PrioDeX: A Data Exchange Middleware for Efficient Event Prioritization in SDN-Based IoT Systems

PrioDeX: a Data Exchange Middleware for Efficient Event Prioritization in SDN-based IoT systems

GEORGIOS BOULOUKAKIS, SAMOVAR, Télécom SudParis, IP Paris, France
KYLE BENSON, RTI, USA
LUCA SCALZOTTO, Injenia S.r.l., Bologna, Italy
PAOLO BELLAVISTA, Univ. of Bologna, Italy
CASEY GRANT, NFPA, USA
VALÉRIE ISSARNY, INRIA Paris, France
SHARAD MEHROTRA, UC Irvine, USA
IOANNIS MOSCHOLIOS, Univ. of Peloponnese, Greece
NALINI VENKATASUBRAMANIAN, UC Irvine, USA

Real-time event detection and targeted decision making for emerging mission-critical applications require systems that extract and process relevant data from IoT sources in smart spaces. Oftentimes, this data is heterogeneous in size, relevance, and urgency, which creates a challenge when considering that different groups of stakeholders (e.g., first responders, medical staff, government officials, etc) require such data to be delivered in a reliable and timely manner. Furthermore, in mission-critical settings, networks can become constrained due to lossy channels and failed components, which ultimately add to the complexity of the problem. In this paper, we propose PrioDeX, a cross-layer middleware system that enables timely and reliable delivery of mission-critical data from IoT sources to relevant consumers through the prioritization of messages. It integrates parameters at the application, network, and middleware layers into a data exchange service that accurately estimates end-to-end performance metrics through a queueing analytical model. PrioDeX proposes novel algorithms that utilize the results of this analysis to tune data exchange configurations (event priorities and dropping policies), which is necessary for satisfying situational awareness requirements and resource constraints. PrioDeX leverages Software-Defined Networking (SDN) methodologies to enforce these configurations in the IoT network infrastructure. We evaluate our approach using both simulated and prototype-based experiments in a smart building fire response scenario. Our application-aware prioritization algorithm improves the value of exchanged information by 36% when compared with no prioritization; the addition of our network-aware drop rate policies improves this performance by 42% over priorities only and by 94% over no prioritization.

CCS Concepts: • **Networks → Programming interfaces**; **Network performance modeling**; • **Computer systems organization → Reliability**; • **Software → Message oriented middleware**.

Additional Key Words and Phrases: Publish/Subscribe Middleware, Event Prioritization, Utility Functions, Queueing Networks, SDN

Authors' addresses: Georgios BouloukakisSAMOVAR, Télécom SudParis, IP Paris, France, georgios.bouloukakis@telecom-sudparis.eu; Kyle BensonRTI, USA, kebenson@ics.uci.edu; Luca ScalzottoInjenia S.r.l., Bologna, Italy, luca.scalzotto@studio.unibo.it; Paolo BellavistaUniv. of Bologna, Italy, paolo.bellavista@unibo.it; Casey GrantNFPA, USA, cgrant@nfpa.org; Valérie IssarnyINRIA Paris, France, valerie.issarny@inria.fr; Sharad MehrotraUC Irvine, USA, sharad@ics.uci.edu; Ioannis MoscholiosUniv. of Peloponnese, Greece, idm@uop.gr; Nalini VenkatasubramanianUC Irvine, USA, nalini@uci.edu.

2577-6207/2019/0-ART0 $15.00
https://doi.org/0000001.0000001

ACM Reference Format:
Georgios Bouloukakis, Kyle Benson, Luca Scalzotto, Paolo Bellavista, Casey Grant, Valérie Issarny, Sharad Mehrotra, Ioannis Moscholios, and Nalini Venkatasubramanian. 2019. PrioDeX: a Data Exchange Middleware for Efficient Event Prioritization in SDN-based IoT systems. *ACM Trans. Internet Things* 0, 0, Article 0 (2019), 30 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

The integration of the *Internet of Things* (IoT) into daily life promises to revolutionize societal-scale operations. It integrates pervasive sensing/actuation, dynamic data analytics and communications, which encourages domains such as transportation, home automation, healthcare, and emergency response to become increasingly IoT-enabled. Smart spaces such as office buildings, tend to increase the deployment of novel networking infrastructures along with state-of-the-art IoT devices; this provides data-driven insights to improve the situational awareness of a space. This is particularly useful in mission-critical applications for enabling timely and reliable communication in smart spaces. Recent smart city efforts such as the SmartAmerica and Global City Teams Challenges have showcased the integration of IoT into a variety of application domains [9, 11, 35, 61].

A distributed data exchange system that manages relevant data flows to/from IoT devices and individuals (data producers and consumers) is a critical centerpiece of IoT deployments. In smart spaces, such devices are deployed at the Edge (closer to individuals) and thus, data exchange systems must support a flexible Edge networking infrastructure to manage data flows with varying quality levels. Producers of data correspond to IoT sensors, *events* to data produced/consumed, and consumers of data to interested entities (i.e., human stakeholders or other IoT devices and services). The data exchange system routes information to actuators (e.g., alarms) or human stakeholders. In mission-critical emergency scenarios, IoT devices can forward raw sensor data to interested recipients (e.g., first responders, medical staff, public safety officers, government officials, etc) through data exchange systems to help coordinate the response effort.

Key challenges arise for enabling timely data exchange to a diverse set of recipients, including: (i) managing heterogeneous information with varying size, format, relevance and urgency; (ii) seamless dynamic integration of new IoT data sources with pre-existing sources and information; (iii) supporting reliable and timely communication over constrained networks – e.g., due to lossy channels and failed components. For instance, during a structural fire, firefighters require timely reception of up-to-date situational awareness information. Given the heterogeneity of this information and the limited networking resources for delivering notifications, we believe event prioritization is necessary in such mission-critical settings. Existing data exchange systems [6, 38, 50, 54, 58, 65] provide mechanisms for event prioritization either by manually assigning priorities to specific data flows or by dynamically assigning them based on the application-level data flows/types (e.g., video data) or even QoS-specific requirements (e.g., delay-sensitive apps). However, such systems cannot be customized to support mission-critical applications with dynamic changes of the app requirements, interested data recipients, data flows/types and the networking conditions.

In this paper we propose **PrioDeX**, an integration middleware that enables timely and reliable delivery of the most critical data to relevant data recipients despite challenging network conditions. PrioDeX unifies smartspace IoT data and edge infrastructures with programmable network infrastructures and domain specific applications (e.g., smart fire fighting). It leverages Edge computing (i.e., publish/subscribe brokers at the network edge) and *Software-Defined Networking* (SDN) to bridge critical application requirements with network state. The main contribution of PrioDeX is the capability of configuring the SDN-enabled physical network to prioritize events according to the situational awareness app-requirements and network resource constraints. We model the Edge

infrastructure using priority queues to estimate performance metrics (response times, delivery success rates) based on the system workload. These are given as input to PrioDeX algorithms to assign priorities and carefully tune packet drop rates (for bandwidth allocation) to active subscriptions.

The PrioDeX middleware combines several novel capabilities and design features. This paper expands upon our previous work [10] to include experiments and experiences with our prototype implementation as well as the formal proof of our new analytical model and derivations of our priority assignment and drop rate policies. The key contributions of this paper are:

- Introducing a cross-layer approach (application, middleware, network) to prioritize mission-critical data exchange in IoT-enhanced smart spaces with SDN-enabled infrastructures (§2).
- Providing an analytical model using queueing theory that estimates performance metrics for cross-layer IoT interactions. This model includes our new multi-class priority queueing model. We use it here to represent an SDN switch, but it is generally suitable for use in other queueing networks (§3).
- Developing novel algorithms that leverage the above queueing model to explore the configuration parameter space for IoT event prioritization and delivery/delay tuning (§4).
- Implementing the PrioDeX prototype that integrates the above algorithms and an OpenFlow-enabled controller to configure the SDN-based underlying infrastructure (§5).
- Evaluating the PrioDeX middleware by: describing our experimental framework that relies on both simulation and prototype (enriched with an emulated network) implementations for configuring, and running experiments; evaluating our middleware's approach; comparing the proposed algorithms' performance; and validating our proposed analytical model (§6).

We conclude this paper in §8 with lessons learned and a look towards future work in this area.

2 OVERVIEW

In this section, we describe an IoT-enhanced structural fire scenario where efficient data exchange is necessary for satisfying situational awareness requirements of first responders. Then, we propose a cross-layer middleware approach to address these requirements via the efficient delivery of mission-critical data from IoT sources to relevant consumers.

2.1 Motivating Use Case Scenario

To motivate the need for timely IoT data exchange and highlight the challenges involved, we begin with an IoT-enhanced fire scenario. During a structural fire, an occupant or automated system activates an emergency dispatch process, which then notifies a local fire department. After some time, a team of *Fire Fighters* (FFs) along with an *Incident Commander* (IC) arrive; the IC is responsible for coordinating the effort from an *Incident Command Post* (ICP) set up onsite. To effectively manage the dynamic response and minimize casualties, injuries, and property damage, the IC requires up-to-date situational awareness information from the building. Today, the IC still derives much of this information from non-digital sources (e.g., human-initiated reports via voice or radio, paper records, etc). However, sensorized smartspaces (equipped with IoT devices) enable access to live data feeds that can generate actionable information in real-time via proper filtering, prioritizing, and analysis. Maintaining up-to-date situational awareness for *Smart Fire Fighting* (SFF) requires the integration and enrichment of static and dynamic data from buildings and IoT infrastructure. Static information such as building floor plans, inspection histories, and presence of hazardous material can be gathered apriori. For example, an emergency operations center may monitor third-party data streams (e.g., weather, social media) and forward relevant information to the ICP. Dynamic information published by IoT devices (in the building and brought by FFs) must be delivered to relevant stakeholders and combined with contextual knowledge to generate situational awareness.

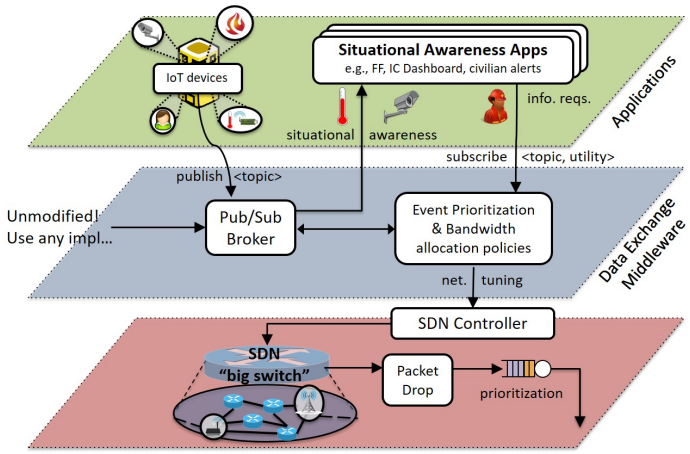


Fig. 1. The PrioDeX cross-layer middleware.

Such information includes: motion sensing, location, occupancy, activity tracking, smoke levels, air flow rate, audiovisual feeds, etc. Different data types vary in importance depending on the situation (e.g., "smoke" > "water pressure") and the stakeholders' (e.g., IC, FFs, residents) data requests. Then, stakeholders visualize or act based on the received information. For instance, an IC may use a tablet running a situational awareness dashboard to monitor the situation and coordinate the response effort. FFs may use some less-intrusive interface (e.g., a heads-up display on their glasses) to receive similar non-voice commands from the IC. A key challenge for SFF is the delivery of mission-critical data for "timely, targeted decision making" in an unreliable, partially available, and congested network environment [30].

2.2 Enabling Efficient Prioritization at the Edge

We now overview how the PrioDeX middleware is designed to address the requirements and challenges of the above the structural fire scenario. We frame our discussions in terms of the three layers depicted in Fig. 1: (i) mission-critical applications; (ii) abstractions representing the physical network infrastructure; and (iii) the data exchange middleware bridging these two layers. Our proposed solution aids in managing the overall system configuration and flow of information. PrioDeX integrates other state-of-the-art technologies: data APIs for interfacing with IoT data (e.g., from Edge devices in the building), a local pub/sub broker (or network of brokers), a thin client middleware running on each subscribing IoT device, and SDN APIs for managing the network infrastructure. It implements novel algorithms and provides middleware APIs for our data prioritization and network management approach. To ensure delivery of the most important events despite network resource constraints (e.g., failures, poor signal strength, limited bandwidth), it **prioritizes events** and **allocates available network bandwidth** according to application requirements.

Application layer. PrioDeX subscriber devices run a client middleware to establish broker connections, retrieve a list of event topics and subscribe to relevant ones. Since different data vary by importance, we propose prioritizing events according to their relative importance to the emergency response effort. To configure this, subscribers register **utility functions** with their PrioDeX subscriptions (see Fig. 1). These functions capture a quantified measure of value for varying rates of event delivery performance. Our proposed algorithms consider these utility functions when configuring the data exchange and network to maximize the users' situational awareness.

Data exchange layer. PrioDeX prioritizes subscriptions according to their subscriber-specified utility functions. It leverages the theoretic analysis we present in §3 to estimate system performance under given configurations. This analysis drives the algorithms presented in §4 that assign discrete

priority classes and allocate available network bandwidth. PrioDeX connects publishers (e.g., Edge devices) and subscribers (e.g., IC, FFs) with the data exchange broker, which performs the actual routing of events. While some existing data exchange implementations and protocols support priorities, configuring them requires specific APIs [47]. Furthermore, many popular options (e.g., the MQTT [43] protocol and associated broker implementations) do not support priorities and so require equal treatment of all events transmitted to the same subscriber. To decouple PrioDeX from the underlying pub/sub broker, which may be specific to the site's Edge devices, we do not employ app-layer (i.e., in-broker) prioritization. Rather, we propose enforcing priorities at the network layer through unified APIs provided by SDN. This approach accounts for both app-level requirements (e.g., utility functions) and network-level state information (e.g., available bandwidth) without mandating (or extensively modifying) specific broker technologies. Hence, PrioDeX essentially extends the data exchange broker/protocol with network and application-aware prioritization.

Network layer. PrioDeX manages the network infrastructure through APIs provided by an SDN controller that likely runs alongside the Edge. It gathers network state information to derive resource constraints. This is combined with the subscribers' information requirements to drive its management algorithms. Zhang K and Jacobsen HA previously advocated for a similar approach [62] of a centrally gathered global view of pub/sub system's state to simplify its management. They refer to this central control approach as *SDN-like* because it separates the pub/sub control and data plane. They further propose integrating SDN with the data exchange middleware, which this centralization cleanly enables. We advocate for this approach in IoT settings when offloading device management and data processing from constrained devices leads to centralized (e.g., cloud-centric) designs. For simplicity of discussion, we consider the ***big switch*** model shown in Fig. 1 that abstracts the entire local physical network into a single virtual SDN switch. This provides a simplified single-network view of the whole distributed system that may span multiple physical heterogeneous networks (e.g., Wi-Fi or local cellular) and different locations.

To enforce event priorities at the network layer, PrioDeX leverages SDN APIs. It configures priority queueing disciplines for packets matching the different subscriptions. However, for the network to distinguish the data exchange-layer concept of subscriptions, we must first translate it to a network-level concept. As shown in Fig.2, we accomplish this through the SDN concept of **network flows**. SDN switches match incoming packets of a particular network flow according header information. For example, OpenFlow considers OSI Layer 2-4 fields such as IP/MAC address, UDP/TCP port, VLAN, etc. To differentiate subscriptions belonging to different network flows, a PrioDeX subscriber maintains multiple **network connections** with the pub/sub broker (e.g., over different OSI Layer 4 port numbers). This may represent different applications running on the same device and/or one application opening multiple connections. The latter case enables the network to distinguish and manage individual groups of subscriptions based on their assigned connection. The data exchange layer dictates this assignment of (possibly multiple) subscriptions to one network connection and its corresponding unique network flow. Subscribers initiate multiple connections and then register each subscription to avoid directly configuring the underlying data exchange broker. PrioDeX also assigns each network flow a priority level by considering subscriber requirements. It configures the SDN switches to forward packets matching these network flows through the proper priority queue. To manage available network resources, PrioDeX also allocates bandwidth to each network flow. It applies **preemptive packet drop rates** that consider the utility of each network flow's subscriptions. We propose dropping lower-priority packets before switch buffers fill up to prevent high delays and dropping of higher-priority packets. §4.3 discusses this concept further and proposes our optimization-based algorithm for setting these drop rates. Our proposal leverages discrete priority classes to drive priority queueing disciplines and defines the best priority assignments rather than assuming them as a given input.

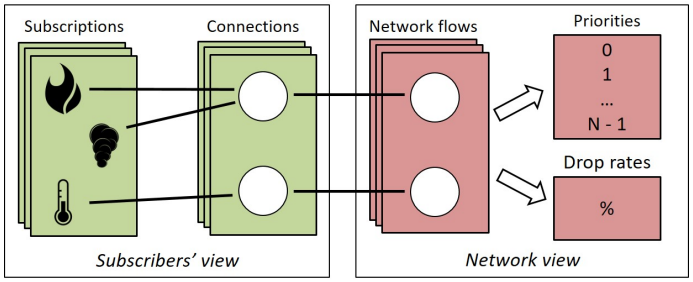


Fig. 2. PrioDeX prioritizes subscriptions at the SDN layer using multiple connections per subscriber.

Notation	Description
Application Layer	
$v_j \in V$	event topics
$s_i \in S$	subscribers
$r_j \in R$	subscriptions
$p_i \in P$	publishers
$\lambda^{pub}_{p_i,v_j}$	topic v_j's pub rate
$\lambda^{sub}_{r_j}$	r_j's delivery rate
Ξ_{r_j}	r_j's success rate
Δ_{r_j}	r_j's end-to-end response time

Notation	Description
Data Exchange Layer	
$b_k \in B$	brokers
$\lambda^{notify}_{b_k,r_j}$	r_j's notification rate
$\Psi : R \mapsto F$	network flow for a subscription
$\Phi : F \mapsto Y$	priority for a network flow
$\Omega : F \mapsto [0, 1]$	packet drop rate for a network flow
Network Layer	
$x_k \in X$	SDN switches
$h_j \in H, H = P \cup S \cup B$	network hosts
$w_{x_k,h_j} \in W, w_{x_k,h_j} \in \mathbb{N}$	bandwidth between x_k and h_j
$G_{v_j} \in \mathbb{Z}_{>0}$	serialized packet size for topic v_j
$z_{h_j,h_i} \in Z, z_{h_j,h_i} \in [0, 1]$	packet *error rate*
$\Gamma : \mathbb{N} \times H \times H \mapsto \mathbb{N}$	transforms event departure to arrival rates (e.g., packet errors)
$f_j \in F$	network flows
$y_j \in Y$	unique *priority classes*

Table 1. Notations of the parameters in our cross-layer data exchange model.

3 PRIODEX FORMAL MODEL

To enable timely and reliable data exchange in IoT systems, existing solutions propose creating performance models that can be leveraged for system tuning. Such models must consider all three layers' characteristics and their effects on each other. Existing efforts typically focus on each layer in isolation to model the performance of middleware systems [14, 36, 49], network infrastructures [7, 29] and more recently SDN infrastructures [23, 55]. In this paper, we model cross-layer interactions by composing and extending previous work [14, 15] at each layer through the unified framework of *queueing theory* [31, 52]. PrioDeX combines queueing theoretic approaches from both the middleware and network layers to construct the representative and extensible 3-layer queueing network shown in Fig. 3. At the middleware layer, M/M/1 queues are used to model the subscription matching process and the delivery of events to subscribers. At the networking layer, M/M/1, multi-class and priority-class queues are used to model packet processing, transmission, and prioritization. When increasing the number of brokers, subscribers and switches, the number of queues leveraged are increased as well. The data exchange middleware bridges the network infrastructure and application layers to enable a novel cross-layer end-to-end performance model. We derive this analytical model to estimate a particular configuration's expected performance.

3.1 Queueing Network Performance Modeling

Refer to Table 1 for the notations used throughout this section.

Application Modeling. Each publisher p_i publishes to a set of topics $V_{p_i} \subseteq V$ (e.g., "smoke"). Let $\lambda^{pub}_{p_i,v_j}$ be the publication rate of events with topic v_j published by p_i per unit time.

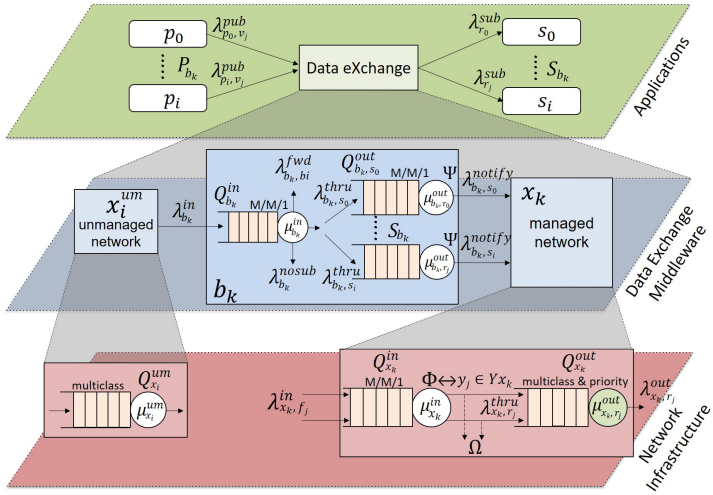


Fig. 3. PrioDeX queueing network model.

Assumption 1. $\lambda^{pub}_{p_i,v_j}$ *is based on a Poisson process.*

We define each subscription as a tuple $r_j = (s_i, v_j, U_{r_j})$ where the utility function U_{r_j} quantifies the information value for subscriber s_i receiving events with topic v_j. Let $R_{s_i} = \{r_j \in R : s_i \in r_j\}$ be the set of prioritized information requests (i.e., subscriptions) for each subscriber s_i. Let $\lambda^{sub}_{r_j}$ be the incoming rate of events matching subscription r_j received per unit time by subscriber s_i.

Let Ξ_{r_j} be the *success rate* of delivering $\lambda^{sub}_{r_j}$, which can be estimated as follows:

$$\mathrm{E}\left[\Xi_{r_j}\right] = \frac{\lambda^{sub}_{r_j}}{\sum_{p_i \in P} \lambda^{pub}_{p_i,v_j}} \tag{1}$$

where the denominator aggregates events produced from publishers for topic v_j (Assumption 1).

Let Δ_{r_j} be the average *end-to-end response time* of events matching subscription $r_j = (s_i, v_j, U_{r_j})$ from the moment they are published until s_i receives them; this is a function of event processing, queueing, transmission and propagation delays (defined later in this section).

Data Exchange Modeling. The data exchange layer represents a network of broker nodes B. We assume that each publisher p_i connects to its *home broker* b_{p_i}, which we define as the broker to which p_i publishes events. A home broker b_{s_i} is defined analogously for each subscriber s_i. Furthermore, we define the set of publishers connected with b_k as $P_{b_k} = \{p_i \in P : b_k = b_{p_i}\}$, the set of subscribers connected with b_k as $S_{b_k} = \{s_i \in S : b_k = b_{s_i}\}$, and the set of subscriptions handled by b_k as $R_{b_k} = \cup_{s_i \in S_{b_k}} R_{s_i}$. A broker b_k forwards events with rate $\lambda^{fwd}_{b_k,b_i}$ to another broker $b_i \in B$ for event delivery to b_i's subscribers. As depicted in Fig. 3, we model each broker b_k using a single inbound M/M/1 queue $Q^{in}_{b_k}$ and multiple outbound M/M/1 queues $Q^{out}_{b_k,s_i}$. By Assumption 1 and the exponentially distributed service rate of $Q^{in}_{b_i}, \forall b_i \in B - \{b_k\}$, we know that $\lambda^{fwd}_{b_k,b_i}$ follows a Poisson distribution. Hence, we can define the arrival rate of events at $Q^{in}_{b_k}$ as the sum of all (post-network transformation) event publication/forwarding rates over all publishers/brokers:

$$\lambda^{in}_{b_k} = \sum_{p_i \in P_{b_k}} \sum_{v_j \in V_{p_i}} \Gamma\left(\lambda^{pub}_{p_i,v_j}, p_i, b_k\right) + \sum_{b_i \in B, b_i \neq b_k} \Gamma\left(\lambda^{fwd}_{b_i,b_k}, b_i, b_k\right) \tag{2}$$

where Γ, which we define later in this section, represents network-layer traffic shaping due to error rates, administrative policies, etc.

Forwarding, replication, or dropping of events based on current subscriptions occurs at the exit of $Q^{in}_{b_k}$. Let $\mu^{in}_{b_k}$ be $Q^{in}_{b_k}$'s service rate for analyzing an incoming event and determining where to forward it (e.g., based on a topic routing tree). We assume $\mu^{in}_{b_k}$ is constant (or averaged) across all topics and independent of current subscriptions. Events not matching subscriptions R_{b_k} are dropped with rate $\lambda^{nosub}_{b_k}$. For each subscriber $s_i \in S_{b_k}$, b_k forwards events matching a subscription $r_j \in R_{s_i}$ to $Q^{out}_{b_k,s_i}$ with rate $\lambda^{thru}_{b_k,s_i}$ for transmission to s_i. Recall that each broker maintains multiple connections (i.e., network flows) with each subscriber. Let $\mu^{out}_{b_k,r_j}$ be the service rate at $Q^{out}_{b_k,s_i}$ that captures the time it takes to map an event matching subscription r_j to the corresponding connection of s_i. It forwards these publications into the network layer with rate $\lambda^{notify}_{b_k,r_j}$. Hence, we define the per-subscriber forwarding rate as:

$$\lambda^{notify}_{b_k,s_i} = \sum_{r_j \in R_{s_i}} \lambda^{notify}_{b_k,r_j} \tag{3}$$

PrioDeX Configuration Parameters. The data exchange layer also represents the PrioDeX configuration service. PrioDeX associates each subscription with one of the *network flows* $f_j \in F$ in order to manage subscription traffic in a network-aware manner. Recall from §2 that network flows represent multiple connections between a subscriber and its home broker. We define the set of network flows for a particular subscriber s_i as $F_{s_i} \subseteq F$. Additionally PrioDeX defines a set of unique *priority classes* $y_j \in Y$ to which each network flow is assigned; this enables network traffic to be managed in an application-aware manner. Note that y_j has higher priority than y_k for $j < k$ – i.e., y_0 is the highest priority. To configure the end-to-end data exchange interactions across all 3 layers, PrioDeX employs the following functions:

$\mathbf{\Psi} : \mathbf{R} \mapsto \mathbf{F}$ is the function that maps subscriptions (i.e., events matching them) to the corresponding subscribers' network flows. Note that we denote $\Psi(s_i, v_j) = \Psi(r_j)$ as the network flow for subscription $r_j = (s_i, v_j, U_{r_j})$ and so $\Psi : S \times V \mapsto F$. As described in §2, this mapping allows the SDN data plane to distinguish packets containing events from each other, based on their subscriptions.

$\mathbf{\Phi} : \mathbf{F} \mapsto \mathbf{Y}$ is the function mapping network flows to priority classes. This defines which priority class (i.e., priority queue) the SDN infrastructure uses for a packet transmitted on network flow f_j. This packet contains event(s) matching subscriber s_i's subscription r_j where $f_j = \Psi(r_j)$. Hence $\Phi \circ \Psi(r_j)$ is subscription r_j's priority.

$\mathbf{\Omega} : \mathbf{F} \mapsto [\mathbf{0}, \mathbf{1}]$ is the function mapping network flows to preemptive packet drop probabilities. By dropping some packets on a network flow, PrioDeX tunes the data exchange configuration more accurately than through priority assignment alone. Somewhat akin to network traffic policing, this technique lowers the bandwidth usage of a network flow so that the aggregate bandwidth needs of all flows does not exceed that which is available. By dropping packets in the lower-priority flows, this prevents switch buffers from filling up and dropping higher-priority packets.

Network Modeling. Publications forwarded to the network layer are encapsulated in packets for transmission by the SDN infrastructure. To simplify our analysis, we leverage the following:

Assumption 2. *The data exchange and applications encapsulate each event in a single packet for transmission through the network.*

Let X be the set of *SDN switches* that connect with the various hosts H. A host h_j may have multiple physical network interfaces/connections to one or more switches and packets between two hosts may traverse multiple routes. However, SDN abstractions support the following assumption that simplifies our analysis:

Assumption 3. *We consider multiple switches/routes between two hosts as aggregated into a single virtual SDN switch/link that captures the underlying physical network topology and characteristics.*

By Assumption 3, we need only to model a single *big switch* serving a publisher or subscriber. Hence, we refer to x_{s_i} as the *PrioDeX-managed SDN switch* that controls traffic between b_{s_i} and s_i.

We refer to x_{p_i} as the *unmanaged SDN switch* that exposes the network characteristics (defined below) of the network channel between b_{p_i} and p_i. Note that PrioDeX does not manage the latter switch because this might conflict with deployment-specific IoT device configurations. To model multiple hosts sharing the same network medium (e.g., a wireless channel), we apply Assumption 3 and model such a channel as one switch serving multiple hosts. Therefore, we define the set of subscribers served by switch x_k as $S_{x_k} = \{s_i \in S : x_{s_i} = x_k\}$, all of their subscriptions as $R_{x_k} = \{\cup_{s_i \in S_{x_k}} R_{s_i}\}$, and all of their network flows as F_{x_k}. Similarly, let $P_{x_k} = \{p_i \in P : x_{p_i} = x_k\}$ be the set of publishers served by x_k.

Let $Q^{um}_{x_i}$ be the queue modeling the *unmanaged switch* x_i that encompasses a *publisher-broker* or *broker-broker* link. By Assumption 2, we have the packet arrival rate for publications and forwarded events at switch x_i as $\lambda^{pub}_{p_i,v_j}$ and $\lambda^{fwd}_{b_k,b_i,v_j}$ respectively. We model $Q^{um}_{x_i}$ as a multi-class queue, which enables us to define the average transmission delay of a packet ($\Delta^{tx}_{r_j}$) based on its size. Each class corresponds to the topic of an event encapsulated within a packet. Hence, we define the expected *serialized size* (e.g., in bytes) of a packet that, by Assumption 2, contains a single event published to topic v_j as $G_{v_j} \in \mathbb{Z}_{>0}$. Using Assumption 3, we have w_{x_k,h_j} as the bottleneck bandwidth available between two hosts (i.e., from the switch x_k serving them to the destination host h_j). Therefore, we can define a per-topic packet transmission rate as:

$$\mu^{um}_{x_i,v_j} = \frac{w_{x_i,b_k}}{G_{v_j}} \tag{4}$$

Equation 4 is used to estimate the average *transmission delay* $\Delta^{um}_{x_i}$ (see 15). We apply Γ to packets departing the switch queue $Q^{um}_{x_i}$ in order to transform event departure rates from a host h_j to event arrival rates at the destination host h_i. To simplify our analysis, we leave retransmission of packets for future work and instead consider only packet error rates. Let $z_{h_j,h_i} \in [0,1]$ be this packet error rate that allows us to model packet drops at the single switch between these hosts. We have the arrival rate of publications (on topic v_j from publisher p_i at broker b_k) as:

$$\Gamma\left(\lambda^{pub}_{p_i,v_j}, p_i, b_k\right) = \left(1 - z_{p_i,b_k}\right) \lambda^{pub}_{p_i,v_j} \tag{5}$$

We define the transformed arrival rate of events forwarded from broker b_i to b_k similarly.

We model each *managed SDN switch* encompassing a *broker-subscriber* link as two different queues: *1)* an M/M/1 queue $Q^{in}_{x_k}$ that feeds into *2)* our newly-proposed queueing model: a non-preemptive priority and multi-class queue $Q^{out}_{x_k}$. By Assumption 2, we therefore have the arrival rate at switch x_k of event-encapsulating packets within a network flow f_j as $\lambda^{in}_{x_k,f_j}$. $Q^{in}_{x_k}$ processes each incoming packet by matching its header contents to a corresponding network flow f_j and determining the assigned priority (i.e., $\Phi(f_j)$). Let $\mu^{in}_{x_k}$ be the service rate at $Q^{in}_{x_k}$ that captures the time required to perform this matching (e.g., an SDN switch TCAM lookup), assign the given priority, and route the packet to the appropriate output port. Note that this might actually capture delays from forwarding packets along a multi-switch route. Before enqueueing the packet at the correct output port, we have the per-subscription arrival rate at $Q^{out}_{x_k}$ as:

$$\lambda^{thru}_{x_k,r_j} = \left(1 - \Omega \circ \Psi\left(r_j\right)\right) \lambda^{notify}_{b_k,r_j} \tag{6}$$

where the switch first applies the dropping policy to each flow (i.e., $\Psi(r_j)$) according to the PrioDeX-computed function Ω.

Multi-class priority queue $Q^{out}_{x_k}$ separates the departure rates of each packet according to its serialized size and the switch's available bandwidth. Note that the assigned priority class affects the response time but not the departure rates of these packets. By Assumption 2, we have the service (i.e., transmission) rate of packets encapsulating events that match subscription $r_j = (s_i, v_j, U_{r_j})$ from SDN switch x_k to subscriber s_i as:

$$\mu^{out}_{x_k,r_j} = \frac{w_{x_k,s_i}}{G_{v_j}} \tag{7}$$

We have the departure rate from $Q^{out}_{x_k}$ as:
$\lambda^{out}_{x_k,r_j} = \lambda^{thru}_{x_k,r_j}$.

We then apply Assumption 2 and Γ to packets departing switch queue $Q^{out}_{x_k}$. Considering packet error rates, we have the arrival rate of events at subscriber s_i matching subscription $r_j = (s_i, v_j, U_{r_j})$ as:

$$\Gamma\left(\lambda^{out}_{x_k,r_j}, b_{s_i}, s_i\right) = \lambda^{sub}_{r_j} = \left(1 - z_{b_{s_i},s_i}\right) \lambda^{out}_{x_k,r_j} \tag{8}$$

3.2 End-to-end Analytical Model

We now leverage our queueing network to derive theoretical performance results. This analysis, the accuracy of which we validate in §6.4, enables PrioDeX to tune the data exchange performance characteristics of end-to-end event response time and delivery success rate. To define Δ_{r_j}, the end-to-end response time of events for subscription r_j, we define the propagation and queueing delays at each layer. Note that the queueing delay in our model captures the real-world processing and network transmission delays.

To simplify our analysis, we exploit the local nature of our target scenario and consider only a single broker (b_k) for the remainder of this section. Future work will explore relaxing this assumption and extending this analysis to include the more general scenario of a distributed broker network enabled by our queueing network model above. By the above assumption, we must define the per-subscription end-to-end response time metric denoted by Δ_{r_j}, which is the expected delay from any such publisher to broker b_k considering both the queueing delay at the intermediate switch x_{p_i} and heterogenous propagation delays. Therefore, we have:

$$\mathrm{E}\left[\Delta^{prop}_{p_i,b_k} + \Delta^{um}_{x_{p_i}}\right] = \sum_{\{p_i \in P_{b_k} : v_j \in V_{p_i}\}} \frac{\Delta^{prop}_{p_i,b_k} + \Delta^{um}_{x_{p_i}}}{|P(p_i, v_j)|} \tag{9}$$

where $\Delta^{prop}_{p_i,b_k}$ is the *propagation delay* (i.e., physical network latency) between a publisher $p_i \in P_{b_k}$ and the broker b_k, $|P(p_i, v_j)|$ is the number of maximum publishers producing events on topic v_j – i.e., $|\{p_i \in P_{b_k} : v_j \in V_{p_i}\}|$ and $\Delta^{um}_{x_{p_i}}$ is the transmission delay of packets passing through the switch. Then, by using (9), we complete the calculation of Δ_{r_j} as follows:

$$\Delta_{r_j} = \mathrm{E}\left[\Delta^{prop}_{p_i,b_k} + \Delta^{um}_{x_{p_i}}\right] + \Delta_{b_k} + \Delta^{prop}_{b_k,s_i} + \Delta_{x_{s_i}} \tag{10}$$

where Δ_{b_k} is the processing delay of events passing through b_k; $\Delta^{prop}_{b_k,s_i}$ is the *propagation delay* between the broker and the subscriber s_i; and and $\Delta_{x_{s_i}}$ is the transmission delay of packets passing through x_{s_i}. The average response time of (10) includes queueing delays at each layer of PrioDeX. Based on the queueing network representing PrioDeX (see Fig. 3), we identify the type of each queueing model and their arrival/processing/transmission rates.

At the data exchange layer we use M/M/1 queues. Based on standard solutions for M/M/1 queues (see page 62, equation 2.26 in [27]), the time that an event remains in the system (i.e., queueing time + service time; also called average delay) is given by:

$$\Delta_{Q_{mm1}}(\mu, \lambda) = \frac{1}{(\mu - \lambda)} \tag{11}$$

At the network layer, we use three different types of queueing models: (i) the M/M/1 queue ($Q^{in}_{x_k}$); (ii) the multi-class queue ($Q^{um}_{x_i}$, unmanaged switch queue) and (iii) the non-preemptive priority and multi-class queue ($Q^{out}_{x_k}$, SDN switch queue). Note that each class corresponds to the topic of

an event encapsulated within a packet. Based on standard queueing theoretic solutions [37], the average delay for events matching a particular subscription r_k is given by:

$$\Delta_{Q_{mcl}}(\mu, \lambda, r_k) \;=\; \frac{1}{\mu_{r_k} \;-\; \mu_{r_k} \;\sum_{r_j \in R} \lambda_{r_j}/\mu_{r_j}} \tag{12}$$

where $\lambda = \{\lambda_{r_j} : r_j \in R\}$ and $\mu = \{\mu_{r_j} : r_j \in R\}$.

Finally, the SDN switch is modeled using the non-preemptive priority and multi-class queue (Q_{x_k}). Hence, the average delay of packets for r_k assigned with y_j is given by:

$$\Delta_{Q_{mclpr}}(\mu, \lambda, r_k) \;=\; \frac{L_{r_k}(\lambda, \mu)}{\lambda_{r_k}} \tag{13}$$

where $\lambda = \{\lambda_{r_j} : r_j \in R\}$, $\mu = \{\mu_{r_j} : r_j \in R\}$ and L_{r_k} is the number of events matching subscription r_k with assigned priority y_c (where $\Phi \circ \Psi(r_k) = y_c$) in the system (queue + server) of Q_{mclpr}. See Appendix A for our proof of (13).

By relying on the above analytical models, we define the average delay of events for any subscription r_j at each node and layer of the PrioDeX queueing network according to (10).

Data Exchange. At this layer, the average delay at b_k (Δ_{b_k}) is given by calculating the queueing delay of events matching r_j at both inbound ($Q^{in}_{b_k}$) and outbound ($Q^{out}_{b_k,s_i}$) queues – i.e., $\Delta_{b_k} = \Delta_{Q^{in}_{b_k}} + \Delta_{Q^{out}_{b_k,s_i}}$. Both queues are of M/M/1 type. For $Q^{in}_{b_k}$, the incoming rate of events is $\lambda^{in}_{b_k}$ and its service rate is $\mu^{in}_{b_k}$; for $Q^{out}_{b_k,s_i}$ the incoming rate of events is $\lambda^{thru}_{b_k,s_i}$ and the service rate is $\mu^{out}_{b_k,r_j}$. Hence, we apply (11) to determine:

$$\Delta_{b_k} = \Delta_{Q_{mm1}}\left(\mu^{in}_{b_k}, \lambda^{in}_{b_k}\right) \;+\; \Delta_{Q_{mm1}}\left(\mu^{out}_{b_k,r_j}, \lambda^{thru}_{b_k,s_i}\right) \tag{14}$$

Network. At this layer, the average delay ($\Delta^{um}_{x_i}$) at the unmanaged switch x_i (*publishers-broker* link) is given by calculating the queueing delay of packets matching r_k at the multi-class $Q^{um}_{x_i}$ queue. Hence, using the analytical model of (12) such a delay is given by:

$$\Delta^{um}_{x_i} = \Delta_{Q_{mcl}}\left(\{\mu^{um}_{x_i,v_j} : v_j \in V\}, \{\lambda^{pub}_{p_i,v_j} : p_i \in P_{x_i}, v_j \in V_{p_i}\}, r_k\right) \tag{15}$$

At the SDN switch x_k (*broker-subscribers* link), the average delay (Δ_{x_k}) is given by estimating the queueing delay for packets matching r_j at both the inbound ($Q^{in}_{x_k}$) and outbound ($Q^{out}_{x_k}$) queues – i.e., $\Delta_{x_k} = \Delta_{Q^{in}_{x_k}} + \Delta_{Q^{out}_{x_k}}$. In the M/M/1 queue $Q^{in}_{x_k}$, packets arrive at a per-flow rate $\lambda^{in}_{x_k,f_j}$ and are served with rate $\mu^{in}_{x_k}$. Hence, by applying (11), $\Delta_{Q^{in}_{x_k}} = \Delta_{Q_{mm1}}\left(\mu^{in}_{x_k}, \lambda^{in}_{x_k,f_j}\right)$.

The outbound queue ($Q^{out}_{x_k}$), a multi-class and non-preemptive priority queue, has a per-subscription packet arrival rate $\lambda^{thru}_{x_k,r_j}$. Its service rates $\mu^{out}_{x_k,r_j}$ capture the specific event/packet size of the corresponding $r_k = (s_i, v_j, U_{r_j})$. Hence, we apply (13) to find:

$$\Delta_{Q^{out}_{x_k}} = \Delta_{Q_{mclpr}}\left(\{\mu^{out}_{x_k,r_j} : r_j \in R_{x_k}\}, \{\lambda^{thru}_{x_k,r_j} : r_j \in R_{x_k}\}, r_k\right) \tag{16}$$

According to Fig. 3 and (10), to estimate the average response time for events matching subscription r_j of a single subscriber, we must consider the propagation and queueing delays for events passing through one broker and two switches. In particular, we have: (i) one multi-class queue in the *publishers - broker* switch (15); (ii) two M/M/1 queues in the broker (14); (iii) one M/M/1 queue and one multi-priority queue in the *broker - subscriber* switch (11,16). The time required to determine the average response time using (10) is typically in the order of a few milliseconds. The time complexity of (10) increases linearly as the number of switches increases (which is proportional to the queues in the network). This is because we must consider the queueing and propagation delays for the events passing via the additional *publisher - broker* and *broker - subscriber* switches.

4 DATA EXCHANGE CONFIGURATION ALGORITHMS

The core algorithms of PrioDeX leverage the above analytical model to configure the SDN-enabled data exchange. Considering current system state and information requirements, they assign priorities and preemptive drop rates to subscriptions (i.e., via $\Phi \circ \Psi$, Ω) in order to maximize subscriber-defined *utility functions*.

4.1 Utility Functions

To capture the relative value of information for different subscriptions, we propose using *utility functions*. Subscribers include a utility function with their subscriptions. Utility functions directly affect rate of successful event delivery Ξ_{r_j} and response time Δ_{r_j}. The overall utility for a subscriber depends on each of its subscriptions' utilities and is defined as:

$$U_{s_i} = \sum_{r_j \in R_{s_i}} U_{r_j}(\Xi_{r_j}) \tag{17}$$

Let $\widehat{U_{r_j}}$ be a subscription's maximum achievable utility: delivering the maximum number of events under ideal network conditions (i.e., no loss, minimal latency, no other traffic).

To further capture the relative value of information between each subscriber, we consider an overall utility of all subscribing first responders. Each subscriber may define different utility functions to capture the fact that each of their needs vary (e.g., the IC may require more situational awareness than individual firefighters). We define the overall utility of the configuration for all subscribers as a sum over each individual subscriber's utility:

$$U = \sum_{s_i \in S} U_{s_i} \tag{18}$$

To model heterogeneous information requirements in our experiments, we generate different utility functions for each subscription. We define the base utility function as:

$$U_{r_j}(\Xi_{r_j}) = \alpha_{r_j} \log(1 + \Xi_{r_j}) \tag{19}$$

where the utility weight α_{r_j} is varied for each subscription.

4.2 Priority Assignment Algorithm

PrioDeX leverages the above quantified utility metrics to assign priorities for each data flow in a manner that aims to maximize the overall system utility. We decouple the assignment of priorities from that of drop rates for two reasons. Prioritization ensures the most important events get through *first*, but it does not necessarily provide guarantees about *how much* data is delivered. Hence, we first assign the priorities and then optimally set the preemptive drop rates to tune bandwidth usage for the network flows in each priority class. Second, this decoupling allows us to explore different policies in these two spaces independently.

Greedy Split. Because the assignment of discrete priorities for maximizing the utility is non-trivial, we propose a heuristic to approximate a solution. It first ranks subscriptions according to their maximum utility $\widehat{U_{r_j}}$ scaled by the corresponding required bandwidth. This measures *information value per unit bandwidth* and lets PrioDeX consider that some high-value subscriptions may consume a lot of network resources. We define this utility weight as follows:

$$\alpha_{r_j} = \frac{\widehat{U_{r_j}}}{G_{v_j} \lambda^{notify}_{b_k, r_j}} \tag{20}$$

We provide a solution to the priority-assignment problem through the following greedy approach:

(1) Sort the subscriptions $r_j \in R_{s_i}$ by (20)

(2) Split this list into $|F_{s_i}|$ sub-lists of approximately equal size
(3) Assign $\Psi(r_j) = F_{s_i}(k)$ for each $r_j \in$ sub-list number k
(4) Split the list of flows $F_{s_i} \forall s_i \in S$ into approximately $|Y|$ sub-lists of approximately equal size
(5) Assign $\Phi(f_j) = y_k$ for each $f_j \in$ sub-list number k

Note that this splitting up of lists handles unequally-sized splits by preferring higher priorities first.

Cluster Split. We also propose the following cluster approach. It consists of the same first three steps as the greedy approach, then:

(4) Split the list of flows $F_{s_i} \forall s_i \in S$ into $|Y|$ sub-groups leveraging the k-means clustering method and the following utility weight (per network flow) to evaluate each data point:

$$\alpha_{f_j} = \sum_{r_j \in f_j} \alpha_{r_j} \tag{21}$$

(5) Assign $\Phi(f_j) = y_k$ for each f_j that belongs to the group with priority y_k.

The main difference between the two approaches is that the greedy split has fixed group size, while the cluster split groups different network flows depending only on their information value (i.e., Eq. 21). Both priority assignments ensure delivery of the highest-priority events if possible. However, an overloaded system will fill switch buffers and lead to high delay and loss of lower-priority events. Hence, we apply preemptive drop rates to avoid such a case.

4.3 Ensuring Queue Stability via Preemptive Drop Rates

Given a priority assignment, subscription utility functions, and the current network state (e.g., bandwidth constraints), PrioDeX further fine-tunes the subscriptions' successful notification rate Ξ_{r_j} by applying a packet dropping policy. This improves the overall utility of the system's configuration by allocating available bandwidth to the network flows. In addition, this bandwidth allocation also ensures *queue stability* throughout the network. That is, if packets arrive at the switches' inbound queues too quickly, the forwarding queues will grow in size until the buffers fill up and packets are dropped. To prevent the dropping of high-value events, PrioDeX preemptively drops lower-priority packets. The algorithms presented in this section determine the probability with which packets of each network flow should be dropped ($\Omega(f_j)$). Here, the goal is to satisfy the situational awareness requirements and the conditions necessary for our analytical model's results to be accurate, while also ensuring queue stability and improving the overall system performance.

Let $\rho_Q = \frac{\lambda}{\mu}$ be the server utilization (i.e., the probability that the server is busy) of the corresponding queue (e.g., $Q^{out}_{x_k}$). By [27], the system remains *unsaturated* (i.e., queue stability is ensured) when $\rho_Q < 1$. For PrioDeX's M/M/1 queues (i.e., $Q^{in}_{b_k}$, $Q^{out}_{b_k,s_i}$, $Q^{in}_{x_k}$) we define: $\rho_{Q^{in}_{b_k}} = \frac{\lambda^{in}_{b_k}}{\mu^{in}_{b_k}}$, $\rho_{Q^{out}_{b_k,s_i}} = \frac{\lambda^{thru}_{b_k,s_i}}{\mu^{out}_{b_k,r_j}}$ and $\rho_{Q^{in}_{x_k}} = \frac{\lambda^{in}_{x_k,f_j}}{\mu^{in}_{x_k}}$. PrioDeX's multi-class queues $Q^{um}_{x_i}$ and $Q^{out}_{x_k}$ have per-topic and per-subscription arrival and service rates, respectively. Thus, we can estimate the per-class server utilization as well as the overall server utilization for each queue as:

$$\rho_{Q^{um}_{x_i}} = \sum_{P_{x_i}} \sum_{v_j \in V_{p_i}} \frac{\lambda^{pub}_{p_i,v_j}}{\mu^{um}_{x_i,v_j}} \tag{22}$$

$$\rho_{Q^{out}_{x_k}} = \sum_{r_j \in R_{x_k}} \frac{\lambda^{thru}_{x_k,r_j}}{\mu^{out}_{x_k,r_j}} \tag{23}$$

To improve successful delivery rate while ensuring queue stability, we propose several algorithms of increasing sophistication below. Note that these algorithmic formulations currently only consider the outbound queue of the SDN switches for this constraint, as tuning the drop rates only affects

$\rho_{Q_{x_k}^{out}}$. Also recall that this queue captures the bottleneck bandwidth of the network route from broker to subscriber. Future work will explore simultaneously balancing the load across data exchange brokers to also ensure stability of their queues within our model.

Each algorithm makes use of a parameter $\tilde{\rho}$ in tuning the system's tolerance to approaching (but never exceeding) the queue saturation point of $\rho_{Q_{x_k}^{out}} = 1$. Clearly, to satisfy the strict inequality $\rho_{Q_{x_k}^{out}} < 1$ we must have $\tilde{\rho} > 0$. Increasing $\tilde{\rho}$ provides ample buffer within the SDN switch queues for resilience against temporary notification rate spikes that might otherwise lead to queue saturation. However, even if this condition is just barely satisfied (e.g., $\tilde{\rho} = 10^{-10}$), queues will still grow quite large and thereby cause high delay. Therefore, the following drop rate policies set Ω such that:

$$\rho_{Q_{x_k}^{out}} = 1 - \tilde{\rho} \tag{24}$$

Flat drop rates: this simple naive policy sets all drop rates equal to satisfy Eq. (24) by solving Eq. (23) for a parameter β such that:

$$\Omega(f_j) = \beta \tag{25}$$

Linear drop rates: this value-aware policy sets the drop rates for each network flow according to its assigned priority level. It solves Eq. (23) for a parameter β that satisfies Eq. (24) with drop rates set to:

$$\Omega(f_j) = \beta\Phi(f_j) \tag{26}$$

Exponential drop rates: similar to *Linear*, this policy sets drop rates according to priority level. It solves Eq. (23) for a parameter β that satisfies (24) with drop rates set to:

$$\Omega(f_j) = 1 - \beta^{-\Phi(f_j)} \tag{27}$$

To compute the parameter β in (25,26,27), we use the analytical solutions presented in Appendix B, i.e., (47) for Flat, (48) for Linear and (54) for Exponential drop rates.

Optimized drop rates: the following convex optimization formulation assigns drop rates to maximize overall utility (see 18). Given the assigned priorities to network flows as input, PrioDeX assigns drop rates by solving the following:

$$\begin{aligned} \text{maximize} \quad & U \\ \text{subject to} \quad & \Omega(f_j) \in [0,1], \forall f_j \in F \\ & \rho_{Q_{x_k}^{out}} \leq 1 - \tilde{\rho},\ \forall x_k \in X \end{aligned} \tag{28}$$

Note that the first constraint defines the feasible domain of assigned drop rates for each network flow f_j, and the second constraint ensures that available bandwidth constraints are met (i.e., queue stability) according to the $\tilde{\rho}$ parameter for each SDN switch x_k. As long as the chosen utility functions are concave (e.g., logarithm such as 19), then (28) can be expressed as a convex optimization problem and efficiently solved. We used CVXPY [2, 22] that solves convex optimization problems to assign drop rates to PrioDeX network flows.

5 PROTOTYPE IMPLEMENTATION

We now present the PrioDeX prototype which implements the cross-layer architecture (Fig. 1 in §2), the underpinning theoretical model (§3) and the algorithms (§4). The main software components developed and technologies used are shown in Fig. 4. Among these components, the *PrioDeX Coordinator Service* (PCS) is a part of the data exchange layer and provides configuration parameters to the network infrastructure through SDN. It runs the algorithms to compute priorities and drop rate policies, which are enforced via an SDN controller that configures SDN switches. The PrioDeX source code and the detailed documentation is provided on: https://github.com/boulouk/priodex.

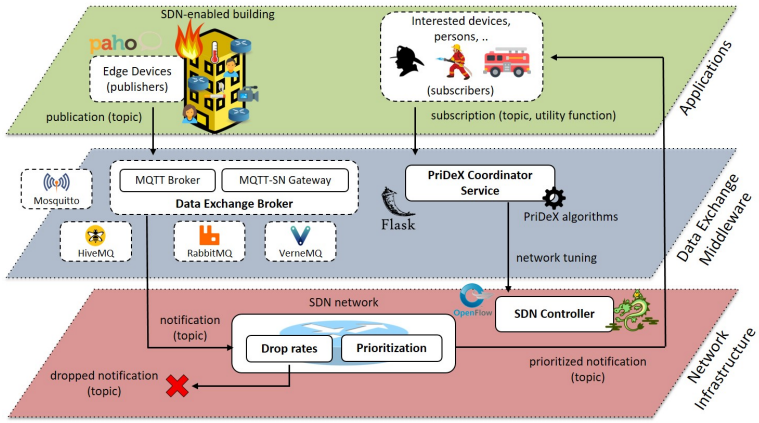


Fig. 4. The PrioDeX cross-layer prototype implementation.

5.1 Cross-layer Prototype Implementation

Application layer. The MQTT Paho library [20] is used from any *PrioDeX publisher* to create an MQTT connection with the data exchange broker. Publishers produce events according to one of two methods: (i) by using probability distributions (e.g., Exponential or Deterministic); or (ii) by using traces created with data coming from real IoT deployments. Subscribers connect to the broker by specifying a topic name and its corresponding utility function to receive relevant events. Each subscriber establishes multiple MQTT-SN connections to the broker with a client library [19] – these allow the network layer to distinguish between different event types (e.g., more/less relevant). Hence, different queueing priority disciplines and event dropping policies can be applied as indicated by the PCS. To establish the subscribers' connections, MQTT-SN is used instead of MQTT because it is implemented over UDP rather than TCP. TCP's re-transmission mechanism interferes with our preemptive packet dropping approach that tolerates losses of less important data due to the constrained bandwidth. However, since UDP does not support fragmentation and reassembly of application-layer events, we assume that events are never fragmented. We additionally limit the event size to 256 bytes (before packet headers) due to the limitations of the MQTT-SN library. The PCS workflow with respect to subscribers is consists of three steps. First, the subscribers coordinate with the PCS as depicted in Fig. 5. Then, the PCS determines the port number of the connection to be used for each subscription. Finally, subscribers open the connections specified by the PCS to the broker and subscribe to each topic through its corresponding connection.

Data exchange layer. In this layer, we follow the publish/subscribe paradigm for event dissemination using the following components:

Data exchange broker. Publishers and subscribers interact with each other via an MQTT-based [43] message broker. While PrioDeX supports any MQTT broker implementation (e.g., EMQ, RabbitMQ), we deployed Moquette [16] because it is lightweight, embeddable, open-source and easy to configure. We also deploy an MQTT-SN gateway [26, 34] co-located with the MQTT broker to translate events from MQTT over TCP (publishers' protocol) to MQTT-SN over UDP (subscribers' protocol).

PrioDeX Coordinator Service. The PCS is the "brain" of PrioDeX. It manages user subscriptions by assigning priorities and drop rates as described in §4. We implemented the PCS as a REST service using the Python library Flask [25]. Subscribers indicate their topics of interest and the corresponding utility functions to the PCS through an HTTP request (i.e., *subscription intent*). Then, the PCS computes priorities and drop rates for the subscriber's network flows, provides to the subscribers the mapping of subscriptions to connections (i.e., network flows), and configures the network layer to enforce the assignment of priorities and dropping policies to network flows.

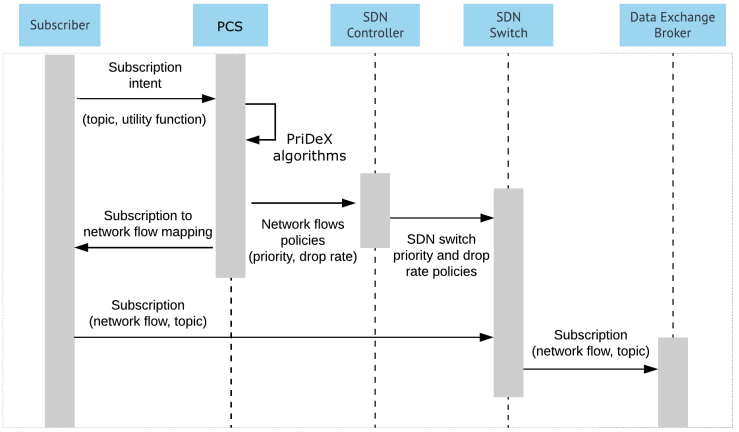


Fig. 5. Subscribers and PCS interaction workflow.

Network layer. The network layer enforces the event prioritization and drop rate policies indicated by the PCS using the OpenFlow [40] protocol. This is performed from the Ryu SDN controller that configures the SDN switches via the following SDN network applications:

(1) The *Topology Application* monitors network traffic to create an internal graph representation (using the NetworkX library [28]) of the network topology. The topology is used to route packets from source to destination.
(2) The *PrioDeX Flow Application* populates the switches' flow and group tables. This application implements different prioritization policies and thus, can be used to tune the networking infrastructure in the case of an emergency (e.g., earthquake, water contamination, etc).

Based on the scenario described in §2, we implement the FireDeX flow application to enforce priority and drop rate policies in the switches' flows and group tables. To identify a subscriber's data flows, the FireDeX flow application matches the packet's header with the network flow information received by the PCS, i.e., the subscriber's IP address and the connection's/network flow's transport layer port number. To set the drop rates, we use the *SELECT* option of the OpenFlow *group tables*. In particular, we set the forwarding and drop probabilities by defining two weighted "buckets" (i.e., options of a group rule); the first bucket represents the actions taken to forward a packet normally and the second bucket represents the actions taken to drop the packet. For example, the rules shown in Listing 1 match packets for the subscriber with IP address 10.0.0.1 and an MQTT-SN connection on UDP port 8888. The forwarding bucket applies priority class 2 (i.e., queue number), while the drop bucket applies a 10% drop rate.

```
FLOW TABLE RULE: ip_address = 10.0.0.1, udp_port = 8888,
    action = (group_identifier, 1)
GROUP TABLE RULE: group_identifier = 1,
    buckets [
        (weight = 90, action = (queue = 2, output_port = 3)),
        (weight = 10, action = drop)
    ]
```

Listing 1. Example rules in flow and group tables.

We configure the priority queues in the switches via Linux TC [5] because OpenFlow does not provide an API to support this. Furthermore, PrioDeX must enforce a random per-packet selection of the buckets option to apply the drop rate rather than the typical approach of hashing packet header fields. This is implemented by leveraging a modified Open vSwitch (OVS) version [45].

5.2 Implementation challenges

We faced the following challenges, and overcame them while implementing the PrioDeX prototype:

- **Differentiating events** at the network layer for policy enforcement. We used SDN network flows to distinguish subscriptions served by different connections. This was necessary because the OVS switches can only inspect a packet's header (i.e., OSI Layers 2-4), and not the payload. Hence, we needed to bring the concept of subscription topics from OSI Layer 7 (app-layer) down to OSI Layers 2-4 for our in-network policy enforcement.
- **Enforcing packet drop rates via SDN** required us to use the modified OVS version described above. As a result, we implemented the drop rate policies through a weighted per-packet selection of bucket options (i.e., drop vs. forward with priority).
- **SDN controller choice:** we moved from ONOS to Ryu because the former does not support group rules for specifying the enqueue action as required by our flow rules shown in Listing 1.
- **The UDP protocol** was used over the TCP protocol when subscribers receive events. Applying drop policies to events over a TCP connection triggers its re-transmission mechanism because the sender does not receive an acknowledgement when the corresponding event is dropped. Hence, subscribers employ the MQTT-SN (over UDP) protocol.
- **Clock-synchronization issues** between publishers and subscribers for gathering accurate performance metrics. To ovecome this challenge we run our experiments on a single machine using Mininet – all applications shared the same system clock.
- **Our experimental network topology** requires additional "dummy" switches for constructing priority queues that are shared across all subscribers. This is due to the fact that each host has its own Ethernet interface connecting it with a switch. Therefore, enqueueing prioritized events may result in one set of priority queues for each subscriber rather than a shared queue across all subscribers as our version accomplishes.

Despite the fact that we have tackled the above challenges, the MQTT-SN control events (e.g., subscription, unsubscription, ACK) are sent to the same UDP port (or network flow) in which we apply the drop rate policies. Therefore, some of the control events may be dropped. To better understand the situation, let us consider the events exchanged between a broker and a subscriber through the SDN infrastructure. The first event sent from the broker to the subscriber (i.e., the subscription's acknowledgement) triggers the creation of the flow/group rules (see Listing 1) associated with the network flow to which the subscription belongs. Subsequently, when the subscriber subscribes to another topic on the same network flow, the second subscription's acknowledgement may be dropped because of the drop rate policy applied. This can delay the subscription process considerably if the assigned drop rate to that network flow is high. One possible solution to overcome the aforementioned problem requires changing the interaction protocol between subscribers and the PCS. In particular, each subscriber notifies the PCS of its intention to subscribe/unsubscribe to/from a topic. Then, the PCS temporarily disables the priority and drop rate policies to allow the subscriber use its network flows and modify its subscriptions. Once the subscriber finishes the subscribing/unsubscribing process, the PCS instructs the SDN controller to re-apply the policies. Note that applying this solution enables us to support policy reconfiguration and manage dynamic conditions (e.g., subscriber churn).

6 EXPERIMENTAL RESULTS

PrioDeX uses the analytical model given in §3.2 to estimate end-to-end response times and success rates for event notifications to interested subscribers. We use this model to evaluate the PrioDeX approach for a given configuration. In particular, we compare our approach's efficacy with that of an unprioritized system and evaluate the trade-off between response times and success rates.

Network Layer		
Parameter	Sim	Prototype
#subscribers ($\|S\|$)	10	10
#publishers ($\|P\|$)	160	10
#flows ($\|F_{s_i}\|$)	9	7
#priorities ($\|Y\|$)	9	7
bandwidth (w_{s_i})	80Mbps	320 Kbps
ρ **tolerance** ($\tilde{\rho}$)	0.1	0.1

Data Exchange Layer			
	Sim	Sim	Prototype
Parameter	Tel. data	Async. events	Async. events
#topics ($\|V\|$)	140	60	7
pub rate ($\lambda^{pub}_{p_i,v_j}$)	6 ∈ [4,7]	4 ∈ [3,5]	1
event size (G_{v_j})	110 ∈ [90,500]	800 ∈ [500,1100]	100
#subscriptions ($\|R_{s_i}\|$)	70	42	70
utility weight (α_{r_j})	2 ∈ [0.01,2]	1 ∈ [0.1,4]	5 ∈ [0.01,100]

Table 2. Default parameters for our experimental configurations.

We use our proposed *greedy-split* priority-assignment algorithm and the *exponential* drop rate policy. Subsequently, we utilize the analytical model to compare the ability of different algorithms to maximize the overall value of information captured. Then, we validate the PrioDeX theoretical model, which includes a multi-class priority queue that represents the prioritization, dropping and transmission of packets in the SDN network infrastructure. We developed an experimental framework that uses both an extended open source queueing simulator as well as the implemented prototype to represent our real-world scenario. We compare the subscribers' end-to-end response times given by the analytical model with those given by the simulation and the prototype. Note that we omit trivial results for validating success rates. In order to improve the figures' legibility, we did not include error bars in our plots as the simulation results' confidence intervals are very small (less than two orders of magnitude from the corresponding mean values presented in the plots). We further validate the model's accuracy under larger numbers of subscribers.

6.1 Experimental Setup

We developed a Python-based experimental framework that models the real-world scenario described in §2 to provide input data for our experiments. The inputs to this framework are the parameters given in Table 2, which will generate configurations for every publisher, subscriber, broker, and the network. We consider two classes of topics that represent events produced from publishers: (i) sensor telemetry readings published periodically from FFs or IoT devices deployed in the building; and (ii) asynchronously-published notifications that indicate real-world phenomena detected from analysis of raw sensor readings. Subscribers correspond to stakeholders such as the IC, FFs and building occupants that subscribe to situational awareness information with varying importance (e.g., "smoke" > "water pressure" for FFs). The parameters in Table 2 represent the average expected value of an exponential distribution. For example, a publication rate or packet size is selected from the given range of values. The actual topics published and subscribed, are chosen uniformly at random from those available. Note that we bound these values to maintain realistic parameters by reproducing a new one if it lies outside the given range. We parameterize a saturated system with high publication rates, overloaded buffers and constrained bandwidth capacity.

PrioDeX ensures low response times and high delivery success rates by using the model presented in §3 which generically captures a wide range of scenarios and system configurations. To reduce the number of variables we explore in our experiments, we only simulate a single (i.e., last-hop) SDN switch between the broker and subscribers. Recall that this represents the bottleneck bandwidth that may cause high transmission delays. Also note that propagation delay and error rates are typically modeled as constant values. Hence, we ignore them for these experiments to focus on analyzing the variable delays that our model aims to capture.

Queueing Network Simulator. After generating these configuration parameters for a single instance of a scenario, our Python-based framework feeds them into a simulator. That is, these parameters correspond to the expected values of the probability distributions from which the simulator draws the actual individual publications' arrival times and packet sizes. Note that we use exponential distributions in order to maintain our assumption of Poisson arrival/service rates.

This simulator extends JINQS [24], a Java simulation library for multiclass queueing networks. JINQS provides a suite of primitives that allow developers to rapidly build simulations for a wide range of queueing networks. We leverage this power and extend JINQS in order to: (i) represent the queueing network introduced in Fig. 3; (ii) implement our new multi-class and non-preemptive priority queueing model; (iii) simulate pub/sub interactions using a set of configuration parameters provided by our Python-based framework. To evaluate PrioDeX, we generate parameters and record the average of 10 runs for each configuration. Each run generates approximately 6,500,000 publications in order to accurately calculate per-subscription response times and success rates. Furthermore, we consider 9 priority classes due to practical limitations of many existing network traffic and data exchange management systems. For example, Linux TC [5] and AMQP 0.9.1 [1] only support 8 and 10 priority queues (one queue per priority class) respectively.

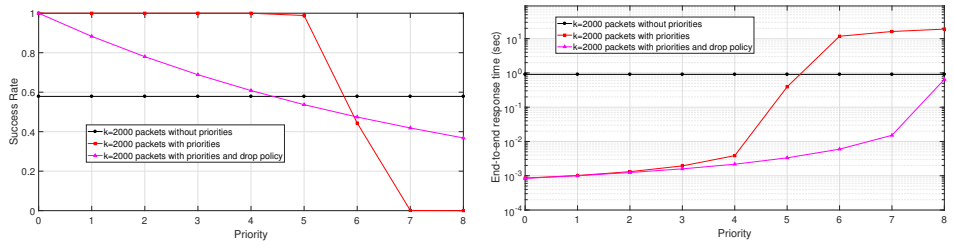
Prototype Emulator. Instead of feeding the generated configuration parameters into a simulator, we use the our prototype that implements a pub/sub system with an emulated network using Mininet [41]. This uses OVS [57] to create a virtual network topology of SDN-enabled switches (in a real Linux networking stack) with realistic delays, bandwidth limits and link loss rates. It connects these switches together as well as to virtual hosts, which are implemented as network namespace-isolated processes. Then, we run our prototype implementation described in §5. OVS switches connect via the SDN southbound protocol OpenFlow [40] to the distributed SDN controller platform Ryu [21] running on the same machine. The publisher/subscriber hosts produce output files from which we calculate the experimental results. The experimental framework configures the managed SDN switches to create a number of priority queues. Because OpenFlow does not support a unified API for creating these queues, we currently perform this using Linux TC [5] that supports up to 8 queues. However, the highest priority queue is used to send the default traffic. Since this would affect the results for that priority queue, we route default traffic through the highest priority queue, and prioritized traffic through the remaining queues. This limits the number of priority queues that we can actually use to prioritize the network traffic to 7.

6.2 Evaluating the PrioDeX Approach

We now compare our approach's efficacy with that of an unprioritized system and a system without preemptive packet drops – this evaluates the trade-off between response times and success rates. We will first discuss the concept of network switch buffers and their limited capacity within the context of PrioDeX in more detail. Recall from §4.3 that we apply drop rates in order to prevent these buffers from filling up, which leads to high queuing delays as well as dropped high priority packets. Recall also from that discussion that we tune the parameter $\tilde{\rho}$ in order to keep these buffers from growing indefinitely. We set $\tilde{\rho} = 0.1$ to prevent our system from being saturated while also ensuring low response times and high delivery success rates. This is used throughout our experiments and adopted as the default in our prototype.

While $\tilde{\rho}$ keeps buffers at a finite size, we must also consider real-world constraints of physical switches: limited buffer capacity. Hence, we now consider applying a buffer capacity of k packets for the simulator's SDN switch outbound queue. This models a real-world switch dropping packets when the buffer fills up. It drops the incoming packet if its priority class is less than or equal to the lowest priority class of those in the buffer. Otherwise, it evicts lower-priority packets to make space in the buffer. We set $k = 2000$ based on reported buffer sizes of various real-world SDN switches[48]. Additionally, we configure this queue in 3 different ways:

(i) No priority assignment or drop policy features (i.e., a simple switch that treats all packets identically and only drops incoming ones when its buffer has filled up)
(ii) Priority assignment only (i.e., no drop rates)
(iii) Both priorities and drop rates (i.e., the complete PrioDeX approach)



(a) Comparing success rates. (b) Comparing response times.

Fig. 6. Success rates vs. response times for no priorities, priorities only, and an added drop policy.

These experiments use the parameters given in Table 2. Figs. 6a and 6b show the success rates and end-to-end response times for each priority class, respectively. Note that priority classes are represented numerically (x-axis) where a lower number has higher priority. Configuration (i) results in a 58% success rate and 0.9 sec response time regardless of assigned priority. Configuration (ii) uses the *greedy-split* algorithm for assigning priorities to each network flow (i.e., to their contained subscriptions and associated packets). The results demonstrate that priority assignment significantly improves both response times and success rates for higher priority subscriptions. In particular, subscriptions with priorities 0-4 have a response time less than 4 ms and 100% success rate. However, the success rate of lower priority subscriptions suddenly decreases while the response time increases to the order of seconds. For instance, those with priority 6 have a 45% success rate and 11 sec. response time. Additionally, subscriptions with priorities 7,8 have very low success rates (almost all packets dropped), while those events successfully delivered have a high response time of 20 sec.

The results for configuration (iii) demonstrate how applying drop rates further improves response time to the order of milliseconds. Specifically, priority 0-6 subscriptions have a response time under 6 ms, whereas those with priority 8 have a response time of 647 ms. The most important subscriptions (i.e., priority 0) have 100% success rate. The PrioDeX *exponential* drop rate policy smoothly decreases the success rate proportional to the priority level. This demonstrates our approach to controlling the success rate based on a subscriber's available bandwidth in order to achieve lower response times. Next, we compare the level of overall utility achieved using the various priority assignment and drop rate algorithms that base their configurations on the subscriptions' utility functions.

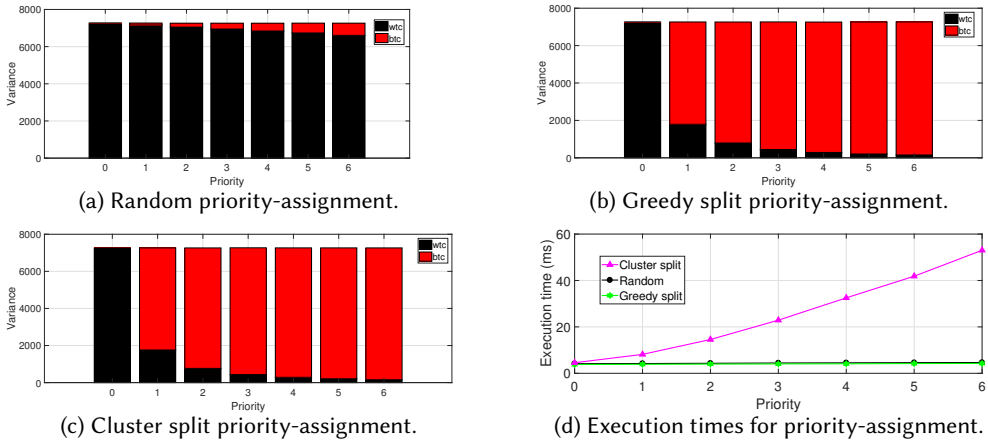
6.3 Comparing Prioritization & Drop Rate Algorithms for Situational Awareness

We now compare our proposed priority assignment algorithms' ability to group similar network flows into priority classes. Each group contains one or more network flows with the same priority class. We define the *within-class (wtc)* and *between-class (btc)* variances denoted by σ^2_{wtc} and σ^2_{btc}, respectively, in order to measure the similarity/dissimilarity of the grouped network flows. We then compare our proposed algorithms' ability to maximize the value of information captured for a given configuration. We measure this as the *achieved utility rate*: the ratio of a subscription's max utility ($\widehat{U_{r_j}}$) to achieved utility, averaged over all subscriptions.

6.3.1 Priority algorithms comparison. We first introduce the metrics used to compare our algorithms. The *within-class* variance per priority represents the spread of each network flow utility (see Eq.21) with respect to the mean utility value of the grouped network flows with the same priority class. This is defined as follows:

$$\sigma^2_{y_k} = \sum_{f_j:\Phi(f_j)=y_k} (\alpha_{f_j} - E[\alpha_{f_j} : \Phi(f_j) = y_k])^2 n_{y_k} \tag{29}$$

where $E[\alpha_{f_j} : \Phi(f_j) = y_k]$ is the average value of network flow utilities with the same y_k and $n_{y_k} = \frac{|f_j \in F:\Phi(f_j)=y_k|}{|F|}$ is the number of network flows with the same y_k divided to the overall number of network flows. We then estimate the total within-class variance for all priority classes as follows:



(a) Random priority-assignment.
(b) Greedy split priority-assignment.
(c) Cluster split priority-assignment.
(d) Execution times for priority-assignment.

Fig. 7. Comparing priority-assignment algorithms policies using wtc/btc variances and their execution time.

$$\sigma^2_{wtc} = \sum_{y_k \in Y} \sigma^2_{y_k} \tag{30}$$

The second metric, *between-class* variance, represents the spread of the mean utility value of the grouped network flows with the same priority class with respect to the mean utility value of all network flows. This is defined as follows:

$$\sigma^2_{btc} = \sum_{y_k \in Y} (E[\alpha_{f_j} : f_j \in F] - E[\alpha_{f_j} : \Phi(f_j) = y_k])^2 n_{y_k} \tag{31}$$

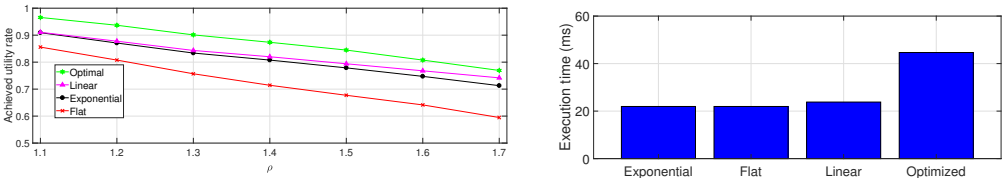
where $E[\alpha_{f_j} : f_j \in F]$ is the average utility value of all network flows.

To summarize, within-class variance measures the similarity/dissimilarity of each network flow in a group (with the same priority class), while between-class variance measures the similarity/dissimilarity of each group of network flows (grouped per priority class) for all priority classes. The main purpose of our approach is to assign similar network flows in a group – hence we aim to minimize the within-class variance. On the other hand, we want dissimilar groups of network flows and thus, we aim to maximize the between-class variance.

Figs. 7a, 7b, 7c show the measured variances according to assigned number of priority classes where priorities were assigned using the random, the greedy split and the cluster split algorithms, respectively. In our experimental setup, we consider 70 network flows assigned with random values. We run each experiment 100000 times and average across the results. As expected, the random algorithm demonstrates the worst behavior (i.e., within-class variance values are very high) than the other algorithms for all priorities. This is because it does not consider the utility values (α_{f_j}) of network flows. By leveraging the networking characteristics in the network utility definition (see Eq.21) when assigning priorities using the greedy and cluster approaches, the within-class variance decreases very rapidly.

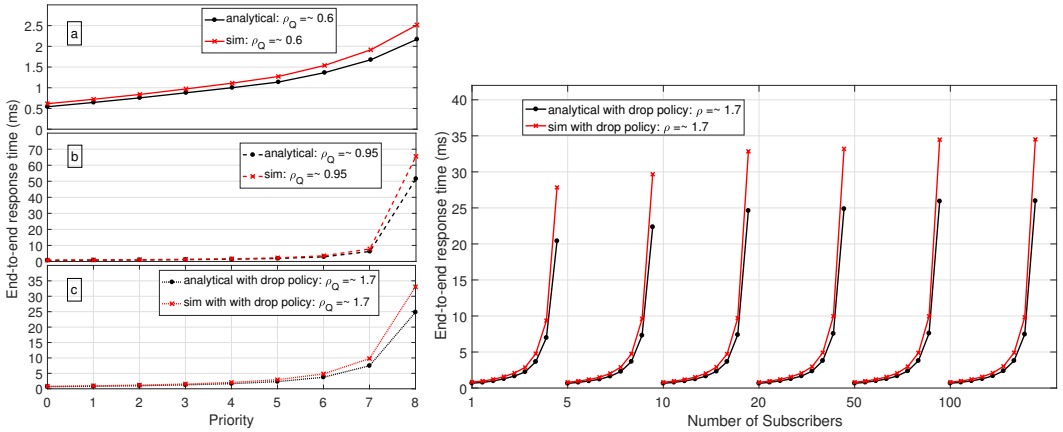
Although these two approaches perform very similar with regard to their ability to grouping similar utility values of network flows, their execution time is different. As shown in Fig. 7d cluster one becomes significantly less efficient as the number of priority classes increases.

6.3.2 Drop rate algorithms comparison. We compare the four drop rate-assignment algorithms outlined in §4.3 (Flat, Linear, Exponential, Optimized). Note these algorithms assume that priority classes have been already assigned to network flows – we leverage the greedy-split algorithm which is the most efficient. To demonstrate PrioDeX's ability to improve situational awareness for heterogeneous data and information requirements, our experiments varied the load of the network



(a) Drop rate policies. (b) Execution times for drop rate policies.

Fig. 8. Comparing drop rate policies and their execution time.



(a) For varying traffic loads ($\rho_{Q_{x_k}^{out}}$). (b) For varying numbers of subscribers.

Fig. 9. Analytical vs. simulation end-to-end response times.

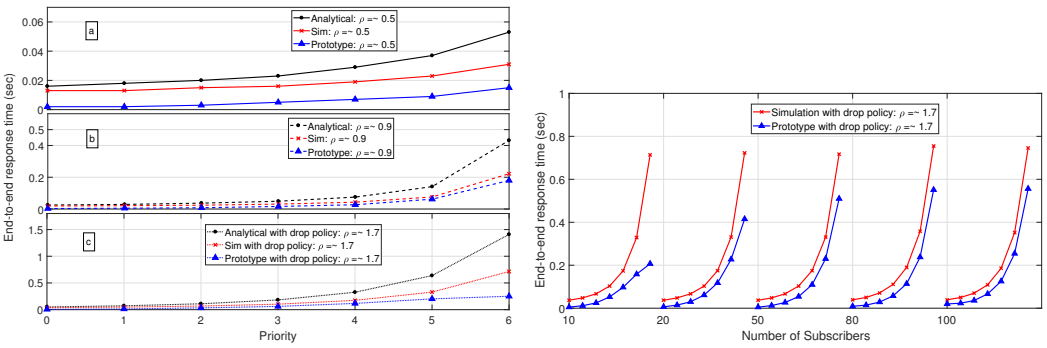
(ρ) by increasing the number of subscriptions. The x-axis in Fig. 8a shows the different ρ values used throughout the experiments while the y-axis represents the achieved utility rate. Fig. 8b shows the execution time of each algorithm with ρ = 1.7 (i.e., overloaded network conditions). As shown in Figs. 8a and 8b, while the optimization-based algorithm captures the highest overall utility rate (i.e., it maximizes situational awareness) it is the least efficient in terms of execution time. The linear and exponential algorithms demonstrate similar utility rates and execution times.

6.4 Validating the PrioDeX Models

To prove the accuracy of the theoretical analysis (presented in §3.2), we now compare the estimated performance metrics with those from the simulator and the prototype implementation.

6.4.1 Simulation-based Validation. Recall that the SDN switch's outbound queue (see Fig. 3) captures the bottleneck bandwidth of the network route from broker to subscriber. PrioDeX uses the corresponding server utilization ($\rho_{Q_{x_k}^{out}}$) to decide the bandwidth tuning by assigning drop rates. Therefore, we parameterize the simulated queueing network to vary the traffic load: *a)* medium-load conditions ($\rho_{Q_{x_k}^{out}}$ = 0.6); *b)* high-load conditions (i.e., close to saturation – $\rho_{Q_{x_k}^{out}}$ = 0.95); *c)* overloaded conditions (i.e., saturated – $\rho_{Q_{x_k}^{out}}$ = 1.7). Note that the saturated case (3rd) corresponds to the default parameters in Table 2. To achieve the medium-load (1st) and high-load (2nd) cases, we set the number of subscriptions for each topic class respectively: (i) 21,15; and (ii) 42,24.

Fig. 9a shows the results of these experiments according to assigned number of priority classes and averaged across all topics, subscribers, etc. Comparing the curves of both the simulated measurements and the analytical results obtained by Eq. (10) reveal our model's high accuracy. We notice small differences for events with lower priority levels. In particular, note priority level 8's



(a) for varying traffic loads ($\rho_{Q_{x_k}^{out}}$). (b) for varying numbers of subscribers.

Fig. 10. Analytical vs. simulation vs. prototype end-to-end response times.

differences: 0.35 ms in Fig. 9aa, 13.98 ms in Fig. 9ab and 8.24 ms in Fig. 9ac. Because the system approaches saturation in Figs. 9ab and 9ac, we deem these results acceptable. In Fig. 9ac, PrioDeX uses our drop policy mechanism to drop packets at the SDN switch and return the system to below saturation (i.e., $\rho_{Q_{x_k}^{out}} = 0.9$ by using $\tilde{\rho} = 0.1$).

We now validate our analytical model's accuracy under varying numbers of subscribers: $|S| = 1, 10, 20, 50, 100$. Among the simulation parameters defined in Table 2, we select to scale up the number of subscribers because this significantly increases the system traffic load – the broker must duplicate events to match the increased number of subscriptions. Additionally, the prioritization and dropping mechanisms are enforced at the broker-subscriber link and thus is highly important to extensively evaluate this part of our system. To maintain the same degree of system saturation (i.e., $\rho_{Q_{x_k}^{out}} = 1.7$), we increase the simulation bandwidth proportional to the number of subscribers: $w_{x_k,s_i} = 8Mbps$. We keep all other simulation parameters according to Table 2. According to these parameters, we measure the simulated mean response times and plot them vs. those calculated using Eq. (10) in Fig. 9b. Note the curve for each number of subscribers that shows response time increasing with the priority class. From this comparison, we see that the absolute deviation between the two curves does not exceed 10 ms across all priority levels. Therefore, our model remains accurate even with higher numbers of subscribers.

6.4.2 Prototype-based Validation. We further validate our analytical model by comparing its estimated response times with the ones derived from a prototype implementation configured under realistic settings. Similar to the simulation-based validation, we first show the results obtained for different traffic loads and then scale up the number of subscribers. We modify the experiments' configuration parameters used in the simulation (see Table 2, Prototype column) to overcome practical issues imposed by Mininet and challenges described in §5.2. In particular, we use one subscription per network flow and we reduce the number of priority classes to 7 because of Linux TC limitations. Note that we use a fixed publication rate and event size – if we vary the publication rate and event size, it takes longer to converge to the expected values of the probability distributions and thus the execution time of each real experiment. Because the analytical model assumes exponential distributions, we also calculate response times using the queueing simulator with realistic parameters introduced from the prototype implementation and its emulated network through Mininet. In particular, we only use asynchronously-published events and we set the SDN output queue's service rate as deterministic rather than exponential. This represents the concept of a switch's bandwidth, that essentially has a constant service rate measured in bytes/second.

Again, we parameterize and deploy PrioDeX under different network load conditions: *a)* medium-load ($\rho_{Q_{x_k}^{out}} = 0.5$); *b)* high-load ($\rho_{Q_{x_k}^{out}} = 0.9$); and *c)* overloaded ($\rho_{Q_{x_k}^{out}} = 1.7$). Fig. 10a shows the

	App-layer requirements	Methodology used	Technologies & Tools	Prototype	QoS metric improvement
Zhang et al. [65]	Prioritized data classes	Bandwidth allocation to prioritized data flows	Apache HTTP & Squid cache servers	✓	Response time
Saghian et al. [50]	Data flows importance	Priority queues	OMNet++ simulator	✗	Response time
Yu et al. [60]	Delay and bandwidth sensitive apps	DPI and Laplacian SVM	SDN, OpenFlow	✗	Network resource utilization
Li et al. [38]	App data types	C4.5 decision tree, priority queues	SDN, OpenFlow	✓	Response time, throughput
An et al. [6]	Delay-sensitive and delay-tolerant traffic	Priority-adjustment algorithms	NS-2 simulator	✗	Real-time requirement satisfaction
Shi et al. [54]	Delay requirements, data semantics	Priority queues based on semantics	SDN, OpenFlow, LDAP	✓	Response time
Bröring et al. [17]	Video QoS constraints	Bandwidth allocation	DNode-RED, SDN, OpenFlow, MQTT	✓	Response time, delivery success rate
Nguyen et al. [42]	Utility functions	Bandwidth allocation to prioritized data flows	Java simulator	✗	Response time, network resource utilization
PrioDeX	Utility functions	Priority queues, analytical models, heuristic & optimization-based algorithms	SDN, OpenFlow, MQTT, MQTT-SN	✓	Response time, network resource utilization, maximizing user's utility

Table 3. Comparison of PrioDeX with related work.

end-to-end response times obtained using 10 subscribers and the aforementioned load conditions – these closely match the response times calculated from both the simulation and the analytical model. The differences observed are due to the following reasons: (i) the analytical model assumes exponential service rates while the simulation applies deterministic service rates to match the constant service rate (bytes/second) in the switch of the prototype; (ii) Mininet lacks the ability to emulate proper queueing delay when transmitting packets and thus there is a significant difference between the response times of the simulation and the prototype. This is because Linux TC (used by Mininet to apply queueing disciplines to any network interface) can emulate bandwidth limitations, packet loss, and network delay. The bandwidth limitation constrains the volume of traffic (i.e., number of bytes) that can be sent per unit of time. However, it does not simulate the actual packet transmission delay ($\frac{G_{v_j}}{w_{s_i}}$) due to the available bandwidth. Hence, Linux TC sends packets at the same speed regardless of packet size (i.e., the transmission delay is constant), which in turn affects the perceived queueing delay. That is, if the available bandwidth is enough to empty the queues, the queued packets do not experience the queueing delay due to the transmission of previous packets.

As described in the simulation-based validation (§6.4.1) the lowest priority events experience the highest response time difference under saturated conditions. Here, we notice constant response time differences across all priority classes in the unsaturated setting (Fig. 10aa). For the saturated and overloaded conditions (Fig. 10ab & 10ac), we observe larger differences for lower priorities due to the Linux TC packet transmission limitations. Because these packets wait in the queues for a longer period, this difference is compounded further by the lack of transmission delay for each queued packet in front of it. Hence, we expect to see a gap between the analytical model and the simulated/emulated results. We then scale up the number of subscribers as shown in Fig. 10b. We parameterize and deploy the system with the following number of subscribers $|S|$ = 10, 20, 50, 80, 100. To maintain the same degree of system saturation (i.e., $\rho_{Q_{x_k}^{out}} = 1.7$), we increase the bandwidth proportional to the number of subscribers. As shown in Fig. 10b, the implementation produces again accurate results that closely match the simulated and analytical ones.

7 RELATED WORK

In this section, we compare PrioDeX against other related data exchange systems for enabling reliable and timely data exchange. We summarize the principal solutions in Table 3 with regard to:

the considered app-layer requirements (if any), the methodology used to improve the system's performance, the technologies and tools leveraged to implement the prototype or perform experiments, and the improvement in terms of QoS metrics. To enable reliable and timely data exchange, existing solutions manipulate data at both the middleware and network layers. Early middleware-based solutions [18, 39, 65] support prioritization or bandwidth allocation based on the available system capacity, data relevance and data importance. More recent middleware solutions assign priorities based on validity span of published data and subscriptions [50] or based on delay and reliability requirements [58]. Currently, standardized message brokers such as RabbitMQ, ActiveMQ, etc, support the assignment of priorities at the publisher side prior to the emission of a message.

With the advent of novel networking technologies (e.g., OpenFlow [40], P4 [13]), advanced capabilities are provided to system designers to customize the underlying network infrastructure. SDN-based approaches [33] have been used for improving network resiliency [3] and handover latency in 5G network environments [4]. Other research into SDN-enabled 5G cellular architectures [56] supports the potential for such interfaces that connect emergency responder devices to the building's internal network. SDN provides a variety of abstractions to represent the underlying physical network. Yu et al [60] leverage SDN to apply *deep packet inspection* (DPI) and *Laplacian SVM* techniques for identifying applications with delay or bandwidth requirements, which are then used to improve the overall network resource utilization. Also, SDN has been leveraged for priority assignment and bandwidth allocation to network flows to satisfy app-specific requirements. These include authorized access to directly manage physical switches, control over virtual (software-based) switches [46] (e.g., running alongside the broker), network virtualization [12] to reserve "slices" of the physical infrastructure, etc. Li et al [38] introduce a middleware solution that assigns priority levels to network flows based on three different classes of data: *expedited forwarding*, *assured forwarding* and *best effort*. Similarly, An et al [6] assign priorities based on the type of data traffic (i.e., delay-sensitive or delay-tolerant traffic), and Shi et al [54] assign priorities based on the semantics of data. Finally, Bröring et al [17] allocate bandwidth to SDN network flows based on video-specific application-level QoS constraints (e.g. min/max frame rate). While the above approaches manage network flows in SDN switches based on application data flows/types, IoT devices in buildings/structures (e.g., sensors, cameras) produce data that varies in size, frequency (periodic samples vs. asynchronous alerts), type, and importance to individual subscribers [8, 51, 63, 64]. Research on *Network Utility Maximization* (NUM) [59] aims to tune the underlying network according to application-level requirements. NUM configures a network (e.g., assigns bandwidth) to serve nodes in a manner that maximizes utility functions to capture a user's degree of satisfaction with the network's performance. However, few prior researchers have investigated discrete priority classes, which we leverage in our approach, within the context of NUM. The authors of [42] propose assigning more bandwidth to users (i.e., via weighting their requests higher) based on their requested priority levels.

Our cross-layer approach and consideration of utility functions sets apart our work from most related SDN research referenced above. Utility functions enable a more flexible configuration of application-level requirements (e.g., information needs) including mission-critical ones. In addition, PrioDeX leverages SDN to manage networking at the Edge for IoT deployments by offloading network configuration tasks from constrained devices and network hardware.

8 CONCLUSION

In this paper we presented PrioDeX: an extensible middleware for timely and reliable IoT data exchange. Our proposed SDN-enabled three-layer approach bridges application-specified information requirements, generic data exchange capabilities, and physical network characteristics for efficient delivery of mission-critical data from IoT sources to relevant consumers. We design a cross-layer

queueing analytical model for estimating system performance metrics. These metrics are used as input to the PrioDeX algorithms for assigning priorities to subscriptions and tune their bandwidth allocation (via packet drop rates) to maximize overall situational awareness. Our experimental results show that our approach greatly improves the performance in terms of information value captured as well as end-to-end delays. PrioDeX can inspire system designers to build the next generation of *Smart Fire Fighting* systems with support for proper filtering, prioritizing and analysis. In addition, application developers can leverage PrioDeX to define the situational awareness information of any emergency response (e.g., active shooter) and QoS-dependent scenarios (e.g., traffic estimation). Research scientists can leverage and further extend PrioDeX's theoretic-grounded models for the QoS evaluation of IoT-enabled smart spaces at runtime so as to support their adaptation in relation with the evolving operating environment.

The modular design of our theoretical model supports the composition of alternative queueing models. Hence it lays the groundwork for many potential extensions and alterations, some of which we will address in future work. In particular, we aim to: consider non-Poisson arrival and service rates by using e.g., G/G/1 queues; convert larger events into many packets (or many events into one packet) by applying the queueing theoretic concept of *batch arrivals* [53]; configure an entire broker network rather than just the local broker at the Edge. In addition, we plan to extend our prototype to include: managing dynamic conditions such as failing publisher devices, and varying network bandwidth/error rates; accurately and efficiently estimating publication rates; considering SDN overhead (e.g., flow table space required, delay for configuration changes and statistics collection); supporting alternative formulations of tunable bandwidth allocation (e.g., traffic policing). Finally, we will build on the PrioDeX prototype to explore further IoT middleware challenges in emergency response settings.

ACKNOWLEDGMENTS

This work was supported by: NSF award CNS 1450768, DARPA agreement # FA8750-16-2-0021, the Inria@SiliconValley International Lab and the research associate team MINES.

REFERENCES

[1] AMQP Working Group 0-9-1. 2008. http://www.amqp.org/specification/0-9-1/amqp-org-download.
[2] A. Agrawal, R. Verschueren, S. Diamond, and S. Boyd. 2018. A Rewriting System for Convex Optimization Problems. *Journal of Control and Decision* 5, 1 (2018), 42–60.
[3] J. Ai, H. Chen, Z. Guo, G. Cheng, and T. Baker. 2019. Improving Resiliency of Software-Defined Networks with Network Coding-based Multipath Routing. In *IEEE Symposium on Computers and Communications (ISCC)*. 1–6.
[4] A. S. D. Alfoudi, S. Newaz, R. Ramlie, G. M. Lee, and T. Baker. 2019. Seamless Mobility Management in Heterogeneous 5G Networks: A Coordination Approach among Distributed SDN Controllers. In *IEEE 89th Vehicular Technology Conf. (VTC2019-Spring)*. 1–6.
[5] W. Almesberger. 1999. Linux network traffic control-implementation overview.
[6] N. An, T. Ha, K.J. Park, and H. Lim. 2016. Dynamic priority-adjustment for real-time flows in software-defined networks. In *17th Intl. Telecommunications Network Strategy and Planning Symposium (Networks)*. IEEE, 144–149.
[7] C. C. Beard and V. S. Frost. 2004. Prioritization of emergency network traffic using ticket servers: A performance analysis. *Simulation* 80, 6 (2004), 289–299.
[8] S. Behnel, L. Fiege, and G. Muhl. 2006. On quality-of-service and publish-subscribe. In *ICDCS Workshops*. IEEE.
[9] K. Benson, C. Fracchia, G. Wang, Q. Zhu, S. Almomen, J. Cohn, L. D'arcy, D. Hoffman, M. Makai, J. Stamatakis, and N. Venkatasubramanian. 2015. SCALE: Safe community awareness and alerting leveraging the internet of things. *Communications Magazine, IEEE* 53, 12 (2015), 27–34.
[10] K. E. Benson, G. Bouloukakis, C. Grant, V. Issarny, S. Mehrotra, I. Moscholios, and N. Venkatasubramanian. 2018. FireDeX: a Prioritized IoT Data Exchange Middleware for Emergency Response. *ACM/IFIP/USENIX Intl. Middleware Conf.* (2018), 279–292.
[11] K. E. Benson, G. Wang, N. Venkatasubramanian, and Y. Kim. 2018. Ride: A Resilient IoT Data Exchange Middleware Leveraging SDN and Edge Cloud Resources. In *2018 IEEE/ACM Third Intl. Conf. on Internet-of-Things Design and*

Implementation (IoTDI). 72–83.
[12] A. Blenk, A. Basta, M. Reisslein, and W. Kellerer. 2015. Survey on Network Virtualization Hypervisors for Software Defined Networking. *CoRR* abs/1506.07275 (2015). arXiv:1506.07275 http://arxiv.org/abs/1506.07275
[13] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. 2014. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.* 44, 3 (July 2014), 87–95.
[14] G. Bouloukakis, N. Georgantas, A. Kattepur, and V. Issarny. 2017. Timeliness Evaluation of Intermittent Mobile Connectivity over Pub/Sub Systems. In *Proceedings of the 8th ACM/SPEC on Intl. Conf. on Performance Engineering*. 275–286.
[15] G. Bouloukakis, I. Moscholios, N. Georgantas, and V. Issarny. 2017. Performance Modeling of the Middleware Overlay Infrastructure of Mobile Things. In *IEEE Intl. Conf. on Communications*.
[16] Moquette broker. 2014. https://github.com/andsel/moquette/.
[17] A. Bröring, J. Seeger, M. Papoutsakis, K. Fysarakis, and A. Caracalli. 2020. Networking-Aware IoT Application Development. *Sensors* 20, 3 (2020), 897.
[18] S. Chakravarthy and N. Vontella. 2004. A publish/subscribe based architecture of an alert server to support prioritized and persistent alerts. In *Intl. Conf. on Distributed Computing and Internet Technology*. Springer, 106–116.
[19] MQTT-SN UDP client. 2016. https://github.com/jsaak/mqtt-sn-gateway.
[20] Paho Java Client. 2008. https://www.eclipse.org/paho/clients/java/.
[21] Ryu SDN controller. 2011. https://osrg.github.io/ryu/.
[22] S. Diamond and S. Boyd. 2016. CVXPY: A Python-Embedded Modeling Language for Convex Optimization. *Journal of Machine Learning Research* 17, 83 (2016), 1–5.
[23] G. Faraci, A. Lombardo, and G. Schembra. 2017. A building block to model an SDN/NFV network. In *2017 IEEE Intl. Conf. on Communications (ICC)*. 1–7.
[24] T. Field. 2006. JINQS: An extensible library for simulating multiclass queueing networks, v1. 0 user guide.
[25] Flask Web Framework. 2010. http://flask.pocoo.org/.
[26] MQTT-SN Transparent Gateway. 2016. https://www.eclipse.org/paho/components/mqtt-sn-transparent-gateway/.
[27] D. Gross, J. Shortle, J. Thompson, and C. Harris. 2008. *Fundamentals of queueing theory*. John Wiley & Sons, 4th edition.
[28] A. A. Hagberg, D. A. Schult, and P. J. Swart. 2008. Exploring Network Structure, Dynamics, and Function using NetworkX. In *SciPy*, G. Varoquaux, T. Vaught, and J. Millman (Eds.). 11–15.
[29] H. Halabian, I. Lambadaris, and C.-H. Lung. 2010. Network capacity region of multi-queue multi-server queueing system with time varying connectivities. In *2010 IEEE Intl. Symposium on Information Theory*. IEEE, 1803–1807.
[30] A. Hamins, C. Grant, N. Bryner, A. Jones, and G. Koepke. 2015. *NIST Special Publication 1191 Research Roadmap for Smart Fire Fighting*. National Institute Of Standards and Technology.
[31] F. He, L. Baresi, C. Ghezzi, and P. Spoletini. 2007. Formal analysis of publish-subscribe systems by probabilistic timed automata. In *Intl. Conf. on Formal Techniques for Networked and Distributed Systems*. 247–262.
[32] R. A. Horn and C. R. Johnson. 2012. Matrix Analysis. *Cambridge, UK: Cambridge University Press* (2012).
[33] T. Hu, Z. Guo, P. Yi, T. Baker, and J. Lan. 2018. Multi-controller based software-defined networking: A survey. *IEEE Access* 6 (2018), 15980–15996.
[34] IBM 2013. *MQTT For Sensor Networks (MQTT-SN)*. IBM.
[35] M. Inoue, Y. Owada, K. Hamaguti, and R. Miura. 2014. Nerve Net: A Regional-Area Network for Resilient Local Information Sharing and Communications. In *Proceedings of the 2014 2nd Intl. Symposium on Computing and Networking (CANDAR '14)*. IEEE Computer Society, Washington, DC, USA, 3–6.
[36] S. Kounev, K. Sachs, J. Bacon, and A. Buchmann. 2008. A methodology for performance modeling of distributed event-based systems. In *11th IEEE Intl. Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*. 13–22.
[37] E. Lazowska, J. Zahorjan, S. Graham, and K. Sevcik. 1984. *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc.
[38] F. Li, J. Cao, X. Wang, and Y. Sun. 2017. A QoS guaranteed technique for cloud applications based on software defined networking. *IEEE access* 5 (2017), 21229–21241.
[39] P. Maheshwari, H. Tang, and R. Liang. 2004. Enhancing web services with message-oriented middleware. In *Proceedings. Intl. Conf. on Web Services,*. IEEE, 524–531.
[40] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. 2008. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.* 38, 2 (March 2008), 69–74.
[41] Mininet. 2016. Mininet: An Instant Virtual Network on your Laptop (or other PC). http://mininet.org/
[42] H. A. Nguyen, T. V. Nguyen, and D. Choi. 2009. How to Maximize User Satisfaction Degree in Multi-service IP Networks. In *2009 1st Asian Conf. on Intelligent Information and Database Systems*. 471–476.
[43] OASIS 2014. *MQTT Version 3.1.1*. OASIS.

[44] T. E. Oliphant. 2006. *A guide to NumPy*. Vol. 1. Trelgol Publishing USA.
[45] Stochastic OVS. 2014. https://github.com/saeenali/openvswitch/wiki/.
[46] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, et al. 2015. The Design and Implementation of Open vSwitch. In *12th Symposium on Networked Systems Design and Implementation (NSDI)* (Oakland, CA).
[47] RabbitMQ. 2018. https://www.rabbitmq.com/.
[48] Buffer requirements. 2008. https://people.ucsc.edu/~warner/buffer.html.
[49] K. Sachs, S. Kounev, and A. Buchmann. 2013. Performance modeling and analysis of message-oriented event-driven systems. *Software & Systems Modeling* 12, 4 (2013), 705–729.
[50] M. Saghian and R. Ravanmehr. 2015. Publish/subscribe middleware for resource discovery in MANET. In *2015 15th IEEE/ACM Intl. Symposium on Cluster, Cloud and Grid Computing*. IEEE, 1205–1208.
[51] P. Salehi, K. Zhang, and H. Jacobsen. 2017. PopSub: Improving resource utilization in distributed content-based publish/subscribe systems. In *Distributed Event-Based Systems (DEBS)*. ACM, 88–99.
[52] A. Schröter, G. Mühl, S. Kounev, H. Parzyjegla, and J. Richling. 2010. Stochastic performance analysis and capacity planning of publish/subscribe systems. In *Distributed Event-Based Systems (DEBS)*. ACM, 258–269.
[53] D. N. Shanbhag. 1966. On infinite server queues with batch arrivals. *Journal of Applied Probability* 3, 1 (1966), 274–279.
[54] Y. Shi, Y. Zhang, H.-A. Jacobsen, L. Tang, G. Elliott, G. Zhang, X. Chen, and J. Chen. 2019. Using Machine Learning to Provide Reliable Differentiated Services for IoT in SDN-Like Publish/Subscribe Middleware. *Sensors* 19, 6 (2019), 1449.
[55] D. Singh, B. Ng, Y. Lai, Y. Lin, and W. K.G. Seah. 2017. Modelling Software-Defined Networking: Switch Design with Finite Buffer and Priority Queueing. In *2017 IEEE 42nd Conf. on Local Computer Networks (LCN)*. IEEE, 567–570.
[56] S. Khan Tayyaba and M. A. Shah. 2019. Resource allocation in SDN based 5G cellular networks. *Peer-to-Peer Networking and Applications* 12, 2 (2019), 514–538.
[57] Open vSwitch. 2016. http://openvswitch.org/.
[58] Y. Wang, Y. Zhang, and J. Chen. 2017. Pursuing Differentiated Services in a SDN-Based IoT-Oriented Pub/Sub System. In *24th International Conference on Web Services*. IEEE, 906–909.
[59] Y. Yi and M. Chiang. 2008. Stochastic network utility maximisation – a tribute to Kelly's paper published in this journal a decade ago. *European Transactions on Telecommunications* 19, 4 (2008), 421–442.
[60] C. Yu, J. Lan, Z. Guo, Y. Hu, and T. Baker. 2019. An adaptive and lightweight update mechanism for SDN. *IEEE Access* 7 (2019), 12914–12927.
[61] Mosterman P.J.-Padir-T. Wan Y. Zander, J. and S. Fu. 2015. Cyber-physical systems can make emergency response smart. *Procedia Engineering* 107 (2015), 312–318.
[62] K. Zhang and H. Jacobsen. 2013. SDN-like: The Next Generation of Pub/Sub. *CoRR* abs/1308.0056 (2013). http://arxiv.org/abs/1308.0056
[63] K. Zhang, V. Muthusamy, M. Sadoghi, and H. Jacobsen. 2017. Subscription covering for relevance-based filtering in content-based publish/subscribe systems. In *IEEE 37th ICDCS*. IEEE, 2039–2044.
[64] K. Zhang, M. Sadoghi, V. Muthusamy, and H. Jacobsen. 2017. Efficient covering for top-k filtering in content-based publish/subscribe systems. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conf.* 174–184.
[65] R. Zhang, C. Lu, T. F. Abdelzaher, and J. A. Stankovic. 2002. Controlware: A middleware architecture for feedback control of software performance. In *Proceedings 22nd Intl. Conf. on Distributed Computing Systems*. IEEE, 301–310.

A MULTI-CLASS PRIORITY QUEUE ANALYTICAL MODEL

We now prove the analytical model that estimates the average response time of events matching subscription r_k in the system (queue+server) of Q_{mclpr}. This is a *non-preemptive multi-class priority* queueing system where each subscription ($r_j \in R$) corresponds to a class and one or more subscriptions can be mapped to priority level $y_j \in Y$.

Based on (13), to estimate $\Delta_{Q_{mclpr}}$ for a given r_k, we accept as input the set of arrival (λ^{sub}) and processing (μ^{sub}) rates:

$$\lambda^{sub} = \{\lambda_{r_j} : r_j \in R\} \tag{32}$$

$$\mu^{sub} = \{\mu_{r_j} : r_j \in R\} \tag{33}$$

As previously discussed, a given subscription (r_k) is mapped to a priority (y_c) as given by:

$$y_c = \Phi \circ \Psi(r_k) \tag{34}$$

Let λ^{prio} be the set of arrival rates and μ^{prio} the set of processing rates per y_j:

$$\lambda^{prio} = \{\lambda_{y_j} : y_j \in Y\} \tag{35}$$

$$\mu^{prio} = \{\mu_{y_j} : y_j \in Y\} \tag{36}$$

Because one or more r_j can be mapped to a y_c, by (34) we can estimate the arrival rate λ_{y_c} of events with assigned priority y_c as follows:

$$\lambda_{y_c} = \sum_{\{r_j \in R : y_c = \Phi \circ \Psi(r_j)\}} \lambda_{r_j} \tag{37}$$

Similarly the processing rate μ_{y_c} is estimated as follows:

$$\mu_{y_c} = \left[\sum_{\{r_j \in R : y_c = \Phi \circ \Psi(r_j)\}} \frac{\lambda_{r_j}}{\lambda_{y_c}} \frac{1}{\mu_{r_j}} \right]^{-1} \tag{38}$$

Similarly, we can estimate arrival and processing rates for any priority y_j. We now rely on (37),(38), and the analysis in Section 3.4.2 of [27] to estimate the waiting time (delay only in the queue) $\Delta_q^{y_c}$ for a given y_c as follows:

$$\Delta_q^{y_c} = \frac{\sum_{y_j \in Y} \frac{\rho_{y_j}}{\mu_{y_j}}}{(1 - \sigma_{y_c - 1})(1 - \sigma_{y_c})} \tag{39}$$

where $\rho_{y_j} = \lambda_{y_j} \,/\, \mu_{y_j}$ and $\sigma_{y_c} = \sum_{i=0}^{c} \rho_{y_i}$ (i.e., the sum of ρ_{y_i} for all priority classes y_i whose priority is higher than or equal to y_c). Let $L_q^{y_c}$ be the average number of priority-y_c events in the queue. From (39), Little's law then gives:

$$L_q^{y_c} = \Delta_q^{y_c} \lambda_{y_c} \tag{40}$$

Finally, let Δ^{y_c} be the average response time of priority-y_c events in the system (queue+server). This is estimated as follows:

$$\Delta^{y_c} = \Delta_q^{y_c} + \frac{1}{\mu_{y_c}} \tag{41}$$

Let $L_q^{r_k}$ be the average number of events in the queue matching subscription r_k with priority y_c. Using (37) and (40) this can be estimated by:

$$L_q^{r_k} = \frac{\lambda_{r_k}}{\lambda_{y_c}} L_q^{y_c} \tag{42}$$

and the average number of priority-y_c events in the system matching subscription r_k is given by:

$$L^{r_k} = L_q^{r_k} \frac{\lambda_{r_k}}{\mu_{r_k}} \tag{43}$$

Finally, by relying on Little's law and (43), the average response time of events matching a given subscription r_k in the multi-class priority queueing system (Q_{mclpr}) is given by:

$$\Delta_{Q_{mclpr}} = \frac{L^{r_k}}{\lambda_{r_k}} \tag{44}$$

B EFFICIENTLY COMPUTING DROP RATE POLICIES

We now detail efficiently computing drop rate policies for the PrioDeX middleware by solving (23) for the flat, linear and exponential drop rate policies. Considering (23) and (24) we aim to find:

$$\rho_{Q_{x_k}^{out}} = \sum_{r_j \in R_{x_k}} \frac{\lambda_{x_k, r_j}^{thru}}{\mu_{x_k, r_j}^{out}} = 1 - \tilde{\rho} \tag{45}$$

We can expand the denominator to rewrite the previous equation considering (6) and (7):

$$\rho_{Q_{x_k}^{out}} = \sum_{r_j \in R_{x_k}} \frac{\lambda_{b_k,r_j}^{notify} G_{v_j}\left(1 - \Omega \circ \Psi(r_j)\right)}{w_{x_k,s_i}} = 1 - \tilde{\rho} \tag{46}$$

where $\Omega \circ \Psi(r_j)$ represents the drop rate for the subscription r_j. (46) is the starting point for each of the following derivations.

We omit the proof for the Flat and the Linear drop rate policies, since it is very similar to the Exponential. We just present the final results.

$$\textbf{Flat: } \beta = 1 - \frac{\sum_{s_i \in S_{x_k}} w_{x_k,s_i}(1-\tilde{\rho})}{\sum_{r_j \in R_{x_k}} \lambda_{b_k,r_j}^{notify} G_{v_j}} \tag{47}$$

$$\textbf{Linear: } \beta = \frac{\sum_{r_j \in R_{x_k}} \lambda_{b_k,r_j}^{notify} G_{v_j} - \sum_{s_i \in S_{x_k}} w_{x_k,s_i}(1-\tilde{\rho})}{\sum_{r_j \in R_{x_k}} \lambda_{b_k,r_j}^{notify} G_{v_j} \Phi(f_j)} \tag{48}$$

Exponential drop rate policy. This policy sets each network flow's drop rate according to its assigned priority level. The drop rate for subscription r_j is equal to the drop rate assigned to its network flow f_j. Hence, considering (27) we have:

$$\Omega \circ \Psi(r_j) = \Omega(f_j) = 1 - \beta^{-\Phi(f_j)} \tag{49}$$

Substituting (49) into (46) we obtain:

$$\sum_{r_j \in R_{x_k}} \frac{\lambda_{b_k,r_j}^{notify} G_{v_j}(1 - (1 - \beta^{-\Phi(f_j)}))}{w_{x_k,s_i}} = 1 - \tilde{\rho} \tag{50}$$

We isolate the constant term β:

$$\sum_{r_j \in R_{x_k}} \lambda_{b_k,r_j}^{notify} G_{v_j}(1 - (1 - \beta^{-\Phi(f_j)})) = \sum_{s_i \in S_{x_k}} w_{x_k,s_i}(1-\tilde{\rho}) \tag{51}$$

$$\sum_{r_j \in R_{x_k}} \lambda_{b_k,r_j}^{notify} G_{v_j} \beta^{-\Phi(f_j)} = \sum_{s_i \in S_{x_k}} w_{x_k,s_i}(1-\tilde{\rho}) \tag{52}$$

Since $\Phi(f_j) \in Y \ \forall f_j \in F$ where $Y = \{0, 1, ..., N-1\}$, we have:

$$\sum_{y \in Y} \beta^{-y} \Big(\sum_{r_j \in R_{x_k}, \Phi(f_j)=y} \lambda_{b_k,r_j}^{notify} G_{v_j} \Big) - \sum_{s_i \in S_{x_k}} w_{x_k,s_i}(1-\tilde{\rho}) = 0 \tag{53}$$

Note that we can express this as a polynomial. Substituting $\alpha = \beta^{-1}$ we get:

$$\sum_{y \in Y} \alpha^{y} \Big(\sum_{r_j \in R_{x_k}, \Phi(f_j)=y} \lambda_{b_k,r_j}^{notify} G_{v_j} \Big) - \sum_{s_i \in S_{x_k}} w_{x_k,s_i}(1-\tilde{\rho}) = 0 \tag{54}$$

We can therefore solve the (N-1)-order polynomial given in (54) to efficiently compute the exponential drop rates. We can solve this polynomial using the algorithm described in [32]. It relies on computing the eigenvalues of the companion matrix. The commonly-used NumPy Python library [44] implements this algorithm.