopted in DCTCP [43] and uses earlier deadline first policy in the flow scheduling. [9] improves the congestion avoidance mechanism in [10] with the aid of packet-pacing NIC. [11] adopts a first in first out (FIFO) policy to schedule bandwidth. However, they require either heavy support from switches or modification of transmission protocols, OS kernel and application modules, making it difficult to deploy upon commodity hardware. They can hardly provide guaranteed bound on the latency and flow deadlines. QJUMP [12] forwards messages into different queues based on their priorities, which is aligned with the intention of multi-level management in JANUS. However, its proposed method is not applicable in stream processing applications in the edge environment due to the limited visibility and control of network devices for manipulating internal data streams. Homa [44], pHost [45] and NDP [46] leverage receiver-driven flow control mechanism to reduce the latency of small messages. However, these switch based mechanisms are highly dependent upon an assumption that the ingress throughput equals to the egress throughput, to be invalid in the IoT-edge-cloud continuum. JANUS firstly develops an effective mechanism of dynamic bandwidth allocation and holistic traffic coordination at the application layer, which is flexible to carry out throughput throttling and bandwidth adjustment over streaming data.

Stream processing in cloud and edge computing. Most of the stream processing platforms [47], [48], [49], [50], [51] rely on datacenter environments to provide centralized streaming services. The advances of edge computing facilitate the shift of cloud-based data processing much closer to the ground, which can significantly reduce the process latency [52]. Frontier [53] develops an edge-based stream processing system for ML applications. However, it focus on reliability of ML applications on edge nodes in a distributed manner. Approxiot [38] mainly focuses on optimizing the performance of analytic tasks rather than considering the queueing delay problem in delivering the streaming data. NebulaStream [54] develops APIs for specifying dataflow programs that can direct data streams to different processing tasks. However, it does not differentiate latency sensitivity of different IoT applications and thus fail to effectively cope with queue delay. JANUS presents an effective traffic scheduling system across the full stack in the IoT-edge-cloud continuum, particularly considering different types of data records and their specific QoS requirements.

Offloading in mobile edge computing. General-purpose offloading [55], [56], [57], [58], [59], [60], [61] in mobile computing mainly targets the problem of task offloading to the cloud, neglecting the impact of messaging across various computing resources. LEO [55] optimizes the energy consumption by performing multiple sensor processing tasks on mobile devices, without considering the dynamicity of IoT network. Despite the consideration of diverse resources, MAUI [56], Code in the Air [58] and Odessa [57] are unaware of queuing delay from edge nodes to cloud. They can benefit from the network adaptation capabilities in JANUS. Wang et al. [62] proposed a Edmonds–Karp algorithm to address a mixed-integer nonlinear programming problem in computation offloading for IoV. Ren et al.[63] formalized the edge-cloud task offloading as a convex optimization problem and resolves it via KKT conditions. Xu et al.[64] tackled task offloading by simple logistics to optimize the QoS metrics. They decomposed the optimization problems into simpler convex forms. However, our system requires a real-time solution that cannot be resolved by such methods.

9 CONCLUSION

We have presented JANUS, a traffic scheduling system for data streams in distributed edge computing. JANUS addresses the throughput mismatch problem where data arrive faster than they can be consumed on an edge node. We formulate two distinct optimization problems and tackle them in JANUS in a practical manner. At the edge node level, JANUS dynamically allocates the uplink bandwidth according to the latency constraint of the application, by giving higher priority to latency-sensitive applications. JANUS actively monitors the traffic loads of a distributed edge computing network to direct data from heavily loaded edge nodes to the less loaded ones to achieve a network-wide load balancing. In the future, we plan to develop other gradient-based algorithms for throughput optimization and evaluate it in a larger-scale environment.

REFERENCES

[1] B. Qian, J. Su, Z. Wen *et al.*, "Orchestrating the development lifecycle of machine learning-based iot applications: A taxonomy and survey," *ACM Computing Surveys (CSUR)*, 2020.

[2] K. Ha, Y. Abe, T. Eiszler, Z. Chen, W. Hu, B. Amos, R. Upadhyaya, P. Pillai, and M. Satyanarayanan, "You can teach elephants to dance: Agile vm handoff for edge computing," in *ACM/IEEE SEC*, 2017.

[3] T. Olsson, E. Lagerstam, T. Kärkkäinen, and K. Väänänen-Vainio-Mattila, "Expected user experience of mobile augmented reality services: a user study in the context of shopping centres," *Personal and ubiquitous computing*, 2013.

[4] H. Lasi, P. Fettke, H.-G. Kemper, T. Feld, and M. Hoffmann, "Industry 4.0," *Business & information systems engineering*, 2014.

[5] A. P. Plageras, K. E. Psannis, C. Stergiou, H. Wang, and B. B. Gupta, "Efficient iot-based sensor big data collection–processing and analysis in smart buildings," *FGCS*, 2018.

[6] L. Smith and M. Turner, "Building the urban observatory: Engineering the largest set of publicly available real-time environmental urban data in the uk." in *Geophysical Research Abstracts*, 2019.

[7] P. Schulz, M. Matthe, H. Klessig *et al.*, "Latency critical iot applications in 5g: Perspective on the design of radio interface and network architecture," *IEEE Communications Magazine*, 2017.

[8] M. Chowdhury and I. Stoica, "Coflow: A networking abstraction for cluster applications," in *ACM HotNet*, 2012.

[9] M. Alizadeh *et al.*, "Less is more: Trading a little bandwidth for ultra-low latency in the data center," in *USENIX NSDI*, 2012.

[10] B. Vamanan, J. Hasan, and T. Vijaykumar, "Deadline-aware datacenter tcp (d2tcp)," *ACM SIGCOMM*, 2012.

[11] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron, "Better never than late: Meeting deadlines in datacenter networks," *ACM SIGCOMM*, vol. 41, no. 4, pp. 50–61, 2011.

[12] M. P. Grosvenor *et al.*, "Queues don't matter when you can {JUMP} them!" in *USENIX NSDI*, 2015.

[13] C.-Y. Wan, S. B. Eisenman, and A. T. Campbell, "Coda: Congestion detection and avoidance in sensor networks," in *ACM SenSys*, 2003.

[14] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, "Hula: Scalable load balancing using programmable data planes," in *Symposium on SDN Research*, 2016.

[15] M. M. Hasan, S. Kwon, and J.-H. Na, "Adaptive mobility load balancing algorithm for lte small-cell networks," *IEEE Trans. on Wireless Communications*, 2018.

[16] A. Banks and R. Gupta, "Mqtt version 3.1.1," *OASIS standard*, 2014.

[17] J. Kramer, "Advanced message queuing protocol," *Linux Journal*, 2009.

[18] S. Bagchi, M.-B. Siddiqui, P. Wood, and H. Zhang, "Dependability in edge computing," *Communications of the ACM*, 2019.

[19] C. Pereira, A. Pinto, D. Ferreira, and A. Aguiar, "Experimental characterization of mobile iot application latency," *IEEE Internet of Things Journal*, vol. 4, no. 4, pp. 1082–1094, 2017.

[20] urban observatory team, *Urban science build data*, 2020. [Online]. Available: https://urbanobservatory.ac.uk/#

[21] M. D. de Assuncao, A. da Silva Veith, and R. Buyya, "Distributed data stream processing and edge computing: A survey on resource elasticity and future directions," *JNCA*, 2018.

[22] D. Wedelin, "An algorithm for large scale 0–1 integer programming with application to airline crew scheduling," *Annals of Oper. research*, 1995.

[23] J. P. Ignizio, "An approach to the capital budgeting problem with multiple objectives," *The Engineering Economist*, vol. 21, no. 4, pp. 259–272, 1976.

[24] P. C. Chu and J. E. Beasley, "A genetic algorithm for the multidimensional knapsack problem," *Journal of heuristics*, vol. 4, no. 1, pp. 63–86, 1998.

[25] O. Mersmann, B. Bischl, H. Trautmann *et al.*, "A novel feature-based approach to characterize algorithm performance for the traveling salesperson problem," *Annals of Mathematics and Artificial Intelligence*, 2013.

[26] Gurobi. [Online]. Available: https://www.gurobi.com/

[27] Ibm cplex. [Online]. Available: https://www.ibm.com/analytics/cplex-optimizer

[28] C. Mommessin, R. Yang *et al.*, "Affinity-aware resource provisioning for long-running applications in shared clusters," *Journal of Parallel and Distributed Computing*, 2023.

[29] R. A. Light, "Mosquitto: server and client implementation of the mqtt protocol," *J. Open Source Software*, vol. 2, no. 13, p. 265, 2017.

[30] A. Videla and J. J. Williams, *RabbitMQ in action: distributed messaging for everyone*. Manning, 2012.

[31] E. Yoneki, "Raspinet: Decentralised communication and sensing platform with satellite connectivity," in *ACM MobiCom*, 2014.

[32] G. Best *et al.*, "Planning-aware communication for decentralised multi-robot coordination," in *IEEE ICRA*, 2018.

[33] D. Schulze and H. Zipper, "A decentralised control algorithm for an automated coexistence management," in *IEEE CDC*, 2018.

[34] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queueing algorithm," *ACM SIGCOMM*, 1989.

[35] J. C. Bennett and H. Zhang, "Wf/sup 2/q: worst-case fair weighted fair queueing," in *IEEE INFOCOM*, 1996.

[36] R. Pan, B. Prabhakar, and K. Psounis, "Choke-a stateless active queue management scheme for approximating fair bandwidth allocation," in *IEEE INFOCOM*, 2000.

[37] Z. Cao *et al.*, "Rainbow fair queueing: Fair bandwidth sharing without per-flow state," in *IEEE INFOCOM*, 2000.

[38] Z. Wen, P. Bhatotia, R. Chen, and M. Lee, "Approxiot: Approximate analytics for edge computing," in *IEEE ICDCS*, 2018.

[39] T. Um, G. Lee, and B.-G. Chun, "Pluto: high-performance iot-aware stream processing," in *IEEE ICDCS*, 2021.

[40] D. Kumar *et al.*, "Aggnet: Cost-aware aggregation networks for geo-distributed streaming analytics," in *IEEE/ACM SEC*, 2021.

[41] linux, *Linux Traffic Control*, 2020. [Online]. Available: https://linux.die.net/man/8/tc

[42] Opensignal, *State of Mobile Networks: UK*, 2020 (accessed 22, 3, 2020). [Online]. Available: https://www.opensignal.com/reports/2018/04/uk/state-of-the-mobile-network

[43] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," in *ACM SIGCOMM*, 2010, pp. 63–74.

[44] B. Montazeri *et al.*, "Homa: A receiver-driven low-latency transport protocol using network priorities," in *ACM SIGCOMM*, 2018.

[45] P. X. Gao, A. Narayan *et al.*, "phost: Distributed near-optimal datacenter transport over commodity network fabric," in *ACM CoNEXT*, 2015.

[46] M. Handley, C. Raiciu *et al.*, "Re-architecting datacenter networks and stacks for low latency and high performance," in *ACM SIGCOMM*, 2017.

[47] L. Mai, K. Zeng *et al.*, "Chi: a scalable and programmable control plane for distributed stream processing systems," *VLDB Endowment*, 2018.

[48] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *IEEE Data Eng. Bull.*, 2015.

[49] A. Toshniwal, S. Taneja, A. Shukla *et al.*, "Storm@ twitter," in *ACM SIGMOD*, 2014.

[50] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *ACM SOSP*, 2013.

[51] S. Venkataraman *et al.*, "Drizzle: Fast and adaptable stream processing at scale," in *ACM SOSP*, 2017.

[52] M. Satyanarayanan, "The emergence of edge computing," *IEEE Computer*, 2017.

[53] D. O'Keeffe, T. Salonidis, and P. Pietzuch, "Frontier: Resilient edge processing for the internet of things," *PVLDB*, 2018.

[54] S. Zeuch *et al.*, "The nebulastream platform: Data and application management for the internet of things," *arXiv*, 2019.