UC Irvine
UC Irvine Electronic Theses and Dissertations

Title
Resilient Communications Middleware for IoT Data Exchange

Permalink
https://escholarship.org/uc/item/3wq0c6tf

Author
Benson, Kyle Edward

Publication Date
2018

Copyright Information
This work is made available under the terms of a Creative Commons Attribution License, availalbe at https://creativecommons.org/licenses/by/4.0/

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA,
IRVINE

Resilient Communications Middleware for IoT Data Exchange

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Kyle E. Benson

Dissertation Committee:
Professor Nalini Venkatasubramanian, Chair
Professor Sharad Mehrotra
Professor Michael Dillencourt
Professor Magda El Zarki

2018

© 2018 Kyle E. Benson

DEDICATION

I dedicate my PhD thesis...

...

...first and foremost to my family. My parents Ed and Laureen are the best anyone could ask for, and not a day goes by that I don't thank God for their love and support.

...

...to my closest friends whose love and support (and at times ridicule) have also shaped the man I am today. Notably the Hoodlums (Rob, Tucker, Matt, Dylan, Alex, and Charles), other Delawareans (Kevin [both big and evil], Jamie, Nick, Brooke, Eric, Shawn, and Bryan), and my newer California friends (Moshe, Kristin, Jon, Kevin, Sky, Mehdi, Olivia, Deisy, and Robin).

...

...to the UCI Associated Graduate Students (esp. Chris, Justin, and Kevin) for providing extra-curricular activities and focus during my graduate studies.

...

... to my University of Delaware mentors, peers, and colleagues: my undergraduate advisor Michela Taufer; co-authors Trilce Estrada and Sam Schlacter; other mentoring professors Kathy McCoy, Terry Harvey, and Lori Pollock; the 2011 Computer Science cohort, especially "Pretty" Steve, Eric, Mary, and Sana.

...

... to my Charter School of Wilmington high school teachers, especially Bill Tressler who first introduced me to Computer Science and Chuck Biehl who helped expand my view of the world through discrete mathematics.

...

... to all the beauty of God's miraculous creation, whether visible or not: the mountains and snow, the oceans and tides, the beaches and birds, the forests and stars.

...

... to all the future humans who will carry on and improve our legacy. I hope my life's work can provide even the smallest of stepping pebbles on your way to the stars. But even more so I hope that love and truth guide you all to a better future where one day humanity might truly know peace.

TABLE OF CONTENTS

Page

LIST OF FIGURES **vii**

LIST OF TABLES **ix**

ACKNOWLEDGMENTS **x**

CURRICULUM VITAE **xi**

ABSTRACT OF THE DISSERTATION **xiii**

1 Introduction **1**
1.1 What is the Internet of Things (IoT)? . . . 1
1.2 What is Data Exchange? . . . 3
1.3 Motivation: Data-Centric IoT Challenges . . . 5
1.3.1 Constrained Devices and Cloud-centric Design . . . 5
1.3.2 IoT Data Exchange Middleware Design . . . 6
1.3.3 Enabling Resilient Internet of Things (IoT) Data Exchange . . . 8
1.4 Thesis Contributions and Organization . . . 10

2 Related Work **13**
2.1 IoT Data Exchange . . . 13
2.1.1 IoT Messaging Protocols . . . 14
2.1.2 IoT Middleware Services . . . 15
2.2 Resilient Network Communications . . . 17
2.2.1 Resilient Network Protocols . . . 17
2.2.2 Redundant Routing Paths . . . 18
2.2.3 Resilient Overlays . . . 19
2.2.4 Large-scale and Geo-correlated Failures . . . 22
2.2.5 Delay-Tolerant Networking . . . 22
2.3 Software-Defined Networking . . . 23
2.3.1 SDN Control Plane . . . 24
2.3.2 SDN for IoT . . . 26
2.3.3 SDN for Pub/sub . . . 27
2.3.4 SDN for Resilience . . . 28
2.3.5 QoS via SDN . . . 28

2.4 Edge Computing . . . 29

3 Our Proposed Middleware Approach to Resilient IoT Data Exchange 31
3.1 IoT in Mission-critical Settings . . . 32
3.1.1 Home & Community Safety . . . 33
3.1.2 Earthquake Detection and Alerting . . . 33
3.1.3 Smart Fire Fighting . . . 35
3.2 Resilient Data Exchange Goals . . . 38
3.3 Our Proposed Middleware Approach . . . 40
3.4 System Architectures & Middleware Design . . . 45
3.5 Leveraging Edge Resources . . . 48
3.6 SDN for Flexible IoT Edge Network Control . . . 49

4 An IoT Deployment Experience 52
4.1 SCALE: Safe Community Awareness and Alerting Leveraging the Internet of Things . . . 53
4.2 System Architecture . . . 54
4.2.1 Cloud Data Exchange for IoT . . . 55
4.2.2 Sensing Client . . . 59
4.2.3 Analytics . . . 66
4.2.4 Actuation . . . 68
4.3 Conclusions & Research Challenges . . . 71
4.3.1 Resilience Concerns . . . 71
4.3.2 Data Exchange . . . 72
4.3.3 Analytics . . . 75

5 Geo-aware Resilient Overlays for Cloud-centric IoT Data Collection 76
5.1 Chapter Overview . . . 77
5.2 Resilient Overlays for IoT Data Exchange . . . 79
5.2.1 Failure Avoidance . . . 79
5.2.2 P2P vs. SDN Overlay Construction . . . 81
5.3 Algorithms for Geo-Diverse Route Selection . . . 82
5.3.1 Model and Notation . . . 83
5.3.2 Geo-diverse Path Heuristics . . . 84
5.4 Experimental Setup . . . 89
5.4.1 Simulation Design . . . 89
5.4.2 Modeling Community Infrastructure Topologies . . . 91
5.4.3 Failure Model . . . 95
5.5 Experimental Results . . . 97
5.5.1 Comparing Geo-diverse Path Heuristics . . . 98
5.5.2 Comparing Other Parameters . . . 100
5.5.3 Sharing Network Resources Between IoT Deployments . . . 101
5.6 Prototype Implementation . . . 102
5.6.1 Fully Peer-to-Peer Overlay Considerations . . . 102
5.6.2 Extending SCALE with GeoCRON . . . 105

5.7 Chapter Summary and Discussion . . . 106
5.7.1 Integrating GeoCRON Into Our Proposed Middleware . . . 107

6 Edge Communications for Resilient IoT Data Exchange 108
6.1 Chapter Overview . . . 109
6.2 Our Approach to Resilient IoT Data Exchange . . . 110
6.2.1 A Driving Scenario: Smart Campus Disaster Response . . . 110
6.2.2 Ride-enhanced IoT Services for Emergency Response . . . 112
6.2.3 Resilient IoT Data Exchange (Ride) Workflow . . . 116
6.3 Ride Algorithms . . . 119
6.3.1 Ride-C – Data Collection in Ride . . . 119
6.3.2 Ride-D – Data Dissemination in Ride . . . 123
6.4 Prototype Implementation . . . 127
6.5 Experimental Evaluation . . . 132
6.5.1 Experimental Setup . . . 132
6.5.2 Ride Evaluation in a Seismic Alerting Scenario . . . 135
6.5.3 Ride-C Performance & Parameter Space Evaluation . . . 137
6.5.4 Ride-D Scalability & Parameter Space Evaluation . . . 142
6.6 Chapter Summary and Discussion . . . 147
6.6.1 Integrating Ride Into Our Proposed Middleware . . . 148

7 Prioritizing Heterogeneous IoT Information Flows at the Edge 150
7.1 Chapter Overview . . . 151
7.2 The FireDeX Approach . . . 153
7.2.1 A Driving Scenario: Fire Fighting with IoT . . . 154
7.2.2 IoT Data Exchange Addressed by FireDeX . . . 154
7.2.3 Enabling Event Prioritization . . . 156
7.3 FireDeX Formal Model . . . 160
7.3.1 Queueing Network Performance Modeling . . . 162
7.3.2 End-to-end Analytical Model . . . 168
7.4 Data Exchange Configuration Algorithms . . . 172
7.4.1 Utility Functions . . . 172
7.4.2 Priority Assignment Algorithm . . . 173
7.4.3 Ensuring Queue Stability via Preemptive Drop Rates . . . 175
7.5 Prototype Implementation . . . 178
7.5.1 Application layer . . . 179
7.5.2 Data exchange layer . . . 181
7.5.3 Network layer . . . 182
7.5.4 Implementation challenges . . . 183
7.6 Experimental Results . . . 187
7.6.1 Experimental Setup . . . 188
7.6.2 Validating our Queueing Network Model . . . 191
7.6.3 Evaluating the FireDeX Approach . . . 193
7.6.4 Comparing Prioritization & Drop Rate Algorithms for Situational Awareness . . . 196

7.6.5 Assessing the FireDeX Prototype . . . 198
7.7 Chapter Summary and Discussion . . . 202
7.7.1 Integrating FireDeX Into Our Proposed Middleware . . . 202

8 Conclusion **204**
8.1 Future Directions . . . 205
8.2 Towards the Future of Resilient IoT Data Exchange . . . 209

Bibliography **211**

Appendices **226**
A Multi-class Priority Queue Analytical Model . . . 226
B Efficiently Computing Drop Rate Policies . . . 229

LIST OF FIGURES

Page

1.1 IoT data exchange architecture . . . 4

2.1 Resilient Overlay Network (RON) routing . . . 20
2.2 SDN control/data plane separation . . . 23
2.3 Virtual network embedding . . . 26

3.1 Smart fire fighting research roadmap . . . 35
3.2 IoT-enhanced structure fire scenario . . . 36
3.3 Smart fire fighting data exchange . . . 37
3.4 Proposed cross-layer middleware approach . . . 41
3.5 Proposed middleware architecture . . . 46
3.6 Chapter progression and SDN abstrations . . . 50

4.1 SCALE data exchange . . . 56
4.2 Smart smoke detector wiring diagram . . . 62
4.3 SCALE Client Architecture . . . 64
4.4 SCALE Client Class Diagram . . . 65
4.5 SCALE server architecture . . . 67
4.6 SCALE dashboard . . . 70

5.1 Community infrastructure monitoring . . . 78
5.2 GeoCRON approach . . . 80
5.3 GeoCRON overlay architecture . . . 81
5.4 Overlay usage in GeoCRON . . . 85
5.5 GeoCRON infrastructure topology . . . 92
5.6 GeoCRON infrastructure topology (less redundant) . . . 93
5.7 GeoCRON failure model . . . 96
5.8 GeoCRON heuristics' results . . . 99
5.9 GeoCRON multipath fanout results . . . 100
5.10 GeoCRON failure probability results . . . 101
5.11 GeoCRON within our middleware architecture . . . 107

6.1 The Ride middleware architecture . . . 111
6.2 Ride's workflow . . . 117
6.3 A prototype of Ride in our physical lab test-bed. . . . 130
6.4 The network topologies used in our experiments. . . . 134

6.5 An example of Ride's failure adaptations 137
6.6 Ride-C configuration results 138
6.7 Congested Cloud Data Path (CDP) results 140
6.8 Varying Maximum Detection Time results 140
6.9 Varying failure probability results 141
6.10 Comparing failure detectors 142
6.11 MDMT-construction algorithm results 144
6.12 MDMT-selection policy results 145
6.13 Ride-D overhead results 146
6.14 Ride within our middleware architecture 148

7.1 The FireDeX cross-layer middleware. 153
7.2 FireDeX middleware architecture 157
7.3 Mapping subscriptions to network flows for prioritization 159
7.4 FireDeX queueing network model. 160
7.5 FireDeX implementation 179
7.6 Web dashboard. 180
7.7 Web dashboard. 181
7.8 FireDeX workflow 182
7.9 FireDeX experimental network topology 185
7.10 FireDeX multiple subscriptions challenge 186
7.11 Analytical vs. simulation results 191
7.12 Results from scaling up # subscriber 193
7.13 Comparing various $\tilde{\rho}$ values. 194
7.14 Evaluating success rates 195
7.15 Evaluating response times 195
7.16 Comparing priority-assignment algorithms 197
7.17 Comparing drop rate policies 197
7.18 Analytical vs. simulation vs. prototype results 200
7.19 Scaling # subscribers with prototype 201
7.20 FireDeX within our middleware architecture 203

8.1 Our complete proposed middleware architecture 210

LIST OF TABLES

Page

4.1 Access/edge network technologies used in SCALE 60

5.1 Latency overhead for overlay routing 106

6.1 Parameters Used in Ride-C 122

7.1 FireDeX formal notation 161
7.2 FireDeX experimental configuration parameters 187
7.3 FireDeX experimental configuration parameters for prototype 199

ACKNOWLEDGMENTS

I would like to thank first and foremost my co-authors and various other peers I worked closely with during my PhD: Georgios Bouloukakis, Guoxi Wang, Qing Han, Luca Scalzotto, Qiuxi Zhu, Kyungbaek Kim, Andrew Yang, Ranga Raj, and Kiyoshi Nakayama. I also thank my various mentors from both my PhD studies and internships: my dissertation committee, especially my advisor Nalini Venkatasubramanian for all her guidance throughout the years; Matthew MacFreier from ViaSat; Daniel Hoffman from Montgomery County, MD and the SCALE team; Robert Szewczyk from Nest; Young-Jin Kim and Marina Thottan from Bell Labs. I further thank the various teams and collaborators who have either contributed to this research or supported it in some capacity: the CSN team (especially Mani Chandy and Julian Bunn); the entire SCALE team (especially Dan Hoffman, John Cohn, and Charles Fracchia); the NFPA (especially Casey Grant) and other authors of the Research Roadmap for Smart Fire Fighting; UCI's Office of Information Technology for discussing campus network topologies; Extreme Networks for providing SDN testbed infrastructure.

My PhD research was supported in part by: NSF awards CNS 1450768, CNS 1143705, CNS 0958520, CNS 1528995; NIST Award # 70NANB17H285; the Bren School of Information and Computer Sciences at UCI; the ARCS foundation; the La Verne Noyes Scholarship program.

CURRICULUM VITAE

Kyle E. Benson

EDUCATION

Doctor of Philosophy in Computer Science — **2018**
University of California, Irvine — *Irvine, California*

Bachelor of Science in Computer Science — **2011**
University of Delaware — *Newark, Delaware*

RESEARCH EXPERIENCE

Graduate Student Researcher — **2011–2018**
University of California, Irvine — *Irvine, California*

Research Intern — **Summer 2016**
Nokia Bell Labs — *Murray Hill, New Jersey*

Undergraduate Researcher — **2010–2011**
University of Delaware — *Newark, Delaware*

SOFTWARE ENGINEERING EXPERIENCE

Software Engineer Intern — **Summer 2015**
Nest Labs — *Palo Alto, California*

Innovation Fellow — **Spring–Summer 2014**
Montgomery County — *Montgomery County, Maryland*

Software Engineering Intern — **Summer 2013**
ViaSat, Inc. — *Carlsbad, California*

LEADERSHIP EXPERIENCE

Vice President of Social Affairs — **2013–2014**
Associated Graduate Students of UCI — *Irvine, California*

Site Assistant Technical Supervisor — **2010–2011**
University of Delaware — *Newark, Delaware*

TEACHING EXPERIENCE

Teaching Assistant — **2012–2018**
University of California, Irvine — *Irvine, California*

MAGAZINE ARTICLE PUBLICATIONS

SCALE: Safe Community Awareness and Alerting Leveraging the Internet of Things **Dec 2015**
IEEE Communications Magazine

REFEREED CONFERENCE PUBLICATIONS

FireDeX: a Prioritized IoT Data Exchange Middleware for Emergency Response **Dec 2018**
ACM Middleware

Ride: A Resilient IoT Data Exchange Middleware Leveraging SDN and Edge Cloud Resources **April 2018**
IEEE Internet of Things Design and Implementation (IoTDI)

Resilient Overlays for IoT-based Community Infrastructure Communications **April 2016**
IEEE Internet of Things Design and Implementation (IoTDI)

Improving Sensor Data Delivery During Disaster Scenarios with Resilient Overlay Networks **March 2013**
Workshop on Pervasive Networks for Emergency Management (PerNEM) as part of IEEE Conference on Pervasive Computing and Communications (PerCom)

SOFTWARE

SCALE Client `https://github.com/KyleBenson/scale_client/`
Python-based event-driven middleware for acquiring data from various sensors (i.e. via Raspberry Pi platform), processing it, and exporting it via multiple networks and data exchange protocols.

Ride middleware `https://github.com/KyleBenson/ride/`
Resilient IoT Data Exchange (RIDE) using SDN and edge computing. Includes a mininet-based simulation framework, REST API adapters for SDN controller interaction, and various graph-based algorithms. Also used in FireDeX project.

ABSTRACT OF THE DISSERTATION

Resilient Communications Middleware for IoT Data Exchange

By

Kyle E. Benson

Doctor of Philosophy in Computer Science

University of California, Irvine, 2018

Professor Nalini Venkatasubramanian, Chair

IoT aims to improve our daily lives by seamlessly integrating heterogeneous devices into the physical space around us through a digital ecosystem of sensor data, communications, analytics, and semi-automated actions. Our exploratory proof-of-concept IoT system, SCALE, motivated our research challenges and served as a testbed for deploying our proposed approach. In particular, we identified the critical need for IoT data exchange: an event-driven pattern using producer-consumer abstractions to connect many heterogeneous devices, services, and users. This provides a seamless communications fabric for large-scale networks of heterogeneous resource-constrained devices. IoT deployments keep system deployment costs and complexity low by off-loading much of the logic to the cloud. Hence, data exchange manages the flow of application-supporting messages in order to gather data at the network edge, process it in the cloud, and then use it at the edge. However, infrastructure failures (e.g. during a natural disaster) or resource limitations (e.g. during emergency response activities) disrupt connectivity with such cloud platforms.

To ensure a high degree of confidence in the resilient operation of mission-critical IoT systems, this thesis proposes middleware solutions to resilient communications in support of IoT data exchange in highly-challenged environments. Our proposed techniques leverage current application information/resilience requirements, physical network topology awareness, and

modern system configuration abstractions (i.e. edge computing and SDN). They account for and adapt the IoT data exchange to failures and other impactful constraints in the underlying network infrastructure. We propose the use of SDN-enabled edge computing for three primary reasons: reliability, performance, and locality (i.e. making use of data produced at the edge there at the edge).

We explore this approach within the context of two different mission-critical scenarios and three projects that build on each other to progressively leverage more edge intelligence as we focus on more local settings. We first leverage a centrally-controlled geo-aware resilient overlay network (GeoCRON) to improve cloud-centric collection of IoT seismic sensor data during geographically-widespread earthquake-induced network infrastructure failures. We then consider the question of whether to collect and process IoT data in the cloud or at the edge for further resilience to cloud connection instability. To this end, the Ride system monitors cloud overlay paths and redirects IoT data flows (when necessary) transparently to IoT devices. Ride also expands on the seismic scenario with earthquake early-warning through a novel resilient local alerting mechanism based on redundant multicast trees. Lastly, we consider balancing the needs of multiple mission-critical applications in an IoT-enabled structure fire response. The FireDeX project models the complete data exchange (i.e. IoT hosts, data exchange brokers, and network infrastructure) and adaptively prioritizes information flows according to information requirements and network resource constraints.

Altogether, the proposed middleware approach enables a more holistic view of and control over the data exchange process. As demonstrated in the individual projects, this occurs at different network and IoT deployment scales. While these contributions are but a few parts of the greater resilient IoT data exchange challenge, they represent a few steps in the direction of that goal.

Chapter 1

Introduction

1.1 What is the Internet of Things (IoT)?

The IoT represents the next stage of technological evolution in the information age. It aims to improve our daily lives by seamlessly integrating heterogeneous physical devices throughout the spaces we frequent regularly. By leveraging pervasive sensor data, communications, and analytics, IoT digitizes physical devices and the environment they co-inhabit with humans. This bridges physical spaces with virtual applications to perform semi-automated actions and/or expose human-centric information services within the physical world.

While pervasive computing has been explored for decades (e.g. the annual IEEE PerCom conference is in its 17th year), the 2010s saw a leap in available products. IoT components became cheap and available enough (e.g. Raspberry Pi in 2012) that more hobbyists and researchers began experimenting with do-it-yourself IoT systems. Economies of scale continue to enable more pervasive IoT deployments in previously-mundane spaces. They also make IoT devices affordable and accessible even to lower-income populations. Over time, this will help bridge the chasm in the digital divide: pervasive computing systems were previously

available only to those who could afford them and possess the expertise required for their setup and configuration.

As we move further into a future full of connected devices, the IoT promises to revolutionize societal-scale operations and influence our daily lives. It integrates pervasive sensing/actuation, dynamic data analytics, and communications. Domains such as transportation, home automation, healthcare, and emergency response are becoming increasingly IoT-enabled; this provides data-driven insights to improve situational awareness. This is particularly useful in mission critical applications, e.g. to enable effective and timely emergency response. Recent smart city efforts such as the SmartAmerica Challenge and Global City Teams Challenge have showcased the integration of IoT into a variety of community settings and application domains [171, 27, 107]. Such global efforts aim to leverage the IoT's promise to improve economic and living conditions for all.

§4 introduces our response to the SmartAmerica Challenge and consequent experiences with an IoT proof-of-concept called SCALE. It aims, through the use of modern connected devices and computer systems, to improve the safety of residents. We focused particularly on lower-income and elderly residents who often do not have access to advanced technologies such as home security systems, smartphones, and computers with Internet connections. To accomplish this goal, we designed an event-driven distributed system to sense safety-related data from devices in homes or on individuals, analyze it locally or within the cloud to detect possible emergency events, and automatically contact individuals (e.g. homeowners, caretakers, even emergency dispatch) to notify them and confirm if there is indeed an emergency. This project revealed a difficult but promising road towards real-world IoT deployments for personal safety, emergency response, and other mission-critical scenarios. This thesis leverages our experiences with SCALE, similar systems, and various mission-critical scenarios (see §3.1) to motivate a plethora of research problems. SCALE also serves as a testbed IoT ecosystem into which we integrate proof-of-concept modules that implement our novel

contributions. This thesis focuses in particular on network communications in support of resiliently connecting the various devices and services in IoT deployments to form a data exchange fabric.

1.2 What is Data Exchange?

IoT deployments require some distributed *data exchange* solution to manage the flow of relevant data to/from devices, systems and individuals (data producers and consumers). This typically takes the form of a middleware that connects the many heterogeneous devices, services, and users through a seamless communications fabric. Rather than completely implementing communication mechanisms from the ground up, such middleware supports reusable patterns from which larger systems can be composed more quickly, efficiently, and with less developer effort. Fig. 1.1 shows how the heavily event-driven nature of IoT ecosystems leads to a natural abstraction of their workflows as data producer/consumer patterns. Sensing devices embedded in the physical space monitor real-world events and produce data associated with them. The data exchange facilitates the transmission of this data via various communications networks to interested data consumers. These human users, actuation devices, or IoT applications/services consume events for further processing, storage, analysis, taking physical action, and/or conversion into higher-level events according to some semantics.

Consider the following common interaction styles supported by IoT data exchange middleware:

- Publish-subscribe (the Message Queue Telemetry Transport Protocol (MQTT) [141], CoAP's *observe* feature)
- RESTful interactions (HTTP, the Constrained Application Protocol (CoAP) [71])

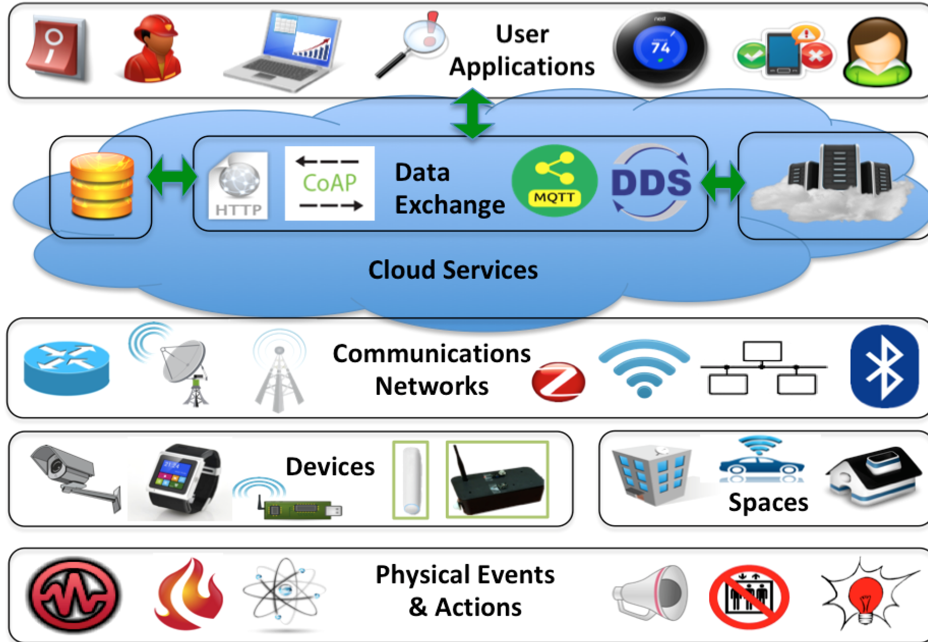


Figure 1.1: IoT deployments typically leverage cloud services and data exchange middleware to alleviate the burden on constrained devices and manage heterogeneity of underlying technologies, data, devices, etc.

- Streaming data (HTTP WebSockets)
- Upload-only for sensors using a gateway or long-range low power wireless technology such as LORA or Sigfox

In this thesis, we take an event-centric approach to IoT data exchange. The event-based nature and distributed broker architecture of pub/sub systems, which decouples devices in time and space, fits the needs of IoT interactions well. As such, we do not directly consider patterns such as upload-only and streaming as they can essentially be boiled down to publishing data related to events of interest, albeit perhaps with some quality of service (QoS) in the latter case. Rather, we primarily consider the **publish-subscribe** pattern in this thesis without loss of generality. We base this assumption on our previous experiences with such systems (e.g. see Chap. 4) and the popular use of pub/sub (e.g. MQTT[141]) in IoT implementations. Hence, we consider IoT sensors as publishers, all manner of data as events, and interested entities (i.e. human stakeholders or other IoT devices and services) as

subscribers. Data exchanges (e.g. a broker implementation) route information to actuators (e.g. alarms), data analytics services that detect new events, or to a local logging database for post-incident analysis and forensics.

1.3 Motivation: Data-Centric IoT Challenges

During the SCALE project's first phase, we identified and confirmed several key challenges for IoT that we detail later in §1.3.3. This thesis makes steps towards addressing three critical challenges: resilience, scalability, and intelligently managing heterogeneous resource-constrained devices. It does this through a networking-centric approach to *resilient IoT data exchange*: connecting many heterogeneous devices, services, and users through a seamless communications fabric. This middleware fabric facilitates the exchange of application-supporting messages under a variety of challenging conditions without requiring complex configurations that account for every different device and application.

1.3.1 Constrained Devices and Cloud-centric Design

IoT devices often tend to be low-powered, inexpensive, embedded systems with little intelligence. Their limited storage (i.e. flash memory and RAM), processing capabilities, and power availability means they must run lightweight highly-optimized software with low code footprint and minimal computation. Hence, resource-constrained IoT devices and deployments keep costs and complexity low by off-loading much of the logic to the cloud. While recent work aims to support processing IoT workloads in-network (see §2.1.2), this often remains infeasible due to e.g. limited device resources or reliance on proprietary network infrastructure. Instead, thin IoT client designs leverage cloud-based services to generate actionable information in response to real-world events.

These cloud services provide the storage and compute capabilities necessary to manage IoT devices and support their applications. The compute capabilities include: serving user-facing applications (e.g. web-based interfaces), infrastructure for managing large volumes of data from disparate sources, event-processing pipelines that compose multiple operations on this data, and machine learning for advanced analytics. Managing IoT deployments includes: securing connections to prevent unauthorized access, over-the-air (OTA) updates with the latest device software, and monitoring devices to ensure proper function.

To connect on-site devices with off-site cloud data centers, these deployments leverage various communications networks. While these connections usually traverse the public Internet backbone at some point, the edge networks are often wireless (e.g. Wi-Fi, cellular, Bluetooth, etc.). Managing devices across heterogeneous networks, especially those without IP or oriented towards low-power communications (e.g. Zigbee and 6LoWPAN), introduces a layer of complexity that cloud-based services and middleware solutions aim to address.

1.3.2 IoT Data Exchange Middleware Design

In conjunction with the aforementioned cloud services, data exchange middleware further alleviates the complexity of managing heterogeneous devices, services, and networks without burdening constrained devices. From our SCALE project (see §4.3.2), we learned that IoT data exchange should enable a flexible loose coupling of devices, apps on those devices, and IoT system designs for simple abstract data-processing. Such abstractions as data analytics pipelines proved highly flexible in adding new components, be they additional software or even hardware (e.g. adding peripheral sensors via networked microcontrollers), to the SCALE system. Especially when considering less-capable embedded devices, this orchestration becomes much simpler with a unified data exchange middleware. Due to these constraints, the data exchange must be designed with simplistic client-side interactions and

relegate more sophisticated coordination to more capable entities e.g. cloud-based servers or brokers.

A hierarchical data exchange architecture with a distributed broker network lends itself well to IoT deployments. "Top-level services" run in the cloud to manage global networks of devices. They may potentially coordinate across "regional" services that cover certain areas. This breakdown by physical location lends itself well to the event-centric nature of IoT for several reasons:

1. Data interests / interactions typically exhibit strong spatial locality (e.g. monitoring our homes)

2. Organic IoT deployments add additional regions over time or break up denses ones into smaller ones

3. Each participating organization can manage its independently-evolving deployment according to its own application domain, system, and policy requirements

At the bottom of this hierarchy lies the finest granularity of the "**edge**": devices physically reside in edge networks and often interact with other nearby actors. Edge services (e.g. a smart Building Management System (BMS)) can support these local interactions by providing lower latency, location-centric services, and local device management. However, they may also facilitate external interactions. For example, many low-powered devices operate at the edge on non-IP networks such as Zigbee and must communicate with outside entities through a **gateway** responsible for conversion to the common IP protocol.

As such, a complete IoT data exchange solution must consider the complexities of coordinating across these various edge networks and hierarchies. This may include challenges such as mobility, protocol translation, and security. In this thesis, we specifically focus on resilient network communications in support of the data exchange overlay. As described

in §3, this takes a step towards bridging the gap between the network and data exchange layers by exploiting data semantics to translate higher-level characteristics into lower-level configurations.

1.3.3 Enabling Resilient IoT Data Exchange

As more individuals come to rely on IoT systems, especially for mission-critical scenarios like those outlined in §3.1, we must clearly ensure a high degree of confidence in their resilient operation: reliability, resource-awareness, and dynamic adaptability in challenging situations. To this end, we narrow the focus of this thesis to this notion of **resilience**: adapting the system in response to various challenges so as to maintain continuity of operations even if in a degraded mode.

The low-cost constrained nature of IoT devices leaves them susceptible to a variety of problems including including faulty components, inaccurate sensing, and software bugs. Resilient operation of applications and services in the presence of failures, disruptions, and other unexpected challenges is a key issue. As IoT systems scale and increasingly rely on cloud services to operate, resilience of these services, the software and algorithms comprising them, and the networks that link them with end devices are crucial considerations. Furthermore, infrastructure failures (e.g. during a natural disaster) or resource limitations (e.g. during emergency response activities) disrupt connectivity with cloud platforms. Such disruptions commonly occur as general Internet service outages, but are far more impactful in extreme events such as natural catastrophes or man-made disasters [48, 100, 52, 123]. While IoT deployments can be used to create dependable awareness and consequently improved decision making in disaster settings, this data must be quickly delivered in the face of massive geo-correlated network outages and resource limitations. Critical applications such as healthcare and emergency response must continue to operate meaningfully (at least in a degraded

service mode) despite cloud and connectivity disruptions.

Most modern IoT deployments rely heavily on cloud intelligence to mitigate such disruptions. However, cloud outages do occur [187, 178] and would leave deployments reliant on this approach inoperative. While adding intelligence to the devices themselves can help alleviate some issues, it increases their cost and complexity. Furthermore, some researchers argue that the cloud-centric approach limits IoT, which would be better served by a data-centric approach [197]. Without complete autonomy, end devices cannot always overcome such challenges, which are especially relevant to the mission-critical scenarios described in §3.1. With the increasingly-widespread deployment of more flexible networking infrastructure (e.g. Software Defined Networking), we advocate for an approach that also leverages intelligence embedded in the network itself. This enables a holistic middleware approach to adaptively translating application-specified resilience requirements and system state into concrete system configurations at multiple layers.

Hence, we focus on resilient network communications. This lower layer must operate effectively in order to support the meaningful exchange of relevant data. By simultaneously considering both the network and data exchange middleware layers of IoT systems, we aim for a more holistic solution than would be possible when considering only one layer at a time in isolation. Clearly, a truly holistic solution to the challenge of resilience must address other layers of the system stack:

- Compute hardware – gracefully handling faults or the degradation over time of cheap components crammed into a small form factor
- Power constraints – greater energy density in batteries, operation in hostile conditions, and energy harvesting
- Sensors – callibration, inaccuracies, missing readings, and longevity

- Application software – bugs that hamper operations or limit the applications' effectiveness; handling missing data or unavailable services

However, we leave these other challenges out of scope for this thesis and maintain a narrow focus on the networking and data exchange middleware layers. The approach outlined below can be leveraged as one part of a more complete solution to the plethora of IoT resilience challenges. Such a middleware should perform under a variety of challenging conditions (e.g. dynamic requirements/constraints, infrastructure failures, etc.) without requiring complex manual configurations that account for every different device and application.

1.4 Thesis Contributions and Organization

This thesis aims to address some of the above challenges through a cross-layer (i.e. middleware and network) middleware approach to resilient IoT data exchange. It proposes the centralized control of pub/sub-oriented information flows and melding these layers through the use of Software-Defined Networking. This network and application-aware control middleware avoids burdening constrained devices, adapts to infrastructure challenges, incorporates evolving information requirements, and supports heterogeneous technologies. We demonstrate four related but distinct approaches within different IoT deployment settings and scales. As we progress through the chapters, each project progressively leverages more edge intelligence and focuses on more local scenarios.

The following is the overall organization and research contributions of the thesis:

- Chapter 2 surveys related work.

- Chapter 3 proposes our approach in greater detail.

- Chapter 4 details our contribution to the research community of an early IoT proof-of-concept called SCALE. It uses novel networking technologies, commodity sensor devices,cloud services, and middleware abstractions to sense, analyze, and act on sensed events in a distributed manner. We present SCALE's architecture, capabilities, and our lessons learned that drive the other research projects in this thesis.

- Chapter 5 describes a centrally-controlled Geographically-Correlated Resilient Overlay Network (GeoCRON). It improves cloud-centric collection of IoT seismic sensor data during geographically-widespread earthquake-induced network infrastructure failures. Our key contributions include the novel resilient seismic sensing scenario, consideration of multiple IoT infrastructure networks within a community-scale setting, and the modeling of this scenario within a simulated experimental framework.

- Chapter 6 considers whether to collect and process IoT data in the cloud or at the edge. It expands on the seismic scenario by exploring both reliable data collection and alert dissemination (i.e. earthquake early-warning) in a smart campus environment. The project titled Resilient IoT Data Exchange (Ride) exploits both cloud and edge resources to maintain network awareness and intelligently choose data exchange paths to facilitate time-critical IoT applications such as the example seismic alerting scenario. We apply SDN and novel algorithms to ensure service stability through cloud path fail-over, edge fail-over, and redundant multicast trees for enhanced local alerting. Our main contributions from this project include a resource-conserving adaptive network probing technique, heuristic-based algorithms for consructing redundant multicast trees for pub/sub dissemination, and novel techniques and heuristics for exploiting network state awareness gathered during the data collection phase when selecting the best available multicast tree for the dissemination phase.

- Chapter 7 explores resilient IoT data exchange in a smart fire-fighting scenario and balances the needs of multiple mission-critical applications. The FireDeX middleware

incorporates mechanisms and algorithms for ensuring reliable communications over multiple networks between a smart building's infrastructure (e.g. sensors, actuators, occupants) and first responders. It enforces event prioritization at the network layer to ensure these responders receive pertinent data in a timely manner despite resource constraints, hostile environments, and heterogeneous protocols and data models. Our primary contributions include a cross-layer analytical model of IoT data exchange performance derived from a theoretical queueing network. We demonstrate the accuracy of our model, significant improvements to the value of information captured during a response effort, and the real-world performance of a prototype system.

- Chapter 8 concludes the dissertation with lessons learned, a holistic view at our entire proposed middleware, and a look forward to future research problems we must address to enable truly resilient IoT data exchange.

Chapter 2

Related Work

We now survey relevant work to provide appropriate background for this thesis. We start with an overview of messaging protocols and middleware services related to IoT data exchange. We then explore techniques for improving communications resilience, which will support the lower-level IoT connectivity and thereby also improve resilience for IoT data exchange middleware. Lastly, we briefly overview research that explores leveraging edge resources and SDN. These emerging technologies and concepts can help enable more resilient IoT data exchange solutions, and so we apply these in designing our proposed middleware approach.

2.1 IoT Data Exchange

Recall from §1.2 that we define IoT data exchange as a middleware solution to manage the flow of relevant data to/from devices, systems and individuals (data producers and consumers). This section overviews various messaging protocols and higher-level middleware services that provide IoT data exchange solutions. The former provides the lower level of connectivity in terms of messaging primitives. The latter builds upon this lower layer to

provide higher-level services with more advanced capabilities and guarantees. While not comprehensive, we aim to provide a survey of typical solutions and how they fail to address all of the resilience challenges outlined in §1.3.3.

2.1.1 IoT Messaging Protocols

IoT developers can either implement their own custom data exchange mechanism or choose from numerous pre-existing **popular IoT protocols** or industry standards. Despite attempts to standardize machine-to-machine (m2m) communication (e.g. oneM2M [180], Thread[1]) as well as frameworks for building IoT applications (e.g. AllJoyn[2], Nest Weave[3]), no one choice has captured enough market share to be considered a de facto standard yet. A Google search reveals that a handful of protocols have been touted over the last few years as ideal for IoT data exchange. One common theme across them is the small footprint of client libraries, which allows their use by constrained devices. They also tend to rely on centralized server(s), simplifying their deployment since most IoT systems rely on cloud services anyway. Each of these protocols came to popularity for different reasons. Hence, a system developer must choose the protocol for a particular deployment based on requirements including performance, reliance on centralized infrastructure, client code footprint, QoS guarantees, and a balancing act of features and customization vs. API simplicity.

Our SCALE project initially incorporated one of the simplest and most popular IoT protocols, Message Queuing Telemetry Transport (**MQTT**) [141]. This open sourced lightweight topic-based publish-subscribe protocol boasts a simple API that enables an asynchronous event-driven programming paradigm. For RESTful architectures, **HTTP** remains a popular option due to its near-universal penetration and ability to directly interact with web services (e.g. forward sensor data directly to cloud apps). RFC 7252 [71] defined a lighter-weight

[1] `https://www.threadgroup.org/`
[2] `https://github.com/alljoyn/alljoyn.github.com/wiki`
[3] `https://nest.com/weave/`

RESTful protocol for more constrained devices. The the Constrained Application Protocol (CoAP) operates over UDP rather than TCP, which better supports battery-powered devices and even low-power wireless networks such as ZigBee. It also provides subscription-like capabilities through its *observe* feature as well as HTTP compatibility through a proxy and UDP-based group multicast. The Data Distribution Service (**DDS**) [142] provides pub-sub style *message bus* semantics and the ability to define the data schema in terms of types, sizes, etc. Along with its support for UDP transport and a plethora of QoS options, these features enable broker-less group communication as well as real-time guarantees. The Extensible Messaging and Presence Protocol (**XMPP**) [159] features integrated support for presence (i.e. device discovery) and its extensible nature allows for configuring additional services such as publish-subscribe and the QoS support lacking from the core protocol. The Asynchronous Message Queueing Protocol (**AMQP**) [14], an enterprise-grade message queue standard, receives much attention for managing data flows among backend services. However, its lack of an embedded device-capable implementation precludes its use in highly-constrained IoT client devices.

2.1.2 IoT Middleware Services

We now consider a few middleware solutions that provide IoT services beyond connectivity and basic data exchange. Such solutions can leverage work in large-scale pub/sub systems [23, 160, 200] and adapt them specifically for IoT. For example, the authors in [79] propose moving IoT designs beyond the cloud using a distributed replicated append-only log for IoT data called the Global Data Plane. The Nodle.io (`www.nodle.io`) project leverages smartphones as gateways to provide Internet connectivity for constrained IoT devices with lower-powered radios (e.g. BLE). This creates a large-scale network for location tracking and data exchange without requiring dedicated infrastructure. The authors of [76] leverage information-centric networking approaches to facilitate in-network processing of IoT data

before it reaches the destination consumer(s). Such data-centric approaches also enable research into managing data exchange configurations according to data processing workload characteristics [129, 183]. Other related research proposes similar centralized control of IoT data exchange [109, 198].

The challenge of **interoperability** is typically addressed through protocol translation. For example, the Eclipse project Ponte [148] converts between HTTP, CoAP, and MQTT by providing thin wrappers around numerous choices of the pub/sub backend and underlying data storage. IFTTT (`http://ifttt.com`) facilitates the interoperation of various IoT services through a cloud platform through which users can define triggers and actions that should be taken in response. For example, one might configure their home to email them whenever someone rings their smart doorbell. The HomeOS project [57] addressed a similar challenge with a different approach. It abstracts smart devices like PC peripherals in an operating system and accomplishes interoperability via drivers. It manages a smart home through a central server that provides local computation and storage The Building Operating System Services (BOSS) project took a similar approach to this [55]. BOSS abstracts devices through drivers, which enables an abstraction layer running the same code on multiple buildings with different underlying physical hardware. It also establishes fault domains.

Other services include supporting sensing at a semantic level. SATWARE [102] approaches this by defining *virtual sensors*, which abstract the underlying sensor feed and expose a semantically-annoted resulting sensor feed. SensorML [38] defines an XML encoding to describe sensors and their capabilities, whether physical or virtual. This can be used to define an ontology of system components and how they work together. The Sentilo platform [9] provides an open source architecture for sensor data management in a smart city. It uses RESTful APIs to provide basic services such as sensor data discovery, data transfer, and pub/sub. It includes a large catalog of sensor types so as to generically support many different data sources.

2.2 Resilient Network Communications

Techniques to handle end-to-end dependability (e.g. due to infrastructure failures) have been designed at different levels of the system stack. In general, they can be considered as **failure avoidance** (i.e. proactive) or **recovery** (reactive). Reactive recovery of network paths may take several seconds or even minutes [67]. During recent hardware maintenance on our local campus data center we measured similar downtimes of ≈45secs. - 5 mins. Other reactive approaches include retransmission or selecting alternative routes as discussed below. Failure avoidance limits the perceived impact of infrastructure challenges and includes techniques such as multi-path transmission or coding in wireless networks.

While we aim to survey general network resilience in this section to lend context to the overall thesis, we only scratch the surface of this topic. Interested readers may look to surveys such as [177] for a more comprehensive investigation. Both [176] and [177] discuss general challenges to networked systems and discuss the proposed *ResiliNets* framework, which aims to formalize the steps and strategies involved in designing and maintaining more robust networks.

2.2.1 Resilient Network Protocols

The original Internet architecture was designed as a scalable high-performance resilient network for delivering data. The Internet Protocol (IP) [106] itself was designed with resilience in mind during the early days of ARPANET. A prime benefit of packet switching is that messages transmitted over the network would be routed by the infrastructure itself. This allowed it to route around failures in the network by updating the routing tables after a failed node or link. Beyond this *best effort* model of packet routing, the Transport Layer provides reliability guarantees through TCP by retransmitting lost packets. This still requires an

available route, but it is resilient to transient failures or occasional packet corruption. While UDP does not provide such reliability, many application-layer protocols (e.g. CoAP [71]) implement their own reliability mechanisms on top of it by retransmitting datagrams similar to TCP. We leave out of scope discussions about resilience using coding and error correction e.g. Ethernet's Frame Check Sequence or layered coding used in many video transmission protocols that tolerates some packet loss without requiring retransmission.

Throughout the Internet's evolution, researchers and engineers applied a variety of techniques to improve its resilience. Some of these techniques generalize and reappear at various points in different networking stacks. For example, the concept of redundancy is applied in terms of transmitting over multiple wireless channels to avoid interference, maintaining redundant networking equipment (e.g. routers, wireless basestation coverage) in case of failure, aggregating multiple physical links in a wired topology in case one fails, and even retransmitting unacknowledged packets when using TCP. Another application of redundancy that we will exploit in this thesis is maintaining multiple network paths between two nodes. This may take the form of recomputing alternative routes in routing infrastructure given knowledge of the complete topology (e.g. OSPF), maintaining different flows and either intelligently choosing between them with a higher level logic or copying data over both (e.g. MPLS, TCP multi-homing), or maintaining loops for fast re-routing of traffic in response to failures (e.g. SONET). These techniques, while capable of quickly and effectively handling transient network failures such as packet loss due to congestion or faulty networking equipment, break down during massive failure scenarios.

2.2.2 Redundant Routing Paths

Many different resilient networking techniques find and exploit redundant (i.e. disjoint) paths [149]. We explored techniques for recovering from double-link failures by reconnect-

ing a spanning tree. Our work decomposed the network into cycles in order to respond to these link failures by activating the correct backup links [136]. These may act as backup routes (i.e. reactive), but failing over to them takes several seconds or even minutes [67]. During recent hardware maintenance on our local campus data center we measured similar downtimes of $\approx$ 45secs. - 5 mins while the routing protocols converged to the alternative route. Hence, proactive techniques that enable multiple simultaneous connections for resilience to failures/packet drops may prevent such downtime. For example, consider the use of multi-homing, which establishes multiple connections at the transport layer to increase resilience to such failures [45]. Similarly, some systems exploit multiple access networks for redundancy [62, 186].

The traditional disjoint paths problem formulation minimizes the total number of edges/vertices shared by several of k different paths between a source and destination. Finding $k > 2$ maximally-disjoint paths of minimal cost is NP-Complete even if we restrict the number of hops in such paths to be ≤ 4 [36]. However, a pair (i.e. $k = 2$) of such paths can be found in polynomial time[181]. To efficiently find $k > 2$ disjoint paths, a polynomial-time minimum cost flow-based algorithm was proposed in [78]. It reformulates the problem to minimize *shareability*: the sum over all edges of the total number of paths (minus one) using that edge.

2.2.3 Resilient Overlays

In the face of failures and congestion, the underlying network infrastructure's routing protocols may take several minutes to utilize alternative routes due to reconvergence of the routing protocols. Several case studies have identified issues with Internet routing, in particular with the Border Gateway Protocol (BGP), during massive failure scenarios. For example, [191] found paths that needlessly passed through other continents following a major earthquake

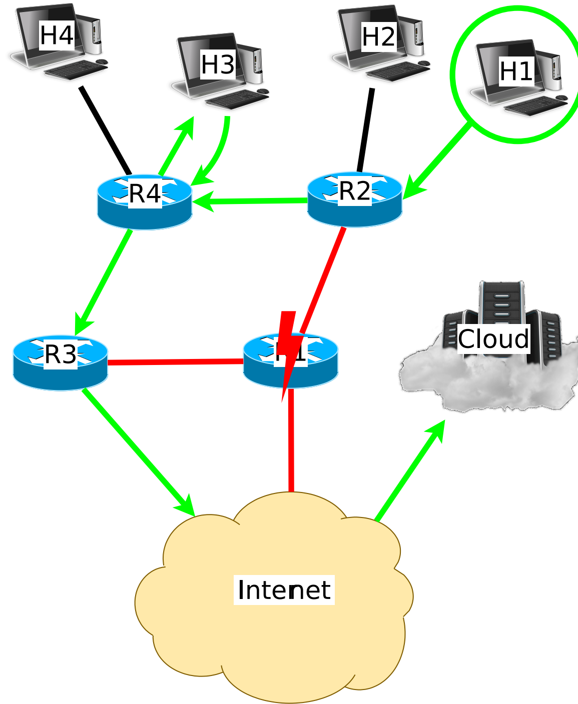


Figure 2.1: A node in a resilient overlay network routes around failures by passing packets through another overlay node.

in Taiwan. It also claimed that BGP policies negatively impact resilience on the Internet by not allowing certain paths. The authors of [82] found that most visible failures in the Internet did not exceed 5-15 minutes while the authors of [119] found that BGP route update convergence could take up to 15 minutes after a fault. During this time, some end-to-end connections may be unavailable because certain paths are non-functional but others may exist that the routing infrastructure is not yet aware of. Hence, we see that we cannot always rely on network routing infrastructure to address all failure scenarios. Part of the reason for this is the simplification of intelligence in the core of the network to improve performance and scalability.

Alternatively, application-layer *Resilient Overlay Networks (RONs)* can leverage alternative network routes to substantially improve delivery of messages, as well as latency, during

failures, unavailability, and congestion [16, 82]. RON nodes try to find an alternative path when the main one fails to deliver a packet. As shown in Fig. 2.1, they do this by acting as intermediary hops within the overlay network. They attempt to contact another node (i.e. peer) in the overlay to see if that node is reachable and has a working path to the desired destination. If it does, then the traffic is routed through this intermediate node to the destination until a more direct path becomes available or less congested.

Indeed, [82] found that "overlay networks can typically route around 50% of failures." When an end-host perceives a failure, or simply wishes to improve chances of delivery by utilizing an alternate path, it chooses such an intermediary overlay peer according to some metrics (e.g. latency, current load) and requests that it route traffic to the destination. Adding this level of intelligence to the routing infrastructure may incur large amounts of additional complexity and cost, but it can also be accomplished with simple end hosts in a peer-to-peer-like fashion. Deploying end hosts for the specific purpose of establishing a RON, or using those that are already part of a distributed sensing effort for this purpose as well, could possibly increase the reliability of a system without having to modify any of the routers in the underlying physical network.

Aside from the original work proposing RON [16], other research has explored aspects of this approach. For flash dissemination of alerts to the public, [67] constructed a reliable application layer multicast forest with multiple parents-to-multiple children for exploiting path redundancy. The authors of [107] studied a regional-area network for Internet access and resilient information sharing in emergency situations. They configure a mesh-topological network composed composed of multiple base stations interconnected with each other. Their system uses a variety of Ethernet-based wired or wireless transmission systems such as optical/metal Ethernet, WiFi, FWA, satellite, and unmanned aerial vehicle (UAV). Our work [29, 28], which we present in Chap. 5, added geo-awareness to the RON concept for resilience to geo-correlated failures e.g. caused by earthquakes.

2.2.4 Large-scale and Geo-correlated Failures

Many previous projects have explored resilience to failures in the Internet, although few have addressed large-scale geographically correlated failures. Most of these works aim to formally model failures and identify strategies for designing more reliable network infrastructure. Large-scale failures tend to be geographically-correlated in nature, whether due to a particularly impactful natural phenomenon (e.g. earthquakes as described in §3.1.2, tornado), a cascading failure from another network (e.g. power grid), or perhaps a targeted attack by human adversaries (e.g. electro-magnetic pulse weapon). Some research attempts to formally model these failures and extrapolate design methodologies for improving network infrastructure reliability from them. Straight line segments drawn through a network topology, failing any intersected links, were used in [137] to study geo-correlated failures. In [97], the most damaging link cuts possible for a given network provider is used to plan a more resilient network. Geo-diverse multipath routing within an autonomous system (AS) is studied in [153], and we borrow and expand on their geo-diversity metrics in Chap. 5. Another consideration is disseminating early warnings/notifications/alerts at short notice to the public. Flash dissemination can leverage application-layer overlays to ensure all recipients are notified as quickly as possible [67].

2.2.5 Delay-Tolerant Networking

When an application tolerates long delays (i.e. $>$ a few seconds) between delivery of messages, it can forego normal assumptions of end-to-end latency made by the traditional TCP/IP suite. In such a case, routing of packets may involve a *store-and-forward* approach in which intermediate routers must wait until the next hop is available to transfer the packet. Delay-tolerant networks often exploit mobility patterns of these intermediate nodes. We leave out of scope this specific focus on wireless, mobile, and delay-tolerant networking.

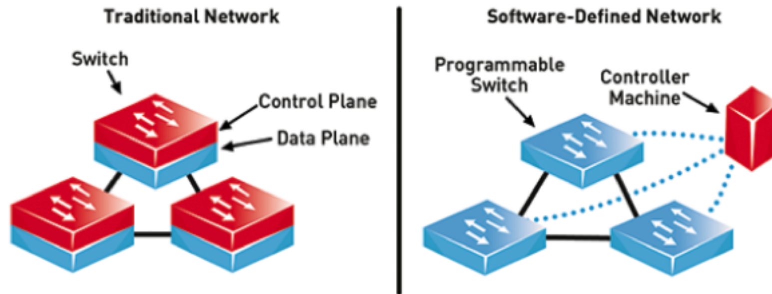


Figure 2.2: SDN separates the control and data plane to logically centralize the control plane. (source: Dungay 2016)

Instead, we refer interested readers to explore a somewhat recent survey on delay-tolerant networking in the context of vehicular networks [145]. We also refer them to our previous work (done outside the context of this thesis) on information gathering that exploits mobile data collectors in the context of a fire response within an unplanned infrastructure-deficient shanty town [108, 151].

2.3 Software-Defined Networking

Software-Defined Networking (SDN) represents a new paradigm in computer networking in which the control plane is separated from the data plane [130]. As shown in Fig. 2.2, the control plane is then logically centralized to make routing decisions globally. It then pushes configurations down to the data plane (e.g. physical switches) to carry out these decisions. SDN APIs (e.g. OpenFlow [130], P4 [37]) provide a unified view of and control over the underlying network infrastructure. OpenFlow defines a protocol for manipulating *flow table rules*, which match packets according to various OSI Layer 2-4 header fields and perform actions such as forwarding out some port or even manipulating the header. An SDN controller observes the underlying network by querying switches for various statistics e.g. packets sent/received, loss rates, etc. This enables the accurate collection and maintenance of evolving network conditions in support of dynamically adapting the data plane.

SDN controllers also expose high-level APIs for developing network applications/services. This enables the fully-automatable merging of network-and-application-awareness, derivation of unique communication requirements, and configuration of the underlying data plane switches.

SDN provides a variety of abstractions to represent the underlying physical network. These include authorized access to directly manage physical switches, control over virtual (software-based) switches [146] (e.g. running inside edge/cloud data centers), network virtualization [35] to reserve "slices" of the physical infrastructure, etc. SDN can provide a simplified single-network view of the whole distributed system that may span multiple physical heterogeneous networks (e.g. building Wi-Fi and local cellular) and different locations. Recent research into SDN-enabled 5G cellular architectures [182] supports the potential for such interfaces that connect e.g. emergency responder devices to a building's internal network.

For more information about SDN beyond the topics covered below, interested readers may wish to read this survey paper [18]. Table 2 in [112] compares different programmable data planes. Some techniques also use SDN for network and application awareness [77].

2.3.1 SDN Control Plane

SDN's logically-centralized control plane means deploying one or more controller services that configure the data plane switches. When a switch finds a packet with no matching rule, it forwards it to the controller. The controller determines how to handle the packet, installs the appropriate rules in the data plane, and forwards the packet back to the appropriate switch. SDN controllers have evolved quite a lot over the years (see [18]), with one of the most recent and more mature options being ONOS [2]. ONOS supports clustering controller instances as well as deploying apps across them.

A single-controller architecture is much simpler, but also represents a single point of failure as well as a performance bottleneck. To successfully distribute multiple controller instances, the global network state gathered from the data plane must by synchronized between them. The authors of [121] explore these state distribution trade-offs within the context of a load balancer application. They found "that SDN control state inconsistency significantly degrades performance of logically centralized control applications agnostic to the underlying state distribution". Hence, SDN presents a tradeoff of managing state distribution for added network intelligence.

The logically-centralized control plane presents a possible single point of failure if not physically-distributed. SDN controllers are typically clustered for scalability and fault-tolerance. However, co-locating them in one part of the network makes it easier for a few link failures to isolate data plane elements from all controller instances. Therefore, the *controller placement problem* has been explored in the literature [101, 68]. It aims to place controller instances in locations that minimize the impact of potential failures.

Other control plane challenges include managing device mobility in distributed controller networks [75]. To handle large volumes of previously-unmatched packets, [152] proposes a "prioritized service model (PSM)" to handle unmatched newcoming packets with higher priority so that they experience delay closer to that of already-matched flows.

Network hypervisors: in addition to the direct data plane programmability offered by SDN controllers, researchers explored a virtualization concept similar to that of the virtual machine OS. A network hypervisor [169, 12] is an SDN controller that manages the physical network but exposes virtual abstractions of that network in the form of e.g. *virtual slices.* Such virtual network tenancy may prove invaluable for rapidly-deployed IoT systems. Network hypervisors must map virtual switch abstractions to physical switches as shown in Fig. 2.3. This process, called *virtual network embedding (VNE)*, may assign one virtual switch to one physical switch without the tenant SDN controller knowing which physical

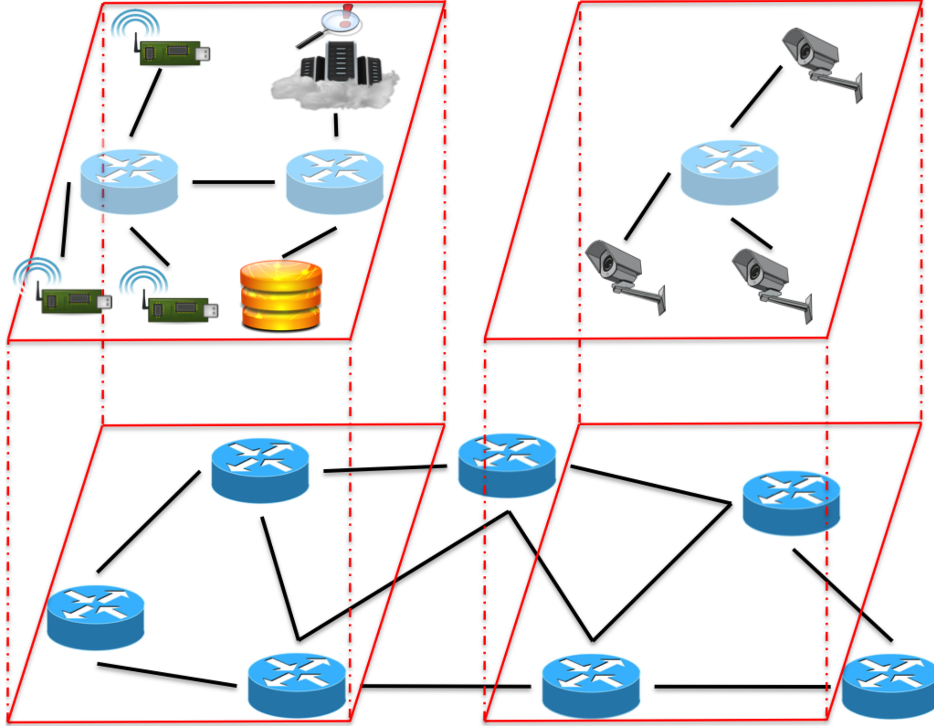


Figure 2.3: A Virtual Network Embedding (VNE) maps SDN software (i.e. virtual) switches to physical infrastructure, thereby abstracting it from IoT devices and services.

switch it is managing. VNE can also map a virtual switch to many physical ones and even multiple virtual switches to a single physical one (i.e. by *slicing* the flow table rules). These abstractions can even support additional services such as automatic fail-over or virtual links (i.e. tunnels) that implement our proposed overlay routing techniques. In this thesis, we assume either direct control over the data plane switches (Chap. 6) or a "big switch" abstraction that encompasses the entire physical network in a single virtual switch (Chap. 7). Hence, we leave further discussion about network virtualization as out of scope and instead refer interested readers to recent survey papers on network hypervisors [35] and VNE [84].

2.3.2 SDN for IoT

SDN helps manage networking for IoT deployments by offloading configuration logic from constrained devices. While many recent preliminary works have proposed the application of

SDN to IoT deployments, few substantial works with in-depth studies and implementations have been published. Some of those of particular note that have include: using P4 [37] to rewrite packets in order to bridge multiple protocols without cloud involvement [185]; Network Calculus and Genetic Algorithms to manage SDN flows and guarantee QoS in heterogeneous networks [69]; managing wireless IoT sensor networks [60]; leveraging a Value of Information metric to configure information processing entities that make up IoT applications at the edge [74]; managing IoT pub/sub protocols as described in §2.3.3. Although it does not consider real-time (i.e. delay-sensitive) management, [196] uses ONOS "traffic steering intents" to control flows between Kafka brokers in an IoT setting for resilience to link errors or traffic congestion.

2.3.3 SDN for Pub/sub

Multiple recent works applied SDN to enable high performance pub/sub through network-level multicast. This started in [110] to provide line-rate pub/sub. Later works investigated: using network header primitives (e.g. IPv6 address) for establishing content-based routing tables with application-layer filtering [33] bandwidth-efficient content based routing [32]; dynamic subscription/advertisement changes [73]; load balancing [11]. Furthermore, [189] investigated differentiating pub/sub subscriptions at the network level and prioritizing them separately to provide bounded queueing delays. Many other research works investigated coupling DDS with SDN for better control of the data exchange process [91, 92, 49, 31]. POSEIDON [92] aims to support different underlying pub/sub protocols, but it requires software agents running on the SDN switches.

2.3.4 SDN for Resilience

Resilience research in SDN predominately focuses on quickly recovering from failures. One mechanism, *fast fail-over* [63], uses predefined backup paths and recovers by changing to an alternate path computed as described in §2.2.2. Some works consider how to compute these paths in SDN-specific settings such as [157], which uses machine learning to forward packets along the best of several available paths. Other consider network-wide resilient management [173] or leveraging redundant routes for IoT devices [162]. In [53], partial-protection of paths is shown to reduce the cost of maintaining backup paths vs. a full-protection scheme. Some research considers both QoS and reliability simultaneously. For example, [54] provisions multiple paths for QoS and fault-tolerance in multimedia networking. The authors of [122] take QoS metrics into account during the process of computing redundant routes, whereas [46] considers reliability as a QoS requirement.

2.3.5 QoS via SDN

Some recent works address prioritization in SDN, although they do not operate at the fine level of granularity that our approach described in Chapter 7 does. These works include: delay guarantees for real-time systems [118]; studies on real SDN switch performance when applying prioritization to TCP flows [59]; SDN & priority queues to enhance improvement of query plans in distributed SQL store [193]. Real-time flows are considered in [15] by assigning them higher priority than the delay-tolerant ones.

While we do not directly address Quality of Service (QoS) guarantees in this thesis, this is an area of extensive research within the SDN community. Examples include: QoS guarantees of bandwidth [116]; adaptive QoS-aware routing in hybrid networks (i.e. containing some non-SDN switches) using a simulated annealing approach [124]; a network calculus-based language for expressing QoS parameters [165]; an extensive comparison on multiple

algorithms for setting up performance-guaranteed traffic tunnels in backbones [184] [42] considers QoS in home networks in terms of 4 different application types: web, file transfer, voice, and video.

2.4 Edge Computing

A common design paradigm for the Internet architecture as well as protocols/applications that rely on it is that of pushing intelligence as close to the **edge** as possible. Not only does this allow the network core to focus specifically on high-speed packet routing by limiting the amount of logic those devices contain, it also allows for a hierarchical design methodology in terms of deploying heterogeneous devices to handle different tasks within a network. For example, customer devices run TCP in order to ensure reliable delivery of data (if possible) through the best-effort Internet service. Gateway routers at the edges of networks handle these customer devices' traffic and may send packets through different flows in the network to offer a particular quality-of-service or shape certain traffic types (e.g. p2p applications). In enterprise networks, middleboxes offer services such as firewalls or caches that would overly complicate finely-engineered routers.

This hierarchical design also allows these networks to scale in an efficient and cost-effective manner. Routers at the edges of the network aggregate data flows from many customer devices and forward them along a fewer number of links into the core of the network. The core routers, which are much fewer in number, will then forward this traffic along high-speed long-haul links to other core routers and back up this hierarchy to the destination customer devices. In this manner, the more complex logic to handle applications and new network services resides in customer devices. With the advent of IoT systems, designers often disrupt this paradigm by further extending the hierarchy as IoT devices are less capable of supporting more complex logic. They often deploy gateway nodes or basestations to collect traffic from

sensor devices (or send commands to devices) and determine how to handle it (e.g. send to cloud or process immediately). These nodes offer opportune points at which to add software intelligence at the network's edge. This can complement the Internet's resilience mechanisms in order to address the aforementioned problems, a concept we will return to in Section 5.

At the IoT data and application layers, we observe a trend towards decentralization through processing and storing data on edge devices and gateways (e.g. Apple HomeKit, Nest Weave, and research projects [62, 117, 70]). Mesh networking solutions [17, 81] avoid the cloud and enable such approaches via direct communications between IoT devices. This edge approach also potentially improves QoS [164, 127] and performance over cloud-only architectures by selectively performing computations or other services on edge resources. For example, [70] decomposes an application into tasks and selectively runs them on either the client or a 2-tiered cloud architecture. We also see investigations into deploying middlebox-like services for in-network data exchange. For example, [194] runs MQTT brokers within the edge network infrastructure itself for improved message forwarding.

Smart buildings are a special case of edge computing in IoT systems because they represent dedicated infrastructure that can be easily exploited as edge resources. Heterogeneous 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 [200, 160, 199, 21]. In existing structures, this complexity is handled by a BMS that locally manages devices and data. Recent work on smart buildings includes ontologies [19] and protocols to support building automation [44], techniques to preserve privacy [132], enabling energy efficiency [190], programmable building operating system services [55], context aware IoT management via SDN [111], and accurate positioning for location based apps [47].

Chapter 3

Our Proposed Middleware Approach to Resilient IoT Data Exchange

We now provide a high-level overview of our proposed approach. We first motivate the mission-critical IoT application scenarios that we use to derive challenges and design requirements. Then we present our approach, its merits, system architecture, and design decisions. We discuss our cross-layer middleware within the context of deriving and leveraging application and network awareness, edge vs. cloud data exchange operation, and configuring IoT edge networks through SDN. We explore our approach in the context of three different projects inspired by and built on top of the initial SCALE efforts. Each focuses on slightly different scenarios, data exchange goals, approaches to resilience, system scales, available state information, degrees of edge involvement, and SDN abstractions. This chapter compares and constrasts the projects that compose the remaining chapters along these dimensions to paint the picture of our approach's overall capabilities.

3.1 IoT in Mission-critical Settings

With the increasing availability and decreasing cost of microelectromechanical systems sensors, several projects have begun exploring the use of these devices in Internet-connected distributed sensing efforts. As IoT deployments find their way into more aspects of our lives, we will come to rely on them for improving how we address emergency scenarios. Examples of such smartspaces include personal spaces such as homes, offices, senior facilities; critical infrastructures such as airports, shipping, and port facilities; organizations such as schools and hospitals. Applications range from surveillance and security to personal safety and situational awareness in emergency response scenarios. While IoT devices are often deployed for a dedicated purpose, such as monitoring critical infrastructures, their data may be repurposed and exploited for new applications if access to this information are made available through open APIs. We now detail a few examples of such mission-critical applications and their challenges presented to IoT data exchange and communications networks.

Such sensor networks could potentially detect these events' onset and warn possibly affected individuals to find shelter, as well as aid first responders through increased situational awareness. However, network failures can severely hamper these networks' ability to gather useful information in a timely manner, especially important for those aimed at monitoring fast-moving destructive physical phenomena such as earthquakes and floods. Such events often result in large-scale geographically correlated failures in addition to serious network congestion as individuals contact each other or request help, exacerbating failures or tying up channels entirely.

3.1.1 Home & Community Safety

As we move beyond purpose-built building/home security systems, we enter an age of solutions deployable and customizable by the building manager/homeowner. Such deployments may leverage other devices in the vicinity to improve outcomes. For example, alarms raised by detection of an emergency may trigger flashing lights or automatically connect a resident with emergency personnel via video call. Several other works in addition to our SCALE project, which explores the question of deploying in-home safety solutions with off-the-shelf hardware as detailed in §4, explore in-home and community safety scenarios. For example, [41] uses a phone's accelerometer to detect when the owner carrying it has fallen down. The authors of [13] expand on this scenario by considering which specific IoT devices to activate in order to perpetually monitor a smart space for possible personal emergencies while considering energy constraints of battery-powered devices. At the wider community scale, IoT device deployments can be exploited to detect emergencies outside the home. For example, [87] uses audio signals (e.g. from ambient microphones deployed in a city) to accurately detect gunshots. Other such devices, whose information could help understand regional impact of phenomena, include weather stations, pollution detectors, and geiger counters [5]. Such applications require a high degree of certainty in the results. As such, they should ensure that data of potential interest is not lost and is delivered as soon as possible. Such a requirement is especially challenging in home environments due to the typical lack of expertise in persons that deploy the IoT infrastructure. For more wide-scale and public deployments, we propose multiple techniques in this thesis that improve communications resilience.

3.1.2 Earthquake Detection and Alerting

We now present an IoT deployment scenario for earthquake-detection and emergency response derived from our ongoing IoT projects and collaborative deployments. We model this

use case after two different volunteer seismic sensing network projects based in California. The Community Seismic Network (CSN) [7, 50] and Quake-Catcher Network (QCN) [51] deploy small low-cost accelerometers in homes, businesses, and schools to monitor seismic activity. These devices report such activity to a cloud service that quickly and accurately detects and characterizes earthquakes. By distributing such situational awareness (i.e. in the form of mass alerts), affected individuals can then take protective action (e.g. "duck-hold-and-cover") while first responders assess damage, coordinate efforts, and direct evacuations.

While some of our earliest work [24, 25] was with QCN, we only present the specifics of CSN within this discussion. Volunteers provide space and power for these sensors, which may be connected to a desktop computer or a dedicated low-power computer, such as a plug computer or Raspberry Pi. When they detect abnormal ground motion, indicative of a possible seismic event, these hosts report messages called *picks* to a cloud server (e.g. Google AppEngine) that processes the information and determines if an earthquake has occurred. This server sits outside the earthquake-prone region from where it safely collects and analyzes recent *picks* for possible earthquakes. By effectively identifying and categorizing earthquakes in a timely manner, the CSN system, in conjunction with traditional seismographs, could enable a real-time fine-grained targeted early warning system and provide people precious seconds to take shelter before a seismic wave propagates to their location. We envision such a deployment expanding in the future as more individuals deploy IoT sensor devices in their homes that often incorporate accelerometers (e.g. home security, smart-phones), which could contribute shaking measurements to the CSN system.

However, earthquake-induced damage can cause communications disruptions (congestion, failure) and cloud connectivity instabilities as discussed earlier. This may result in lost or delayed sensor data captured during and immediately after an earthquake. The most heavily-impacted regions suffer the highest data losses but also most need timely reliable alerts for protecting life and property. To support an early warning system despite severe congestion

and packet loss (i.e. due to sudden traffic spikes from people contacting each other as well as failures caused by the tremor), the *picks* must arrive at the server for analysis within a few seconds of the event. Such failures will often be localized due to the geographic scope of an earthquake as well as cascading failures introduced by e.g. failures in the power grid.

3.1.3 Smart Fire Fighting

In 2015, the US National Institute of Standards and Technology (NIST) and National Fire Protection Association (NFPA) published the Research Roadmap for Smart Fire Fighting (Res. Roadmap) [94] that "establishes the scientific and technical basis" of Smart Fire Fighting (SFF). This interdisciplinary collaboration brought together members of industry, government, the fire science practice, and academia to identify technologies and key challenges in SFF. In response, we initiated a project to address some of these challenges (see Fig. 3.1) with distributed middleware solutions for data management.

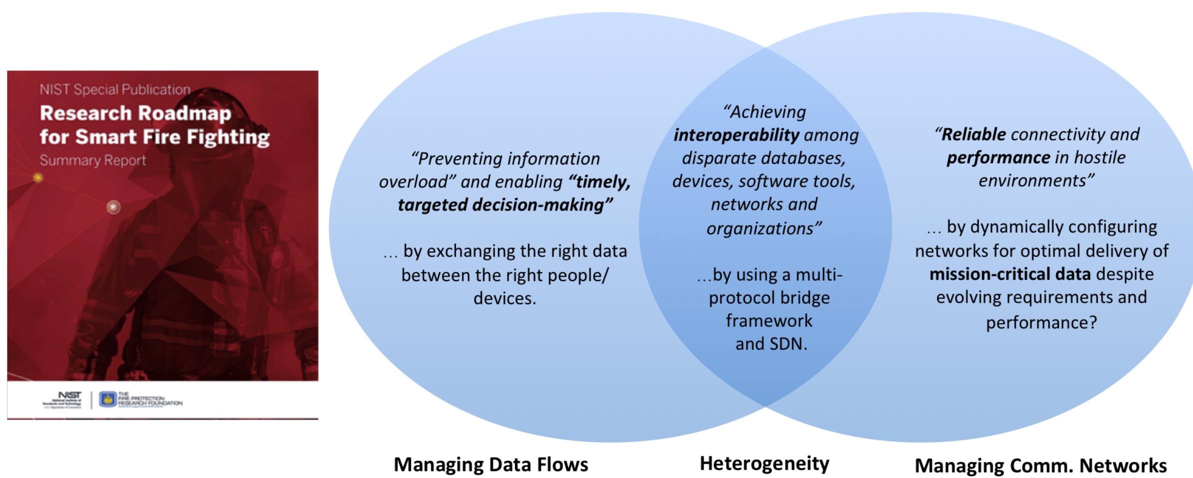


Figure 3.1: The NIST/NFPA Research Roadmap for Smart Fire Fighting [94] identified several key challenges that guide our explorations in the FireDeX project.

During a fire, an occupant or automated system activates the emergency dispatch process depicted in Fig. 3.2. A local fire department(s) responds by sending a team of fire fighters (FFs). An Incident Commander (IC) coordinates 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. Today the IC still derives much of this information from non-digital sources (e.g. human-initiated reports via voice and/or FF radio, paper records) and collates it in an ad-hoc manner to guide operations. However, smartspaces equipped with IoT devices enable access to live data feeds that can generate actionable information in real-time through proper filtering, prioritizing, and analysis. Such a data-driven approach to improving FF outcomes fuses sensor data producers with consumers (e.g. analytics, IC) and actions (e.g. device actuations, alerts) as shown in Fig. 3.3. This improves the efficacy of the classical fire response workflow, thereby driving the emergence of *SFF* technologies.

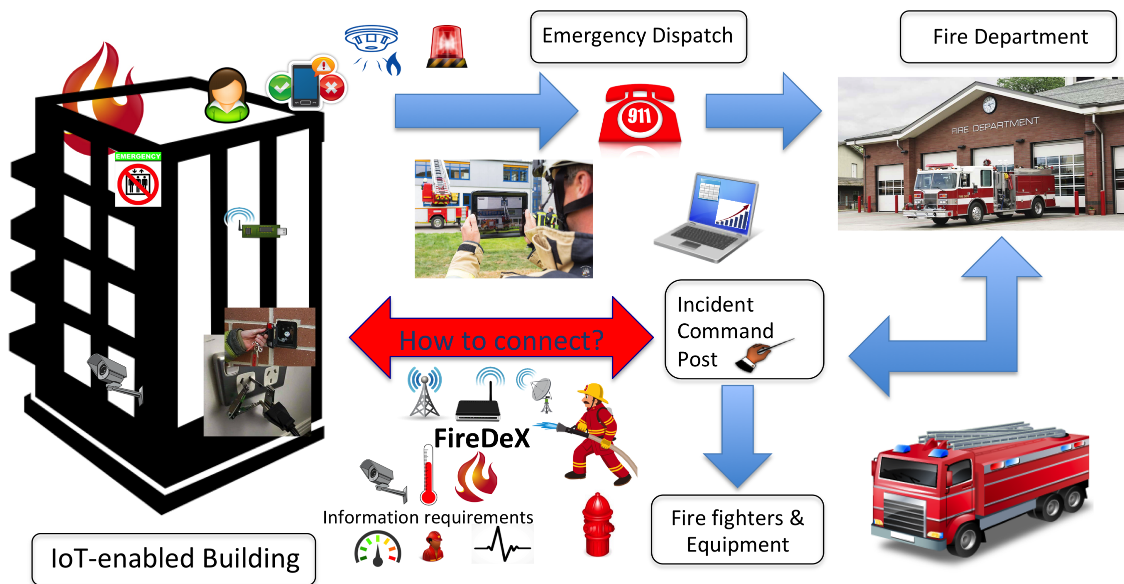


Figure 3.2: To address our target scenario of fire response in an IoT-enabled structure, GeoCRON addresses fundamental challenges in exchanging data between the FFs and the IoT-enabled building infrastructure.

Maintaining up-to-date situational awareness for 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. This data may be served by off-site cloud data exchange. For example, an Emergency Operations Center (EOC) serves a particular region's emergency response

services with inter-agency coordination as well as cloud computing infrastructure to support SFF services. An EOC may even 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 subscribers (e.g. analytics services) and combined with contextual knowledge to generate situational awareness. Such information includes: motion sensing, location/occupancy/activity tracking, smoke levels, air flow rate, audiovisual feeds, etc. Different data types vary in importance to responders depending on the situation (e.g. "smoke" > "water pressure" when initially sizing up the event and vice versa during active fire suppression).

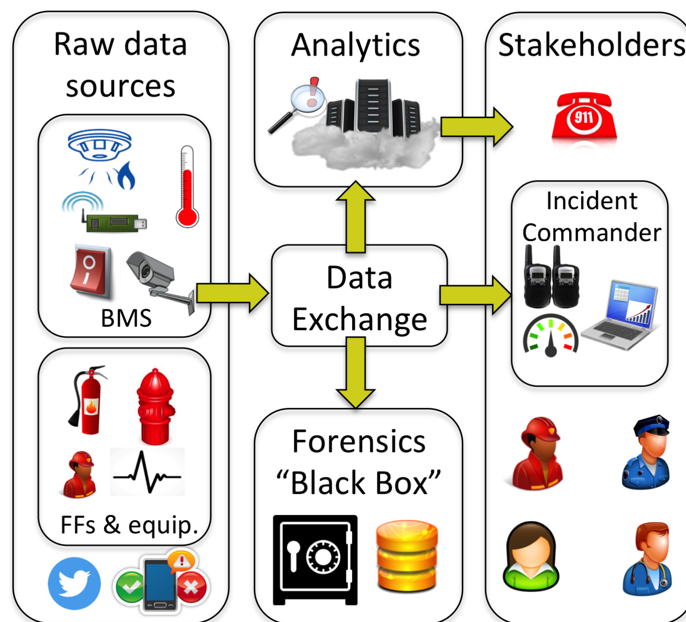


Figure 3.3: To address our target scenario of fire response in an IoT-enabled structure, GeoCRON addresses fundamental challenges in exchanging data between the FFs and the IoT-enabled building infrastructure.

Stakeholders (e.g. IC, FFs, residents) request, visualize, and act on data as shown in Fig. 3.3. An IC may use a tablet (or similar device) running a situational awareness dashboard to monitor the situation and coordinate the response effort. The IC assesses building occupancy for coordinating evacuations, tracks the locations and biometrics of FFs to ensure their safety and effectiveness, and detects environmental hazards within the building such as

high temperature or smoke levels. FFs may use some less-intrusive interface (e.g. a heads-up display (HUD) on their mask or glasses) to receive similar data or non-voice commands from the IC. A key challenge for SFF is delivering mission-critical data for "timely, targeted decision making" in an unreliable, partially available, and congested network environment [94]. Given the heterogeneous value of events and limited resources for delivering notifications, we believe event prioritization is necessary in such mission-critical settings. To this end, we propose the FireDeX data exchange middleware to manage the flow of situational awareness information. It assigns and enforces event priorities according to the requirements and capabilities of three system layers: application, data exchange, network.

Some other works have addressed some SFF-specific challenges. One investigates equipment for dropping sensor devices to support indoor localization and tracking of FFs, which helps ensure their safety if they become lost inside a smokey building [126]. Another considers edge computing for video transmission that provides increased situational awareness [192]. A related scenario that we leave outside the scope of this thesis is a shanty fire scenario. Different from the above SFF structure fire scenario, a shanty town has little planned infrastructure and poses many hazards such as highly-flammable structures and chemicals. Our previous work in this area investigated path planning and mobile ad-hoc networking techniques for mobile data collectors (e.g. volunteers on bikes) to gather and report situational awareness in support of response efforts.

3.2 Resilient Data Exchange Goals

In support of the above mission-critical IoT scenarios, we identify data exchange requirements that our approach targets throughout this thesis. Such applications require resilient timely data exchange to make use of relevant information in driving situational awareness and thereby effective response. As previously discussed, our solution cannot extensively modify

or complicate constrained IoT devices. Furthermore, it should not completely rely on cloud platforms in case of cloud connectivity instability.

The the Safe Community Awareness and Alerting Network (SCALE) project targeted the in-home safety scenario (see §3.1.1) at the community-scale. As a prototype demonstration effort, we initially aimed to quickly build a working data exchange that could support publishing events and subscribing to notifications about them as described in §1.2. SCALE's evolution from a small demonstration project to multiple deployments spanning several application domains and continents presented new requirements. IoT deployments lasting months to years must execute reliably over time, with minimal administrative intervention, and under changing connectivity and device conditions. See §4.3 for more details about the challenges encountered during the SCALE project that led us to further explore resilience as described in the following chapters.

The GeoCRON project (Chap. 5) arose from our effort to target the community-scale seismic monitoring setting described in §3.1.2. It aims to collect data from IoT seismic sensors spread throughout this wide-area community. However, a major earthquake may cause large-scale geo-correlated failures within this region. Therefore, we aim to address the challenge of collecting this data despite network infrastructure failures that lead to cloud connectivity disruptions. Furthermore, we explore how to accomplish this without direct control over the underlying public Internet infrastructure.

Ride (Chap. 6) expands on the seismic scenario to include earthquake detection and early warning within a campus setting. As such, we relax the preceding assumption about control over the underlying infrastructure. Instead, we explore added control over the *edge network infrastructure*. Hence, we move the resilient communications logic from the IoT devices themselves (as we did in GeoCRON) to the network itself through the use of SDN. We explore resilience to cloud connection disruptions through one of Ride's primary concerns: whether information should be collected and processed in the edge data center vs. in the

cloud. Furthermore, we explore how to leverage SDN capabilities at the edge to improve local data dissemination (i.e. alerting). However, we don't explore the wide-area geo-correlated failures within that chapter because infrastructure failures would be experienced more or less homogeneously across a local campus region. Lastly, Ride explores how to accomplish these goals without introducing significant overhead and consuming additional resources.

FireDeX (Chap. 7) balances the needs of multiple mission-critical applications and data consumers. It targets the smart fire fighting setting described in §3.1.3. It aims to manage data exchange flows within a smart building during an active fire response. Such a dynamic setting introduces challenges of heterogeneous information: what data must be delivered to which consumers. Hence, FireDeX also considers the relative importance of different data types to different data consumers. This large volume of data further complicates this scenario by introducing resource constraints such as limited network bandwidth. When network conditions deteriorate and cannot support every outstanding request, it addresses the challenge of which data should be delivered first. This ensures that subscribers can make use of the best available information in a timely manner and maintain a high degree of situational awareness.

3.3 Our Application-and-Network-Aware Resilient Communications Middleware

To address the challenges outlined previously, we propose a cross-layer middleware solution to resilient IoT data exchange. It builds on the classic IoT deployment approach of cloud-centric data exchange discussed in §1.2 and applied in the original SCALE project (see §4.3.2). Therefore, this *integration middleware* combines together several other middleware solutions in order to bridge the three layers shown in Fig. 3.4. Bridging these layers enables

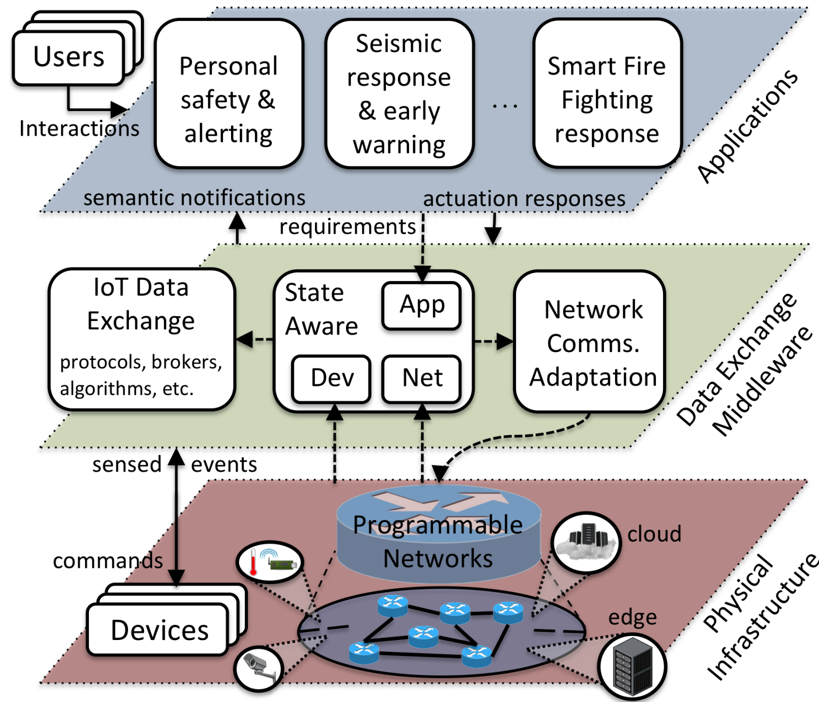


Figure 3.4: We propose a cross-layer middleware approach to help bridge the semantic gaps between the data and infrastructure layers.

our middleware to leverage application and network awareness to improve communications resilience.

Resilience techniques to deal with limitations/disruptions in community IoT networking infrastructures can be addressed at two different levels: a lower *connectivity* level (how to deliver reliably) and a higher *messaging* level (what to deliver for improved utility). GeoCRON and Ride focus on the former by exploiting redundant paths and edge resources to ensure communication of critical safety data to the backend cloud where it is stored and analyzed. FireDeX focuses on the latter by prioritizing messages in a manner that maximizes the overall value of information gathered.

Connectivity level resilience requires awareness and control over the underlying network infrastructure whereas messaging level resilience requires some amount of interaction with the data exchange layer. Our proposed middleware gathers network state and application semantics (i.e. resilience requirements) to leverage when adapting information flow in the

IoT system. By configuring the data exchange middleware and underlying networking infrastructure according to this cross-layer awareness, this unified end-to-end approach bridges semantic gaps between the information and infrastructure layers. Much of the network state information is collected by the SDN controller directly from the data plane switches/routers and made available to our middleware through various SDN APIs.

GeoCRON ensures more reliable collection of data at a cloud service despite geo-correlated infrastructure failures throughout the sensed region. It achieves reliable message delivery by exploiting geographically-redundant network routes to *avoid failures*. It extends the notion of Resilient Overlay Network (RON) [16], which routes around failures and congestion in the wired Internet. Our proposed *Geographically-Correlated Resilient Overlay Network (GeoCRON)* [29, 28] gathers the underlying routing infrastructure topology and locations of both IoT nodes in the overlay and routers in the underlay. It uses this information to establish multiple geo-diverse routes in the overlay according to the number requested by an application's resilience requirements. Sending multiple copies of each message along these geo-diverse routes improves the chances of successful message delivery (to the cloud data exchange) during large-scale geo-correlated failures.

Ride expands on the seismic scenario by also considering alert dissemination after the collection and consumption of relevant events. It extends existing IoT data exchange solutions to more resiliently collect, process, and disseminate events of critical interest to humans (e.g. disaster alerts). Ride achieves this goal by determining whether to collect and process data in the cloud or at the edge. It approaches reliable message delivery through *failure recovery* rather than avoidance and, in the event of a complete cloud outage, it leverages edge resources to maintain continuous data exchange operations. Hence, it prefers the cloud whenever possible (i.e. for more high-performance regional IoT services), but fails over to the edge when it becomes unavailable. Ride pre-configures data flows for reliable operation and dynamically responds to evolving network conditions (e.g. failures, traffic spikes). It borrows

from GeoCRON the approach of establishing multiple geo-diverse overlay links to the cloud data exchange. However, it also monitors these virtual cloud links through a novel resource-conserving adaptive network probing technique. IoT applications register with Ride and provide resilience requirements for this monitoring service (e.g. detection time, false positive rate, etc.). The monitoring service gathers cloud data path link information (e.g. round-trip-time, loss rate, jitter). During deteriorated conditions, Ride re-routes IoT data flows through an alternative path. When no such path is available, it re-routes them to a backup edge service until one of the cloud data paths recovers. This allows seamless operation under both normal and failure conditions. Ride further pre-configures multiple disjoint local multicast-based alert dissemination paths for edge-mode operation. This approach decreases network resource consumption by avoiding packet duplication during local alerting. It builds these multicast trees according to the locations of subscribers registered for resilient alert topics and the alerting application's requested degree of resilience (i.e. number of disjoint multicast trees). Ride gathers local network topology information from the SDN controller and local link status estimated by the data exchange service according to the routes traversed during the data collection phase. Its novel path-selection scheme leverages this potentially imprecise network state awareness.

FireDeX addresses resilience at the messaging level by ensuring delivery of the most important events (according to subscriber requirements) despite resource constraints. It essentially *prioritizes messages* and *allocates available network resources*. It collects information requirements directly from the subscribers in the form of subscriptions: an event topic and an associated utility function that quantifies the value of information captured given a rate of successful message delivery. It also collects network state information (i.e. latency, available bandwidth, and error rates) collected through SDN APIs. The FireDeX algorithms combine this cross-layer awareness and a queueing theory-based analytical model to configure the system for optimal performance. They first assign priorities to subscriptions and then allocate bandwidth to the network flows serving these subscriptions through pre-emptive packet

drop rates. These drop rates prevent the delay or loss of important data due network buffers filling up.

Altogether, these approaches contribute to our vision of a resilient cross-layer communications middleware for IoT data exchange that leverages existing technologies/software whenever possible. Furthermore, such an integration middleware solution has several advantages over a top-to-bottom custom solution:

- **Global optimization** requires a unified framework that simultaneously considers state and requirements of each layer. This provides a more holistic view than each individual solution isolated at its layer would. For example, we can bridge the pub/sub subscription requirements with network state awareness for better configuration of both layers.
- **Implementation agnostic** design supports various implementations of each composed middleware solution. For example, our solution is agnostic to which MQTT broker is used as long as it properly supports the MQTT protocol. New solutions can easily be swapped in to provide additional features without having to re-implement them in our core middleware code.
- **Lightweight client middleware** adds minimal complexity to the IoT devices themselves. Most necessary services and complex algorithms run in the cloud/edge and pass configurations to the lightweight client middleware when necessary.
- **Heterogeneous** devices, platforms, networks, and physical resource constraints can be better abstracted by a unified framework. The specific underlying technologies can be abstracted by the composed middleware services e.g. SDN OpenFlow abstracts the physical network technologies in use.
- **Multi-protocol support** is especially important for IoT ecosystems in which highly-constrained devices require specific protocols with minimal feature sets. Protocol bridging can be easily accomplished at the middleware layer without major changes to the

software (i.e. through plug-ins). For example, we could support many different data exchange protocols (e.g. MQTT and CoAP) without modifying the clients to implement this support.

- **Cascading failures** further hamper resilient operation. A more holistic view can detect and mitigate potential problems at one layer that might affect another.

3.4 System Architectures & Middleware Design

This thesis explores the proposed approach to resilient communications for IoT data exchange in the various scenarios described in §3.1 and shown in Fig. 3.6. These scenarios determine different scales, designs, and resilience challenges in system architectures. The SCALE project (see Chap. 4) targeted the in-home safety scenario (see §3.1.1) at the community-scale. Because it was a demonstration project, we were more concerned with quickly assembling loosely-coupled services supported by a central cloud-based data exchange broker. As such, we learned a lot about IoT deployment resilience through both the initial Smart America Challenge (first half 2014) and subsquent experience leveraging SCALE as a prototype IoT testbed, i.e. a classic IoT deployment. Through this we identified several of the key research challenges that we consider in §1.3.3.

Over several years following the initial SCALE project, our research group extended SCALE to target different application scenarios including water infrastructure [96], in-home senior care [13], and sensor data upload planning for mobile sensing [201]. Each of these scenarios brought heterogeneous requirements that led to different different deployment scales, system design considerations, and proposed research approaches to improving the IoT deployments' resilience. While we leave the specifics of these projects out of scope and instead refer interested readers to the respective publications, this thesis focuses on the earthquake detection/alerting and smart fire fighting scenarios described in §3.1.2 and §3.1.3 respectively.

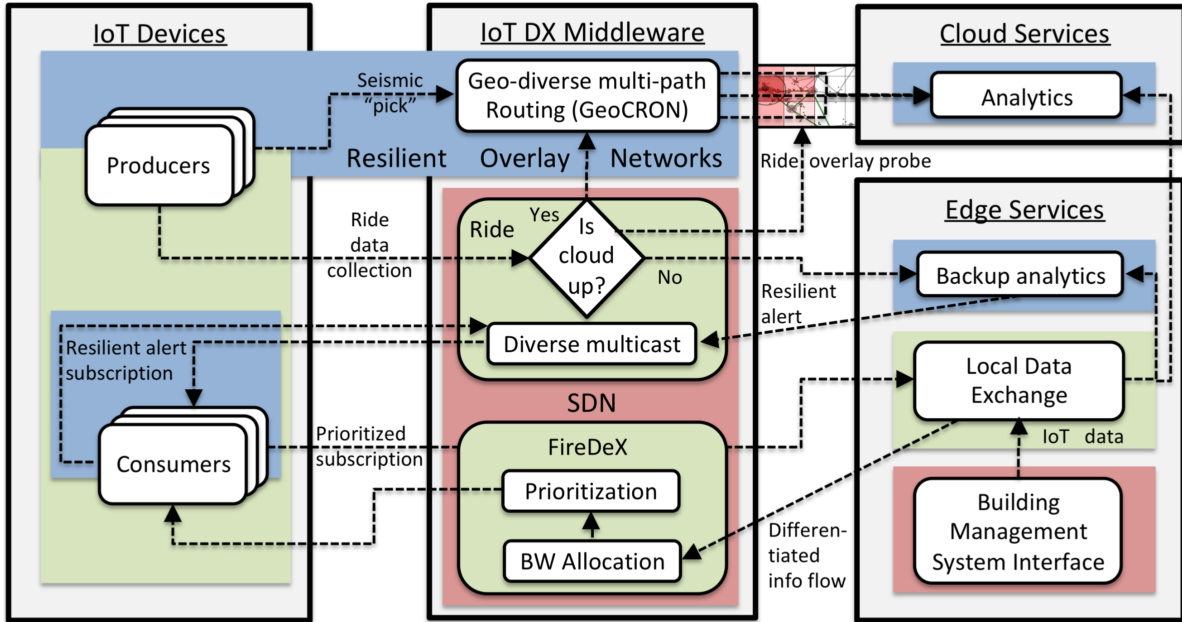


Figure 3.5: Our proposed middleware minimizes intelligence added to devices and instead concentrates it in the network and at the edge.

Fig. 3.5 depicts the system architecture components of our proposed middleware. It maintains the IoT thin client design by minimizing the intelligence added to devices and instead concentrates it in the network and at the edge. Altogether this middleware bridges producers and consumers with decoupled applications around a loosely-coupled data exchange fabric. The remainder of this thesis explores the techniques proposed to address resilience challenges in each scenario. We also discuss our prototype implementations of each project and how we incorporate additional logic into either the IoT devices themselves, the network, or in a controller service running at the cloud and/or edge.

GeoCRON aims to collect data from IoT sensors spread throughout a community-scale deployment despite regional geo-correlated failures. When we first consider the cloud-centric earthquake detection scenario, we assume little control over the underlying public Internet infrastructure due to the IoT devices' spread over a large geographic area. Instead, we essentially consider the IoT producer devices as edge resources to which we add a thin middleware logic that facilitates an application-layer geo-aware overlay network. We centrally configure

this overlay through a data exchange middleware service residing in the cloud (not pictured). Incorporating the geo-diverse multi-path routing techniques into SDN abstractions provided by network service providers would further remove a lot of the producer devices' logic necessary for operating the overlay networks.

Ride expands on the seismic scenario by considering a more local region (i.e. campus area) in Ride. Hence we incorporate both cloud connectivity as well as edge services: local compute for backup analytics and SDN management abstractions for the smart campus network. As such, we view this setting as one small region within the larger community-scale setting that GeoCRON targeted. Ride closes the IoT application loop by facilitating end-to-end resilient data collection, backup analytics, and alert dissemination. It extends the overlay path approach to maintaining cloud connectivity by monitoring and adapting the overlay paths. However, it does so by coordinating between the SDN infrastructure and a networking probing service that runs on both the edge and cloud. It sends probes from the edge through the overlay path(s) to the cloud service, which in turn responds along the same route. The resilient alerting mechanism leverages network-layer multicast and as such does not require modifications to the consumers. Rather, it coordinates between the local data exchange and SDN to configure the redundant multicast trees for any registered resilient alert subscriptions.

FireDeX considers balancing the needs of multiple mission-critical applications and data consumers during the smaller-scale smart fire fighting scenario. It accomplishes this by prioritizing events within a smart building during an active fire response, which ensures delivery of the currently most-critical data first. While we assume we have control over the building-area network through SDN, we do not directly consider the building network topology and instead rely on a higher-level abstraction as described below. This requires the consumers to incorporate additional subscription logic that coordinates with a FireDeX service. They communicate the relative importance of their subscriptions to each other, which FireDeX then uses to compute optimal bandwidth allocations and priority queueing

disciplines for each active subscription. These are enforced at the network-level (i.e. in the SDN switches) so as to be generalized across a transport-layer protocol rather than requiring data exchange (i.e. application)-layer support/modifications. For example, AMQP supports prioritization, but MQTT does not. Hence, we essentially extend MQTT (really MQTT-SN) to add priority queueing disciplines.

Note the evolution of the role edge services play in our architecture as the chapters progress. We start with only IoT devices and cloud services, move to an edge-centric exploration of resilient IoT data exchange, and then consider how to coordinate the needs of various consumers at the edge. This determination of edge vs. cloud is a key part of our approach as well as the above middleware architecture that implements it.

3.5 Leveraging Edge Resources

In this thesis, we argue that a data exchange solution exploiting **edge infrastructure** can enhance localized situational awareness, system outcomes, and event responses (especially in the absence of stable cloud connectivity). IoT edge computing further exploits the fact that events generated in the physical world, as well as consumer interest in them, exhibit spatiotemporal correlations and locality. For example, users in the vicinity of an emergency are interested in alerts and notifications that enable them to take protective action; nearby actuators (e.g. sirens, elevators) should automatically respond to the event even in the absence of stable cloud connectivity.

Fig. 3.5 depicts how we implement our approach to exploiting edge resources and where we locate software artifacts that implement this logic. Because IoT deployments would not typically have direct control over the public Internet, they can take further advantage of edge resources by carefully configuring them along with the local network. This can provide

further guarantees such as QoS or fine-grained routing control. Therefore, we claim that such deployments should exploit both cloud and locally-managed *edge computing* solutions whenever possible. This improves their ability to capture and leverage application and network awareness for dynamically configuring the data exchange in support of mission-critical applications.

A crucial question when edge resources are involved is whether data processing should happen in the cloud or at the edge. Recall that traditional IoT approaches forward all data to the cloud. We essentially assume this approach in the GeoCRON project where the IoT devices themselves are the only exploitable edge resources. However, the seismic data must be aggregated centrally to make use of it and so we only consider uploading this information to the cloud. The edge devices instead provide the networking infrastructure necessary to establish a resilient overlay for more reliable data collection. We first address this edge vs. cloud issue in the Ride project. Not only do we use edge resources as a backup during cloud unavailability, but we exploit the SDN-enabled control over the local edge network to improve local alerting through our novel resilient multicast mechanism. While the FireDeX model allows a distributed broker network, we focus on the local edge setting. In that case, we abstract away whether data exchange and processing happens in the edge vs. cloud. Instead, we focus on improving the performance and utility of the overall data exchange process. Note that the generic model adopted in FireDeX allows for cloud data exchanges, sources, and consumers. Their presence in the cloud would simply change the expected network performance characteristics.

3.6 SDN for Flexible IoT Edge Network Control

We leverage **Software-Defined Networking (SDN)** to gather awareness of and effect control over the underlying networking infrastructure. As explained in depth in §2.3, the

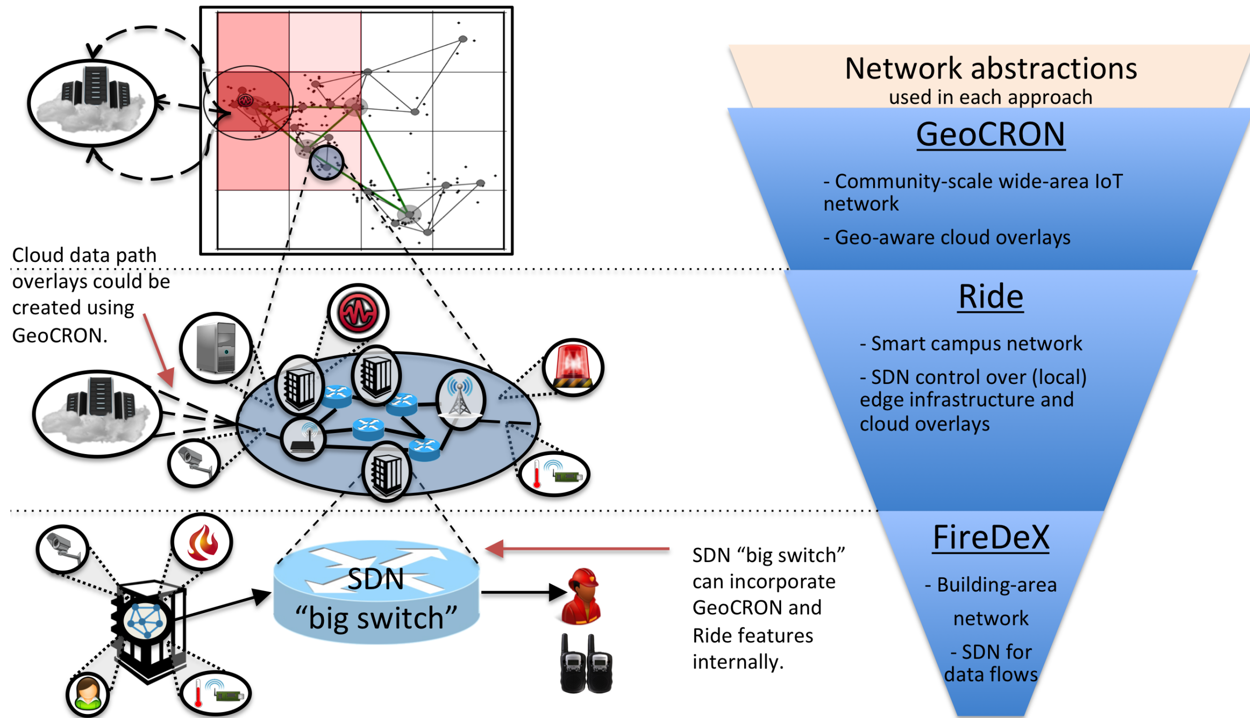


Figure 3.6: As we progress in the chapters, we explore more localized (i.e. edge) approaches and consequently more abstract SDN representations of the network infrastructure.

SDN philosophy exposes a logically-centralized control plane and unified software APIs (e.g. OpenFlow [130]). This enables the accurate collection and maintenance of evolving network conditions. In turn, we leverage this in support of dynamically adapting IoT data exchanges with minimal modifications to constrained devices. Throughout our explorations in this thesis, we assume progressively more mature SDN abstractions as shown in Fig. 3.6. We start by using SDN APIs to create and maintain *resilient overlays* [16]. That is, we treat the public Internet routes between IoT devices and the cloud, which we typically have no administrative control over, like virtual SDN links.

At first, we consider only these overlay links. GeoCRON constructs a resilient overlay using the individual IoT devices as peers. It centrally determines the resilient paths to be followed by sensor data packets being uploaded to the cloud. The peers act as SDN switches in forwarding messages through the right sequence of peers that avoids likely infrastructure failures. The peers forward packets in this manner because we assume limited to no direct

control over the Internet routes that packets traverse. Clearly, we could relax this assumption if an ISP provides SDN APIs for such direct control.

Ride utilizes these overlay links to provide multiple disjoint paths through which devices in the edge network can contact the cloud service. However, it also monitors these virtual links and selects the best available if some become challenged. If none are available, it moves to operating in edge mode by re-routing messages to an edge cloud service. In edge mode, Ride additionally configures the SDN components in the local edge network (where we do assume direct control). This improves local data collection as well as alert dissemination through the use of a resource-conserving resilient multicast mechanism. The flexibility of SDN enables constructing, configuring, and selecting from disjoint multicast trees in this approach. Ride thus ensures cloud connectivity through any available network paths, fail-over to edge backup services during extreme connectivity challenges, and more resilient event routing than traditional approaches.

FireDeX mostly considers the local edge network and abstracts both physical routes and virtual links as a single *SDN big switch*. This simplified view of the underlying network removes the consideration of how to route packets and allows us to instead focus on the prioritization of messages according to information requirements and resource constraints. We designed our queueing network model used in FireDeX so that future work can relax this assumption and explore e.g. different physical routes for different priority classes.

Chapter 4

An IoT Deployment Experience

With the increasing pervasiveness of computers in our daily lives, the IoT concept transitions from a future prediction to real-world deployments. With this manifestation comes a myriad of possible applications from manipulating devices in our homes to large-scale automation of industries and public utilities. To explore possible applications with societal-scale impacts, the White House Presidential Innovations fellows and NIST issued the SmartAmerica Challenge [172] in 2013. In response, we formed a public/private/academic partnership with multiple organizations to envision, design, build, and demonstrate the **Safe Community Awareness and Alerting Network (SCALE)**. This chapter details the SCALE motivation, workflow, architecture, design choices, challenges we encountered, and lessons learned. Since 2013, SCALE grew to include more domains and organizations across multiple continents. The conclusions drawn from this experience inspired much of our future research in IoT (including the remaining chapters in this dissertation). We also leveraged it as a prototype testbed in many of these projects.

4.1 SCALE: Safe Community Awareness and Alerting Leveraging the Internet of Things

A common human-facing aspect of IoT applications is that they aim to improve our quality of life through inexpensive commonly-available technology. While home security systems have existed for decades, they are rather expensive services and only in recent years have we seen components become cheap and available enough that hobbyists experiment with do-it-yourself systems. It seems natural then that an open system, made possible with these recent advances, should be created to improve the lives of underserved populations that previously could not afford such advanced home security and safety monitoring systems. Therefore, SCALE primarily targets the in-home safety scenario described in §3.1.1. We envisioned it as a community-scale deployment of modern connected devices and computer systems to improve the safety of residents. Our goal was to cater specifically to lower-income and elderly residents who often do not have access to advanced technologies such as home security systems, smartphones, and computers with Internet connections.

To accomplish this goal, we designed an event-driven distributed system to sense safety-related data from devices in homes or on individuals, analyze it locally or within the cloud to detect possible emergency events, and automatically contact individuals (e.g. homeowners, caretakers, even emergency dispatch) to notify them and confirm if there is indeed an emergency. We implemented a prototype of this system and deployed it in Montgomery County, Maryland, USA to enable rapid integration of components and testbeds from different partners.

The immediate goals of the SCALE project are:

- Demonstrate our ability to extend a connected safe home to everyone at a low incremental cost

- Jump-start a live testbed for identifying and researching IoT challenges (e.g. middleware, networking, etc.)
- Identify suitable sensors, data schemas, and algorithms for detecting possible emergency events
- Implement and test workflows for cloud-based analytics and alerting
- Demonstrate an open data platform for connecting disparate systems with minimal coordination

4.2 System Architecture

We designed SCALE as an event-driven middleware platform. Devices upload sensed events to a cloud data exchange and the analytics service monitors them for possible emergencies. It sends residents emergency alerts to confirm or reject. Other interested individuals (i.e. emergency dispatch personnel) visualize events through a dashboard. This section discusses the high-level requirements, logical components, architectural design decisions, and implementation details of the system prototype. It first discusses a *cloud data exchange* and then the components of the system that perform *sensing, analysis, and actuation.*

We aimed to implement SCALE in a modular and open manner such that it could be repurposed or extended to support many different applications, not just safety and alerting-related ones. This approach also enables individual components to be swapped out for equivalent ones with little extra effort. We designed all of the components around the concept of loose integration, then relied on an *open data hub* to integrate them into a system. In this way, multiple implementations of the *sensing client, analytics server, and alerting dashboard* could be active simultaneously and iteratively developed in parallel.

4.2.1 Cloud Data Exchange for IoT

As described in §4.3.2, IoT deployments must facilitate machine-to-machine (m2m) communication for exchanging IoT data (sensed events, analytics, alerts, etc.). In SCALE, we propose the Data in Motion Exchange (DIME) system shown in Figure 4.1. We envisioned DIME as an open communications hub for IoT that simplifies the development and deployment processes.

DIME allows any device or service to publish or subscribe to any other data feed, regardless of the protocols used at the device level. This simple loose coupling enables developers to incorporate new services and devices without the need to modify existing ones. Not only does this simplify system evolution but it also creates a level playing field for innovation. Any party can introduce new capabilities, or improvements to existing ones, to the system with minimal need for coordination among current components. They can perform analysis on sensed data, or even higher-level events, and contribute the results back to the exchange, driving science and innovation faster as more devices connect.

4.2.1.1 Sensed Event Data

To build an exchange for IoT data, we first defined the type of data that DIME should handle. We decided to treat raw sensed data and higher-level events equivalently. This aligns with our concept of *virtual sensors*, previously proposed in [102]. *Virtual sensors* abstract low-level data by processing sensor data streams, which may be directly or indirectly derived from physical sensor devices, and exposing higher-level semantics through advanced analytics. Furthermore, by treating incoming data as events, we can develop middleware abstractions for disparate services to treat remote virtual sensors the same as local physical ones.

Recognizing the rich amount of information contained within a higher level event as well

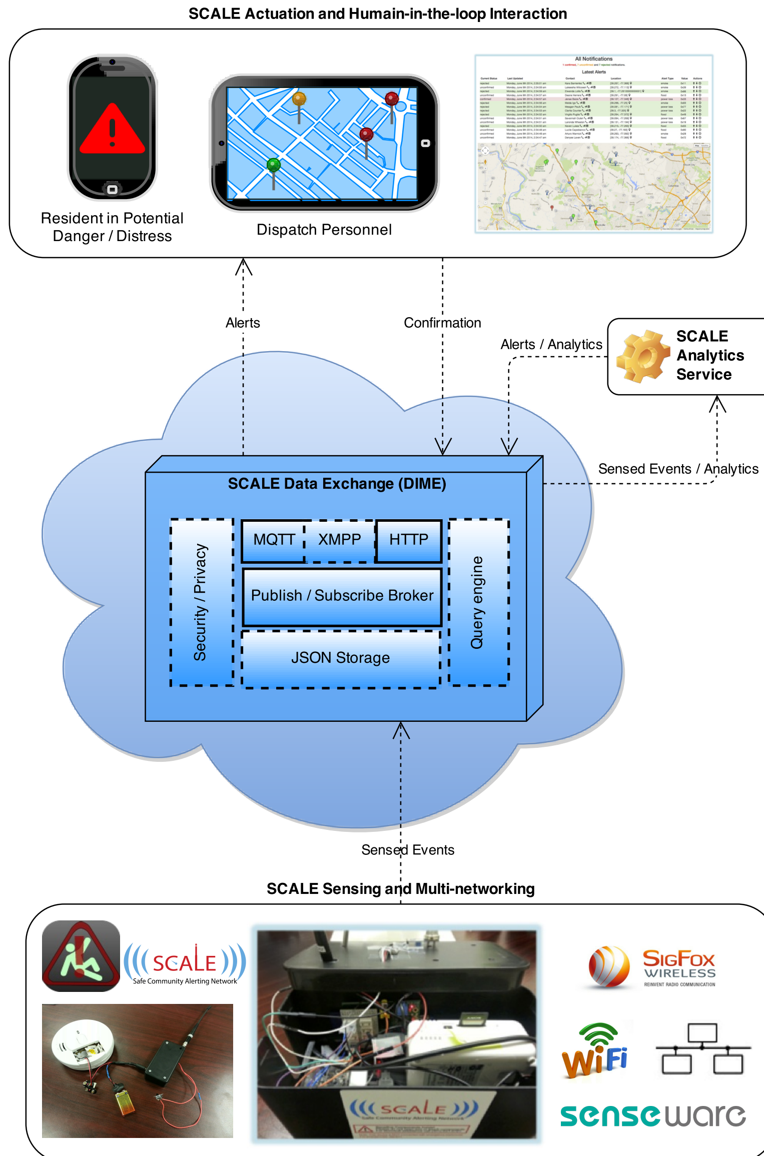


Figure 4.1: DIME facilitates the exchange of data between main SCALE components. DIME Components shown in solid boxes have been implemented, and those in dashed boxes remain as future work.

as subtle device differences that affect lower-level events, we wanted a well-adopted flexible schema that could allow, but not require, inclusion of additional information fields beyond what is necessary to convey the sensor reading. These additional fields should not break the schema or require all entities in the system to understand them.


```
{"d" :
  {"event": "temperature",
  "value": 55.5,
  "units": "celsius",
  "timestamp": 12345678,
  "device": {"id" : "1233",
  "type": "sheevaplug",
  "version": "1.0",
  "manufacturer, chipset, MAC address...":
      "Globalscale Technologies, ..."},
  "location":
      {"lat": 33.3,
        "lon": "-71"},
  "cond": {"time" : 1234, "value" : 95,
      "threshold" : {"operator" : ">",
          "value" : "95"}},
  "prio_class":  "high",
  "prio_value": 0...10,
  "schema":
      {"source":
        "schema.org/scale_sensors.json"}}}
```


Listing 1: An example of the JSON-formatted sensor data schema SCALE uses to encapsulate sensed events.

For simplicity and flexibility, we opted to use JSON to format the data for transmission to the broker as it provides a commonly-used self-describing format supported by mature software modules. We defined what we thought was a reasonable starting point for the schema. described in Listing 1. It includes information about the platform (hardware, operating system, etc.), sensor (device type, identifier, etc.), data (units, value, timestamp, priority, etc.), a pointer to the specific schema in use to facilitate interoperability, and any other miscellaneous domain-specific information developers wish to include. One should note that

we do not believe this schema comprehensive and rather envision a system where different domains could define their own schema and publish information about how to interpret it so as to encourage interoperability between vendors/systems. Such an approach could make use of an interface such as HyperCat [34] that identifies available devices/data, how to access them, and how to interpret the results.

4.2.1.2 Current Implementation

In its current form, DIME uses MQTT[141], a fast, lightweight, publish-subscribe-style protocol. It was developed by IBM for lightweight telemetry, donated to open source, and has since gained popularity for use as an m2m protocol for IoT data. The publish-subscribe model allows multiple servers to collect data from DIME and multiple clients to send it without requiring any configuration on our part. The DIME server currently uses the open source Eclipse Paho MQTT broker [144]. While Paho could be run anywhere, we used IBM's MessageSight [1] software appliance, which handles millions of concurrent data streams, running on the IBM SoftLayer Cloud.

In DIME, sensor data is published to a particular topic, which consists mainly of a device identifier and sensed event type. Other services, such as the SCALE Server, subscribe to this data by a particular device, sensor type, or just to all data.

For compatibility, DIME also provides a RESTful interface, implemented via HTTP, initially residing on the SCALE Server for ease of deployment. This interface translates incoming data into the proper format and publishes them via MQTT. In this manner, we quickly implemented DIME as a simple MQTT server, though we plan to extend it to directly support other protocols (e.g. HTTP and XMPP).

[1] `http://www-03.ibm.com/software/products/en/messagesight`

4.2.2 Sensing Client

This section describes the development and deployment of several SCALE Clients that sense, minimally analyze, and report data to DIME for ingestion by the analytics server.

4.2.2.1 Networking Technologies

SCALE uses various types of networking technologies in order to facilitate communications among heterogeneous sensor devices. The specific choice of networking technology typically depends on the sensor application as well as the constraints of the facility in which the sensors are deployed. In most installations, the limited reach of the facility's cabling infrastructure to the sensor installation points precludes the widespread use of wired networking technologies (e.g. Ethernet) to support the sensor network. Hence, wireless should be supported to provide at least one viable option.

Various wireless networking options exist, such as Wi-Fi, Bluetooth and ZigBee. Table 4.1 shows the list of edge networks that SCALE used in its various applications. Wi-Fi is the most well-known wireless networking technology, but its high power profile can be a limiting factor. Bluetooth, on the other hand, has one of the lowest power profiles, yet has a limited range. ZigBee represents an intermediate solution built on IEEE 802.15.4, which incorporates a low-power wireless specification enabling mesh networking of sensor devices. 3G is good for outdoor deployment where WiFi access is limited, but it is costly. Ultra narrow band (UNB) can be used for long range communication. It, however, supports only low bandwidth communication and requires additional infrastructure (such as base tower). As these examples illustrate, the power envelope and distance between sensors in a particular sensor application can dictate the choice among wireless networking options.

Our first SCALE prototype included the standard Wi-Fi and Ethernet connections as well

Table 4.1: Access/edge network technologies used in SCALE

Network	Features	Application
UNB	Ultra narrow band, long range radio, low bandwidth	CSN
Zigbee	Low power wireless mesh networks	Senseware
WLAN	High bandwidth wireless with infrastructure support	CSN
WiFi Adhoc	Mesh networking, adhoc networks	CSN
BlueTooth and BLE	Low power, low data volume	Assisted living, fall detection for elderly
3G/Cellular/GPRS	Wide area coverage, outdoor deployment	Air quality monitoring, noise sensing
Wired Ethernet	Wired infrastructure	Configuration and management

as Sigfox ultra-narrowband (UNB) wireless adapters. UNB allows for long-range, low-power, low-bandwidth uplinks. Sigfox provided a UNB basestation to install in Montgomery County. We were able to deploy several SCALE devices with Sigfox UNB adapters in Rockville, Maryland and send data to DIME via the basestation from up to several kilometers away, despite using lower-powered basestation and client adapter antennas.

Sigfox adapters send data in 12 byte packets and so MQTT was not an option. Instead, we coded the data to fit within this packet and created the aforementioned HTTP interface where Sigfox directed this data. When received at the SCALE server, the data is translated into the proper schema and published via MQTT so that it appears identical to other sensors' data.

We also integrated Senseware's proprietary mesh networking solution into the SCALE system as described below.

4.2.2.2 Hardware Platforms

We wanted a flexible client platform to allow deploying heterogeneous sensors, devices, and networking technologies. Some clients may plug into a stable power source and Internet access to support a multitude of sensors and more advanced local data analytics, while others may be battery-powered and just upload raw sensed data via wireless. To support the pervasiveness of these systems and address the latter of these device types by reducing reliance on home Internet access, crucial for our mission to support underserved populations, we aimed to integrate platforms and technologies that could provide long-range low-power connections. To address both styles, we chose to use commodity off-the-shelf components wherever possible, which had additional benefits of reducing infrastructure costs; increasing the number of possible integrated devices and sensors; reducing development costs by leveraging extensive community support; allowing other researchers, hobbyists, and new team members to easily understand our design so that they may copy and extend it.

We first built a general-purpose sensor box named FlexSCALE that supports many different sensors and network adapters. Interested readers can refer to our directions on how to build their own FlexSCALE box[2]. The compute units and sensors are housed within a large cable box to protect wires and maintain a cleaner facade. Environmental sensors (e.g. light and temperature) were fastened on top such that they protruded from holes in the lid, gaining external access with minimal wiring exposed. The initial version housed both a Raspberry Pi and a Sheevaplug, each running a form of Debian Linux, as the compute units. We transitioned to just using the Raspberry Pi to simplify platform support and handle a greater variety of peripherals thanks to I/O ports and pins other than USB and Ethernet.

Each FlexScale box has light (luminance), explosive gas, passive infrared (motion detection) sensors, an accelerometer (acting as a seismograph), and thermometer as well as a Wi-Fi

[2] `https://github.com/KyleBenson/scale_client#building-a-scale-box`

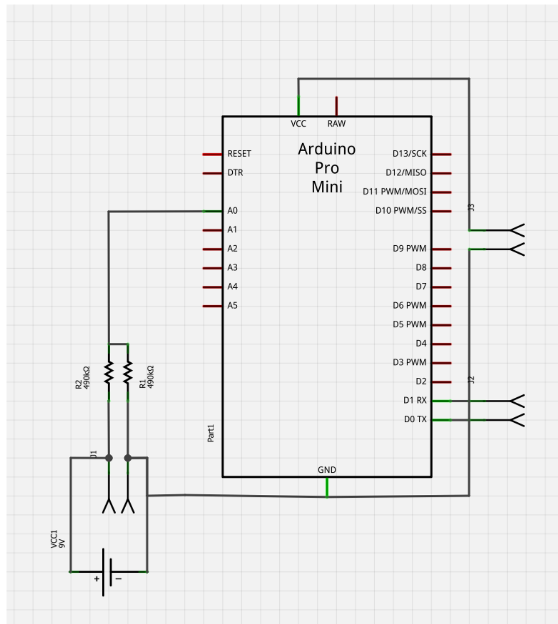


Figure 4.2: Wiring diagram for the hacked smoke detector device.

dongle and a Sigfox UNB adapter. A powered USB hub supported the two USB sensors and two USB network adapters on the Sheevaplug and older Raspberry Pis.

In contrast with the larger and more extensible FlexSCALE Box, we experimented with dedicated devices to monitor a single sensor and report its readings with almost no analysis. We were particularly interested in retrofitting existing household sensing devices and connecting them with the SCALE service. Therefore, we modified an off-the-shelf 9-volt smoke detector and attached it to an Arduino Micro for the purpose of monitoring the voltage level of the battery. See Figure 4.2 for the wiring diagram used for this device. The Arduino constantly sends (every ~4 sec.) the measured voltage level to DIME via a Sigfox adapter. If this level drops significantly, indicative of the alarm going off, the server sends an alert. The theory here is that the alarm consumes more power than just the sensor itself and so the additional function drops the voltage level of the battery significantly. The Arduino and Sigfox devices fit into a small project box, similar in size to a mint tin.

To complement the aforementioned dedicated and flexible sensing platforms, we also built an Android application for personal fall detection. It analyzes the device's accelerometer

readings using the algorithm presented in [41]. Upon detecting a user falling, the applications presents them with an option to cancel the alert, thus preventing false alarms, before a countdown timer expires and the phone publishes the alert via MQTT to call for help.

To test and showcase how existing proprietary systems could integrate with DIME and SCALE with minimal modifications, we partnered with Senseware, a Virginia-based startup. They build modular sensor devices that transmit data via mesh networks to a gateway for upload to a web-based cloud service. The user-friendly devices are easy to deploy and can have a variety of connected sensors (i.e. air quality, humidity), making them an ideal candidate to expand the SCALE testbed with commercial hardware. Senseware integrated their sensors' data by forwarding it to a Senseware-specific HTTP endpoint to facilitate this connection similar to how we integrated Sigfox devices. The data still goes through their full cloud service as well and therefore is visible in their own dashboard as well the SCALE one.

4.2.2.3 Software Design

We wanted a cross-platform extensible software package that runs on the majority of devices. This package should be modular and support plugging in different component implementations (e.g. new sensors or network protocols) without disrupting other modules. Adding or changing hardware components should not require any software changes but should rather be handled through a simple configuration file. We also needed a lighter-weight client software stack for constrained devices, but this would likely be less reusable as these often require device-specific optimizations and/or libraries that would decrease flexibility and portability.

Interested readers can find the most up-to-date version of the SCALE client software at `https://github.com/KyleBenson/scale_client`. Figure 4.3 shows the first prototype FlexSCALE software we built to address the above requirements. Figure 4.4 shows an updated class diagram for the software that more clearly elaborates on the inheritance model

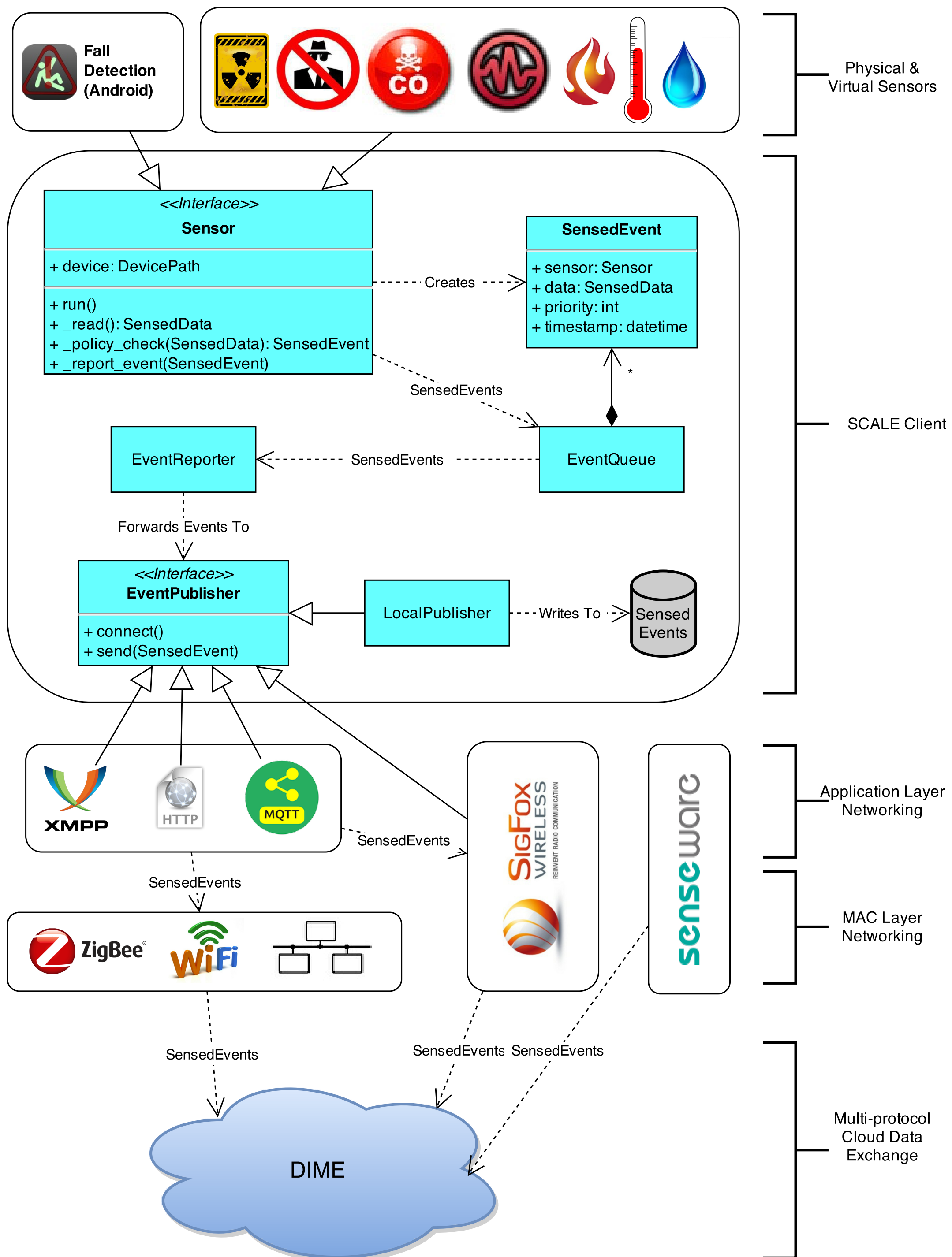


Figure 4.3: The original SCALE Client architecture.

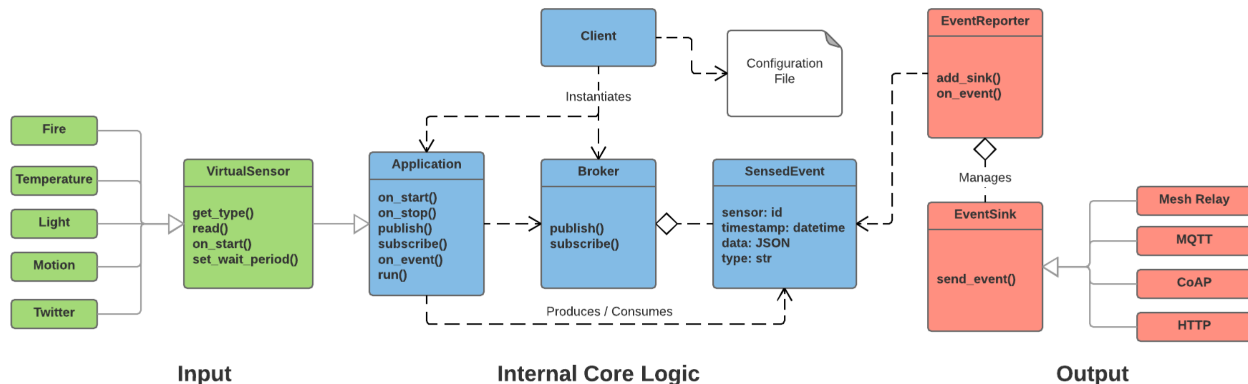


Figure 4.4: The latest SCALE Client class diagram.

we used. This Python package connects with various sensor devices attached to the compute system, records data, and publishes events according to some policy. Data originates at an instantiation of the abstract Sensor class, which allows us to rapidly connect new sensor types and define new *virtual sensors*. Sensors create SensedEvents, which encapsulate the sensor data schema described in Section 4.2.1.1, and place them in a queue for reporting to DIME or further analysis by relevant VirtualSensors.

Each networking protocol that connects the Client to DIME is abstracted with a concrete instantiation of the EventPublisher class. Similar to adding new sensors, this allows us to easily add new protocols and API endpoints with minimal additional code. It currently supports MQTT via Wi-Fi or Ethernet, Sigfox ultra-narrowband (UNB), and local storage. EventPublishers also provide a degree of control over quality of service (QoS), currently just in the form of transmitting higher-priority events first. We added this feature early on to address the UNB transmitters' low bandwidth.

We used SaltStack [3] for configuration management: remotely deploying and updating software on the sensor boxes. We chose SaltStack because it is highly scalable, supports redundant master servers, is written in the Python language used throughout SCALE, and (most importantly) connects with devices deployed behind network address translators (NATs)

[3]http://www.saltstack.com/

as are commonly found in residential homes. Our configurations for the original SCALE deployment can be found at `https://github.com/KyleBenson/ScaleSaltConfig`.

4.2.3 Analytics

The SCALE analytics service monitors sensor data and events streaming from DIME and publishes detected emergency events, which may trigger alerts to individuals when appropriate as described in Section 4.2.4.1. Refer to Figure 4.5 and the description below for how we designed and implemented the analytics server.

We implemented the analytics engine as an asynchronous event-driven Python server that acts on sensed events in accordance with their type using appropriate event-handlers. Thus, adding new sensor and event types only requires additional programming by end application developers, not those responsible for server development. The code implementing this server is available at `https://github.com/KyleBenson/SmartAmericaServer`.

The server, deployed on the IBM BlueMix platform, receives sensed data through Eclipse Paho's MQTT client [144] and routes it to the appropriate event-detection function. These functions, which we refer to as *virtual sensors*, convert lower-level events to higher-level ones (e.g. alarm buzzing to smoke detected to possible fire), escalating events and publishing them back to DIME. When a possible emergency event is detected, SCALE may alert a resident as described in Section 4.2.4.1.

We used the above incremental approach as it allows different server components to live on or replicate across separate machines and locations, improving scalability, response times, modularity, reliability, and ease of creating an audit trail. An audit trail exposes intermediate events to external entities, which helps in building trust in particular event sources (i.e. sensing devices, event-detection algorithms) and adding new hooks for separate services to

Figure 4.5: Depiction of data flow and database tables in the analytics and alerting services.

make use of these states. We accomplished this distributed approach using the Celery task queue manager [4] that distributes event-handling across worker processes.

Some historical storage of recent events is necessary to detect changes over time and disambiguate sensor readings indicative of the same event. We used the Django framework's object-relational mapping (ORM) to abstract the PostgreSQL database tables seen in Figure 4.5 as Python objects. Periodically, the database removes old events, though in the future we will instead archive them for historical analysis and audit purposes.

4.2.4 Actuation

This subsection describes SCALE's mechanisms for interacting with and alerting human users.

4.2.4.1 Alerting

Once possible emergency events are detected, concerned individuals must be notified in a timely, reliable, accessible, and interactive manner. Users will receive alert messages after connecting their home monitoring devices to SCALE and registering these devices and contact information with the alerting system. Ideally, this system could eventually integrate with emergency dispatch centers to automatically alert authorities. To mitigate false-positives, it supports a confirmation step in which the user determines whether the emergency is real and emergency personnel should be alerted.

Because SCALE especially aims to make the system as accessible as possible, especially for lower-income and/or less technologically-savvy users, it does not require access to a computer or smartphone when receiving and acting on alerts. It supports simple phone calls

[4] `http://www.celeryproject.org/`

so that users with land lines, but no cell phones, can still use it. It does support SMS (text messaging) as most people in the US nowadays have cell phones, especially since government programs such as [5] exist to provide them to low-income residents.

While a smartphone application is a potential future addition, we opted to use an Internet telephone service for alerting. We chose Twilio, which has a rich API for programming interactions with users through the server's web interface, to issue SMS/phone call alert messages and handle correspondence with participants (event confirmation/rejection, registration/unregistration, contact method preferences, etc.).

When the analytics subsystem detects a possible emergency, it sends an Alert message through MQTT to instruct the alerting subsystem to contact the registered user(s) of the device from which the sensor data originated. This contact info is retrieved from the database as shown in Figure 4.5, and the database stores an Alert entry representing this communication. When the contact responds, the database updates the state of the Alert to *rejected* if the user responds with "emergency" or presses 1 and *event confirmed* if the user responds with "okay" or presses 2.). If no one responds within some amount of time (currently 30-60 seconds) of initiating the alert, a trigger fires that escalates the emergency event. Currently, it is set to confirm the event, but public officials likely would adopt a different policy that perhaps dispatches an individual to investigate further rather than scrambling an entire unit.

4.2.4.2 Dashboard

To help dispatch personnel visualize alert events and sensed data, we built a dashboard for the SCALE system. We wanted an intuitive, lightweight, web-based solution so we could later port it to mobile devices and borrow functionality for a smartphone application aimed at residents for monitoring their personal SCALE deployment(s). Figure 4.6 shows the main

[5] `http://www.fcc.gov/lifeline`

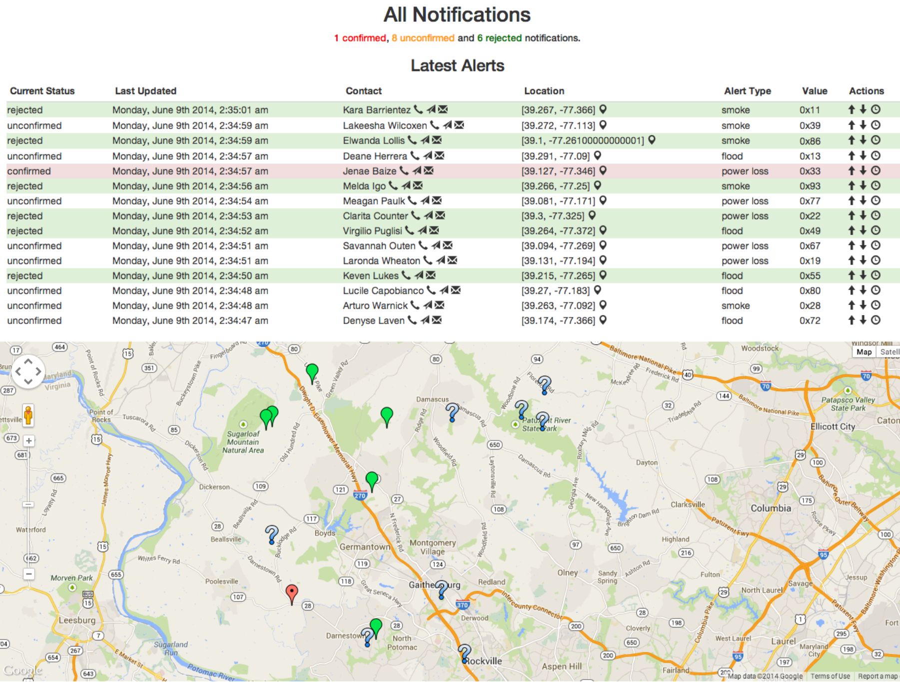


Figure 4.6: The main view of the SCALE dashboard.

user interface.

The main view of the SCALE dashboard presents a list of recent alerts, their locations in a Google Maps view, and a summary of the number of high, medium, and low priority events. It includes contact information about the individual alerts and a currently non-functional interface for calling, texting, or emailing residents. The user can also confirm or reject events manually. A second view presents raw sensed events as they arrive, which is simply for debugging purposes. The dashboard, also hosted on BlueMix, is built on top of software designed by BioBright. It is written using Node.js at the backend, Javascript and Twitter Bootstrap in the front-end, and a browser MQTT client.

4.3 Conclusions & Research Challenges

Our experience in designing, developing, and deploying the first iteration of SCALE described in this chapter has proven the feasibility of a distributed IoT approach to improving resident safety at a low incremental cost. This initial exploration underwent many extensions and expanded to include additional application domains across multiple continents. Because SCALE serves as a research testbed, much further engineering and development would be necessary before it could be deployed in a real mission-critical setting. However, the lessons we learned helped drive future research and development for IoT. We discuss some of these topics below and present our future plans and vision for SCALE. Some of the conclusions we draw here directly influenced the projects described in Chapters 5,6,7.

4.3.1 Resilience Concerns

From a hardware perspective, our biggest lesson learned was that *cheap sensors break*. We purchased many of our devices online for under $10-20 (US) and some failed. The explosive gas sensors in particular tended to burn out after a few uses. While some of these issues can be alleviated by using better quality components, this likely drives up the price of the device without completely ensuring reliable operation and so care must be taken to plan for these issues.

Regarding power, we found that the number of peripherals on the Raspberry Pi required a higher-amperage power adapter. We also used a powered USB hub to support all of the USB sensors and wireless adapters used on the Sheevaplug. We experimented with battery backup and determined that the system dies after about 10 hours. Future work will explore graceful degradation of devices (turning off network adapters, adjusting sampling rates, etc.) to improve this battery life.

We also found working with prototype devices inside a large project box easier than trying to cram them all into a small one. We frequently needed to troubleshoot or change hardware and so having room to work simplified this process. Because we found using both a Sheevaplug and a Raspberry Pi in one box to be troublesome due to the former's constraints in supporting peripherals and having an outdated platform, we are interested in finding a box with a smaller form factor as the small cable boxes were still viewed as large and imposing. We are considering 3D printing our own design so we have ample yet not excessive room and custom fittings for all the components.

From a software perspective, we found the design of the client around simple abstract pipeline components to be very flexible when adding new hardware support.

Since the first SCALE effort, we experimented with additional networked sensor devices to extend the coverage of a SCALE deployment and improve its resilience. We added support for an ad-hoc Wi-Fi mode that supports distributed emergency detection and alerting even during power and network failures by having FlexSCALE boxes exchange data with each other directly. We also added inexpensive battery-powered microcontrollers with attached sensors and IEEE 802.15.4-based wireless so additional sensors can be deployed throughout a residence without requiring additional FlexSCALE boxes.

4.3.2 Data Exchange

While we found MQTT suitable for rapidly developing an IoT system, we did find it limited due to its simplistic lightweight approach. Below we outline some considerations for future IoT protocol standardization efforts and security considerations for data management in IoT systems.

4.3.2.1 Standards Considerations

When designing our analytics system and topic hierarchies, we found MQTT's lack of support for fine-grained queries somewhat limiting. It does not handle ranges at all and the expressiveness of wildcards cannot match that of regular expressions. For example, to perform a query over a target geography one would need to define a tag for that geography, which limits flexibility for defining new targets. Our current inefficient solution is to subscribe to all events and filter them based on content. Future protocols should consider the desire to issue such queries and filters as sorting through the results by content on the client-side may be intractable with the larger-scale systems the IoT vision promises.

A major advantage of MQTT is its lightweight nature and simplicity. Future IoT standards should follow this model, while allowing for extensions that provide additional services when, but only when, developers/deployers wish to use them. For example, the size of the DDS [142] standard may intimidate some newcomers, whereas getting started with MQTT takes only a matter of minutes. Protocol designers must keep in mind that many IoT developers will enter the market with little systems experience or come from a Web 2.0 background. As such, providing a simple intuitive starting point, perhaps with RESTful APIs, for them to develop systems will help lower the barrier to entry, resulting in more projects with diverse applications. This approach appears to have worked very well with Node.js, which has enjoyed rapid adoption in part due to allowing the developer community the freedom to pick from a variety of options for accomplishing a given task rather than specifying one standard way. To further lower this barrier, future standards should also allow developers to use familiar tools, languages, etc. whenever possible. For example, they should emphasize interoperability with other protocols, such as how CoAP [71] can interoperate with HTTP.

4.3.2.2 Security and Privacy

While we did not implement security mechanisms in SCALE beyond requiring SSH keys to remotely access devices, we did discuss security and privacy implications throughout the project and plan to address them in future versions. MQTT supports authentication and identification directly, but not authorization. It can be run using TLS so that the username and password used for authentication are encrypted during transmission. Identification is handled using a unique identifier or a public digital certificate, with the latter clearly involving management of keys. Some MQTT server implementations provide authorization as an added service. In a scenario where user privacy and integrity of the data and communications is crucial, such a server should be used. This allows the server to determine which client devices have access to which resources, i.e. which topics they are allowed to publish and subscribe to. This would prevent unauthorized individuals from retrieving readings from devices they do not own as well as prevent publishing of information to a topic representing a different device. This does not, however, validate the actual data in question, which could still be faked by an individual with the proper secret keys.

One open concern is that of the devices' physical security. As they are located in residents' homes they could be physically tampered with, moved, or have their code modified by knowledgeable users. This could result in undefined behavior, misleading event reports, or completely spoofed data. This is one of the main reasons for involving human-in-the-loop sensing in order to confirm events before notifying emergency personnel. Whether this step is truly enough to ensure correctness of the data in question is a policy question outside the scope of this work.

4.3.3 Analytics

When building the SCALE Server, we aimed to design the analytics module to promote code reuse such that new complex analytics could be derived from existing components with minimal modification. We successfully used an event-driven model that listens for sensed events of some type, transforms them to represent a higher-level conceptual event, and then publishes them back out through DIME for further analysis or immediate action. In this manner, the analytics became a pipeline through which data streams flow, allowing other pipelines to hook in at any step to work off intermediate results for a different purpose. For these reasons we are considering redesigning the analytics module using more suitable libraries such as a stream processing system. We hope this will simplify the implementation and eventually provide a query language suitable for less-technical users to create queries and glean useful information from IoT data.

We envision the future of IoT platforms as enabling non-programmers to build and deploy analytics by merely composing these functions together and changing exposed parameters. An example attempt at building such a development environment is Node-RED [139], a visual graph-based programming platform for IoT that closely matches this paradigm.

Chapter 5

Geo-aware Resilient Overlays for Cloud-centric IoT Data Collection

In this chapter, we begin our exploration of resilient communications for IoT data exchange by considering cloud-centric data collection. We begin this investigation in a highly-challenging earthquake-detection scenario and propose techniques for collecting data from a distributed seismic sensor network despite wide-area infrastructure failures. Because we assume no direct control over the routing infrastructure and start with only the IoT devices and cloud analytics service themselves, we propose a resilient overlay network (RON)-based approach. In this way, we essentially treat the IoT devices as edge resources that facilitate an SDN-like mechanism for routing around possible network failures. The cloud service configures them according to application resilience requirements and the underlying physical network topology. As we put together the complete approach proposed in this thesis, the reader will understand how the techniques and algorithms discussed in this chapter can easily be adopted for direct management through SDN APIs when possible.

5.1 Chapter Overview

This chapter focuses on cloud-centric resilient data collection within the context of the earthquake-detection application described in §3.1.2. Hence, we address the IoT data exchange challenge of large-scale geographically-correlated failures due to a high-magnitude earthquake. Such failures hamper the communication of critical safety data to the backend cloud service where it is stored and analyzed. We therefore propose exploiting redundant network paths to improve data collection. To accomplish this, we extend the the notion of resilient overlay networks (RON), originally designed in the context of wired networks, to IoT deployments. For a detailed background on RON, see §2.2.3.

To capture and exploit the geographically-correlated nature of the deployed infrastructure and associated damaged regions in disasters, we propose the use of *Geographically-Correlated Resilient Overlay Network (GeoCRON)* [28, 29]. We design the GeoCRON middleware to extend the RON concept with awareness of the geographic placement of nodes in the underlay. It uses this information, as well as knowledge about the underlying routing infrastructure, to choose multiple geo-diverse routes in order to improve the chances of a message reaching the destination during large-scale geo-correlated failures.

We study this approach within the context of massive network failures affecting the two different community IoT systems within the same geography shown in Fig. 5.1. In addition to the **seismic sensing network**, we also consider a smaller infrastructure-based network of wireless devices embedded in a **smart water distribution network**. The latter reports water pressure changes at pipe junctions that may be indicative of leaks that require human intervention. See §5.4.2 for more details about this system. We consider the GeoCRON approach both with and without the additional water sensing infrastructure that enables cross-deployment data exchange collaboration.

To study geo-correlated failures on this combined network infrastructure, we utilize simula-

Figure 5.1: A general IoT community monitoring example with emphasis on water infrastructure.

tions due to practical concerns that prevent real-world experimentation. We evaluate the use of our proposed GeoCRON middleware to route messages along geo-diverse paths in order to improve the chances of data delivery. GeoCRON measures path geo-diversity through various formulations, and so we apply these various measures to create a family of heuristics for choosing geo-diverse paths and compare their performance with each other. The proposed heuristics fall into one of two classes: those with only knowledge of peers' locations and those with knowledge of the underlying network infrastructure and its components' locations.

Key contributions of this chapter include:

- Discussing existing as well as new algorithms to exploit path geo-diversity for the construction, maintenance, and effective use of GeoCRON overlays in a realistic multi-network geo-correlated failure scenario. (§5.3)
- Deriving sensing and communications network topologies based on regional water demand, which we leverage as a proxy for population density. We use these topologies

to inform GeoCRON about the structure of the underlying network infrastructure. (§5.4.2)

- Validation of the diverse GeoCRON techniques with appropriate geo-diversity metrics using extensive simulations and analysis of improved resilience to communications network failures. (§5.5)

- A prototype implementation of the GeoCRON reliable communication methodology in the SCALE platform to understand operational and deployment challenges in a real world community setting. (§5.6)

5.2 Resilient Overlays for IoT Data Exchange

We now describe how to use the IoT devices within a resilient overlay for geo-correlated failure avoidance.

5.2.1 Failure Avoidance

Fig. 5.2 depicts GeoCRON's network-aware approach to resilient communications. When a GeoCRON node tries to send sensor data to the cloud server(s), it tries to maximize the delivery rate of the data by sending multiple copies along disjoint paths to each known server. In our earlier work [29], we explored sending one message and awaiting a timeout, which indicates a possible network failure. The node would then try a different path, up to a predetermined number of retries. However, we found that this frequently required a full 5-10 seconds to deliver the majority of the messages. Given the low-latency requirements of our seismic sensing scenario, we instead opted to send all message attempts at the same time to decrease latency (i.e. setting the message timeout to 0). We note that other IoT systems,

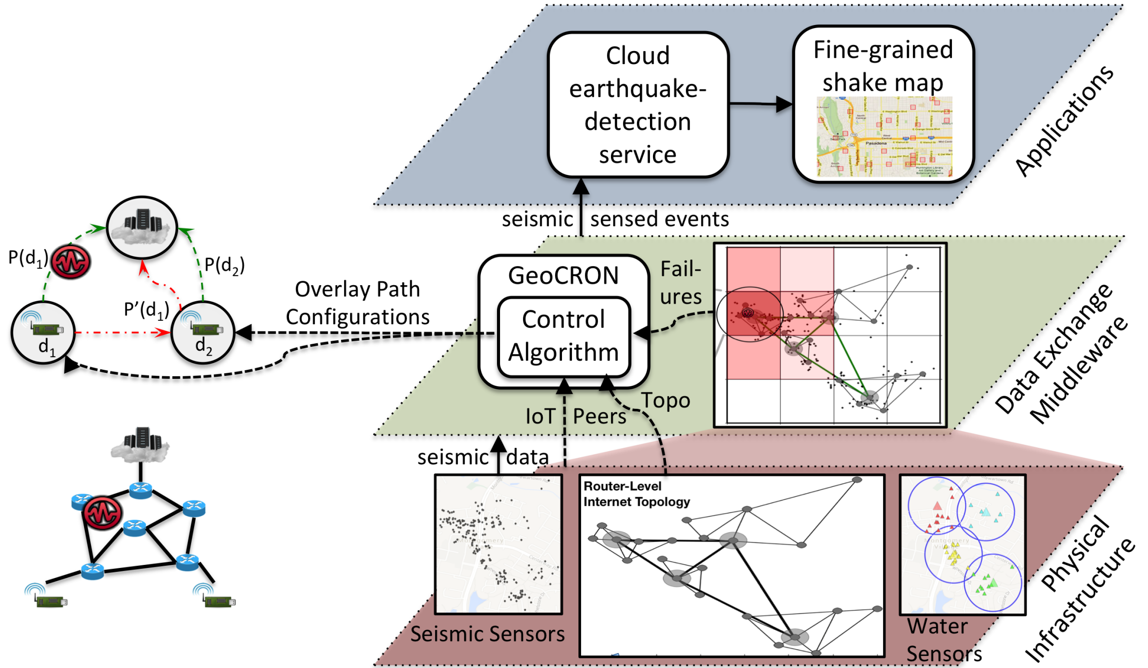


Figure 5.2: The GeoCRON approach leverages infrastructure topology information to avoid failures through geo-diverse overlay routing.

such as the water sensing network, may opt to relax this constraint and use a hybrid of these two techniques, but we save this study for future work.

We define the *multi-path fanout* k as the number of message copies sent to each server. This value is configurable within the client code and is experimentally determined to maximize data delivery without introducing too much congestion into the network. One could also envision a more intelligent selection of k based on knowledge of the network topology and assumptions regarding the impact of the failures, perhaps derived from the level of shaking perceived by the seismic nodes. The first of these packets is sent directly to the server without any overlay hops to minimize latency and overhead. The remaining $k-1$ are sent using geo-diverse overlay paths chosen from the available peers as described in §5.3. In [95], the authors discovered that the vast majority of node pairs only require a single overlay hop in order to exhibit the same diversity as multiple hops. Therefore, we use as each possible path a 2-hop overlay path where the first hop is some overlay peer and the second is the

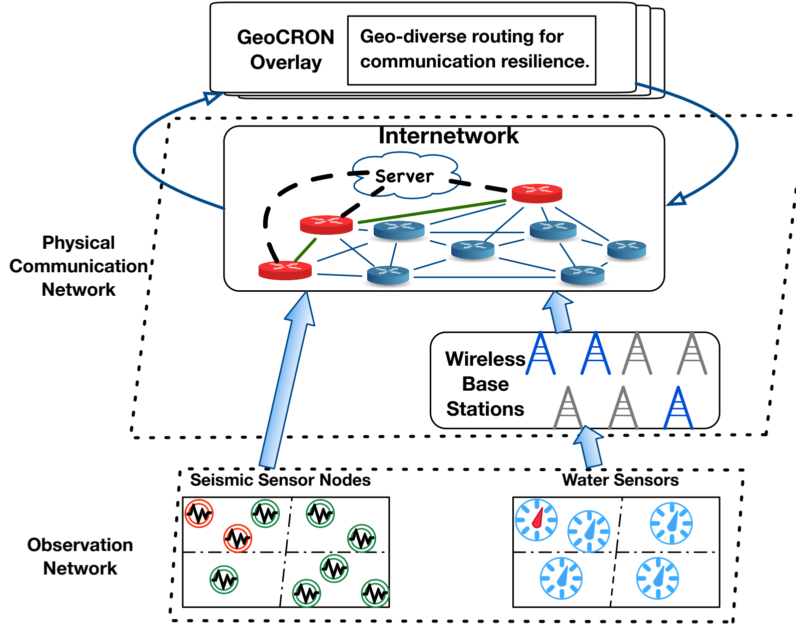


Figure 5.3: An overview of the layered multi-system architecture being studied.

destination server. The list of overlay peers from which to choose is configured by the cloud server as described below.

Corresponding to the two deployment scenarios, we consider a resilient overlay network (RON) comprised of two types of nodes, i.e. seismic sensing nodes and the wireless basestations that receive the water sensor readings (see Fig. 5.3). These two deployments clearly differ in terms of timing constraints on the data delivery, last-hop connectivity, and physical placement of network nodes.

5.2.2 P2P vs. SDN Overlay Construction

GeoCRON routes around network failures using end hosts (i.e. IoT devices) with stable Internet connections and ample power supplies. This avoids reliance on Internet Service Providers adopting and deploying new technologies (i.e. SDN). Networked systems commonly follow the paradigm of pushing intelligence towards the edges of the network to streamline the core

and allow heterogeneous logic on these edge devices. Because many IoT devices, including our model seismic sensing nodes, feature general CPUs, we can easily add software intelligence to enable middlewares for sharing resources between IoT devices. This allows us to share networking capabilities and implement the aforementioned RON approach to significantly enhance communications resilience in IoT systems. Thus, we exploit the existence of a multitude of end-host devices without requiring Internet Service Providers to offer additional services in their networks. Our proposed IoT deployments allow us to add this logic by running the GeoCRON overlay on the seismic nodes, which run on commodity computers, and water sensor basestations, which are assumed to be running high-performance hardware.

Because GeoCRON nodes report data to a cloud service for analysis and storage, we consider GeoCRON a *hybrid peer-to-peer network*. Each node must at least know the locations and IP addresses of its peer clients; specific heuristics will request additional information (e.g. physical route to the other node) in order to facilitate the choice of which route to use. Note that we assume a *centralized controller* (i.e. the cloud server) configures the overlay such that clients need not maintain contact with more hosts other than those they might use as overlay hops. This fits with our general SDN-like approach; the overlay peers can even be thought of as virtual switches in such an architecture. However, one might consider designing a fully-P2P implementation of the GeoCRON system for more decentralization. Such a design must consider which peers each node maintains contact with (i.e. neighbors in the overlay). See §5.6.1 for a discussion about some of these considerations.

5.3 Algorithms for Geo-Diverse Route Selection

Resilience to Internet failures has been extensively studied, though few works address massive geo-correlated failures. See §2.2 for an overview of some approaches. Here we define the algorithms GeoCRON uses for geo-diverse route selection.

5.3.1 Model and Notation

This section introduces notation that we use to model and reason about GeoCRON formally.

Network Model Let $G = (V, E)$ be the graph defining the network under consideration, where V is the set of nodes representing routers and end hosts and E is the set of undirected edges representing physical links between nodes. Each edge $e \in E$ is assigned a weight $w \in \mathbb{N}$ to represent the latency (as measured in milliseconds) of the links. For the purposes of this study, we assume the latency of a link is constant (other than queueing delays) and bidirectional.

Node Locations Each node $v \in V$ is assigned a physical location as measured in geographic coordinates. Let $loc : v_i \rightarrow (x_i, y_i)$, for $x_i, y_i \in \mathbb{R}$, be the function mapping each node v_i to its physical coordinates and $dist : (v_0, v_1) \rightarrow \mathbb{R}$ be the function mapping each node pair to the physical distance between their locations.

We consider each node as belonging to some region, which is some geographic area within the entire plane under study. Let R be the set of regions under consideration, $reg : V \rightarrow R$ map each node to the region it is located in, and $R_D \in R$ be the location of the disaster. In our discussions, we consider each region as a cell within a grid and so reg assigns each $v \in V$ to the cell whose bounds v's location falls within, which could be gleaned from GPS, IP address, or user-specified data. Note that although these approaches may not give perfectly accurate location information, the coarse granularity of reg means that they should reasonably suffice for our purposes.

Overlay and Server Nodes Let $S \subset V$ be the servers (sinks) to which each sensor node reports data to.

Network Paths When a sensor node sends a message to some server $s \in S$ the packet will travel a particular path through the network as determined by the underlying infrastructure.

Let $p = (v_0, e_0, v_1, ..., e_{n-1}, v_n)$, for $v_i \in V, e_i \in E$, be a sequence of nodes and interconnecting edges that represent such a path. When a sensor node chooses to send such a message using overlay nodes as intermediaries, we may consider only the sequence of overlay peers rather than the entire physical path. As such, let $h = (o_0, ..., o_n)$, for $o_i \in O$, be the sequence of overlay peer hops taken by such a message, where $o_n \in S$ is the final destination server. Currently, we only consider 2-hop overlay paths in which the second hop is the destination server s_0. That is, $h = (o_0, ..., s_0)$. Let also $P = \{p_0, p_1, ...\}$ be the set of all possible paths in G and $path : h \rightarrow p \in P$ map overlay paths to physical topology paths. This is done by joining together the physical paths from the source peer o_0 to the first hop o_1 with with the path from the first hop o_1 to the second hop o_2 and so on.

Modeling Path Diversity In order to assess the diversity of a potential path choice, we must define some model for quantifying it. We define diversity as a measure of how different two paths are in terms of shared components, proximity of component locations, or even both. The goal is to identify paths that are less likely to suffer geo-correlated disruptions at the same time. Let $D(p_a, p_b) : \{p_a, p_b\} \rightarrow \mathbb{R}$ be the abstract diversity function for comparing two physical paths. Let $D(h_a, h_b) : \{h_a, h_b\} \rightarrow \mathbb{R}$ be the abstract diversity function for measuring the diversity between two overlay paths. These function templates are used to measure the diversity according to one of the concrete heuristics defined in §5.3.2.

5.3.2 Geo-diverse Path Heuristics

In this section, we propose a family of techniques and heuristics used to rank the geo-diversity of various overlay path options. These heuristics are used by GeoCRON (see Algorithms 1 and 2) to select the best overlay paths with which to deliver data. When a GeoCRON node o_1 has data to send to the server, it first sends the data directly without use of the overlay as described in §5.2. For the sake of discussion, let us just consider the case of a

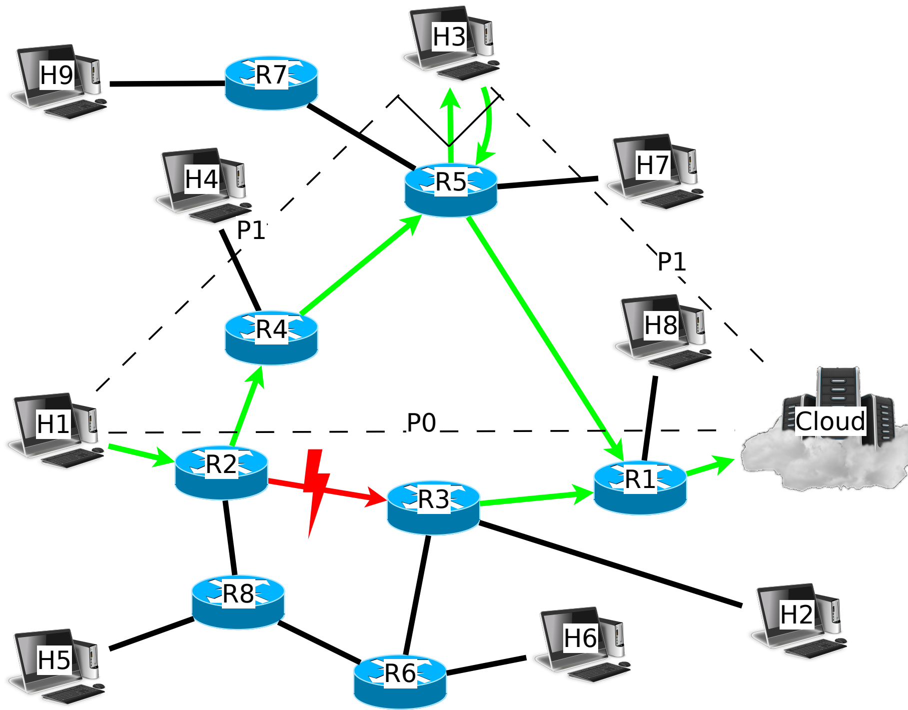


Figure 5.4: A GeoCRON node chooses an overlay based on geographic information.

Algorithm 1: Generalized single-path resilient overlay selection algorithm. Note that GetDiversityScore and GetTieBreaker are *virtual* functions in our implementation, and so their return values depend on the specific heuristic in use.

```
1  Function GetBestPath(s)
2      D* ← 0
3      h* ← NULL
4      for o ∈ O do
5          h ← 2-hop overlay path to s via o
6          D ← GetDiversityScore(h)
7          if (D > D*) or (D = D* and D*! = NULL and GetTieBreaker(h*, h) < 0.0)
             then
8              D* ← D
9              h* ← h
10     if h* is NULL then
11         ThrowNoValidPathException()
12     return h*
```

Algorithm 2: Generalized multi-path resilient overlay selection algorithm

```
1  Function GetBestMultiPath(k, s)
2      ClearCurrentMultiPath()
3      h ← one-hop direct overlay path to s
4      AddToCurrentMultiPath(h)
5      H ← ∅
6      while H.size() < k
7          h ← GetBestPath(s)
8          H.add(h)
9          AddToCurrentMultiPath(h)
10     return H
```

single server s acting as the destination for this data. Let $p_0 = (o_1, e_1, v_2, \ldots, s)$ be this non-overlay direct path to the destination s. If configured with $k > 1$, o_1 will also send out additional packets along geo-diverse overlay routes $P^* = (p_1, \ldots, p_{k-1})$ to s. Clearly it is impractical to assume that o_1 will have knowledge of the failures along overlay path $p_1 = (o_1, e_1, v_2, \ldots, o_i, \ldots s)$ before either attempting the path or receiving some (possibly out-of-band) communications regarding the failures. Therefore, GeoCRON nodes must make the best local decision possible regarding which geo-diverse paths to use.

In our earlier work [29], we tested various heuristics that chose paths $(p_1, \ldots, p_{k-1})$ using only the locations of the overlay nodes. We found little benefit from these heuristics over the baseline (random choice of peer), and so our discussions here consider knowledge of the underlying physical network path p_1 and even the locations of the routers therein. Therefore, when determining p_i's *diversity*, o_1 may consider information such as the components of the path $v, e \in p_1$, the location of each router $v \in p_i$, and the location of the overlay peer choice $o_1 \in O$. By exploiting this knowledge, these heuristics aim to pick paths that are more diverse (less correlated) and therefore improve the resilience of the system to failures.

Below we discuss the implementation of various heuristics (other than *Random* and *Ideal* since they are straightforward) used to choose geo-diverse overlay paths. They have varying degrees of awareness regarding existence and locations of network infrastructure, including

the end devices. In order of decreasing topology and location knowledge, these are:

- *Ideal* (oracle heuristic that finds any working path),
- *Gsford* (router proximity-aware),
- *AreaDistance* (router minimum distance and path area-aware),
- *Intersection* (shared router and link-aware),
- *Random* (uniformly random path choice).

Gsford: This heuristic is derived from some our previous work [113, 114]. In [113], we considered lessening the impact of geographically-correlated failures by picking geo-diverse paths considering whether their routers were within some threshold distance (T_{dist}) of one another. As detailed in Algorithm 13, the very first overlay peer chosen is the one with the lowest latency. All subsequent path choices are compared with all other currently chosen paths and penalized by one point for each router on those paths within T_{dist} of a router on this path. The diversity value is the inverse of this penalty, and so the chosen path will be the one with the lowest penalty score. This technique was applied to a geo-social notification system (GSFord) in [114], where the heuristic gets its name from.

AreaDistance: This heuristic is based on a geodiversity metric proposed in [153]. The goal of this heuristic is to choose physical paths that are as far away from each other as possible. They define the geodiversity of two paths according to:

$$D_g(P_b, P_a) = \alpha d_{min}^2 + \beta A \tag{5.1}$$

Where $\alpha, \beta \in [0, 1]$ are configurable weight parameters, d_{min}^2 is the square of the minimum distance between any two routers in P_b, P_a, and A is the area of the polygon bounded by the locations of all components in P_b and P_a. *AreaDistance* is called to evaluate the geodiversity

Algorithm 3: Path diversity scoring function used in the GSFord system [113, 114]. Paths are penalized for having routers within a threshold distance of a router on another path.

```
1  Function GsfordGetDiversityScore(h, T_dist)
2      H ← GetCurrentMultiPath()
3      if H.size() = 1 then
4          return 1/GetLatency(h)
5      proximity ← 0
6      p_0 ← path(h)
7      for h' ∈ H do
8          p_1 ← path(h') for router v_1 ∈ p_0 do
9              for router v_2 ∈ p_1 do
10                 if dist(v_1, v_2) < T_dist then
11                     proximity ← proximity + 1
12     diversity ← 1/(proximity + 1) // avoid divide by 0
13     return diversity
```

of each currently chosen path and the current possible path choice. For each path, its aggregate geodiversity is chosen as the minimum of those when compared with all of the other paths. The justification for this method is that although a path may be very diverse from another, it could also be extremely correlated with another yet, and so it is geodiverse only in so far as it is geodiverse from *all* currently chosen paths. Ties are broken by taking the choice with the lower path length (number of routers and links).

Path Intersection: This heuristic is also based on a diversity metric proposed in [153]. However, it is not truly a geodiversity metric as it does not consider the physical locations of the topology components. Rather, it simply considers whether or not the two paths share the same components. It measures the size of the paths' intersection, hence its name, and measures the diversity of two paths according to the following metric:

$$D(P_b, P_a) = 1 - \frac{\mid P_b \cap P_a \mid}{|P_a|} \tag{5.2}$$

Where $|P_a| \leq |P_b|$. Note that our implementation takes $min(|P_a|, |P_b|)$ as the factor in the

denominator to ensure $|P_a| \leq |P_b|$. Note also that we must ensure both paths have at least one component each in order to avoid dividing by 0. Just like *AreaDistance*, *Intersection* computes the diversity between the path under consideration and each currently chosen path, assigning the aggregate diversity for this path as the minimum of all these. It also breaks ties by choosing the path of shorter length.

5.4 Experimental Setup

Due to practical considerations with respect to deploying an IoT network within critical infrastructure and testing its performance in an earthquake, we implemented and studied our system in a simulation-based environment. We opted to use the ns-3 [8] network simulator because we wanted to be able to modify the source code to fit our scenario. Below we describe our simulation design and implementation, the code for which is freely available for others to download, use, and modify [1]. We then present the experiments we ran with this simulation and discuss the results.

5.4.1 Simulation Design

To support collecting physical path information, we extended ns-3's *NixVectorRouting* model, which is used for more computationally efficient routing in large networks. This new function returns the physical path that will be used to reach a destination (identified by its IP address).

To read our network topology and sensor locations (see §5.4.2), we created a new TopologyReader model that extends the InetTopologyReader. This new model reads in all of the node and link information the same as for Inet, but it stores the different types of nodes in

[1] The interested reader can find all of our ns-3 code on the *geocron* branch at `https://github.com/KyleBenson/ns3`

different NodeContainers so that we can properly configure each group separately. It also sets the locations of nodes for use in the failure model.

We implemented a GeoCRON Application in ns-3 that attempts to upload sensor data to the server(s) via multiple geo-diverse routes at a specific time. Each Application adds a slight random delay (uniformly random within a range of 1.5 seconds) to this send time when sending packets via the overlay so as to avoid high packet loss rates that we encountered initially due to too many nodes sending messages at the exact same instant. We chose a small delay and to send all multi-path messages at once because the seismic scenario requires low latency. Furthermore, each node would be sending messages at a slightly different time in a real scenario due to e.g. detecting the seismic wave at different times. Overlay forwarding was handled by adding a new header that specifies the IP address of the peer hops, which is used by overlay peers to determine where to forward a packet to. When a server receives a message, it logs this fact in a trace file and responds with an ACK through the same overlay path.

The parameters to consider for an experiment are all specified by command line configuration. These include: disaster location, base failure probability p(fail), number of geo-diverse paths to attempt, which heuristics and their (optional) model parameters, and the number of times to run each unique configuration so as to average results over many different applications of the failure model, random heuristic choices, etc. The simulator iterates over all combinations of specified parameters applying them in a particular order so as to ensure a consistent comparison, with respect to one parameter e.g. application of failure model, between configurations. This lessens the amount of variance we see between treatment groups and gives a more accurate estimation of the underlying distribution(s) that determine the results.

5.4.2 Modeling Community Infrastructure Topologies

Our earlier work [29] considered only the seismic sensor network and generated network topologies from the Rocketfuel [175] and BRITE [131] projects. However, we expanded on this to include the water sensor network and we consider both as participating in the GeoCRON overlay. As such, we now demonstrate a novel method for synthetically generating a communications network topology from the water infrastructure topology as well as place seismic sensors based on population density estimates derived from water customer demand.

We **model** the water sensor network based on the realistic water network provided by EPANet [154, 179], a simulation framework from the Environmental Protection Agency (EPA). It consists of 118 pipelines, 96 junctions (i.e. pipe joints), a pump, a valve, a storage tank, and 2 reservoirs located throughout a 9.26 km $\times$ 7.77 km geographical region. Each junction has its own level of demand (i.e. consumption), and each pipeline has different properties including length, diameter, roughness coefficient, and status (i.e. open or closed). To mimic a real-world setting, we place approximately 50 sensors in this network at sporadic junctions since these interconnection points are more prone to failures [6] and easier to instrument. The sensors periodically send pressure/flow rate values to the cloud service for leak detection. Because modern water networks typically do not have such sensor instrumentation currently built into them, we assume that these sensors are mostly retrofitted into the infrastructure. Therefore, we assume that sensor data collection will be done wirelessly. Because these sensor devices will likely be low-powered and frequently battery-operated, we model the wireless network after a low-power long-range technology such as Sigfox's ultra-narrowband, which we used extensively in the SCALE project. Multiple wireless base stations (BSs) would cover the water network and forward sensor data to the cloud through either the wired infrastructure or direct BS-to-BS wireless links (e.g. microwave antennae).

To **construct** two realistic network topologies (one less redundant and one more redundant)

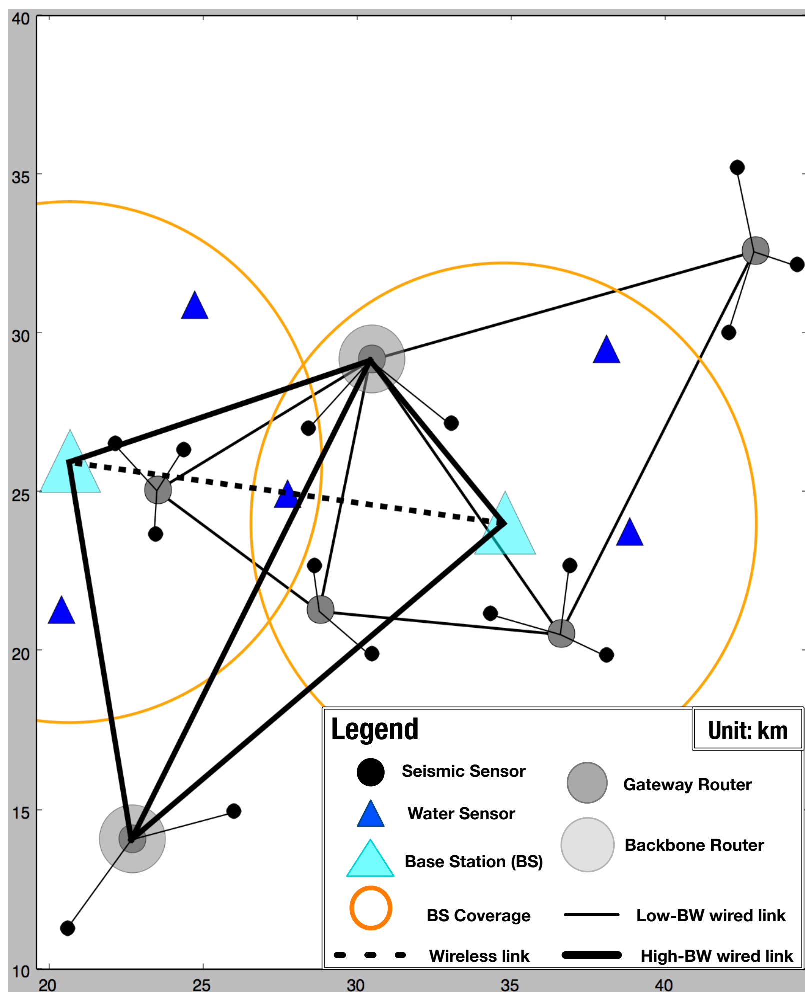


Figure 5.5: Partial network topology as an example to show different node and link types. Other topology plots will follow this legend and exclude the legend for clarity.

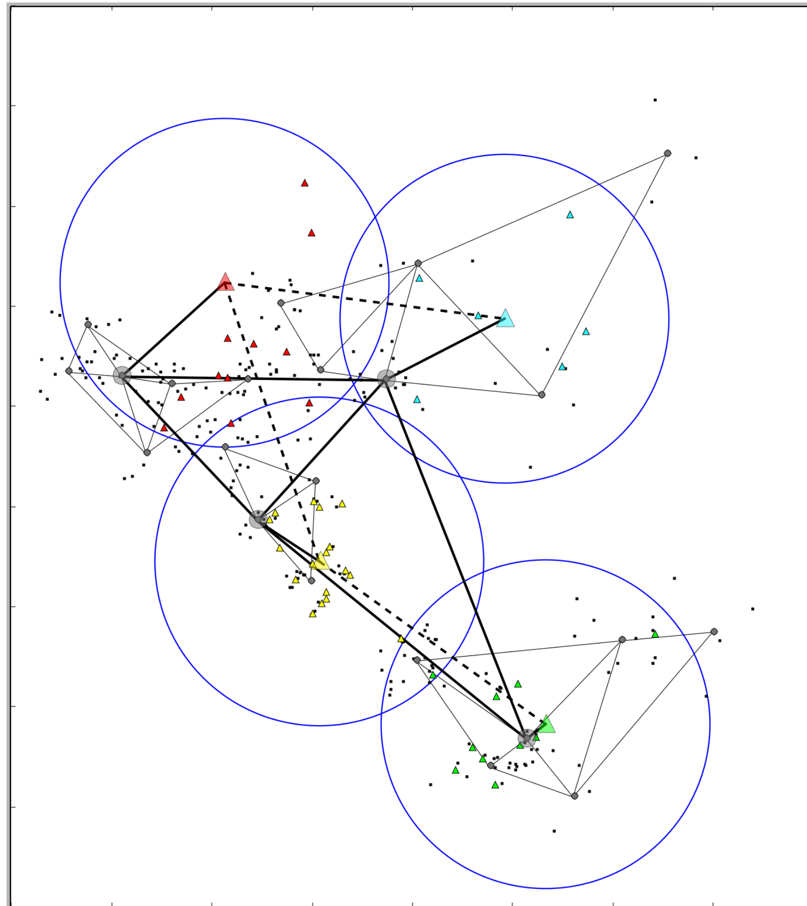


Figure 5.6: The less redundant of our two network topologies.

for this study, we apply the approach given in the IGen project [150]. We started with the sensorized water distribution network described in §5.4.2 as our target geography from which to build communications and seismic networks alongside. By applying real-world network design techniques to constructing synthetic networks [150], we performed the following steps to generate the topologies:

1. Use the water demand values at each junction to derive an estimate of population density around that point. Normalize these demands across the total demand of the whole network to get the density.

2. Use these densities to randomly place 225 nodes representing customer locations (businesses, residential neighborhoods, etc.), each of which have community seismic sensors running on their network.

3. Place 20 additional router nodes based on K-Means with respect to the seismic nodes' locations.

4. Use the techniques adopted by the network topology generator IGen [150], we build a full network topology interconnecting the routers and splitting them into gateway routers and backbone ones. 4 backbone routers are chosen from all the router nodes using K-Medoids. The backbone routers are linked in a full mesh topology, whereas the gateway routers are linked into a sparse mesh using a Delaunay Triangulation. This is an efficient way of obtaining a cost-effective topology with redundancy. It produces alternate paths between nodes, while minimizing the number of such paths [150]. We also removed a few redundant long-haul connections between backbone routers as this would reduce network construction costs.

5. Augment the resulting network to have some long-haul connections outside of the city through a few different paths to represent connections to the cloud service where the data should all be uploaded.

6. Connect each of the customer nodes to the closest gateway routers.

7. Place long-range wireless basestations (BSs). We deploy 4 in the less redundant topology and 7 in the other using K-Means [98] to place each BS at the center of sensor clusters and then move them slightly to ensure full coverage (2km range) over the water sensors. The basestations, which we modeled after the Sigfox ultra-narrowband network we worked with during the SCALE project, are each linked to the closest backbone router via a wired link as well as to the closest basestation via a wireless link. The wireless links between basestations represent microwave links used to link radio towers, especially useful during emergencies when wired connectivity may be affected. We modeled this after Montgomery County, which deploys microwave technology on many its government buildings and radio towers to ensure continuity of operations even during large outages.

8. Placed the cloud data center (server node) far outside target geography because we assume nodes will always try to send sensor data outside of the affected region. This mimics the CSN system, which runs its cloud service outside of California to lessen the possibility of data being trapped within a failed region. The server is connected to 3 of the backbone routers so that there exist redundant options for the overlay to utilize if the default path fails. While we only place a single server node in this topology, it actually represents a connection to the cloud. That is, it represents many servers in different locations, but for the purposes of studying connectivity to any of them we only need a single node.

5.4.3 Failure Model

The failure model assumes an earthquake occurs immediately before each client reports sensor data and that all non-server nodes and all links within the region under study are

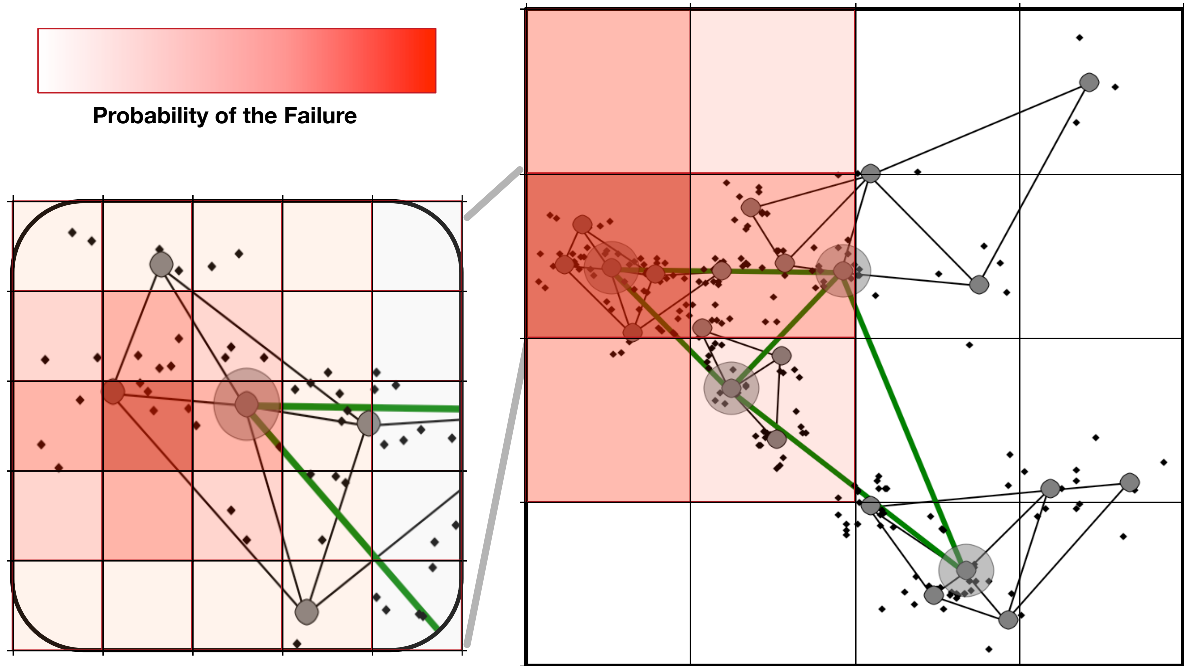


Figure 5.7: The earthquake-inspired failure model used in our simulations.

failed with a particular probability (p(fail)). We assume the server (cloud data center) resides far enough away from the region under study so as to remain unaffected by the disaster. The motivation for this assumption comes from the CSN deployment, which specifies its server instance to run outside of California in order to lessen the chance of data being trapped within the area affected by the earthquake. Our earlier work [29] assumed a uniformly-random failure model due to a lack of fine-grained node locations. Here we have exact node locations and so consider a more realistic version in which nodes closer to the earthquake epicenter are more likely to fail. We loosely base this model on our previous experiences with simulating seismic wave propagation and ground shaking intensity [24, 25].

As depicted in Fig. 5.7, non-server nodes and links fail with a probability inversely proportional to the network component's distance from the earthquake's epicenter according to the

following equation:

$$\frac{p(fail)}{2^{D*S/B}} \tag{5.3}$$

Where p(fail) is the base failure rate input into the simulation, D is the distance from the component to the earthquake epicenter, S is a factor that determines how quickly $p(fail)$ decreases with distance D, and B is the length of the square boundary representing the entire region under study.

All failures happen at the same time (before the Applications attempt data upload), although we are exploring the use of a more sophisticated model that would allow for dynamic evolving failures to represent e.g. propagating seismic waves, secondary failures, aftershocks, and other disasters such as tornadoes, floods, etc.

5.5 Experimental Results

This section describes the results of our simulation experiments. We ran each unique configuration of simulation parameters 24 times. For each unique parameter configuration, we averaged the results of all the runs to lessen the effects of edge cases and better compare the experimental groups with each other.

To quantitatively compare each experimental group, we use the delivery rate of the individual nodes' original sensor data message. That is, a message counts as delivered if at least one copy of it reaches at least one server. This message count is normalized by the number of *active* (non-failed) nodes so that this delivery rate falls in the range $[0, 1]$. The plots below show this delivery rate as a function of time. Note that the starting point of the curves in

these plots represents the performance without the overlay. This is because the messages sent directly to the server go out first and the overlay messages are sent after a short time delay. Note that in the legends each experimental group is labeled with the name of the heuristic and the following parameters in brackets:

- f - failure probability
- k - multipath fanout
- D - distance threshold for *Gsford* heuristic

5.5.1 Comparing Geo-diverse Path Heuristics

Before comparing the various heuristics with each other, we had to identify a somewhat narrow set of parameters to run them with in order to avoid the combinatorial explosion of exploring every possible configuration, which would make each simulation run prohibitively long. We settled on a p(fail) of 0.1, though we also experimented with 0.2, 0.3, and 0.5 (see below). For the multipath fanout, we used $k = 5$ (though we also used $k = 3$, $k = 9$, and $k = 17$ as described below) based on the findings in [153] that some topologies showed strong increases in diversity for $k < 4$ whereas others showed strong increases for $k < 7$.

For *Gsford's* T_{dist} parameter, we ran experiments on the values $T_{dist} \in \{20, 30, 40, 50, 75, 100, 1000, 2500\}$. The *Gsford* heuristic performs better for smaller T_{dist} values, which intuitively makes sense as setting this value to 1 would essentially turn it into an approximation of the *Path Intersection* heuristic, which we show below performs the best. We settled on $T_{dist} = 30$ as performing generally well both in this current experimental setup as well as in some previous studies using different randomly generated topologies.

We also ran experiments on different disaster locations and on both the less redundant

and more redundant topologies we created. The results described below hold across these different parameters as well.

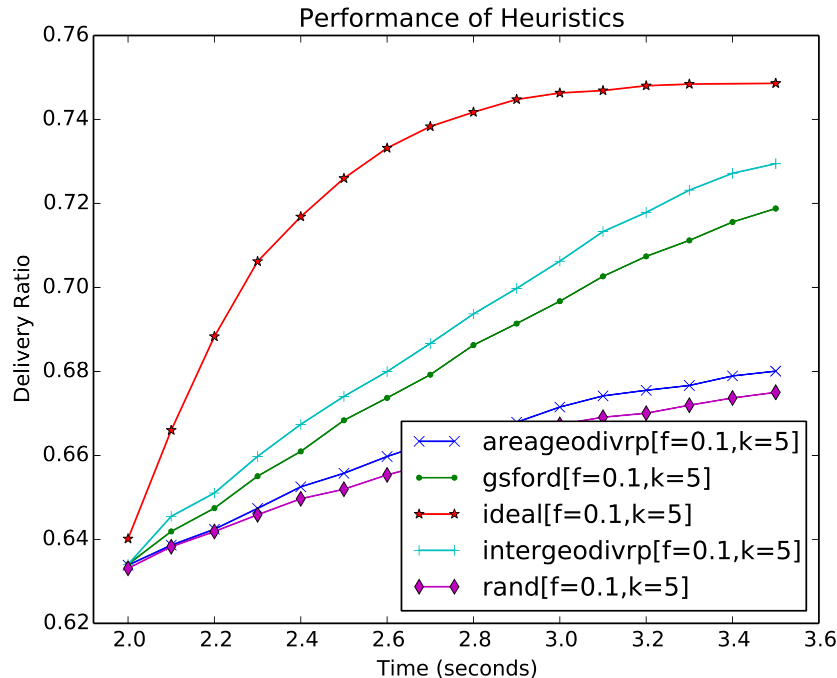


Figure 5.8: The delivery rate for each of the heuristics.

Fig. 5.8 shows the results comparing all the heuristics described in §5.3.2. Recall that the *Ideal* heuristic represents the upper bound on the delivery rate we can achieve with overlay routing. We see that *Gsford* and *Path Intersection* perform similarly, though the latter has a clear slight advantage. It is interesting to note that *Path Intersection* would be easier to implement in a real system as it does not need to know the physical locations of the routing components along a physical path. Because of the above two facts, we chose *Path Intersection* as our best non-optimal heuristic and use it in a few experiments described below. The *AreaDistance* heuristic does not appear to perform particularly well, especially as it is usually matched by the *Random* heuristic, which has the simplest implementation of all.

5.5.2 Comparing Other Parameters

We also ran experiments to explore how the multipath fanout and failure probability affect the delivery rate.

Fig. 5.9 shows the results for using $k \in \{3, 5, 9, 17\}$ with the *Path Intersection* heuristic. We see that using $k = 5$ achieves almost as high an improvement as with higher values, which appears to reproduce the results in [153]. When considering the detrimental effects of too many message copies flowing through an already-challenged network, we believe using a smaller value for k to be ideal.

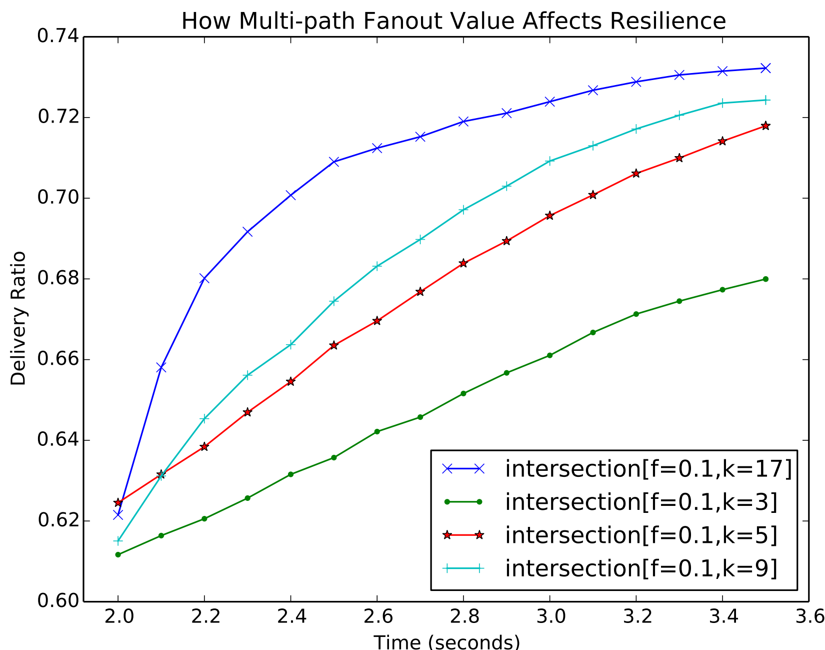


Figure 5.9: The delivery rate for varying multipath fanout values.

Fig. 5.10 shows the upper bounds (using the *Ideal* heuristic) on delivery rates for 1 and 2 servers and $f \in \{0.1, 0.2, 0.3, 0.5\}$. This demonstrates how impactful a slight increase in p(fail) can be on the network's performance. We see that the curves tighten with higher p(fail) values, indicating that the performance improvement becomes less during more failures as fewer working paths are available. When the percentage of network components

failing goes beyond 25% it appears as though the delivery rate expected drops below acceptable values, even with the use of RONs.

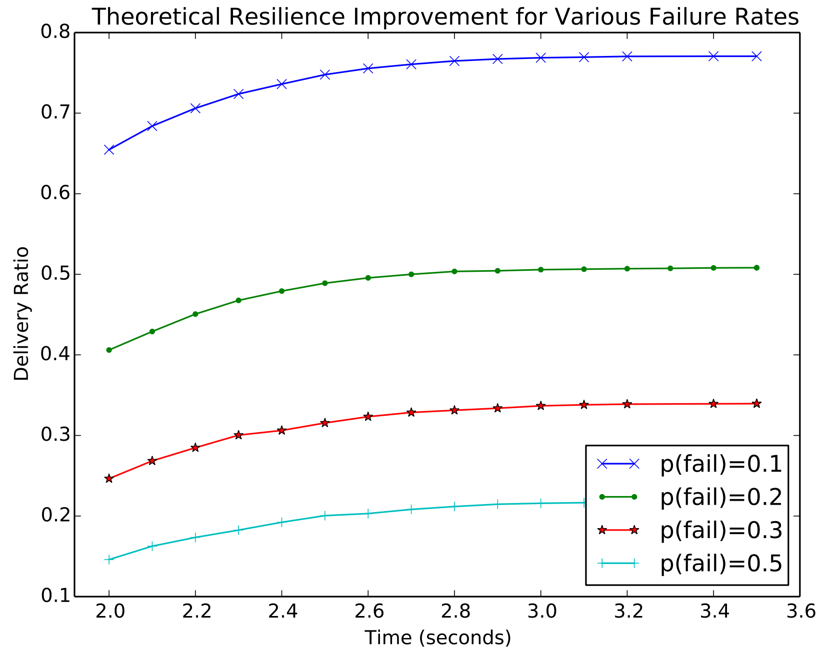


Figure 5.10: The theoretically optimal delivery rate for even higher values of p(fail).

5.5.3 Sharing Network Resources Between IoT Deployments

To study the possibility of both IoT deployments joining the same GeoCRON overlay and sharing network resources, we ran some experiments in which we applied three different treatments of which nodes participate in the overlay: only seismic, only water sensor basestations, and both types of nodes. The previously presented results above are from this last case where all IoT nodes actively participate. For this experiment, we isolated only the results of one network or another (i.e. only seismic nodes both in the case where they were the only ones participating and also when they could use the water sensor basestations as an overlay).

We found that combining the overlays did not actually have a significant impact, either nega-

tively or positively, on the delivery ratio. We believe that this result is due to a combination of several factors. First, the density of the seismic sensor node deployment and the fact the overlay topology is fully connected in the simulations means that the seismic nodes are already capable of finding nearly all of the same paths using only seismic node peers and so the addition of using the basestations does not open up many further possibilities. Second, the structure of our network topology is such that the basestation nodes are already highly resilient and so do not benefit from using the seismic nodes as overlay peers. Because the basestations are connected to each other with long-range wireless links that do not fail when the failure model is applied, they can easily contact each other to route around failures. We actually noted that the use of GeoCRON resulted in achieving a 100% delivery ratio for most scenarios when p(fail) = 0.1. Therefore, we see that the application of GeoCRON does generalize across different types of IoT networks beyond the seismic sensing scenario. Because of these findings, we intend to repeat these experiments with different network topologies and scenarios in the future in order to determine whether these results generalize across further networks and IoT deployments.

5.6 Prototype Implementation

We now discuss practical system design considerations as well as our initial prototype. To begin testing GeoCRON in real-world settings, we created an initial prototype implementation as an extension to the SCALE project described in Chap. 4.

5.6.1 Fully Peer-to-Peer Overlay Considerations

We now consider some practical implementation considerations if one were to implement GeoCRON as a fully peer-to-peer network rather than our proposed centrally-controlled (i.e.

SDN-like) architecture. In order to pick an overlay peer to help create a geo-diverse path to the destination, each GeoCRON client must know about other clients in the network. All GeoCRON nodes must at least know the locations and IP addresses of these peer clients; specific heuristics will request additional information (e.g. physical route to the other node) in order to construct the overlay. As IoT networks can scale to a large number of nodes, it is clearly impractical for each client, which typically runs on a low-powered embedded computer, to maintain this information for every other client in the overlay. Therefore, we must restrict the number of other peers known by each client to a subset of the entire network.

The cloud service instances maintain knowledge of the full network and associated metadata, including underlying route information, to simplify bootstrapping new nodes in the network. When a GeoCRON node comes online, it contacts one of the servers and retrieves a list of geo-diverse peers according to the metrics described in Section 5.3. These peers use tools such as *traceroute* to gather information about the paths between them and both the server(s) and other nodes, including physical routers and link latencies. This data is uploaded to the server(s), where the path geo-diversity algorithms are run to select the best overlay path choices for each node.

The *maintenance* of the overlay would be a crucial consideration if our system involved more *node churn* (e.g. mobile nodes). However, the in situ placement of the sensors means the overlay topology and node locations are expected to remain essentially static except due to failures, in which case nodes are not expected to come back online quickly. Our current architecture thus does not concern itself with overlay maintenance, but we do intend to incorporate mobile nodes and study it in the future. It is well-known that maintaining an peer overlay where each node in the overlay knows about $O(\log(n))$ (n being the number of nodes) other nodes will allow the network to scale to practical sizes. Similar to other overlay-based systems [114, 128], GeoCRON could bootstrap nodes with the aforementioned centralized

approach and then allow nodes to gossip with known peers to refine these initial choices according to some metric. For example, [128] used an simulated annealing-like approach in which connections are chosen to decrease the distance between peers while assuring a low probability of disconnect during random failures by keeping an average node degree. To address more sophisticated and realistic route choice strategies, we are exploring assigning peers based on geo-spatial metrics so that clients are aware of both nearby peers as well as those in diverse areas. We are currently considering three different approaches:

- Assigning peers with a probability inversely proportional to their distance from the client node
- Choosing peers to be as uniformly distributed spatially as possible by breaking the area under consideration into a grid and picking some number of peers from each cell
- Choosing peers based on clusters (a structured approach) by breaking the area into a hierarchical grid and choosing a predetermined number of peers from each cluster

These approaches would guarantee a client knowing about peers that are both nearby and distant. This would allow for a client to contact another peer far away in order to request information about additional peers in this distant region, similar to the method used by Pastry [155] for contacting far away peers based on some key, which in this case would be location.

Furthermore, by introducing such *gossip* mechanisms to GeoCRON in addition to overlay routing, nodes can further coordinate with each other during a disaster to learn about new peers and paths, especially if contact with the server(s) has been disrupted. In this manner, they may share information about perceived failures and known good paths to further improve adaptation to dynamic events. This can be further improved if multiple devices are located within a local area network, especially if equipped with wireless technologies. Note

also that such a mechanism can be exploited to implement rich techniques for in-network processing of sensor readings e.g. data compression, local event detection, and content aggregation.

5.6.2 Extending SCALE with GeoCRON

In our initial implementation, a SCALE client using the GeoCRON overlay feature will contact its SCALE server and request a list of geo-diverse peers. We implemented a location-sensing module to collect the geographic locations of SCALE nodes, which the server uses when running the geo-diverse path algorithms, based on their IP address or a user-specified configuration. The SCALE client middleware contains a group of modules, referred to as *EventSinks*, that handle reporting sensed events to the data exchange using the appropriate networking technologies. The GeoCRON *EventSink* chooses a configurable number of overlay peers, packs necessary information (e.g. server IP address) into a Google protocol buffer (protobuf [4]), and transmits this header along with the sensed event to each chosen overlay peer over UDP. The overlay peers read the header info and forward the packet to the requested server.

To determine the overhead incurred by the overlay routing, we set up an experiment with SCALE devices. We configured a GeoCRON overlay on 6 Raspberry Pis to send sensor data (over Wi-Fi) directly to a laptop acting as the server as well as through each other. The server recorded the timestamps at which the different packets were received so we could determine the difference in latency from the direct message and the overlay message. We ran the experiment a number of times and varied the number of such messages sent (100 and 1000) and the number of hops in the overlay (1-5). The results (see Table 5.1) indicate very low latency increases when using the overlay. The latency increase appears non-linear as we add more hops, but it has high variance, possibly due to effects of operating system

# hops	min	max	mean	stdev
1	0.0002279	0.42008	0.02377	0.02418
2	0.0004408	0.35384	0.04049	0.03435
3	0.0004480	0.29621	0.05441	0.03706
4	0.0006402	0.34069	0.07287	0.04285
5	0.0004800	1.61702	0.12674	0.10012

Table 5.1: Latency difference in seconds between overlay and direct packet for 100 messages.

scheduling. Future work will include setting up an experimental testbed for studying the resilience improvement of using our GeoCRON implementation in a real-world setting as well as repeating the above experiment with devices not on the same local area network.

5.7 Chapter Summary and Discussion

In this chapter we discussed the concept of communications resilience in IoT deployments. To motivate our research, we discussed two IoT systems (seismic and water infrastructure sensing) and the design of a realistic network topology to support them. We proposed the use of *Geographically-Correlated Resilient Overlay Networks (GeoCRON)* for improving these systems' data delivery during a large-scale geo-correlated failure event (earthquake). This middleware runs on IoT systems where inexpensive devices deployed in communities communicate information with remote cloud platforms. We presented and evaluated (in simulations) several heuristics for choosing multiple geo-diverse overlay paths in IoT deployments with varying degrees of knowledge regarding the underlying network topology. We also discussed the design and implementation of an initial prototype system for GeoCRON in the context of the SCALE IoT platform.

5.7.1 Integrating GeoCRON Into Our Proposed Middleware

GeoCRON lays the foundation of our overall middleware approach proposed in this thesis. It provides a resilient cloud-centric data exchange through an application-layer geo-aware resilient overlay network. The prototype implementation described previously incorporates the data exchange logic shown in Fig. 5.11 that runs in the producers and cloud service. However, during severe network outages that cause very high rates of infrastructure failures, GeoCRON's ability to avoid failures decreases dramatically as shown in Fig. 5.10. Hence, this cloud-centric approach alone cannot support such mission-critical applications as earthquake detection and subsequent early warning. Next, we continue building up our proposed middleware by expanding on the seismic scenario with the addition of edge resources to facilitate backup analytics and early warning alerts.

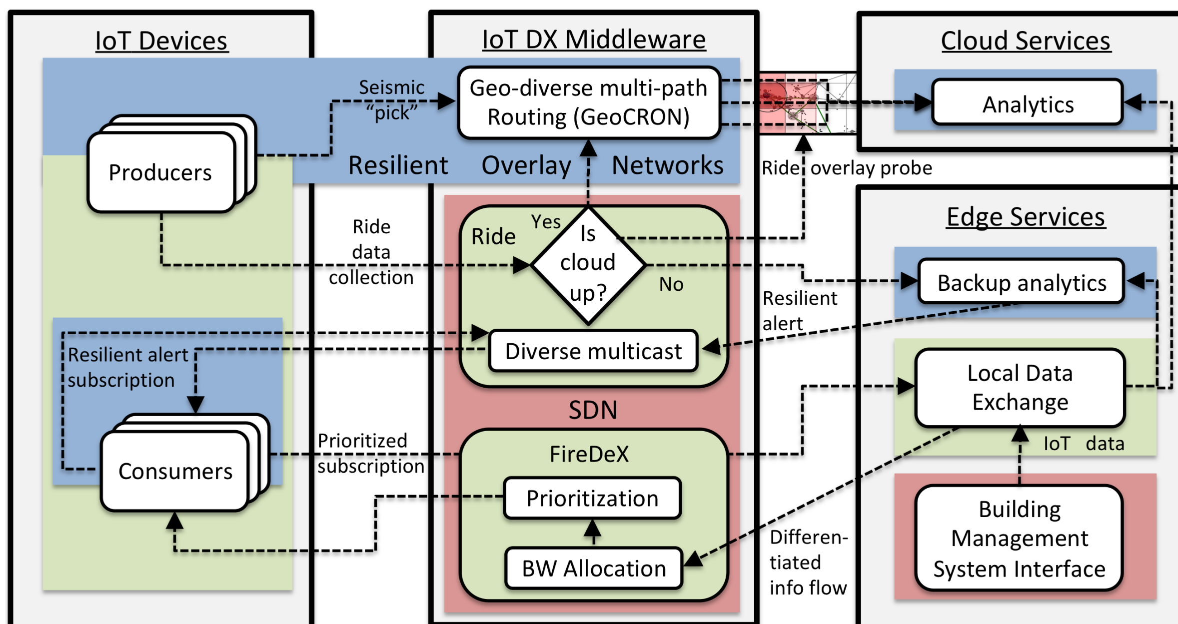


Figure 5.11: GeoCRON adds data exchange logic to the producers for a cloud-centric overlay-based resilient data exchange approach.

Chapter 6

Edge Communications for Resilient IoT Data Exchange

In the previous chapter (5), we explored cloud-centric data collection in the seismic monitoring scenario. However, we also identified the challenge of cloud connectivity instability during such scenarios in §1.3.3. To this end, recall that this thesis advocates for a middleware-based approach to resilient timely IoT data exchange for mission-critical applications without modifications to constrained IoT devices or complete reliance on cloud platforms. Consider this challenge along with the fact that an organization such as a university campus manages its own infrastructure and services. Therefore, such deployments can and should exploit both cloud and locally-managed *edge computing* solutions. In conjunction with capturing **application and network awareness**, this approach further improves our middleware's ability to dynamically configure the IoT data exchange and underlying network in support of mission-critical applications. This chapter explores this concept and also introduces the data dissemination challenge of IoT data exchange. We explore this within the seismic response scenario given in §3.1.2. We expand this scenario in this chapter to include the concept of *early-warning* in contrast to the cloud-centric collection-only approach in the previous

chapter.

6.1 Chapter Overview

In this chapter, we design and develop the **Resilient IoT Data Exchange (Ride)** middleware. It uses edge resources and SDN to gather network awareness and application resilience requirements. It leverages this awareness to pre-configure data flows for reliable operation and dynamically respond to evolving network conditions (e.g. failures, traffic spikes) and critical events (e.g. earthquakes). Ride's novelty lies in its integrated cross-layer approach to enhancing IoT data **collection** from devices and situational awareness **dissemination** to other devices and users (Ride-C and Ride-D respectively). Ride-C employs a novel resource-conserving cloud connection monitoring approach. It probes multiple network overlay paths to the cloud service and, during deteriorated conditions, re-routes IoT data flows through an alternative path or to a backup edge service. This allows seamless operation under both normal and failure conditions. Ride-D pre-configures disjoint local multicast-based alert dissemination paths for edge-mode operation. To this end, it leverages and extends approaches to multicast-based pub/sub discussed in §2.3.3. Its novel path-selection scheme leverages network state information obtained from Ride-C and the SDN infrastructure. By adapting information flows in the IoT system based on application semantics (i.e. resilience requirements) and network state, this unified end-to-end framework bridges semantic gaps between the information and infrastructure layers. In contrast to many of the works discussed in §2, Ride's cross-layer integrated approach leverages both edge and cloud infrastructure, including SDN services. It specifically targets providing mission-critical applications with more resilient and resource-conserving IoT data collection and alert dissemination. Hence, two key aspects of IoT deployments drive our design of Ride: edge computing and SDN.

As previously discussed in §3.5, leveraging edge resources enhances localized situational

awareness. By focusing on the local edge network in an earthquake early-warning scenario (see §6.2.1), we exploit this locality and enhance the data exchange process at the edge during cloud connectivity instability. Ride accomplishes this by leveraging local compute resources and greater flexibility in configuring the underlying infrastructure. It coordinates with edge networking infrastructure that we assume provides SDN capabilities and programmable interfaces.

Ride utilizes SDN APIs to create and maintain *resilient overlays* [16] (see §2.3 and §2.2 for more information). It treats the public Internet routes to the cloud, which we typically have no administrative control over, like virtual SDN links. By configuring the SDN components in the local edge network (where we do have control), this approach ensures cloud connectivity through any available network paths, fail-over to edge backup services during extreme connectivity challenges, and more resilient event routing than traditional approaches. Furthermore, implementing this intelligence in the programmable network infrastructure enables the Ride approach without extensively modifying and adding complexity to constrained IoT devices.

6.2 Our Approach to Resilient IoT Data Exchange

Using a driving earthquake alerting and emergency response scenario, we advocate for the resilient IoT data exchange need and Ride's SDN-based edge computing approach to it.

6.2.1 A Driving Scenario: Smart Campus Disaster Response

Recall the earthquake detection and alerting scenario detailed in §3.1.2. The Ride project focuses on this scenario within a *smart campus* environment: a small community (e.g. university or corporate campus) instrumented with IoT sensors, actuators, and semi-automated

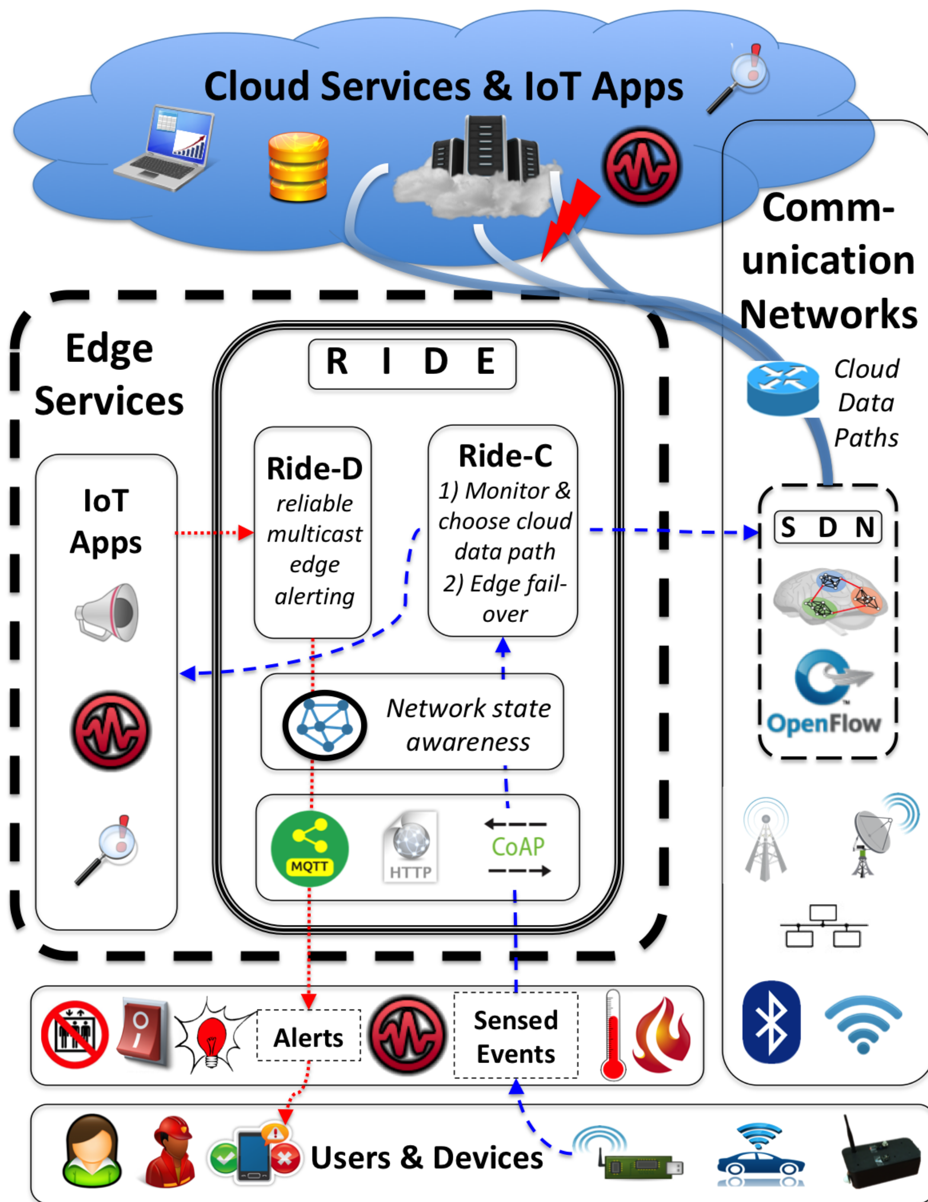


Figure 6.1: The Ride middleware leverages edge cloud resources (without IoT device modifications) for network and application-aware resilient data exchange.

intelligence. Such infrastructure allows the deployment of emergency response applications to improve the safety of community members. To improve reliable seismic data accessibility and alerting during the critical initial seconds of an earthquake, we advocate Ride's combined proactive/reactive approach that leverages all available (i.e. still-functional) resources, especially those at the edge of the network. Ride thus extends a traditional IoT data exchange solution to, when configured with appropriate resilience parameters, support the stringent timing requirements of applications such as an earthquake early-warning system: reliable rapid sensor data collection, event-detection, and real-time alerting.

6.2.2 Ride-enhanced IoT Services for Emergency Response

The geo-correlated nature of seismic events, alert recipients, and related failures in the scenario above illustrates the need and value of managing and processing IoT data flows at the edge in a network and application-aware manner. Therefore, we propose Ride-enhanced alerting service that pre-configures cloud and edge resources to capture and quickly deliver mission-critical sensed events to the public cloud service for regional emergency response coordination. In response to public cloud connectivity issues, it redirects this data to edge services for rapid and reliable generation of local awareness until such connectivity is restored. We treat edge services as logically-centralized, although they can be physically-distributed. Hence, edge services remain available during emergencies; future work will coordinate multiple edge instances to handle service failures. We can also envision a city-scale early-warning system as a hierarchical network of collaborating edges (i.e. systems-of-systems), each of which independently manages their own networks and edge infrastructure.

In designing Ride's architecture (see Fig. 6.1), we adopted a practical approach that considers IoT deployment characteristics and constraints derived from our previous experiences. Our primary design philosophy, avoiding modifications to constrained IoT devices and associated

protocols (e.g. CoAP and MQTT), led us to implement fail-over functionality using edge services and SDN rather than e.g. device-chosen broker fail-over due to timeouts. This also encouraged a protocol-agnostic design that extends in multiple ways the abilities of traditional messaging-layer IoT data exchange protocols, thereby easing adoption by existing deployments. Thanks in part to SDN, we designed Ride's technology-agnostic approach to exploit physical (route) redundancy in ensuring resilient data capture and delivery. This includes leveraging heterogeneous networking technologies: local wired/wireless, Internet overlays, long-range wireless such as LoRa/SigFox, and cellular, which is often congested during earthquakes.

Note Ride's generic design applies in other emergency response scenarios (e.g. tsunamis, wide-spread fires, terrorist attacks, etc.) to maintain time-and-mission-critical services during wide-area infrastructure failures, albeit with slightly less-stringent resilience requirements. Therefore, we treat application-specific analysis techniques (e.g. earthquake analysis) as black boxes. We focus instead on the following two-step process of resiliently **collecting** sensed events and **disseminating** alerts (Ride-C and Ride-D). This jointly enables a unified resilient framework while separating concerns of where to collect and process data from how to disseminate resulting alerts.

6.2.2.1 Ride Data Collection (Ride-C)

Ride-C configures resilient IoT publisher-to-data exchange event collection flows. It tracks and adapts to local or cloud failures and determines whether further processing should occur at the cloud or edge. Our approach monitors current network state and available communication paths to determine when and how to collect and propagate sensed events from IoT devices to cloud services when available or edge services when not. It captures network state awareness and embeds it in the IoT workflow using an SDN controller's APIs to manage physical (or virtual) SDN-enabled switches.

Ride-C creates and manages resilient overlays: multiple Internet paths from local gateway routers to the cloud that administrators typically have no direct control over. We treat each overlay path as a virtual SDN link and refer to it as a CDP. To avoid complicating and burdening resource-constrained IoT devices, Ride-C monitors the cloud connection itself from the edge by probing each CDP (i.e. similar to ping). We use a custom UDP datagram containing a sequence number and timestamp for the probe rather than ICMP echo requests since service providers' firewalls often block ICMP packets. This further enables directly detecting a cloud service process's status as it may have crashed while the cloud server VM still replies to ICMP requests. The probe travels through its assigned CDP to a simple cloud echo server and then back to the Ride-C service. There a control loop analyzes the probes' Round-Trip Time (RTT) to gather network metrics (e.g. link latency, packet loss) and determine if a particular CDP should be avoided due to failure or congestion.

Upon detecting such problems, Ride-C responds by failing over to an alternative cloud path or redirecting to edge services transparently to IoT devices. For simplicity, we assume a *first feasible* path policy using a strict ordering of CDP preferences to maintain cloud connectivity when possible. We leave out of scope the complex question of determining CDP preferences in terms of: cost, network administrator policies, the interplay of multiple applications simultaneously vying for resources, etc. Our novel *adaptive active network probing* technique minimizes overhead while accounting for application-specified resilience requirements (e.g. failure detection time). Directly querying the SDN switches' packet counters to calculate packet loss rate could not provide this level of control and configurability. Nor could it detect but gracefully account for changes in the CDP's underlying physical routes as evidenced by a significant change in latency or jitter.

6.2.2.2 Ride Data Dissemination (Ride-D)

Ride-D uses an unmodified cloud data exchange when possible or resilience-enhanced edge alerting during periods of cloud connection instability (i.e. Ride-C redirected sensed events to the backup edge service). SDN enables Ride-D's novel network-aware multicast-based group communication mechanism for reliable alerting. Before a failure/congestion event, it configures the SDN data plane with multiple pre-constructed Maximally-Disjoint Multicast Trees (MDMTs) (see Fig. 6.2b). At alert time, Ride-D leverages up-to-date local network awareness embedded in the data exchange workflow by Ride-C (i.e. publication routes) and itself (i.e. subscriber responses to previous alerts) to intelligently choose from these multiple component-diverse physical path choices. Because of the time-critical nature of *alerts*, it must quickly select the ideal MDMT and therefore avoids online querying of the SDN control or data planes.

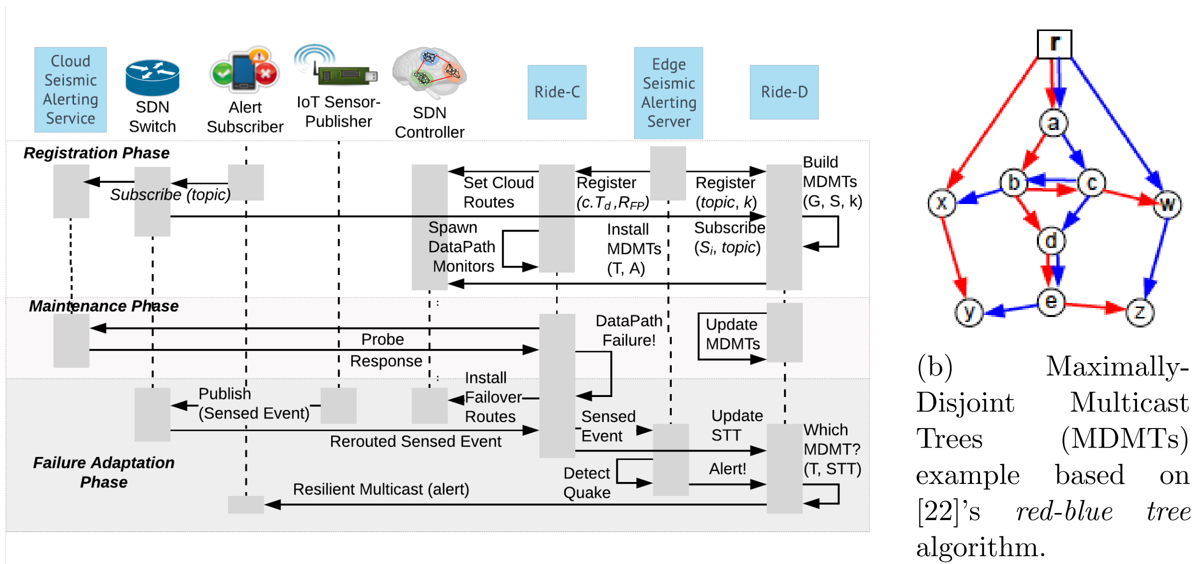
Similar to a few other recent systems [73, 11], Ride-D utilizes the logically centralized control plane and programmable data plane of SDN in conjunction with a pub-sub broker to translate the pub-sub paradigm into network-level multicast. However, it does so to enable resilience in a manner transparent to the client IoT devices and requiring only a thin middleware layer at the edge server application. The only data exchange protocol requirement for Ride-D is network-layer multicast support; §6.4 discusses supporting different protocols. We chose to use network-level multicast rather than an application-layer reliable multicast mechanism in order to improve resource efficiency (i.e. minimal packet duplication and bandwidth usage). In edge environments infrastructure cost constraints (e.g. bandwidth and thin IoT device clients) and challenges introduced by temporary emergency scenarios may prohibit purely-unicast-based alerting. Furthermore, maintaining alternative paths for each alert subscriber, as opposed to each alert group, increases system state and overhead (e.g. data structures, SDN flow tables, and maintenance thereof). Because the preconfiguration phase uses SDN's centralized control plane, it can create and use more resilient multicast trees than those

made by many traditional techniques (e.g. PIM-SM [65]) that build single distribution trees on-demand in response to join requests from subscribers and publishers. This flexibility can also incorporate additional application requirements, e.g. bandwidth guarantees, cost metrics, prioritization, etc. although we leave these out of scope.

To further improve alert reliability, Ride-D can intelligently retransmit alerts that have not yet reached all subscribers. It retries sending the alert after a configurable timeout period as long as some subscribers remain unreached and the number of retries has not exceeded a specified maximum. To facilitate this retry mechanism, it maintains an *alert context* for a given message and topic to track the necessary state: which subscribers have not yet been alerted, which MDMTs were recently used, and how many more retries remain. As each subscriber acknowledges successful receipt of the alert, Ride-D updates the alert context so that these retries ignore that subscriber and any MDMT links that only serve it. It also updates the STT with the response's route back along the used MDMT. Retries ignore the most-recently-selected MDMTs so that e.g. if some were used $x + 1$ times for this alert only those used x times will be considered. This extension to the approach originally presented in [30] improves path-selection diversity and helps overcome the inaccuracy of our network state estimation approach described in §6.3.1.2. When most subscribers have been alerted, Ride-D may switch over to contacting the remaining ones individually via unicast for more precise path selection. Note that we leave the implementation and study of this retransmission extension for future work due to the outstanding implementation challenges we discuss later in §6.4.

6.2.3 Ride Workflow

Ride's workflow executes at the network edge in three phases (see Fig. 6.2a): 1) a priori host registration and network configuration; 2) on-line network state analysis and maintenance;



(a) Ride's workflow consists of three phases shown here as differently-shaded regions starting at the top.

(b) Maximally-Disjoint Multicast Trees (MDMTs) example based on [22]'s *red-blue tree* algorithm.

Figure 6.2: Ride's resilient IoT data exchange workflow and diverse multicast tree-based alert dissemination.

3) event-time failure-detection, adaptation, and alerting.

First, Ride **registers and configures** the participating hosts and network components. It exposes an API (e.g. as an SDN controller northbound API) for the Ride-enabled edge service to register: 1) its application resilience requirements and the available CDPs with Ride-C; 2) its time-critical alert topic (e.g. "seismic-alert") and the desired resilience level (number of MDMTs) with Ride-D. Each IoT subscriber/publisher sends a normal subscription/advertisement message that the SDN data plane forwards to both the unmodified pub-sub broker and the Ride edge service. Working with the SDN controller's APIs, Ride processes this information to set up resilient data collection and alert dissemination routes from/to the relevant publishers/subscribers as detailed in §6.3. This includes configuring SDN switches (be they physical hardware or software implementations) for Ride-C's CDP probing/monitoring mechanism, its publication collection routes, and Ride-D's MDMTs.

Second, Ride **maintains these configurations** in the online phase: it recalculates routes

and updates flow rules in response to network dynamics e.g. topology changes, evolving traffic patterns, handling (un)subscribe requests from clients, etc. It monitors the CDPs for potential failures and gathers network state awareness during data collection as described in §6.3.1.

Third, Ride **adapts to failure events** to maintain service availability. Upon detecting a CDP failure, Ride-C redirects cloud data exchange traffic through a different CDP if one is still available or to the edge server if not. In the latter case, *address translation* allows constrained IoT hosts to remain unaware of this change and seemingly continue publishing data to the cloud network address (i.e. IPv4). The SDN switches translate this destination address from that of the cloud server to the edge's, route data packets to the edge, and translate the source address of replies back from the edge server address to that of the cloud seamlessly. When the CDP recovers, Ride-C reverts this redirection and return to normal cloud operation.

During fail-over to edge services, Ride-D enables network-aware alert dissemination at the edge. It selects the best of its pre-configured MDMTs, thereby improving resilience to local failures and conserving limited network resources. Alert packets are sent to a network address (e.g. IPv4) assigned to the selected MDMT. SDN data plane switches forward packets matching that address along the computed dissemination routes. We also use address translation here to avoid requiring complicated multicast configuration and software support on constrained IoT subscribers. The last hop SDN switch translates the packets' destination address into that of the subscriber so that the alert appears as a unicast message from the server. Our current implementation (see §6.4) uses OpenFlow's [130] *flow rules* for packet forwarding/address translation and *group tables* for multicast. However, the Ride paradigm could utilize alternative SDN technologies, addressing schemes other than IPv4, and even incorporate non-SDN switches using tunneling.

6.3 Ride Algorithms

This section details, in the context of its aforementioned three-phase workflow, Ride's novel techniques for network and application-aware resilient event collection from IoT publishers and dissemination of critical alerts to locally-interested users and actuating IoT subscribers.

Refer to the following notation for the algorithms outlined here. **Ride models the network topology** as an undirected graph $G(V, E)$ with vertices (network switches, routers, and hosts) $V(G)$ connected by links $E(G)$. A route traversing link e incurs a weighted cost w_e (e.g. bandwidth, power consumption, routing table entries). We denote the set of *sensor-publishers* as P, the *subscribers* interested in receiving *alerts* as S, the *cloud service* as $c \in V$, the Ride-enabled edge service as $r \in V$, and the MDMTs as a set T where $k = |T|$ and $\forall T_i \in T, \{r\} \cup S \subset T_i \subseteq G$. We model the CDPs as a set of virtual links $D = \{e \in E(G) : e = (c, y)\}$ for the Internet-connected gateways $y \in V(G)$.

6.3.1 Ride-C – Data Collection in Ride

6.3.1.1 Configuring resilient data collection

Ride-C first selects the primary CDP and configures resilient data collection routes through it. In IoT alerting systems, multiple co-located sensors may generate and send similar sensed events to the server during an emergency. Therefore, we consider a data collection approach for preferring that at least some of these publications can be used for emergency event-detection rather than emphasizing collecting all of them. We compare two policies for building routes and associated flow rules from each registered publisher $p \in P$ to the assigned gateway router y: 1) **shortest path** finds the absolute shortest path (in terms of w_e) between p and y; 2) **diverse path** finds maximally-disjoint paths (i.e. they share a minimum number of common nodes/links) from each $p \in P$ to y, although it prefers shorter

ones when considering equally-disjoint paths.

The latter method exploits topological redundancy in the network to increase the reliability of IoT data collection due to multiple sensed events traversing the same failed link being less likely. See §2.2.2 for a discussion about about the challenges in computing diverse paths and details about the algorithm proposed in [78] that we use to accomplish this for $k > 2$ in polynomial time. Because this algorithm finds multiple paths between two vertices, Ride-C first adapts G by adding a new *virtual node* v^d and edges between each $p \in P$ and v^d, using v^d as the new source node for [78]'s algorithm. Note that this approach does not guarantee that each publisher gets some route to the data exchange. For those that do not, we simply add the shortest path to ensure complete connectivity as expected.

Ride-C then configures the CDP monitoring mechanism (see Alg. 4) for each registered CDP. To optimize resource consumption, it minimizes probing frequency overhead while meeting the application-specified requirements of 1) maximum detection time T_D and 2) failure/congestion-detection false positive rate upper bound, R_{FP}.

Ride-C initializes this process with a learning phase in which it analyzes the CDP's steady-state condition to calculate the proper adaptive probing parameters: interval I and timeout T_o. In this phase, it sends a new probe as soon as it receives the last reply or times out after T_D. Upon gathering enough acknowledgements, it calculates the CDP's packet loss rate P_l and average RTT, RTT_a. We define the requested false positive rate $R_{FP} = (P_l)^N$ as the probability of N consecutive packet losses. Given these parameters, Ride-C calculates the minimum number of sample probes $N_B = \lceil \log_{P_l} R_{FP} \rceil$ it needs to collect before marking a CDP congested or failed. It then concludes the initialization phase by setting the initial probe interval to: $I = \frac{T_D}{\lceil \log_{P_l} R_{FP} \rceil}$.

Algorithm 4: Ride-C Probing and Adaptation

```
1  while True                                          // On-line Adaptive Probing
2      Send a probe on CDP
3      if the acknowledgement is received within T_o then
4          Update sliding window with new RTT
5      else
6          Update sliding window with packet loss indicator
7      P_l, RTT_a ⟵ Calculate new metrics in W
8      if RTT_a > I or last N_B elements in W are all packet loss indicators then
9          return UNAVAILABLE
10     else
11         N_B ← ⌈log_{P_l} R_FP⌉
12         I ← T_D / N_B
13         T_o ← 2 * RTT_a
14         Wait I
```

6.3.1.2 On-line maintenance of network state awareness

As shown in Alg. 4, Ride-C continues updating its resource-conscious application-aware parameters in the steady-state. It revises the CDP's estimated RTT, RTT_a, using an exponential moving average method with a smoothing factor of 0.8, which we chose based on TCP's round-trip time estimation [163]. Ride-C sets the probe's timeout $T_o = 2 * RTT_a$ to ensure it meets the T_D requirement. Upon receiving probe acknowledgements or timeout events, it updates the CDP's packet loss rate and then probing interval as before. Ride-C detects possible failure or congestion as evidenced by N_B consecutive timeouts or significantly increased latency: $RTT_a > I$. It cannot detect failures and mark a CDP *unavailable* within T_D while satisfying R_{FP} if $RTT_a > I$ due to not collecting enough samples within T_D.

During edge mode operation, Ride-C continues CDP monitoring but also estimates the currently-functional local network topology from sensed events collected at the edge. Rather than (or in addition to) waiting for control plane updates derived from link-level failure detection in the network data plane, it leverages its own data plane activity for an *online link*

Table 6.1: Parameters Used in Ride-C

Category	Parameter	Description
Application Specified	T_D:	Maximum detection time
	R_{FP}:	Maximum false positive rate
DataPath Metrics	P_l:	DATAPATH packet loss rate
	RTT_a	Exponential moving average of probe round-trip-time
Detector Parameters	N_B	Min. # samples collected before detector determines CDP state
	I:	Probe interval
	T_o:	The timeout of the probe

state estimation technique. This complements existing network resilience techniques (e.g. packet retransmission) within a distinctly IoT setting by leveraging application-awareness for a time-critical collect-and-disseminate data exchange solution. Ride-C matches recently-collected events with its pre-configured sensor-publisher routes. It adds each of these routes to a graph data structure called the *Successfully Traversed Topology (STT)* that it continually maintains to represent the network components recently (within $\approx$ 2sec.) verified as functional. Note that these *STT* node/link states are non-definitive estimates of the current state: presence in the *STT* could indicate a recently-functional but now-failed component, while absence could have no significance. By embedding this estimation in edge service-bound data flows as incremental updates to the shared *STT*, this cooperative method enables Ride-D to leverage Ride-C's network state awareness to improve resilient local alert dissemination as described later.

6.3.1.3 Active fail-over adaptation

Ride-C responds to a CDP disruption by triggering a fail-over mechanism. It determines: 1) what fail-over actions to perform upon CDP state changes and 2) what flows to generate and push to the SDN-enabled switches for implementing these actions in the physical network. If another CDP remains *available*, Ride-C redirects IoT data collection through it by adapting the SDN data plane as described in the initialization phase. In the case that all the CDPs

are marked *unavailable*, Ride-C will redirect sensed events from publishers to the edge server. It builds these redirection routes and their associated flow rules using the same policies as for CDP redirection, except with the edge server r as the destination instead of a gateway switch y. After this fail-over, Ride operates in edge mode and leverages Ride-D for resilient local alert dissemination.

6.3.2 Ride-D – Data Dissemination in Ride

We now define Ride-D's algorithms for network-aware reliable multicast-based alert dissemination. The Ride middleware uses Ride-D to disseminate *alerts* to locally-interested *subscribers*. For resilient (i.e. to failures and congestion) *alerting*, Ride-D configures the k MDMTs T to share a minimal number of edges/vertices as shown in Fig. 6.2b and discussed in §6.3.2.1.

6.3.2.1 Background on Multicast Tree Construction

The classical problem of constructing a single multicast tree T_i of minimum cost that includes the root and every subscriber (i.e. $\{r\} \cup S \subset T_i$ is referred to as the *Steiner tree problem*. This problem is NP-Hard in general for general graphs, and so heuristic-based algorithms that provide bounded approximations have been proposed [104]. Despite this NP-Hardness, disjoint path research [61, 22] found that a pair (i.e. $k = 2$) of edge-disjoint directed spanning trees can be found in polynomial time, even in a distributed manner. This produces "edge-disjoint directed spanning trees for a 2-edge-connected digraph". These maximally-(edge and vertex)-disjoint trees, rooted at a distinguished node (i.e. r) and spanning all of V, guarantee recovery from a single non-cut node or link failure. Clearly, our earthquake-induced failure scenario would cause > 1 component failures. Hence, we explore networks with greater connectivity and > 2 maximally-disjoint (i.e. they share a minimal number of

edges/vertices) trees for reliable multicast.

6.3.2.2 Configuring MDMTs a priori

We briefly describe the MDMT-construction algorithms below and invite the reader to find more details in the respective references and performance comparisons in §6.5.4:

• ***steiner*** approximates the Steiner trees using the somewhat naïve approximation method described in [104]. It finds the minimum spanning tree of the metric closure subgraph. That is, it converts path lengths between terminal nodes into edge weights for a new graph (i.e. a metric closure subgraph) and finds its minimum spanning tree (MST). Each iteration finds one MDMT and increases the used edges' weights (by either doubling the weight or adding the max weight of all edges) to disincentivize their use in the next iteration. We adopt two versions of this technique for our comparison heuristics by 1) doubling ($w'_e = 2w_e$) or 2) adding the max weight ($w'_{e_1} = w_{e_1} + \max_{e_2 \in E(T_i)} w_{e_2}$). This algorithm has an approximation ratio of $2 - \frac{2}{|S \cup r|}$.

Runtime complexity: $O(|S|(|E| + |V| \log |V|))$.

• ***diverse-paths*** iteratively adds each subscriber $s \in S$ to the MDMTs, ordered by the minimum-path distance from r. Each iteration generates k maximally-disjoint paths from r to s using the same diverse path-finding algorithm [78] as Ride-C's diverse path routing policy. The k paths to the first s form the initial MDMTs. It selectively adds each path to one of the k MDMTs with which it has maximal overlap. Hence, This greedy approach aims to form the trees with lower total cost while maintaining disjoint paths. Because this can result in a non-tree graph, we first find the MST and then iteratively remove all non-terminal vertices of degree ≤ 1 until only necessary edges and vertices remain.

Runtime complexity: $O(k(|E| \log k + |V| \log |V|))$.

- ***red-blue*** incorporates the concept of *red-blue trees* shown in Fig. 6.2b. Recall from §6.3.2.1 that this approach finds $k = 2$ edge-disjoint directed spanning trees in polynomial time [61, 22]. We adopt the *SkeletonList* data structure and algorithm proposed in [22]. It colors *every* edge (not just a tree) in the graph red and/or blue (cut edges are colored both red and blue) in $O(|V| \cdot |E|)$ time. This more efficiently handles topology updates (i.e. dynamic adding and removal of nodes and links). Faster ($O(|V| + |E|)$ algorithms [61] must be fully re-computed after topology updates since they color just those edges in the spanning trees. This coloring partitions G into two maximally-disjoint directed acyclic graphs (DAGs) with respect to the edges. For $k > 2$ (k a positive power of 2), we recursively apply the procedure on the resulting red and blue graph to greedily further subdivide the graph. Note that SkeletonLists enforce directionality on edges, and so we must convert our undirected graph into a directed one first. To compute a Steiner tree from one of these DAGs, we combine the shortest paths to every $s \in S$ into the tree before finally converting the graph back into an undirected graph for use as a multicast tree.

Runtime complexity: $O(k|V| \cdot |E|)$.

6.3.2.3 On-line MDMT maintenance

Ride-D modifies MDMTs in response to network topology/state and subscription updates. Note that we leave the challenge of minimizing MDMT modifications (i.e. to reduce overhead from forwarding plane changes) as out of scope. Interested readers might look closer at [22] for an example of how to handle dynamic topology changes (e.g. dynamic adding/removal of nodes/links/subscribers) using the SkeletonList data structure. We instead assume a simple approach of completely recomputing the MDMTs and focus our contributions on intelligent MDMT-selection as described next.

6.3.2.4 Event-time failure response

Alg. 5 details Ride-D's alerting mechanism. We now describe how its network state and failure-aware **MDMT-selection policies** leverage our novel link-state estimation technique (STT) to determine each T_i's suitability for delivering the alert despite recent failures. We empirically compare these policies later in §6.5.4. Note that the algorithms in Alg. 5 incorporate the intelligent retransmission mechanism described in §6.2.2.2. With this additional mechanism, each alert response from a subscriber is used to update the STT with the route back along T_i. Note that the policies' objective functions break ties randomly. Furthermore, they prefer alerting unreached subscribers by considering each T_i as the MDMT with already-reached subscribers and their unique path links removed.

• ***min-missing-links*** selects the MDMT having the fewest links not present in STT. This policy therefore aims to avoid failed links as possibly indicated by their absence from the STT. It also prefers smaller trees, which it uses to break ties.

Objective function: $-|\{e \in E(T_i), e \notin E(STT)\}|$

• ***max-overlap-links*** selects the MDMT sharing the highest proportion of its links in common with STT, thereby decreasing the likelihood of failures along the MDMT. Note that we scale by $|T_i|$ (i.e. calculate a proportion rather than a discrete total of overlapping links) to alleviate a preference for larger trees. Due to Steiner trees spanning a subset of the graph (each MDMT contains possibly different non-terminal nodes), it differs slightly from *min-missing-links* because of this scaling. These policies also make different selections because of the STT's inherent uncertainty mentioned previously: preferring known good links vs. avoiding potentially bad ones.

Objective function: $\dfrac{|\{e : e \in E(T_i), e \in E(STT)\}|}{|E(T_i)|}$

• ***max-reachable-subscribers*** considers complete paths rather than individual links. It

selects the MDMT that can reach the most subscribers assuming only the links in STT are up. Again, the STT's uncertainty means this assumption may lead this policy astray.

Objective function: $|\{s \in S : PathExists(STT \cap T_i, r, s)\}|$

- ***max-link-importance*** combines the STT-uncertainty-avoidance of *max-overlap-links* with the complete path consideration of *max-reachable-subscribers.* It selects the MDMT whose intersection with STT has the highest total *link importance* (i.e. the number of paths from the root to the subscribers that traverse that link). Note that an implementation should pre-compute each edge's importance, which takes $O(|T_i|)$, to improve run-time performance. Also note that we scale the objective function by the total possible link importance to avoid preferring larger trees. Furthermore, an implementation could easily incorporate the notion of heterogeneous *priority* for different subscribers by assigning different importance values to their respective links.

Objective function: $\dfrac{\sum_{e \in (E(T_i) \cap E(STT))} |\{s \in S : e \in GetPath(T_i, r, s)\}|}{\sum_{e \in E(T_i)} |\{s \in S : e \in GetPath(T_i, r, s)\}|}$

While we omit the formal proof, each metric essentially computes the intersection of T_i and STT in linear time. Although they use this result differently, each implementation has a **runtime complexity** of $\mathbf{O(k(|T_i| + |STT|)}$).

6.4 Prototype Implementation

To demonstrate Ride's improvement to an IoT data exchange's resilience, we developed a prototype implementation and proof-of-concept testbed in our lab. We implemented the core Ride algorithms and integrated them with our SCALE [27] IoT middleware (see §4.2 for details) to use as the edge alerting service. This complete prototype implements the proposed architecture (Fig. 6.1) by leveraging RESTful CoAP APIs to manage the workflow

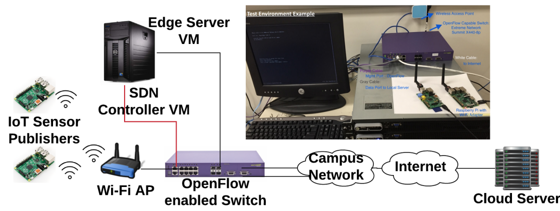
Algorithm 5: Ride-D network-aware multicast alerting algorithms for the configuration and alerting phases.

```
1  Function ConfigureMDMTs(S, topic, r, k, G, algorithm)
2      T ← BuildMDMTs(algorithm, G, S, r, k)
3      for T_i ∈ T do
4          addresses ← InstallMulticastTreeFlowRules(T_i)
5          for s ∈ S do
6              InstallResponseFlowRules(s, r, GetPath(T_i, s, r)
7      RegisterMDMTs(T, topic, addresses)
8  Function SendAlert(msg, topic)
9      Metric ← MDMT selection policy objective function
10     alert ← MakeAlertContext(msg, topic)
11     retries ← GetMaxRetries(topic)
12     while UnreachedSubscribers(alert) ≠ ∅ or retries > 0
13         S ← UnreachedSubscribers(alert)
14         for T_i ∈ GetMDMTs(topic) do
15             M_i ← Metric(S, GetRoot(T_i), T_i, GetSTT())
16         M*, T* ← max {(M_i, T_i) : i ∈ [1..|M|]}
17         address ← GetAddressForMDMT(T*)
18         SendMulticast(msg, address)
19         retries--
20         RecordMDMTUsed(T*, alert)
21         RegisterAlertResponseCallback(OnSubscriberACK, alert)
22         wait(timeout)
23 Function OnSubscriberACK(s, alert)
24     MarkSubscriberReached(s, alert)
25     p ← GetPath(MdmtUsed(alert), s, GetServer(alert))
26     UpdateSTT(p)
```

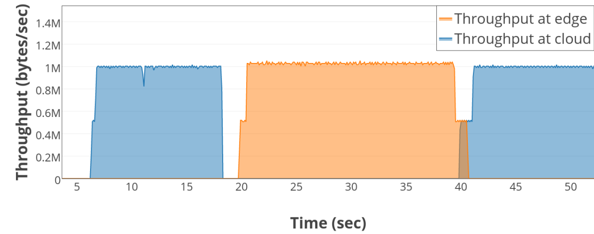
describe in §6.2.3, Fig. 6.2a. We invite the reader to try out Ride and find more details in our source code repository: `https://github.com/KyleBenson/ride`.

The earliest Ride prototype, just like the initial SCALE prototype, forwarded events to a cloud MQTT broker. If unavailable, Ride redirected these data flows to an edge MQTT broker, which required the client's network stack to detect a change in the underlying TCP state machine and re-connect with the new broker. The latest prototype described below instead prefers the UDP datagram-based protocol CoAP, integrated via CoAPthon [72], in order to support connection-less RESTful interactions for constrained IoT devices. This interaction style simplifies OpenFlow-based redirection of sensor-publishers to alternative CDPs or edge services and also enables Ride-D multicast alerting. We also aim to incorporate an extension to the UDP-based MQTT-SN [105], which is designed for low-power devices e.g. sensor networks. We demonstrated (in a limited lab setting) the possibility to apply Ride's address translation techniques on MQTT-SN for edge redirection of data collection and multicast-based alert dissemination. However, most MQTT-SN implementations use different topic IDs for each subscriber, which prohibits our multicast-based alerting. Hence, we leave exploring this avenue for future work.

Fig. 6.3a depicts our lab's **real-world testbed** that we used in our initial proof-of-concept. *Unmodified* SCALE devices publish environmental sensed events to an MQTT [141] broker for visualization via our web-based dashboard or further processing by an analytics service. The SCALE devices associate with a Wi-Fi AP connected to the pictured switch, which routes data to either the edge or cloud broker instances. An ONOS [64] SDN controller connected to the SDN switch's management port controls its forwarding plane routing using the OpenFlow [130] protocol. Both the edge server and SDN controller VMs are hosted on a VMware ESX Hypervisor machine connected to the switch. Using our initial SCALE prototype of Ride, we conducted a first proof-of-concept experiment in our lab test-bed to demonstrate CDP fail-over to the edge MQTT broker. We simulated a **broken link** by



(a) Our experimental testbed setup.



(b) Results from our initial cable-pulling experiment.

Figure 6.3: A prototype of Ride in our physical lab test-bed.

unplugging the Ethernet cable connecting the switch to our campus network. Fig. 6.3b shows the observed throughput of IoT data measured at the cloud broker stop after this network outage and pick up a few seconds later at the edge broker. Soon after reconnecting the Ethernet cable, we see the primary CDP recover as evidenced by the cloud broker throughput.

For our more comprehensive experimental setup (§6.5) based on the seismic alerting scenario, SCALE client devices run 3 different mock seismic alerting applications modeled after CSN: 1) a publisher to upload seismic sensed events at a pre-defined time; 2) an alerting service (running on both cloud and edge servers) to aggregate these readings (i.e. detect an earthquake) over a two-second period and publish a *seismic alert*; 3) a subscriber that records the results of these alerts (i.e. when they were received, which seismic readings were captured in them) for measuring performance.

We implemented Ride's logic on the edge server as modular Ride-C and Ride-D Python middleware services. We developed an SDN controller REST API adaptation layer that requests an updated topology from the SDN controller. Ride then runs path-finding and multicast tree-building algorithms on the network topology using the popular NetworkX [90] graph algorithms library. It builds publisher routes and MDMTs, convert them into OpenFlow flow rules, and install these rules in the SDN data plane via the controller's REST API. This approach enabled more rapid prototyping, modular testing, and flexibility

than targeting a single SDN controller platform.

Ride-C pre-configures data collection routes from each registered sensor-publisher to the cloud service. While we use static flow rules for these routes to improve *STT* accuracy, Ride could also support dynamic routes by having the SDN switch at each hop tag packets in a manner similar to [66]. These tags could then be used to build the *STT*. Ride-C spawns a local threaded client and simple cloud-based UDP echo server to monitor each registered CDP as described in §6.3.1. Upon deeming a CDP congested or failed, Ride-C marks it unavailable and publishes this observation locally for use by other applications. In response to this unavailable event, it checks the state of the other CDPs and uses the SDN data plane to re-route IoT traffic through an available one or to the edge server (using address translation flow rules) until a CDP recovers.

During normal cloud operation, the seismic alerting service simply publishes alerts to each subscriber using unicast. After fail-over to the Ride-D-enabled edge service, it receives and processes sensed events originally addressed to the cloud. It analyzes them for seismic events while Ride simultaneously updates the *STT*. When issuing an alert, it uses the shared *STT* graph to select the best available MDMT and send the singular alert packet to the subscribers using the associated multicast address. This address includes both a destination IPv4 address and UDP source port to ensure responses are routed back along the correct MDMT. This enables Ride-D to collect further network/alert state information for the intelligent retransmission mechanism described in §6.2.2.2. Note, however, that we had difficulty implementing the retransmission mechanism due to limitations with the CoAPthon library (i.e. it ignores responses to multicast messages). Hence, our current implementation only sends a single multicast alert, and so our results in §6.5.4 do not include retransmission. While using multicast trees reduces total bandwidth consumption, we must also address potential flooding issues by not sending alerts too frequently. Therefore, we include minimal-sized alert data to avoid this due to a lack of flow control. CoAP's flow control mechanisms

are intended for reliably-transmitted messages (multicast messages must be non-reliable [71]). Hence, we also later explore prioritizing data to provide some flow control (i.e. in the form of targeted packet drops) in the data plane.

6.5 Experimental Evaluation

This section evaluates Ride using our prototype implementation. We describe the experimental setup (including synthetic network topology), overall results from our experiments, and finally delve deeper into the parameters that affect Ride's individual algorithms' performance.

6.5.1 Experimental Setup

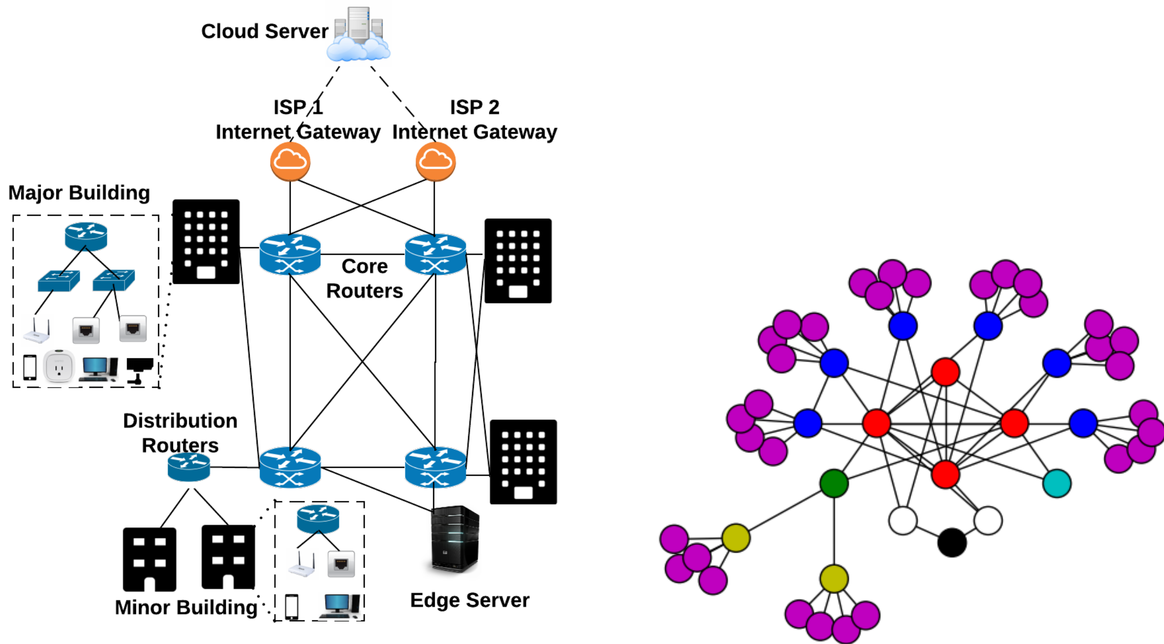
Due to practical limitations (i.e. limited number of physical SDN switches and the difficulty of creating repeatable failure scenarios in a real network), we implement larger-scale experiments with Mininet [1]. This emulation environment uses Open vSwitch (OVS) [3] to create a virtual network topology 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 and run our aforementioned Ride-enabled SCALE seismic clients. OVS switches connect via the SDN southbound protocol OpenFlow [130] to the distributed SDN controller platform ONOS [64] running on the same machine.

6.5.1.1 Synthetic Network Topology

To lend a realistic setting to our experiments, we wrote a Python script to randomly generate a **synthetic campus network topology**, inspired by our university's network, with realistic link characteristics (e.g. bandwidth, latency). Fig. 6.4 shows its hierarchical structure that represents buildings as individual routers, each serving multiple end-hosts and 2-connected to a full mesh of four core routers. Every major building (85%) router connects with two different core routers. A few buildings (e.g. two for the same department) connect directly together. Each distribution router, which also connects with two different core routers, serves several minor buildings (15%). Every building has an internal two-level tree network (i.e. floor/rack switches) serving the end-hosts, although we flattened this due to more scalable simulations and no exploitable redundancy (i.e. due to the tree structure and the shared-risk group that is a buildings' physical structure and electrical wiring). The distinguishing *smart* features of our synthetic campus topology are: 1) edge server(s) (i.e. data centers) connected with two core routers and 2) multiple cloud CDPs comprised of higher-latency links between a public cloud data center node and Internet gateway routers that each connect with two core routers. Our script randomly generates such a topology with varying parameters: number of buildings, hosts, and core routers; redundant connections between buildings (e.g. two buildings for the same department); link characteristics of bandwidth, latency, and cost.

6.5.1.2 Experimental Framework

We use a custom Python-based scenario configuration framework that: 1) reads a synthetic network topology file; 2) constructs it using Mininet; 3) randomly selects and configures hosts as sensor-publishers and/or alert subscribers; 4) executes the experiment by applying a network failure model at pre-determined times; 5) and records results to determine Ride's



(a) Smart campus network structure.

(b) Synthetically generated smart campus topology.

Figure 6.4: The network topologies used in our experiments.

performance. As indicated by the event flow captured in Fig. 6.5, the publishers constantly upload generic IoT traffic (ẽvery 100ms) as well as a seismic sensed event at each of the following failure model steps: 1) after 20 simulated seconds, disabling the primary CDP to represent a distant earthquake and demonstrate Ride-C fail-over; 2) disabling the remaining CDP 20 seconds later, which demonstrates fail-over to the edge and Ride-D-based alerting; 3) disabling nodes/links in the local campus network with a configurable uniformly random probability that represents the geospatially-uniform shaking experienced within a local campus region during a nearby earthquake; 4) 20 seconds later, the primary CDP recovers to demonstrate Ride-C's return to normal (cloud) operation.

Our framework initializes the experiment with the following configuration parameters: the number of publishers/subscribers, the local failure model's uniform probability, the campus topology file (described below), Ride's algorithm/policy parameters (e.g. k, T_D, etc.), and the number of experiment runs. For each run, it chooses the group of publishers and

subscribers uniformly at random from the available end-hosts (overlap allowed). To better compare multiple experimental treatments, we can optionally maintain the same sequence of publisher/subscriber/failure/routing configurations through the use of random number generator seeds.

6.5.1.3 Evaluating Performance

We calculate three main metrics to assess Ride's performance: 1) *reachability*, an approximation of alerting service availability, is the portion of subscribers that successfully receive alerts; 2) *latency* is the delay from when a publisher creates a seismic sensed event until a subscriber first receives an alert derived from it. 3) *overhead* is either the number of probe packets (Ride-C) or total link cost of a route (Ride-D).

We use these metrics to compare Ride with two non-Ride configurations: 1) when $k = 0$ the edge service uses *unicast*-based alert dissemination over the shortest paths; 2) we calculate an *oracle* upper bound on *reachability* by a) removing the failed nodes and links from the topology originally read from a file to create the Mininet network and b) calculating the percent of subscribers reachable from the edge/cloud servers in the remaining topology.

6.5.2 Ride Evaluation in a Seismic Alerting Scenario

This section uses the above scenario to demonstrate Ride's ability to monitor and adapt network state for resilient event collection and timely alert dissemination despite failures. The example run in Fig. 6.5 shows CDP failures as visible gaps in data collection and spikes in alert dissemination. Note that Ride-C quickly fails over to an alternative CDP in the first gap and successfully delivers alerts quicker and more completely than later alerts that must contend with local network failures. After the local failures, Ride enters edge-mode operation

(orange section in middle) and Ride-D disseminates seismic alerts rather than the cloud's basic unicast approach. Note the increase in alert latency over time (green dots trending upwards) due to CoAP's reliable transmission mechanism. It times out after 2-3 seconds of not receiving an acknowledgement and re-sends the seismic event (publisher-to-broker) or alert (broker-to-subscriber). This can lead to increasing event collection and dissemination over time as evidenced by the red/green and yellow bars, respectively, appearing several seconds after the initial event. Note that we discard alerts delivered $>$ 10secs. after the event as they have limited use in seismic early-warning. When returning to cloud operation, we note the lack of event collection gap as the edge continues receiving events until cloud redirection completes.

Our emulated experiments validate the benefit of exploiting SDN-enabled edge resources for resilience in such settings. With a cloud-only approach, the data exchange would experience complete failure during the middle segment. Instead, it only misses a few seconds worth of data collection and alert dissemination. This loss, especially during fail-over to the edge, indicates needed improvements to the SDN-enabled fail-over mechanism. Even with Mininet's zero-latency control plane configuration, the time required to adapt the data plane by installing flow rules drastically impacts both reachability and timeliness. Hence, we are exploring additional strategies such as pre-installation of partial re-routing paths.

Fig. 6.9 shows the performance of event collection and alert dissemination for varying failure probability. We see that for very high failure rates, further network redundancy is needed. The *disjoint* publisher routing algorithm seems to improve alert dissemination slightly by producing a more complete STT. We plan to investigate this further in future work. To further explore and improve Ride configurations, the following two sections isolate the Ride-C and Ride-D mechanisms.

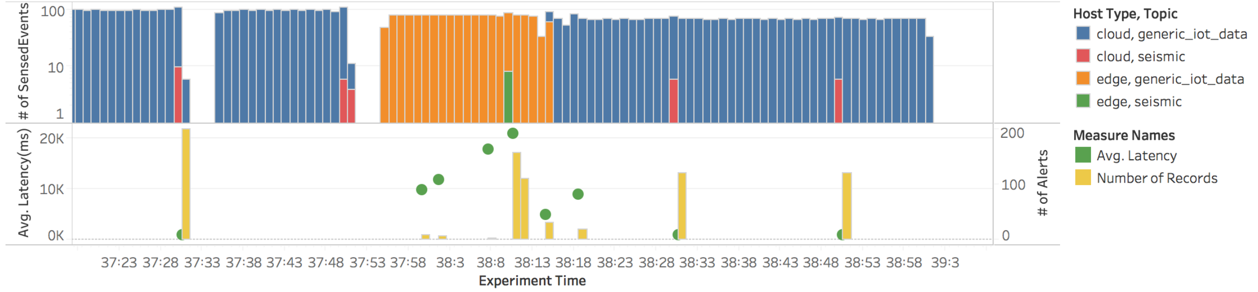


Figure 6.5: Ride's failure adaptations during an example execution of our seismic alerting scenario.

6.5.3 Ride-C Performance & Parameter Space Evaluation

Our initial experiments with Ride-C demonstrate its efficacy and a potential generic application. We used a modified form of the experimental setup described above with a 1% link loss rate and 10-second link failures. The hosts all publish 10 packets/sec. of data to the cloud through a randomly chosen Internet gateway router. We run this experiment with three different configurations: 1) *without Ride-C* - no failure detection or failover is enabled, 2) *with Ride-C* - the default Ride-C failure detection and failover mechanisms are enabled (i.e. if Ride-C detects gateway-to-cloud link failures, it forwards the affected data flow to the other unfailed link), and 3) *with Ride-C and an edge buffer*, which we describe later. At the end of each experiment run, the total number of packets that publishers sent and the number of packets received at the cloud server are collected to calculate the total packet loss rate. Fig. 6.6 shows how Ride-C performed in the CDP failure experiment with these three different configurations and varying numbers of publishers. Overall, in the configuration where Ride-C is not enabled, the experiment results in the highest packet loss. The packets that attempt to traverse the affected link during failure are all lost. The packet loss improves in the configuration with Ride-C default failure detection and failover mechanisms enabled. After detecting the failure, Ride-C redirects the affected flows through the unfailed CDP. In this second configuration, only the packets sent through the affected link between the time when the failure started and when the failover action triggered are lost.

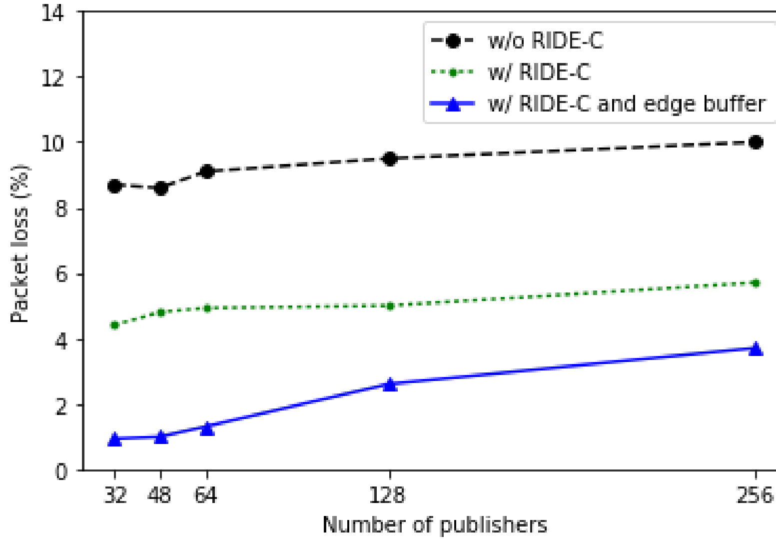


Figure 6.6: Packet loss in the CDP failure experiment using three different configurations.

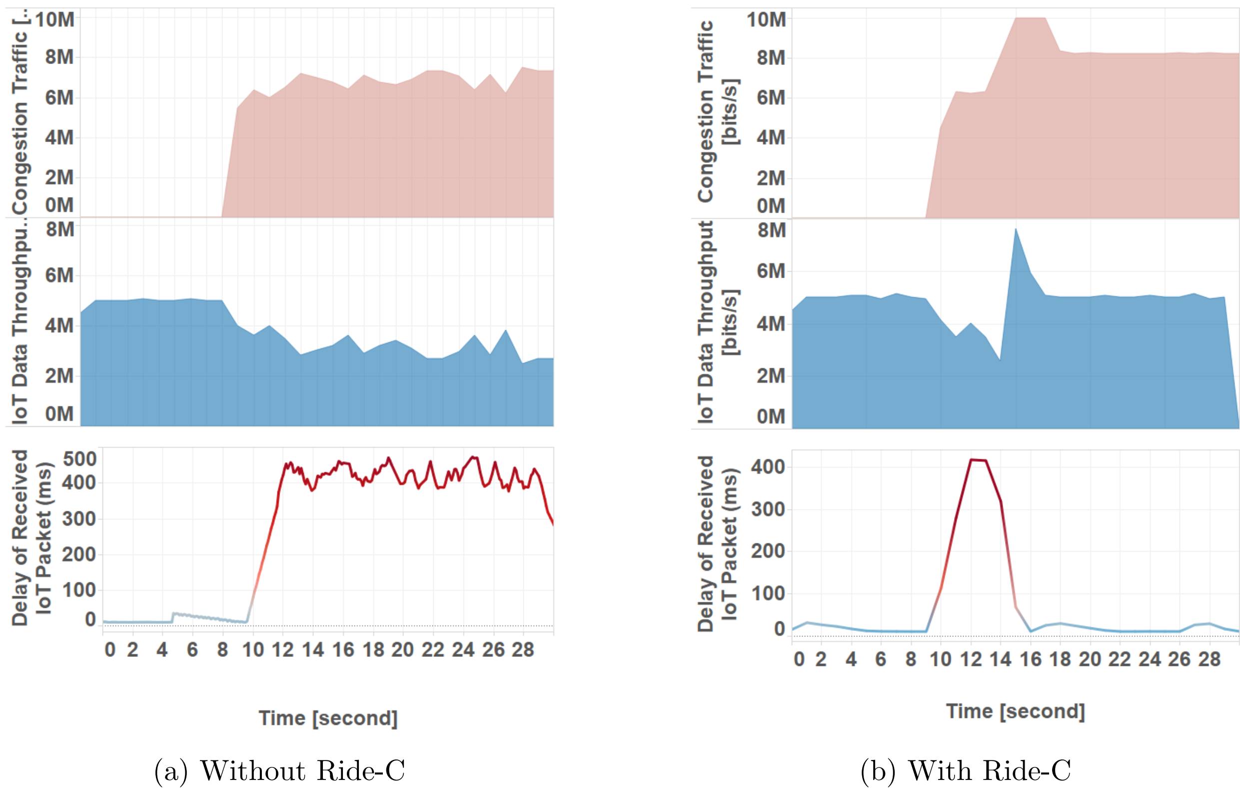
The third configuration further improves the packet loss by running an edge buffer program on the application edge server. The server leverages the network-awareness that Ride-C provides to use this buffer as temporary storage for high-priority *sensed event* packets that might otherwise be lost during a transient cloud disruption. The circular buffer program receives a copy of each *sensed event* packet that is sent to the cloud server and stores a fixed number (2650 by default) of these packets. As a simple proof-of-concept, we initially implemented this step on the publisher, but will later implement it as a network function leveraging SDN features transparent to end hosts. The publisher sends a copy of the *sensed event* packet to both the edge and cloud servers at the same time. Upon detecting the CDP failure, Ride-C sends a notification to the edge buffer program. The edge buffer program then starts to send all the recently-buffered *sensed event* packets to the cloud through the other unfailed CDP.

By temporarily maintaining local copies of this data at the edge after forwarding it through the cloud CDP, we decreased packet loss rates at the cloud service by ≈75% as compared with only buffering in publishing devices, which are often resource-constrained in IoT settings. For lower numbers of publishers, the packet loss sits around the preset gateway-to-cloud link

packet loss rate of 1%. This means that the packets normally lost during the failure are instead recovered and forwarded by the edge buffer program. As the number of publishers increases and exceeds a certain value, the packet loss increases due to more published packets but a constant limited buffer size. When the number of packets lost during failure exceeds the buffer storage capacity, the excess packets are lost. However, this configuration shows a significantly lower packet loss than the previous configurations since some of the data is recovered. This scenario therefore serves as an example of network-aware services that can be developed using Ride-C.

We also tested how Ride-C works during a **congestion scenario** rather than complete link failure. We first set the rate limit on egress ports connecting to the campus network at 10Mbps. A host connected to this SDN-enabled switch creates congestion traffic by sending data to another host in the campus network using *iperf* in UDP mode at 8Mbps. We configured a static flow in the switch to always forward the congestion traffic through one of the campus network egress ports. Fig. 6.7a shows the congestion's impact on IoT traffic without Ride-C. The throughput of IoT data at the cloud decreases and the delay of received IoT packets increases significantly. Fig. 6.7b shows how Ride-C handles the congestion scenario. After detecting the congestion, the IoT traffic that is currently forwarded through the congested port is instead forwarded to an unaffected port. We observe the recovery of IoT data throughput and delay a few seconds after the congestion starts.

Now we evaluate the different configurations of Ride-C and its failure-detection algorithm. We setup experiments to evaluate its failure-detection-and-correction time and overhead (# probe packets) under varying parameters (e.g. the application-specified requirement T_D). Fig. 6.8 shows how Ride-C always meets the required T_D, which closely matches the observed failure detection time (linear trend). Note that different maximum detection time requirements change the detection time and overhead. Also note that the Ride-C detector can always detect the failure within the maximum detection time. It also shows the trade-off



(a) Without Ride-C

(b) With Ride-C

Figure 6.7: The throughput and delay of IoT data under the influence of a congested CDP.

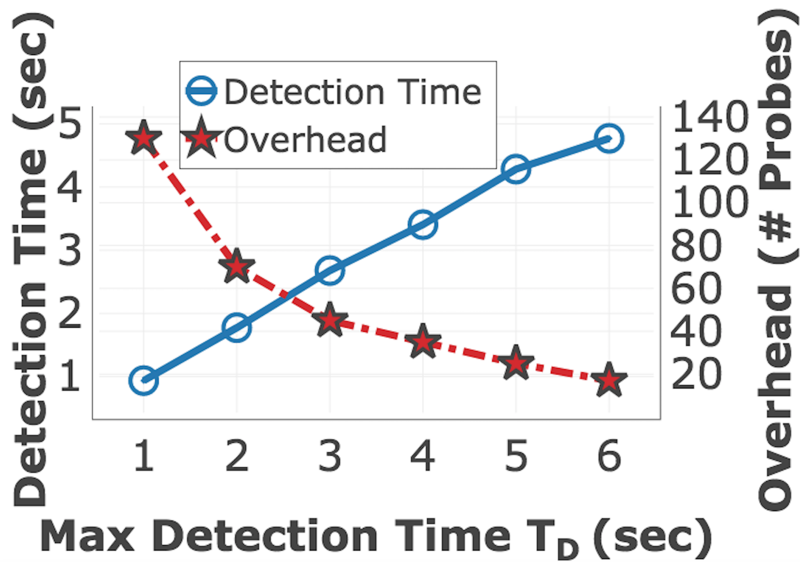


Figure 6.8: Varying Maximum Detection Time T_D showcases Ride-C's resource-conserving adaptive probing.

between T_D and the probing overhead, which decreases significantly as T_D increases from 1 second to 3 seconds. This suggests that if an application can tolerate a few more seconds of failure detection time, it can lower the probing overhead significantly.

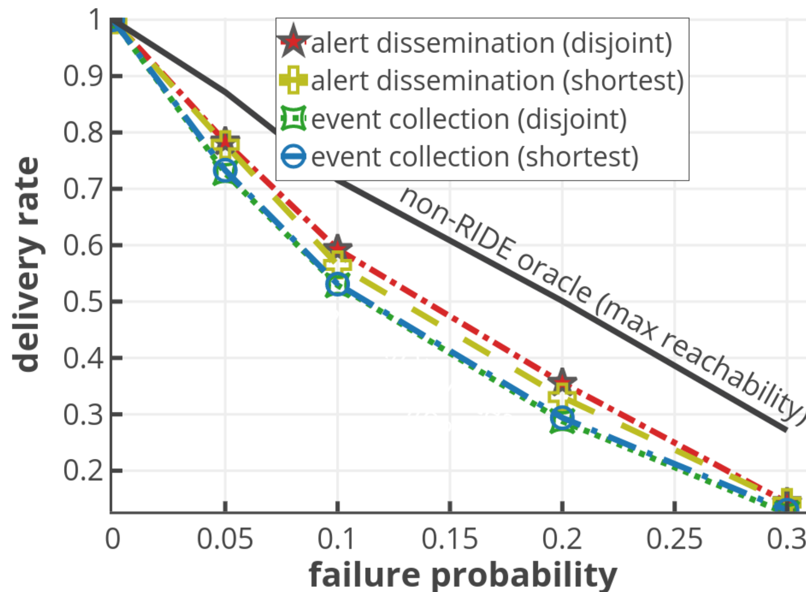


Figure 6.9: Varying network failure probability shows setting collection routes using disjoint method improves alerts' delivery rate

We also compared the two different routing policies (shortest/disjoint) given in §6.3.1, but found that they perform almost identically as shown in Fig. 6.9. Clearly the known hard problem of diverse path routing presents an area ripe for improvement as previously discussed.

We also compared the Ride-C failure-detector's performance with two other failure detectors from [16] and [58]. Both of the detectors are based on a *PULL style* method with which the detector sends probes to a target and decides its liveness based on replies. The Resilient Overlay Networks (RON) [16] failure detector sends the probes with long intervals in its normal state. After a probe timeout, it sends subsequent probes with a shorter interval. If all these fast-transmitted probes timeout, the RON reports a failure event.

The B-AFD failure detector proposed in [58] is an adaptive version of the RON detector. It

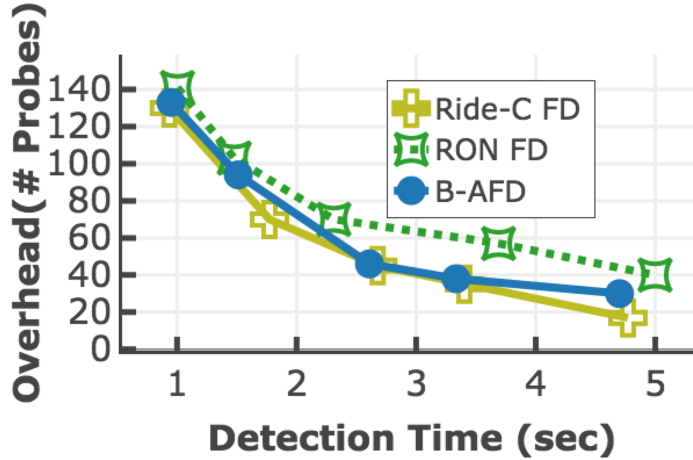


Figure 6.10: Comparing failure detector (FD) overhead of Ride-C and two related works.

takes QoS requirements like maximum detection time, mistake recurrence time, and mistake duration to dynamically reduce the probing overhead. We ran experiments with all three failure detectors several times to compare their performance. To make them detect the failure at a certain time, we set maximum detection time T_D for Ride-C and B-AFD to (1,2,3,4,5) seconds. Since RON has no T_D parameter, we manually configured its parameters to achieve a similar failure detection time. Fig. 6.10 compares their actual failure detection time and probing overhead. It shows that Ride-C and B-AFD detect the failure with much lower overhead while still satisfying the T_D requirement. Compared with B-AFD, Ride-C tends to detect the failure with lower overhead but slightly longer detection time.

6.5.4 Ride-D Scalability & Parameter Space Evaluation

To evaluate Ride-D's ability to resiliently disseminate alerts in larger settings and different configurations, we isolated the Ride-D phase of our experiments with a larger topology. However, scaling issues (Mininets performance degrades with > 100 end-hosts and > 30 switches) necessitated a *simulation* framework version. It uses Python's NetworkX [90] graph algorithms library to manipulate the topology and directly calculate the subscribers'

reachability (given a single alert transmission attempt) for each constructed MDMT in the face of earthquake-induced failures. Our emulated and simulated studies otherwise utilize the same experimental setup and synthetic network topology structure (Fig. 6.4). This allowed for careful control of the parameters, isolating the experiment from the effects of system configuration issues (e.g. SDN controller, Mininet performance, process scheduling, etc.), developing and testing the algorithms before completing the network stack, and exploring the parameter space more quickly due to lower overhead. This version directly applies Ride-D's core logic to build k MDMTs for the specified algorithm and a network topology read from a file rather than the SDN controller. Hence, we used a pared-down version of the Ride-D implementation classes to construct the MDMTs without requiring SDN controller interactions. We did this in part by implementing a dummy version of the SDN REST API adapter described in §6.4 that reads the topology from a file instead of from the SDN controller.

Each run of the simulation version sets up the scenario as described previously, but it directly by: 1) removing the failed nodes and links from each MDMT; 2) calculating the percentage of subscribers reachable from the edge server via the remaining topology; 3) aggregating this value across all runs with the same experimental treatment. (i.e. finds the minimum, maximum, mean, and standard deviation) It similarly calculates the two aforementioned non-Ride comparisons of *oracle* and *unicast*. It also calculates the two special comparisons by removing the failed nodes and links from the original topology to determine reachability via: any remaining path (*oracle*) or the normal shortest paths from edge server to subscribers (unicast baseline non-multicast approach). Note that we ignore results for which no subscribers are reachable in the remaining topology (i.e. oracle's reachability is 0) as no recovery is possible in such a situation.

We vary the aforementioned parameters with default values of: 200 publishers, 400 subscribers, failure probability=0.1, a 200-building topology file, k=4, the *red-blue* MDMT-

construction algorithm, the Ride-C *diverse* publisher-routing policy, and 100 runs. For each MDMT-selection policy, we calculate the STT based on which publishers are still connected to the edge via their Ride-C-assigned routes, execute the policy, and record the reachability of its choice. We also record the minimum, maximum, and mean reachability of all k MDMTs as worst, best, and random selection policy results.

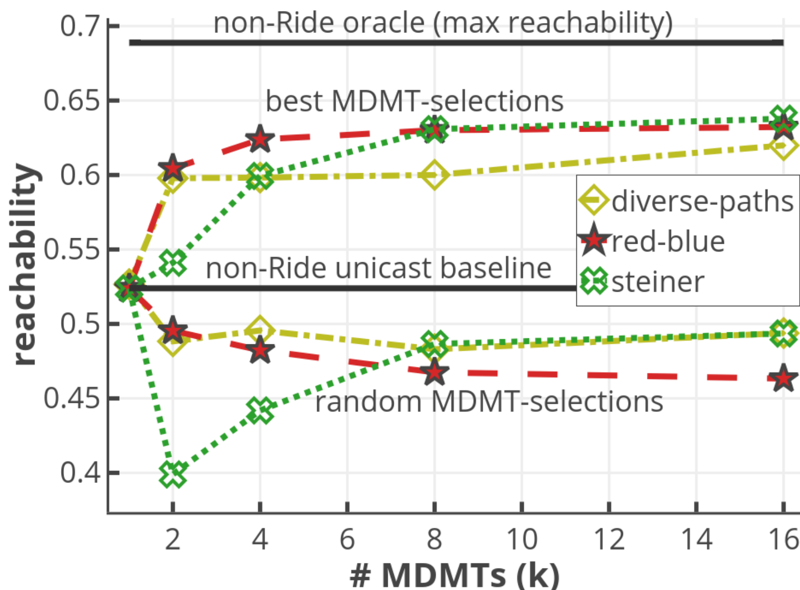


Figure 6.11: Comparing MDMT-construction algorithms shows careful MDMT-selection performs better than unicast.

Fig. 6.11 compares the different MDMT-construction algorithms. Note that both the *random* and *worst* (not pictured) curves perform worse than *unicast* due to multicast tree structures' lack of redundancy. However, the best MDMT choice results prove how an intelligent MDMT-selection policy can effectively support resilient multicast-based alerting. Without enough information (i.e. STT accuracy), however, *unicast* should be preferred since MDMT-selection would be as good as (or even worse than) *random*.

We note that *red-blue* outperforms the other algorithms for smaller k and that $k > 4$ provides insignificant improvement, hence our recommended default of $k = 4$ MDMTs. We also recommend not using *steiner* for $k \in 2, 4$. By varying the failure probability parameter for each algorithm (Fig. 6.9 shows the results for *red-blue*), we found significant reachability im-

provements for lower values (0.05-0.35). Beyond that (not pictured), they converge towards *oracle*, indicating that no strategy could address such high failure rates.

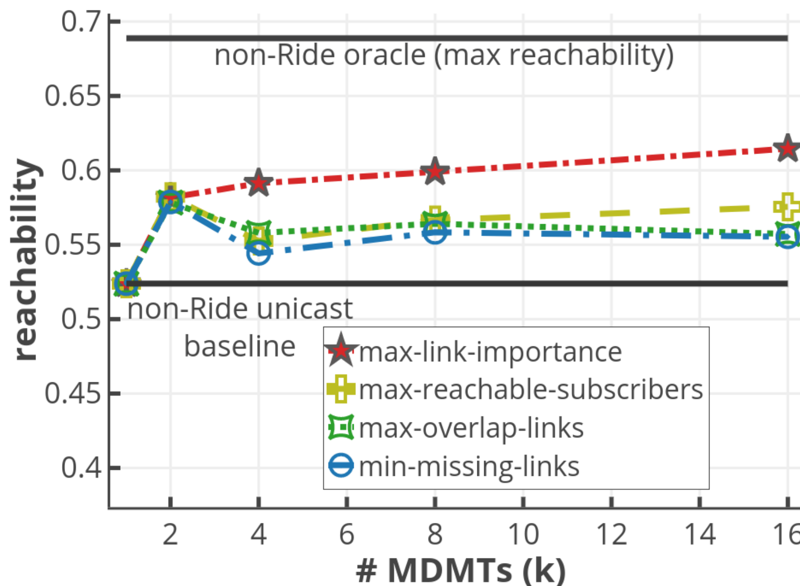


Figure 6.12: For $k > 2$, the *max-link-importance* MDMT-selection policy performs best (*red-blue* construction algorithm pictured).

Fig. 6.12 compares the MDMT-selection policies for *red-blue* (the other construction algorithms produced similar results). It validates Ride-D's network-aware approach of choosing the best MDMT based on Ride-derived network state; all of our policies perform better than unicast and random MDMT choice. However, our recommended policy *max-link-importance* achieves the highest average *reachability* for $k > 2$. This is likely due to its hybrid approach that considers both individual links and complete paths. For $k = 2$, the policy does not seem to matter and so the simplest should suffice.

Fig. 6.13 shows Ride-D's improvement in overhead over traditional unicast alerting. These results show unicast maintaining a constant link cost per subscriber successfully alerted whereas Ride-D incurs less incremental cost per subscriber thanks to multicast's data transmission efficiency. We also note from Fig. 6.13 that the number of subscribers vs. publishers has no effect on *reachability*.

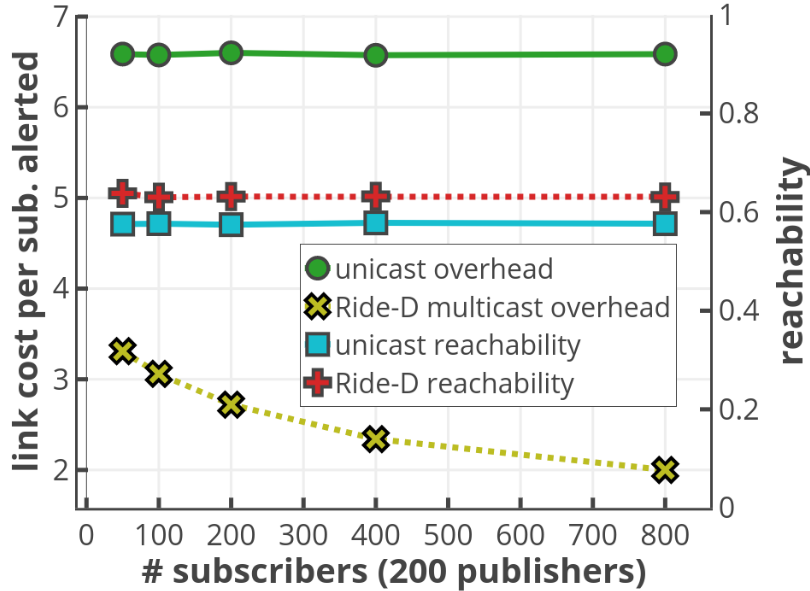


Figure 6.13: Multicast-based dissemination improves overhead vs. unicast while reachability remains unaffected by increased # subscribers.

Some results not pictured varied other scenario parameter. The results all indicated that Ride-D requires a certain level of STT accuracy for effective MDMT-selection. For example, increasing the ratio of publishers to subscribers up to 1:4 improves this selection. Similarly, Ride-D can tolerate up to about 40% publication loss rate (e.g. due to congestion) for a 1:2 publisher:subscriber ratio. Furthermore, the campus network topology's size (i.e. # buildings) and number of redundant connections appears to have little effect on reachability for our campus network topology. Explorations into more redundant topologies could help confirm this finding or identify structures that enable more benefit from the Ride-D approach.

Increasing from $k = 1$ (no redundancy) to $k = 2$ and then $k = 4$ actually *decreases* the *worst* (and sometimes *random*) reachability for each MDMT-construction algorithm. However, it does typically increase the *best* reachability as seen in Fig. 6.11. This intuitive result indicates that choosing too large a value for k does not just increase resource usage but may also decrease reliability due to not enough available redundancy in the topology. Interestingly, *diverse-paths* appears to perform worse for $k = 4$, which indicates that our method for assigning the disjoint paths to MDMTs may need improvement. Experiments on multiple

failure probabilities for these k values demonstrated the same trends.

6.6 Chapter Summary and Discussion

This chapter demonstrated our cross-layer network and application aware approach to resilient communications for IoT data exchange in the context of Ride. This SDN-enabled edge service middleware facilitates network-awareness by monitoring network conditions and adapting to failures/congestion in public cloud IoT data flows for event collection. In the event of cloud unavailability, it also enables resilient emergency alert dissemination to interested users and IoT devices by intelligently selecting from multiple redundant multicast-based topic distribution trees. We framed this discussion in the context of an IoT-based seismic monitoring and alerting application running both in the cloud and at the edge for resilience to earthquake-induced network failures/congestion. Our prototype implementation and emulation/simulation-based results indicate Ride's efficacy.

While our approach does slightly increase overall system complexity, it does so mainly in the edge cloud service. The Ride middleware extends existing IoT data exchanges without modifying them, the IoT devices, or potentially even the cloud services. The registration process enables independently using Ride with only the most-critical IoT services. Administrators must determine which services require such enhancement and their desired level of resilience. Real-time critical apps require lower T_D (e.g. < 1sec.) whereas those that tolerate some delay but must remain operable can use higher values. Less-critical apps that tolerate some alerting loss can use $k = 2$, whereas we recommend $k = 4$ for the most stringent of mission-critical applications such as our seismic scenario. Furthermore, adaptive probing intervals and multicast actually conserve network resources as shown in §6.5. Hence, administrators must weigh the benefits of this conservation with the increased deployment complexity.

6.6.1 Integrating Ride Into Our Proposed Middleware

Ride builds upon the middleware approach and architecture (Fig. 6.14) proposed in this thesis by incorporating edge resources into the data exchange process. It extends the cloud-centric data collection approach used in GeoCRON by monitoring and adapting the overlay paths to failures and disruptions. Through this adaptation to the data collection phase, Ride answers the question of whether to collect and process data at the edge or in the cloud. Furthermore, it improves resilient dissemination of local alerts when operating in edge mode. Our resilient multicast-based data dissemination mechanism closes the IoT application's loop by leveraging both application resilience requirements and network state awareness gathered during the data collection phase. In this manner, it improves the data dissemination process without modifying data consumers. Instead, it focuses the logic for this mechanism in an edge service that coordinates with the SDN infrastructure. The prototype described in §6.4 incorporates modules that implement these mechanisms and algorithms into SCALE.

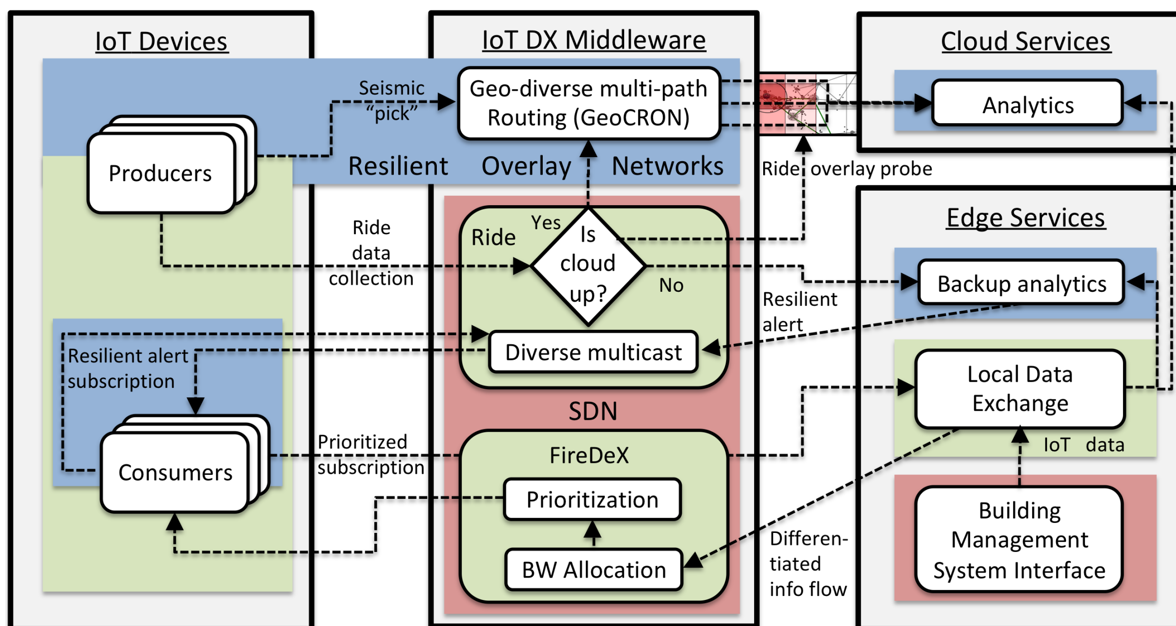


Figure 6.14: Ride adds data exchange logic to the data exchange and SDN layers.

At this stage, our prototype middleware fully supports an end-to-end mission-critical IoT application. However, it does not address the question of which subscriber(s) are most

important to notify. Furthermore, it does not consider the needs of multiple IoT safety applications that share edge resources. Next, we explore this question and expand on our proposed middleware by expanding on its enhanced subscription capabilities to incorporate prioritizing the most important data flows.

Chapter 7

Prioritizing Heterogeneous IoT Information Flows at the Edge

Consider the process of IoT data exchange in support of enhanced situational awareness and response during the Smart Fire Fighting (SFF) scenario described in §3.1.3. Note the diversity of stakeholders with interest in a variety of data: emergency response personnel (e.g. FFs, medical staff, public safety officers, government officials) and civilian community members (e.g. evacuees, their family members or caretakers, residents impacted by smoke or street closings). Key challenges arise in such settings including: managing heterogeneous information with varying size, format, relevance, urgency, etc.; seamless integration of new IoT data sources with pre-existing sources and information on the fly; supporting reliable and timely communication over constrained networks (e.g. due to lossy channels and failed components).

This chapter explores these challenges within the context of our FireDeX project. We now consider how to manage such heterogeneity of data and information requirements in a more localized setting (i.e. the immediate area around the structure on fire). We take the previous

chapters' approaches to maintaining resilient connectivity as a given and instead focus on prioritizing information flows according to user requirements and network resource constraints. In a SFF setting, needs may evolve over time and these constraints may vary as a result of both changing human activities and infrastructure challenges due to the fire itself. This exploration into managing data exchange at the edge expands our previous approaches by also balancing the needs of multiple actors and mission-critical IoT applications simultaneously.

7.1 Chapter Overview

To overcome the above challenges, we propose **FireDeX**, an integration middleware that unifies (a) smartspace IoT data and infrastructure (e.g. a Smart Building Management System (BMS)) with (b) programmable network infrastructure (e.g. through SDN) and (c) domain specific applications (e.g. smart fire fighting systems and apps brought onsite by emergency responders). FireDeX aims to support timely and reliable delivery of the most critical data to relevant subscribers despite challenging network conditions. It utilizes the cross-layer (i.e. application and network-aware) approach for IoT data exchange shown in Fig. 7.1 and described here. It leverages edge computing (i.e. data exchange brokers at the network edge) and SDN to bridge critical application requirements with network state. This approach facilitates platform and device independent adaptation of IoT communication at the middleware layer.

Using SDN, FireDeX configures the underlying physical network to prioritize messages according to the requirements (i.e. subscriptions with utility functions that quantify situational awareness) and underlying network resource constraints (e.g. bandwidth, error). Through the use of priority queues and carefully tuned packet drop rates (i.e. for bandwidth allocation), it ensures timely delivery of the most important data possible. To this end, we construct an extensible queueing theoretic model to abstract the cross-layer flow of data. We

present our formal analysis that our proposed novel algorithms leverage to configure the data exchange. These algorithms manage active subscriptions by separately assigning priorities to each and then allocating bandwidth to them. This enables timely and reliable delivery of the most critical data to relevant subscribers despite challenging network conditions.

The FireDeX middleware combines several novel capabilities and design features, notably the following key contributions:

- Applying a cross-layer approach (application, middleware, networking) to prioritizing mission critical IoT data exchange during a fire response scenario in an IoT-enhanced smart building with SDN-enabled edge infrastructure. (§7.2)
- Formulating an extensible formal model of these three layers based on the unified framework of queueing theory. 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. (§7.3)
- Leveraging the above queueing model to explore the configuration parameter space and derive novel algorithms that prioritize IoT events and tune notification delivery/delay. FireDeX leverages SDN to configure the underlying physical network with priority queueing disciplines and carefully tuned packet drop rates (i.e. for bandwidth allocation). (§7.4)
- Designing, implementing, and evaluating a prototype middleware that incorporates the above algorithms. It coordinates with an OpenFlow-enabled SDN controller to configure the network infrastructure. We discuss our design decisions, challenges overcome, and practical considerations identified during this experience. (§7.5)

In §7.6, we evaluate the FireDeX approach by: describing our experimental framework for randomly generating, configuring, and running experiments; validating our proposed analyt-

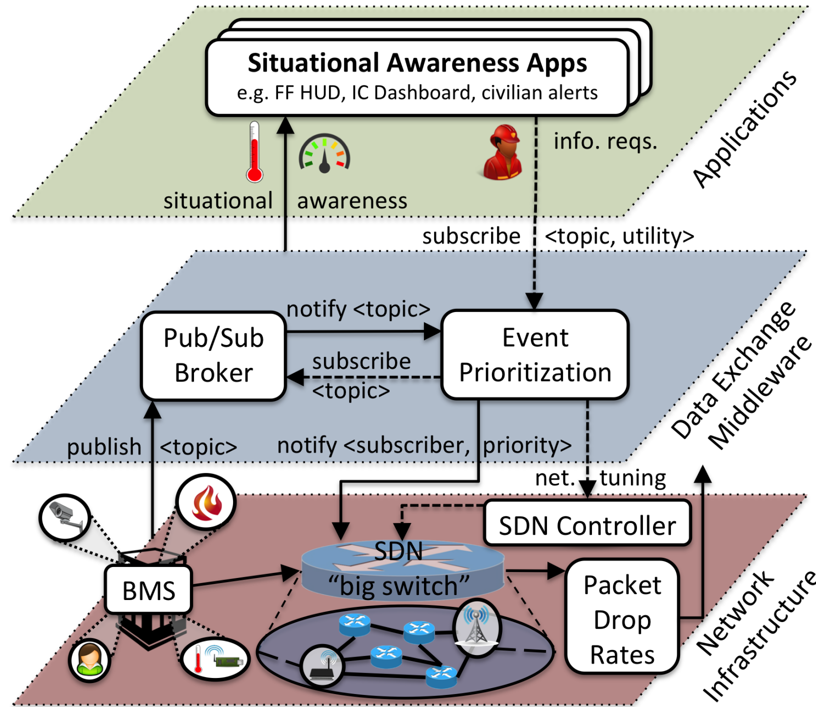


Figure 7.1: The FireDeX cross-layer middleware.

ical model; evaluating our middleware's performance and capabilities both in simulation and an emulated network setting that incorporates our prototype implementation; comparing the various proposed algorithms' performance.

7.2 The FireDeX Approach

To motivate the need for prioritized IoT data exchange, the challenges involved, and our proposed approach, we begin with a motivating IoT-enhanced structural fire scenario. We discuss related work and then describe our proposed approach.

7.2.1 A Driving Scenario: Fire Fighting with IoT

See §3.1.3 for details about our targeted smart fire fighting scenario. We leverage it to define the challenges that FireDeX addresses and to lend a more realistic setting to our experiments. In particular, we target the varying information requirements of subscribers and deliver the most important of heterogeneous information flows according to currently available network resources.

7.2.2 IoT Data Exchange Addressed by FireDeX

Several research challenges for mission-critical IoT data exchange arise from the above driving SFF scenario.

Heterogeneity of devices and information: As described in §2.4, IoT-enabled buildings produce a wide variety and large volume of heterogeneous information potentially relevenat to emergency response efforts. To manage both scale and heterogeneity, we design FireDeX as an edge middleware that leverages existing IoT infrastructure and services managed by a third-party BMS. Its pub/sub approach integrates capabilities of new devices/-tools brought on scene by responders, while separating operational and ownership concerns. Note that external entities often lack the knowledge, access, and expertise to reconfigure local devices. This might conflict with existing configurations customized by building IoT administrators.

Managing smart spaces at scale in real time: Tuning data collection parameters at each device (e.g sampling rate, resolution) for individual subscribers is not always viable, especially as the number and diversity of devices scales. FireDeX's edge broker approach naturally supports scalability; geographically-dispersed subscribers interconnect through a distributed network of data exchange brokers. This network may be hierarchical: top-level

brokers running in cloud data centers serve a large region with many local brokers running in edge data centers to serve a smaller local area (e.g. one or a few buildings, a campus). FireDeX can further leverage work in large-scale pub/sub, in-network data processing, and more deeply exploiting data semantics as described in Chapter 2.

Managing unreliable IoT networks: Communications are often constrained in crisis scenarios. FireDeX leverages SDN to manage networking for IoT deployments by offloading network configuration tasks from constrained devices and network hardware. See §2.3 for a more detailed overview of SDN.

Research on Network Utility Maximization (NUM) [195] 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 that capture a user's degree of satisfaction with the network's performance. Few prior researchers have investigated discrete priority classes, which we leverage in our approach, within the context of NUM. The authors of [138] propose assigning more bandwidth to users (i.e. via weighting their requests higher) based on their requested priority levels. Similarly, [161] manages IoT devices to maximize utility by allocating bandwidth and offloading processing, but the authors do not use SDN or consider the data exchange middleware and prioritized application requirements. Our cross-layer approach and consideration of utility functions sets apart our work from most related SDN research referenced above.

Modeling cross-layer data exchange interactions using Queueing Theory: To analyze IoT data exchange performance we must consider all three layers' characteristics and their effects on each other. However, existing efforts typically focus on each layer in isolation. Therefore, we model cross-layer interactions by composing and extending previous work at each layer through the unified framework of *queueing theory* [99, 166]. Queueing Petri Nets (QPNs) enable accurate performance prediction in pub/sub systems [115, 158]. Alternatively, [39, 40] model and analyze the performance of pub/sub and middleware

protocols using Queueing Networks (QNs). While QPNs have an advantage over QNs in representing parallelism, QNs provide convenient primitives to construct well-formed performance models for efficient analysis [188]. Furthermore, QNs have been extensively applied to model network infrastructure performance [86, 88, 93, 20] and more recently SDN infrastructure [80, 170, 174].

7.2.3 Enabling Event Prioritization

We now overview how the FireDeX middleware addresses the above challenges. See §7.5 for a detailed discussion of how we implemented this middleware. We frame our discussions in terms of the three layers depicted in Fig. 7.1: mission-critical applications, abstractions representing the physical network infrastructure, and the data exchange middleware that bridges these two to manage the overall system configuration and flow of information. As shown in Fig. 7.2, FireDeX integrates other middleware technologies: data APIs for interfacing with IoT data (i.e. through the BMS), a local pub/sub broker, a thin client middleware running on each subscribing IoT device, and SDN APIs for managing local network infrastructure. It implements the proposed 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: FireDeX subscriber devices run a client middleware to establish broker connections, retrieve a list of event topics, subscribe to relevant ones, and report data exchange/network channel statistics. Because 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 FireDeX subscriptions.

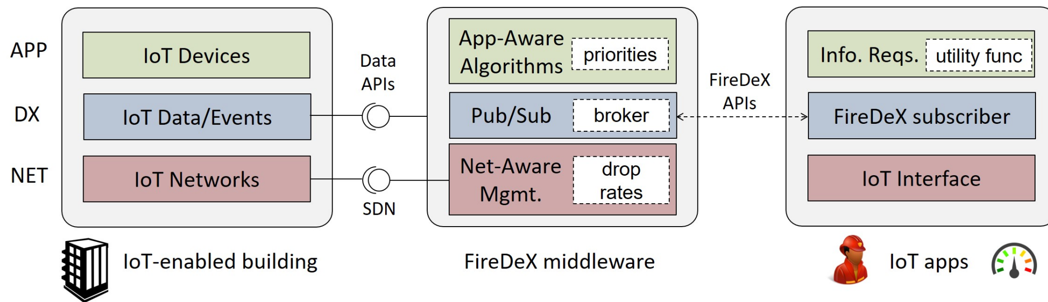


Figure 7.2: FireDeX middleware architecture

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 users' utility (i.e. situational awareness).

Data exchange layer: FireDeX prioritizes subscriptions according to their subscriber-specified utility functions. It leverages the queueing theoretic analysis we present in §7.3 to estimate system performance under a given configuration. This analysis drives the algorithms presented in §7.4 that assign discrete **priority classes** and allocate available network bandwidth. FireDeX connects subscriber clients and the BMS data APIs with the pub/sub broker, which performs the actual routing of events.

While some existing data exchange implementations and protocols support priorities, configuring them requires specific APIs [147]. Furthermore, many popular options (e.g. the MQTT [141] protocol and associated broker implementations) do not support priorities and so require equal treatment of all events transmitted to the same subscriber. To decouple FireDeX from the underlying data exchange broker, which may be specific to the site's BMS, we do not employ application-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 application-level requirements (e.g. utility functions) and network-level state information (e.g. available bandwidth) without mandating (or extensively modifying) specific broker technologies. Hence, FireDeX essentially extends the data exchange broker/protocol with network and application-aware prioritization.

Network layer: FireDeX manages network infrastructure through APIs provided by an SDN controller that likely runs alongside the BMS (i.e. at the edge). It gathers network state information to derive resource constraints. It combines these with the subscribers' information requirements to drive its management algorithms. The authors of [198] previously advocated a similar approach of centrally gathering a global view of a 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 where offloading device management and data processing from constrained devices typically leads to centralized (i.e. cloud-centric) designs. For simplicity of discussion, we consider the ***big switch*** model shown in Fig. 7.1 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. building Wi-Fi and local cellular) and different locations.

To enforce event priorities at the network layer, FireDeX 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.7.3, 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: IP/MAC address, UDP/TCP port, VLAN, etc. To differentiate subscriptions as belonging to different network flows, a FireDeX subscriber maintains multiple **network connections** with the pub/sub broker (e.g. over different 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 middleware layer dictates this assignment of (possibly multiple) subscriptions to one net-

work 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. FireDeX 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.

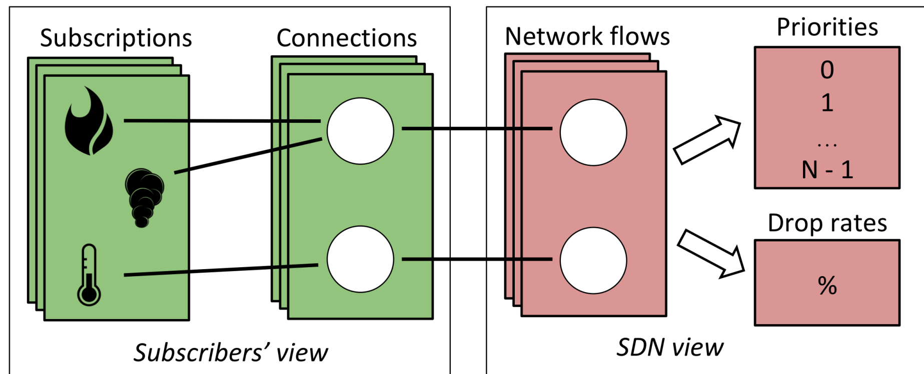


Figure 7.3: FireDeX differentially prioritizes subscriptions at the SDN layer using multiple connections per subscriber.

To manage available network resources, FireDeX 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. §7.4.3 discusses this concept further and proposes our optimization-based algorithm for setting these drop rates. This algorithm is partly inspired by the aforementioned research in Network Utility Maximization (NUM). However, in FireDeX subscribers actually define the utility functions according to their information needs, and so they indirectly cooperatively control the assignment of bandwidth. Furthermore, our proposal leverages discrete priority classes to drive priority queueing disciplines and calculates the best priority assignments rather than assuming them as a given input.

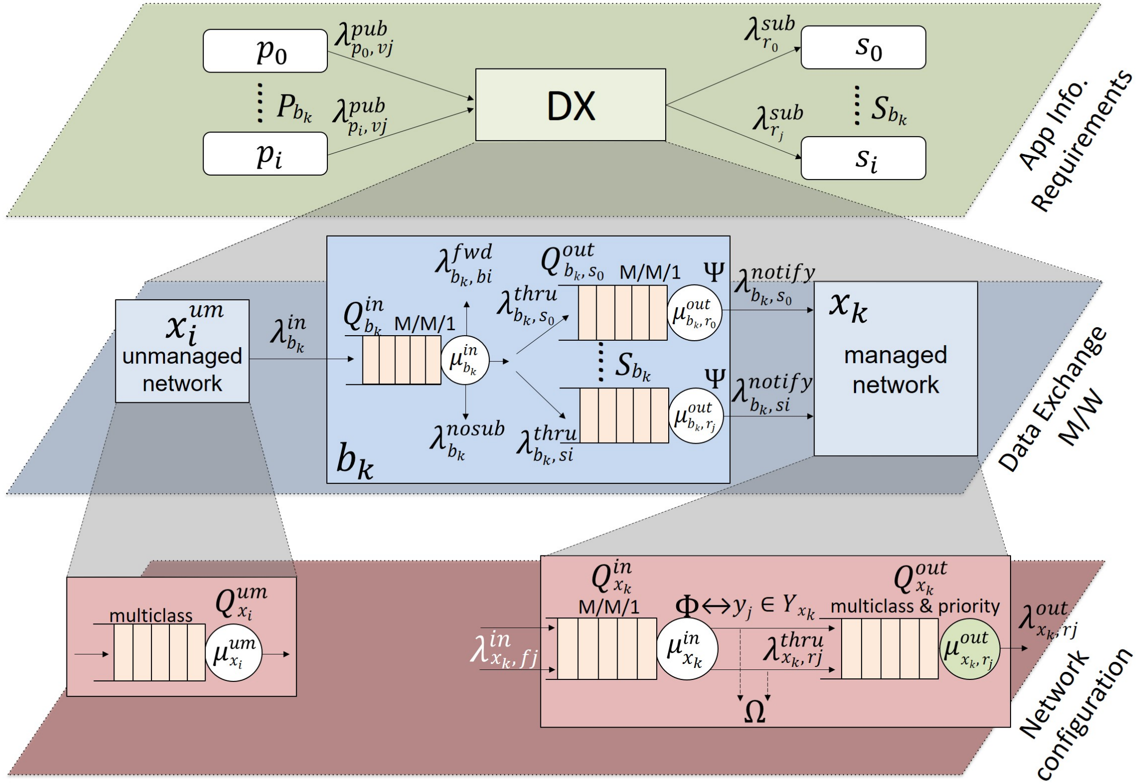


Figure 7.4: FireDeX queueing network model.

7.3 FireDeX Formal Model

From the above scenario, we formulate a generalized model for prioritized data exchange in mission-critical settings. FireDeX combines queueing theoretic approaches from both the middleware and network layers to construct the representative and extensible 3-layer queueing network shown in Fig. 7.4. 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.

Table 7.1: Notations of the parameters in our cross-layer data exchange model

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 publication 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
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*

7.3.1 Queueing Network Performance Modeling

Refer to Table 7.1 for the notations used throughout this section.

7.3.1.1 Application Modeling

Let $V_{p_i} \subseteq V$ be the set of topics each p_i publishes to e.g., "smoke", "temperature", etc. Let $\lambda^{pub}_{p_i,v_j}$ be the publication rate of events with topic v_j published by p_i per unit time.

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 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 events matching subscription $r_j = (s_i, v_j, U_{r_j})$ to subscriber s_i. By Assumption 1, we can estimate Ξ_{r_j} (i.e. by summing Poisson process rates to produce the rate of an aggregate Poisson process) as:

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

Let Δ_{r_j} be the *response time*: the average end-to-end delay of events matching subscription $r_j = (s_i, v_j, U_{r_j})$ from the moment they are published until s_i receives them. Below we calculate this metric, which includes event processing times, network delays, etc.

7.3.1.2 Data Exchange Modeling

The data exchange layer represents a network of broker nodes B. We assume that each publisher/subscriber connects with a single broker that we refer to as its *home broker*: b_{p_i} is the broker that p_i publishes to and b_{s_i} is the broker that subscriber s_i receives events from. 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 eventual consumption by one of the latter's subscribers. As depicted in Fig. 7.4, 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 $\lambda^{fwd}_{b_k,b_i}$ is Poisson. 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:

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

Note that Γ, which we define in §7.3.1.3, 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 of broker b_k's subscribers $s_i \in S_{b_k}$, it 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 calculate the per-subscriber forwarding rate as:

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

FireDeX Configuration Parameters: The data exchange layer also represents the FireDeX configuration service. FireDeX 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 §7.2.3 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 FireDeX defines a set of unique *priority classes* $y_j \in Y$. It assigns each network flow to a priority class for managing network traffic 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, FireDeX employs the following functions:

- $\Psi : R \mapsto F$ is the function mapping 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 §7.2.3, this mapping allows the SDN data plane to distinguish packets containing events from each other based on their subscriptions.
- $\Phi : F \mapsto Y$ 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.

- $\Omega : F \mapsto [0, 1]$ is the function mapping network flows to preemptive packet drop probabilities. By dropping some packets on a network flow, FireDeX more accurately tunes the data exchange configuration 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 available. By dropping packets in the lower-priority flows, this prevents switch buffers from filling up and dropping higher-priority packets.

7.3.1.3 Network Modeling

Publications forwarded to the network layer are encapsulated in packets for transmission by the SDN infrastructure. To simplify the analysis used in our queueing model, 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 *FireDeX-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 FireDeX 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. We therefore 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 calculate 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 $G_{v_j} \in \mathbb{Z}_{>0}$. By 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:

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

This enables calculating the average *transmission delay* $\Delta^{um}_{x_i}$ of packets in $Q^{um}_{x_i}$ (see §7.3.2). 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, by Assumption 3, 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:

$$\Gamma\big(\lambda^{pub}_{p_i,v_j}, p_i, b_k\big) = \big(1 - z_{p_i,b_k}\big)\ \lambda^{pub}_{p_i,v_j}$$

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, the switch first applies the dropping policy to each flow according to the FireDeX-computed function $\Omega(f_j)$. Because our model does not consider further packet drops in $Q^{in}_{x_k}$ or before $Q^{out}_{x_k}$, we have the per-subscription arrival rate at $Q^{out}_{x_k}$ as:

$$\lambda^{thru}_{x_k,r_j} = \Big(1 - \Omega \circ \Psi\big(r_j\big)\Big)\ \lambda^{notify}_{b_k,r_j} \tag{7.1}$$

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 therefore 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:

$$\mu^{out}_{x_k,r_j} = \frac{w_{x_k,s_i}}{G_{v_j}} \tag{7.2}$$

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:

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

7.3.2 End-to-end Analytical Model

We now leverage the above queueing network to derive theoretical performance results. This analysis, the accuracy of which we validate in §7.6.2, enables FireDeX to tune the data exchange performance characteristics of end-to-end event response time and delivery success rate. To calculate Δ_{r_j}, the end-to-end response time of events for subscription r_j, we calculate 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) in 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 calculate the per-subscription end-to-end response time metric as:

$$\Delta_{r_j} = \mathrm{E}\Big[\Delta^{prop}_{p_i,b_k} + \Delta^{um}_{x_{p_i}}\Big] + \Delta_{b_k} + \Delta^{prop}_{b_k,s_i} + \Delta_{x_{s_i}} \tag{7.3}$$

where $\Delta^{prop}_{b_k,h_j \in H}$ is the *propagation delay* (i.e. physical network latency) between the broker and another host h_j (b_k or s_i). $\Delta^{um}_{x_{p_i}}$ and $\Delta_{x_{s_i}}$ are the transmission delays of packets passing through switches x_{p_i} and x_{s_i} respectively. Δ_{b_k} is the processing delay of events passing through b_k.

We must estimate the heterogeneous propagation delays for this subscription's events from each possible publisher on topic v_j – i.e. $\{p_i \in P_{b_k} : v_j \in V_{p_i}\}$. By our single broker assumption, we have this as the expected delay from any such publisher to broker b_k. In the same manner, we estimate the queueing delay at the intermediate switch x_{p_i}. Therefore, we have:

$$\mathrm{E}\Big[\Delta^{prop}_{p_i,b_k} + \Delta^{um}_{x_{p_i}}\Big] = \sum_{\{p_i \in P_{b_k} : v_j \in V_{p_i}\}} \frac{\Delta^{prop}_{p_i,b_k} + \Delta^{um}_{x_{p_i}}}{|\{p_i \in P_{b_k} : v_j \in V_{p_i}\}|}$$

The average response time of (7.3) includes queueing delays at each layer of FireDeX. Based on the queueing network representing FireDeX (see Fig. 7.4), 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 [120], we have the time that an event remains in the system (i.e. queueing time + service time; also called average delay) given by:

$$\Delta_{Q_{mm1}}(\mu, \lambda) \; = \; \frac{1}{(\mu \; - \; \lambda)} \tag{7.4}$$

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 ($Q^{um}_{x_i}$) and *iii)* the non-preemptive priority and multi-class queue ($Q^{out}_{x_k}$). As already pointed out, we model the transmission of packets inside the unmanaged switch queue ($Q^{um}_{x_i}$) using a multi-class queue (each class corresponds to the topic of an event encapsulated within a packet). Based on standard solutions [120], the average delay for a particular subscription r_k is given by:

$$\Delta_{Q_{mcl}}(\mu, \lambda, r_k) \; = \; \frac{1}{\mu_{r_k} \; - \; \mu_{r_k} \; \sum_{r_j \in R} \lambda_{r_j}/\mu_{r_j}} \tag{7.5}$$

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:

$$\Delta_{Q_{mclpr}}(\mu, \lambda, r_k) \; = \; \frac{L_{r_k}(\lambda, \mu)}{\lambda_{r_k}} \tag{7.6}$$

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 (7.6).

By relying on the above analytical models, we calculate the average delay of events for any subscription r_j at each node and layer of the FireDeX queueing network according to (7.3).

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 (7.4) to determine:

$$\Delta_{b_k} = \Delta_{Q_{mm1}}\Big(\mu^{in}_{b_k}, \lambda^{in}_{b_k}\Big) \; + \; \Delta_{Q_{mm1}}\Big(\mu^{out}_{b_k,r_j}, \lambda^{thru}_{b_k,s_i}\Big) \tag{7.7}$$

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 (7.5) such a delay is given by:

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

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 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 (7.4), $\Delta_{Q^{in}_{x_k}} = \Delta_{Q_{mm1}}\Big(\mu^{in}_{x_k}, \lambda^{in}_{x_k,f_j}\Big)$.

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 (7.6) to find:

$$\Delta_{Q^{out}_{x_k}} = \Delta_{Q_{mclpr}}\Big(\{\mu^{out}_{x_k,r_j} : r_j \in R_{x_k}\}, \{\lambda^{thru}_{x_k,r_j} : r_j \in R_{x_k}\}, r_k\Big) \tag{7.8}$$

7.4 Data Exchange Configuration Algorithms

The core algorithms of FireDeX 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*.

7.4.1 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. They depend on the rate of successful event delivery Ξ_{r_j}. The overall utility for a subscriber depends on each of its subscriptions' utilities and is defined as:

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

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

contend with).

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 FFs). We define the overall utility of the configuration for all subscribers as a sum over each individual subscriber's utility:

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

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

$$U_{r_j}(\Xi_{r_j}) = \alpha_{r_j} \log(1 + \Xi_{r_j}) \tag{7.9}$$

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

7.4.2 Priority Assignment Algorithm

FireDeX leverages the above quantified utility metrics to assign priorities for each data flow in a manner that aims to maximize the overall 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.

Because the assignment of discrete priorities to maximize 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 metric essentially measures *information value per unit bandwidth* and lets FireDeX consider that some high-value subscriptions may consume a lot of network resources. We define this metric as:

$$\frac{\widehat{U_{r_j}}}{G_{v_j} \lambda_{b_k, r_j}^{notify}} \tag{7.10}$$

To approximate a solution to the priority-assignment problem, we propose the following greedy approach for each subscriber s_i:

1. Sort the subscriptions $r_j \in R_{s_i}$ by (7.10)
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} 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 (e.g. $|F_{s_i}| > |Y|$) by preferring higher priorities first.

This priority assignment ensures 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.

7.4.3 Ensuring Queue Stability via Preemptive Drop Rates

Given the above priority assignment, FireDeX further fine-tunes the subscriptions' successful notification rate Ξ_{r_j}. Based on the requested subscription utility functions and current network state (e.g. bandwidth constraints), it applies a preemptive packet dropping policy. This improves the overall utility of the system's configuration by essentially allocating available bandwidth to the network flows. Crucially, 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, FireDeX preemptively drops lower-priority packets. The algorithms presented here determine with what probability packets of each network flow should be dropped ($\Omega(f_j)$) to improve situational awareness while ensuring queue stability.

Not only does ensuring queue stability improve system performance, it also satisfies conditions necessary for our analytical model's results to prove accurate. Let $\rho_Q = \frac{\lambda}{\mu}$ be the server utilization (i.e. probability that the server is busy) of the corresponding queue (e.g. $Q^{out}_{x_k}$). By [89], the system remains *unsaturated* (i.e. queue stability is ensured) when $\rho_Q < 1$. For FireDeX'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 $\rho_{Q^{in}_{x_k}} = \frac{\lambda^{in}_{x_k,f_j}}{\mu^{in}_{x_k}}$. FireDeX's multi-class queues $Q^{um}_{x_i}$ and $Q^{out}_{x_k}$ have per-topic and per-subscription arrival and service rates respectively. We determine their server utilization as:

$$\rho_{Q^{um}_{x_i}} = \sum_{P_{x_i}} \sum_{v_j \in V_{p_i}} \frac{\lambda^{pub}_{p_i,v_j}}{\mu^{um}_{x_i,v_j}} \tag{7.11}$$

$$\rho_{Q_{x_k}^{out}} = \sum_{r_j \in R_{x_k}} \frac{\lambda_{x_k,r_j}^{thru}}{\mu_{x_k,r_j}^{out}} \tag{7.12}$$

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 $\tilde{\rho} > 0$. Setting $\tilde{\rho}$ even higher provides ample buffer within the SDN switch queues for resilience to temporary notification rate spikes that might otherwise lead to queue saturation. 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 (refer to Fig. 7.13 for an example).

Therefore, the following drop rate policies set Ω such that:

$$\rho_{Q_{x_k}^{out}} = 1 - \tilde{\rho} \tag{7.13}$$

Flat drop rates: this simple naive policy sets all drop rates equal to satisfy Eq. (7.13) by

solving Eq. (7.12) for a parameter β such that:

$$\Omega(f_j) = \beta \tag{7.14}$$

Linear drop rates: this more information value-aware policy sets the drop rates for each network flow according to its assigned priority level. It solves Eq. (7.12) for a parameter β that satisfies Eq. (7.13) with drop rates set to:

$$\Omega(f_j) = \beta\Phi(f_j) \tag{7.15}$$

Exponential drop rates: similar to *Linear*, this policy sets drop rates according to priority level. It solves Eq. (7.12) for a parameter β that satisfies (7.13) with drop rates set to:

$$\Omega(f_j) = 1 - \beta^{-\Phi(f_j)} \tag{7.16}$$

Appendix B presents derivations for how exactly to efficiently solve the above drop rate heuristics.

Optimized drop rates: the following convex optimization formulation assigns drop rates to maximize overall utility. Given the previously-assigned priorities as input, FireDeX assigns

drop rates by solving the following convex optimization problem:

$$\begin{aligned} \text{maximize} \quad & U \\ \text{subject to} \quad & \Omega(f_j) \in [0,1], \forall f_j \in F \\ & \rho_{Q^{out}_{x_k}} \leq 1 - \tilde{\rho},\ \forall x_k \in X \end{aligned} \tag{7.17}$$

Note that the second constraint ensures available bandwidth constraints are met (i.e. queue stability) according to the $\tilde{\rho}$ parameter.

As long as the chosen utility functions are concave (e.g. logarithm) within the feasible domain of assigned drop rates, then (7.17) can be expressed as a convex optimization problem and efficiently solved. Hence, we define a utility function, such as that given in (7.9), that takes as input our analytical model for determining $\lambda^{sub}_{r_j}$. Because this is an affine function over the drop rates, we can optimally solve for drop rates that maximize the overall system utility. We used CVXPY [10, 56] to implement this approach in the FireDeX middleware.

7.5 Prototype Implementation

We now present a prototype implementation of the FireDeX middleware. It implements the analytical model and algorithms discussed in §7.3 and §7.4 respectively. Consider this implementation according to the three layers shown in Fig. 7.5. The FireDeX Coordinator Service (FCS) extends the data exchange broker by managing the network infrastructure through SDN. It runs the algorithms to compute priorities and drop rate policies, and then it configures the SDN switches through an SDN controller to enforce these policies. We invite the reader to try out FireDeX and find more up-to-date details than we could fit below by visiting our source code repository: `https://github.com/boulouk/firedex`.

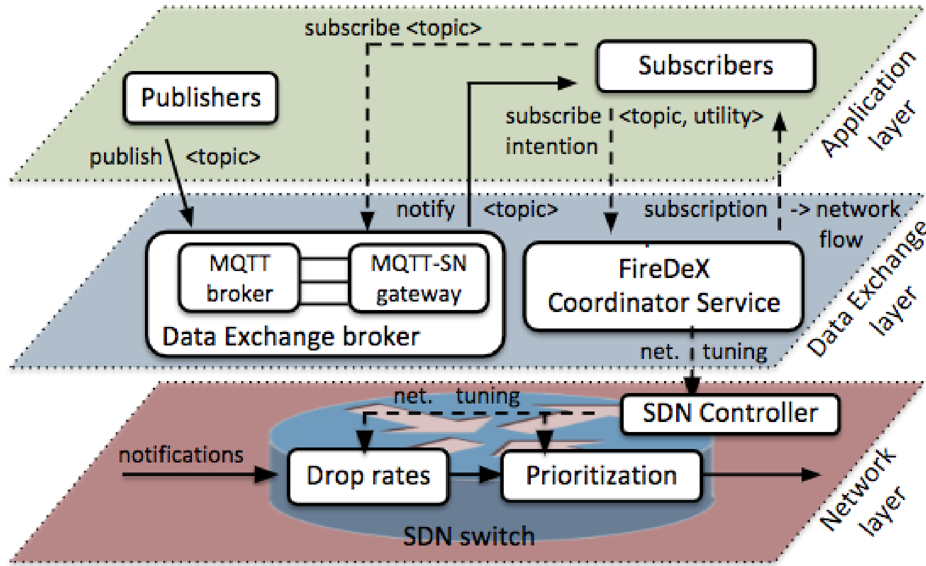


Figure 7.5: The FireDeX cross-layer middleware.

7.5.1 Application layer

The *FireDeX publishers* connect to the data exchange broker through an MQTT connection via the MQTT Paho library [144]. Currently, the data generated by the publishers is simulated via Poisson and Deterministic distributions. Future work will consider replacing it with data coming from real IoT deployments.

The *FireDeX subscribers* connect to the data exchange broker to receive relevant events. They subscribe to these by specifying a topic and the corresponding utility function. Each subscriber establishes multiple MQTT-SN connections to the broker with a client library [135]. Multiple connections allow the network layer to distinguish different event types (i.e. more/less relevant) as described in §7.5.3. Hence, it can apply different priority queueing and event dropping policies as instructed by the FCS. For the subscribers we use MQTT-SN instead of MQTT because it relies on UDP rather than TCP. The latter's re-transmission mechanism interferes with our preemptive packet dropping approach that tolerates some loss of sensor data under constrained bandwidth. Note that UDP does not support fragmentation and reassembly of messages. Therefore, we assume that messages are never fragmented and

Figure 7.6: Web dashboard.

limit their size at the application layer to 256 bytes (before packet headers) due to limitations in the MQTT-SN library. The subscribers first coordinate with the FCS as depicted in Fig. 7.8. This interaction determines the connection they should use for each subscription. Then they open the connections specified by the FCS to the broker and subscribe to each topic through its corresponding connection.

We implemented both the publisher and subscriber applications in Java.

To enable the subscribers' interaction with the FireDeX middleware we implemented a prototype of web dashboard that can be used to receive relevant events specifying topic and the correspondent utility function (see Fig. 7.6 and Fig. 7.7).

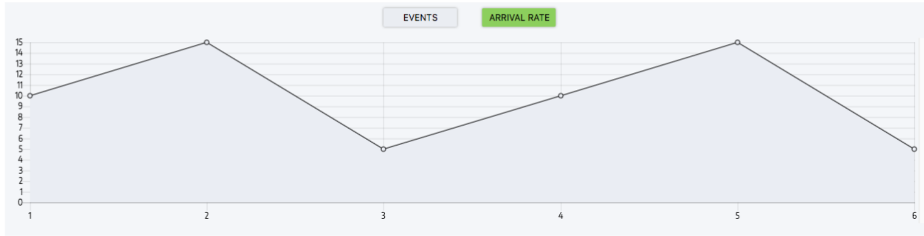


Figure 7.7: Web dashboard.

7.5.2 Data exchange layer

The data exchange layer consists of two components: the unmodified data exchange broker and the FCS.

Data exchange broker. The data exchange layer supports the publish/subscribe paradigm for event delivery. An unmodified MQTT [141] broker facilitates this exchange between publishers and subscribers. While FireDeX supports any MQTT broker implementation (e.g. VerneMQ, HiveMQ, RabbitMQ), we deployed the Moquette [133] broker because it is lightweight, embeddable, open-source and easy to configure. We also run an MQTT-SN gateway [105, 134] co-located with the MQTT broker. It translates from MQTT over TCP (publishers' protocol) to MQTT-SN over UDP (subscribers' protocol).

FireDeX Coordinator Service. The FCS is the "brain" of the FireDeX middleware and follows the workflow shown in Fig. 7.8. It manages user subscriptions by assigning priorities and drop rates as described in §7.4 and shown in Fig. 7.3. The FCS physically resides either in the local network (i.e. building on fire) or a remote location. We implemented the FCS as a REST server using the Python library Flask [85]. The subscribers indicate their topics of interest and corresponding utility functions to the FCS through a HTTP request (i.e. *subscription intent*). The FCS computes priorities and drop rates for each subscriber's network flows, responds with the mapping of subscriptions to connections (i.e. network flows), and then configures the network layer (i.e. SDN infrastructure) to enforce these policies.

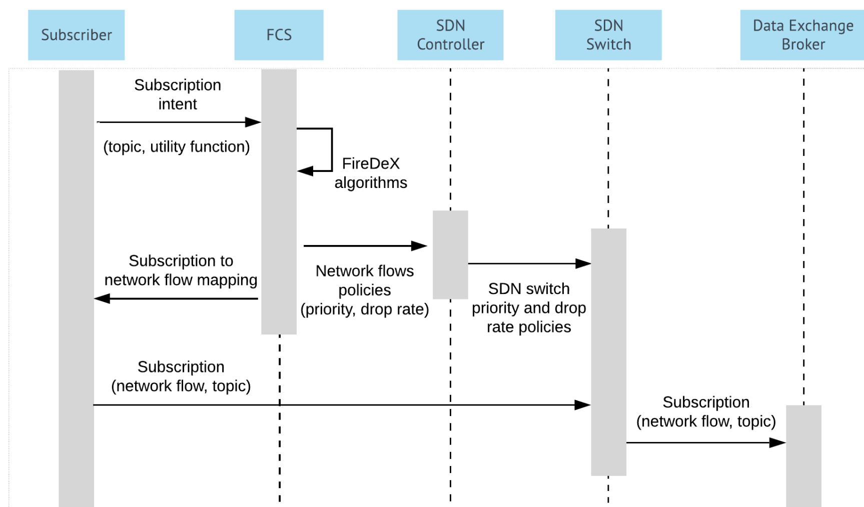


Figure 7.8: Subscribers and FCS interaction workflow.

7.5.3 Network layer

This layer enforces event prioritization and drop rates configured through the SDN protocol OpenFlow [130]. The SDN controller configures the SDN switches with these policies at the direction of the FCS. We implemented two SDN applications through the Ryu [156] SDN controller to facilitate this coordination.

The *Topology Application* monitors network traffic to create an internal graph representation (using the NetworkX library [90]) of the network topology. The topology is used to route packets from source to destination.

The *FireDeX Flow Application* populates the switches' flow and group tables. These tables contain OpenFlow rules that enforce the priority and drop rate policies specified by the FCS. To identify a subscriber's traffic on the network the FireDeX Flow Application matches the packet's header with the network flow information received by the FCS. Network flow information includes the subscriber's IP address and the connection's (i.e. network flow's) transport layer port number. We use the *SELECT* option of OpenFlow *group tables* to set

the drop rate as the weight for one of two different "buckets" that drops the packet rather than forward it normally. The example rules in Listing 7.1 match packets for the subscriber with IP address 10.0.0.1 and an MQTT-SN connection on UDP port 8888. It applies priority class 2 (i.e. queue number) and 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 7.1: Example rules in flow and group tables.

We configure priority queues on the switches via Linux TC [125] since OpenFlow has no unified API to support this. Furthermore, FireDeX must enforce a random per-packet selection of the buckets option (i.e. for the drop rate implementation) rather than the typical approach of hashing packet header fields. Hence, we leverage a modified Open vSwitch (OVS) version [143] that implements this.

7.5.4 Implementation challenges

We resolved several challenges while implementing this prototype:

- **Differentiating events** at the network layer for policy enforcement. We used SDN network flows to distinguish subscriptions served by different connections at the network layer. This was necessary because the OVS switches can only inspect the packets' header (i.e. OSI Layers 2-4), and not the packets' payload. Hence, we needed to bring

the Layer 7 (application layer) concept of subscription topics down to Layers 2-4 for this in-network policy enforcement.

- **Enforcing packet drop rates via SDN** required us to use the modified OVS version described above. This enabled us to implement the drop rate policies through 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 specifying the `enqueue` action as required by our flow rules shown in Listing 7.1.

- **The UDP protocol** was used rather than TCP in our subscribers. Applying drop policies to packets (i.e. events) over a TCP connection triggers its re-transmission mechanism because the sender (i.e. data exchange broker) does not receive an acknowledgement message when the corresponding packet (i.e. event) is dropped. Hence, we used MQTT-SN over UDP so that a packet drop would actually drop that particular event.

- The **Web Socket implementation in JavaScript** that we used in the web-based dashboard implementation proved challenging. It does not allow specifying the local port to which the MQTT socket should bind when opening a connection. Instead, we implemented a gateway between the Java-based subscriber application and the web-based dashboard. This enabled our proposed mechanisms by differentiating the data flows at the gateway and enforcing the FireDeX policies before forwarding events to the subscriber dashboard.

- **Clock-synchronization issues** between publishers and subscribers made gathering accurate results difficult. This is part of the reason for running our experiments on a single machine in Mininet: all applications shared the same system clock in this manner.

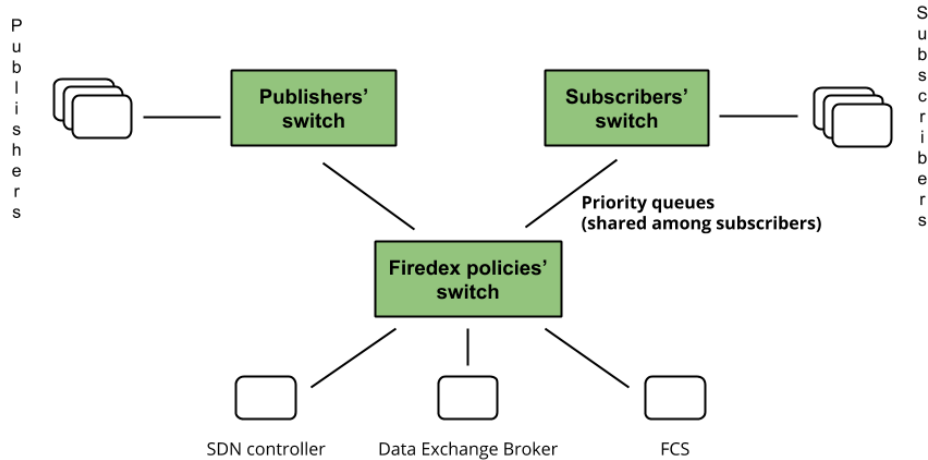


Figure 7.9: Network topology structure used in the the prototype implementation experiments.

- **Our experimental network topology** required additional "dummy" switches for constructing the priority queues shared across all subscribers (see Fig. 7.9). This is due to the fact that each host has its own Ethernet interface connecting it with a switch. Therefore, enqueueing prioritized messages would result in one set of priority queues for each subscriber rather than sharing the queues across all subscribers as our version described here accomplishes.

However, one challenge remains open. The MQTT-SN control messages (e.g. subscription/unsubscription/acknowledgement messages) are sent to the same UDP port (i.e. network flow) on which we apply the drop rate policies. Therefore, some of them may be dropped. To better understand the situation, consider Fig. 7.10 that shows the messages exchanged between broker and subscriber through the SDN network infrastructure. The first message sent from the broker to the subscriber (i.e. the first subscription's acknowledgement message) triggers the creation of the flow/group rules (see Listing 7.1) associated with the network flow to which the subscription belongs. However, when the subscriber subscribes to another topic on the same network flow, the second subscription's acknowledgement message may be dropped because of the drop rate policy applied. This can delay the subscription establishment process for a very long time if this network flow has a high drop rate as-

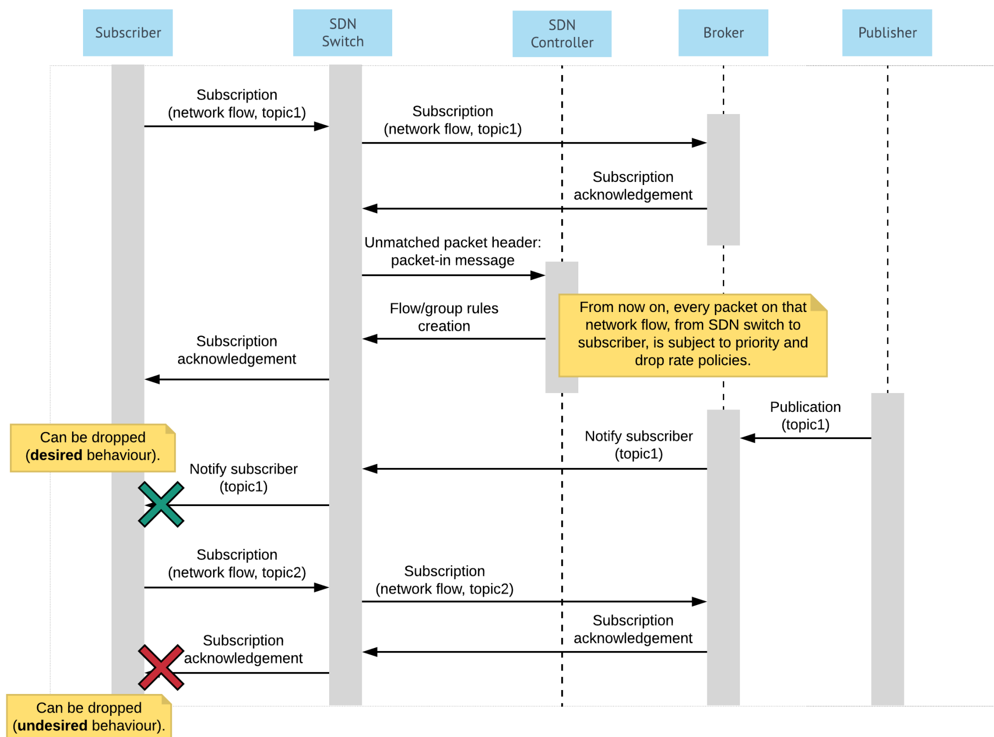


Figure 7.10: Supporting multiple subscriptions per connection is challenging due to dropping control packets.

signed. One possible solution to overcome the aforementioned problem requires changing the protocol between subscribers and FCS. In this case, the SDN controller would create the flow/group rules without setting the drop rates. The subscriber informs the FCS once it finishes establishing all of its subscriptions. Then the FCS can instruct the SDN controller to apply the drop rates policies. However, this complicates the implementation considerably, and so we currently only consider one subscription per network flow in our experiments to avoid this problem.

7.6 Experimental Results

Table 7.2: Default parameters for our experimental configurations used in the simulated experiments.

Data Exchange Parameters					
	#topics ($\lvert V \rvert$)	**pub rate** ($\lambda^{pub}_{p_i,v_j}$)	**event size** (G_{v_j})	**#subscriptions** ($\lvert R_{s_i} \rvert$)	**utility weight** (α_{r_j})
Telemetry data	140	$\text{Exp}(\frac{1}{6}) \in [4,7]$	$\text{Exp}(\frac{1}{110}) \in [90,500]$	70	$\text{Exp}(\frac{1}{0.5}) \in [0.01,2]$
Async. events	60	$\text{Exp}(\frac{1}{4}) \in [3,5]$	$\text{Exp}(\frac{1}{800}) \in [500,1100]$	42	$\text{Exp}(1) \in [0.1,4]$

Network Parameters					
#subscribers ($\lvert S \rvert$)	**#publishers** ($\lvert P \rvert$)	**#flows** ($\lvert F_{s_i} \rvert$)	**#priorities** ($\lvert Y \rvert$)	**bandwidth** (w_{s_i})	ρ **tolerance** ($\tilde{\rho}$)
10	160	9	9	80 Mbps	0.1

FireDeX uses the analytical model given in §7.3.2 to estimate end-to-end response times and success rates for event notifications to interested subscribers. We validate this analysis by using and extending an open source queueing simulator to represent our proposed system. We compare the subscribers' end-to-end response times given by the analytical model with those given by the simulation. Note that we omit the 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 greater numbers of subscribers.

After validating our model, we then use it to evaluate the FireDeX approach for a given configuration. We do this using both the simulator as well as a Mininet-based emulated framework that incorporates our prototype implementation. 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. We use our proposed priority-assignment algorithm, which we call *bandwidth-adjusted-prio*, and the *exponential* drop rate policy. Subsequently, we utilize the analytical model only to compare different algorithms' ability to maximize the overall value of information captured.

7.6.1 Experimental Setup

We developed a Python-based framework that models the real-world scenario given in §7.2.1 and provides input data for our simulations. We configure it to consider two classes of topics that represent events: sensor telemetry readings published periodically and asynchronously-published notifications that indicate real-world phenomena detected from analysis of raw sensor readings. This framework leverages the probability distributions and parameters given in Table 7.2 to generate random configurations for each publisher, subscriber, the broker, and the network. For example, it selects the publication rate and packet size of events from the given distributions. Note that we bound these values to maintain more realistic parameters by redrawing a new one when it lies outside the given range. Note that the actual topics published and subscribed to are chosen uniformly at random from those available.

The model presented in §7.3 generically captures a very 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 subscriber. Recall that this represents the bottleneck bandwidth and 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 instead on the variable delays our model aims to capture.

7.6.1.1 Queueing Network Simulator

After generating these configuration parameters for a single instance of a scenario, the above framework feeds them into a simulator to drive its pseudo-random number generators. 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 [83], 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 to: *i)* represent the queueing network introduced in Fig. 7.4; *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 FireDeX, we generate parameters and run the simulator 10 times for each configuration and then average across these. Each run generates approximately 6,500,000 publications 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 [125] and AMQP 0.9.1 [14] only support 8 and 10 priority queues respectively.

7.6.1.2 Prototype Emulation

The experimental framework we developed configures an emulated network using Mininet [1]. This uses OVS [3] 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 and run our FireDeX middleware implementation described in §7.5. OVS switches connect via the SDN southbound protocol OpenFlow [130] to the distributed SDN controller platform Ryu [156] running on the same machine. The publisher/subscriber hosts themselves produce output files from which we calculate the results depicted below.

The experimental framework configures the managed SDN switches to create a number of priority queues (the configurable parameter $|Y|$). Because OpenFlow does not support a unified API for creating these queues, we currently do this using Linux TC [125], which 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 the default traffic through the highest priority queue, and the 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.

Note that Linux TC 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). This in turn further 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

delay of previous packets. The aforementioned issues have to be considered while evaluating experiments results.

7.6.2 Validating our Queueing Network Model

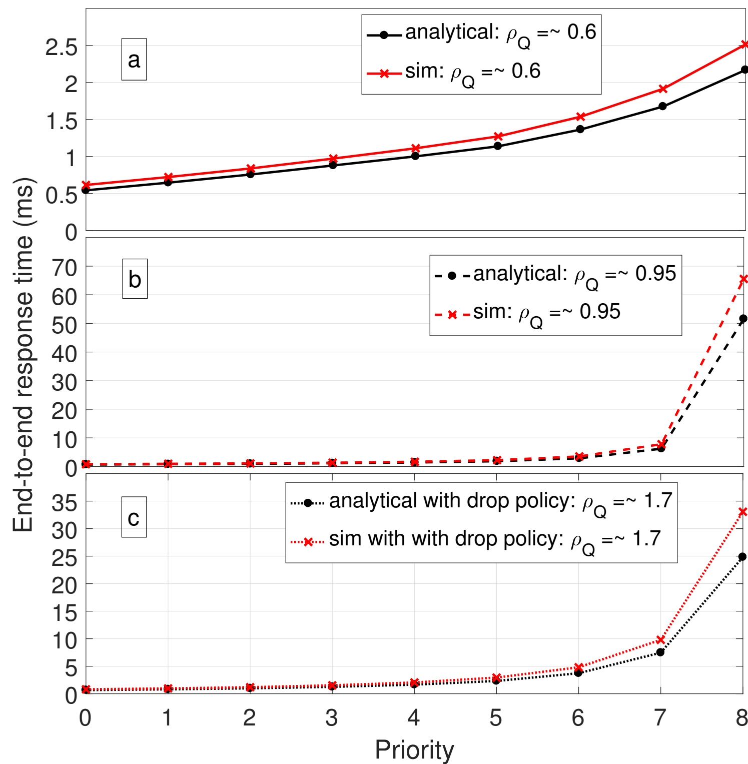


Figure 7.11: Analytical vs. simulation end-to-end response times for varying traffic loads ($\rho_{Q_{x_k}^{out}}$).

To prove the accuracy of the theoretical analysis we developed in §7.3.2, we now compare its estimated performance metrics with those calculated from the aforementioned simulator.

7.6.2.1 Varying traffic loads

Recall that the SDN switch's outbound queue (shown in Fig. 7.4) captures the bottleneck bandwidth of the network route from broker to subscriber. FireDeX 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 system's network traffic load: *a)* medium-load conditions ($\rho_{Q^{out}_{x_k}} = 0.6$); *b)* high-load conditions (i.e. close to saturation – $\rho_{Q^{out}_{x_k}} = 0.95$); *c)* overloaded conditions (i.e. saturated – $\rho_{Q^{out}_{x_k}} = 1.7$). Note that the saturated case (3rd) corresponds to the default parameters in Table 7.2. 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. 7.11 shows the results of these experiments according to assigned priority class and averaged across all topics, subscribers, etc. Comparing the curves of both the simulated measurements and the analytical results obtained by Eq. (7.3) reveal our model's high accuracy. We notice small differences for events with lower priority levels. In particular, note priority level 8's differences: 0.35 ms in Fig. 7.11a, 13.98 ms in Fig. 7.11b and 8.24 ms in Fig. 7.11c. Because the system approaches saturation in Figs. 7.11b and 7.11c, we deem these results acceptable. In Fig. 7.11c, FireDeX uses our drop policy mechanism to drop packets at the SDN switch and return the system to below saturation (i.e. $\rho_{Q^{out}_{x_k}} = 0.9$ by using $\tilde{\rho} = 0.1$).

7.6.2.2 Scaling up number of subscribers

We now validate our analytical model's accuracy under varying numbers of subscribers: $|S|$ = 1, 10, 20, 50, 100. To maintain the same degree of system saturation (i.e. $\rho_{Q^{out}_{x_k}} = 1.7$), we increase the bandwidth proportional to the number of subscribers: $w_{x_k,s_i} = 8Mbps$. We keep all other parameters according to Table 7.2. According to these parameters, we measure the simulated mean response times and plot them vs. those calculated using Eq. (7.3) in Fig. 7.12. 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.

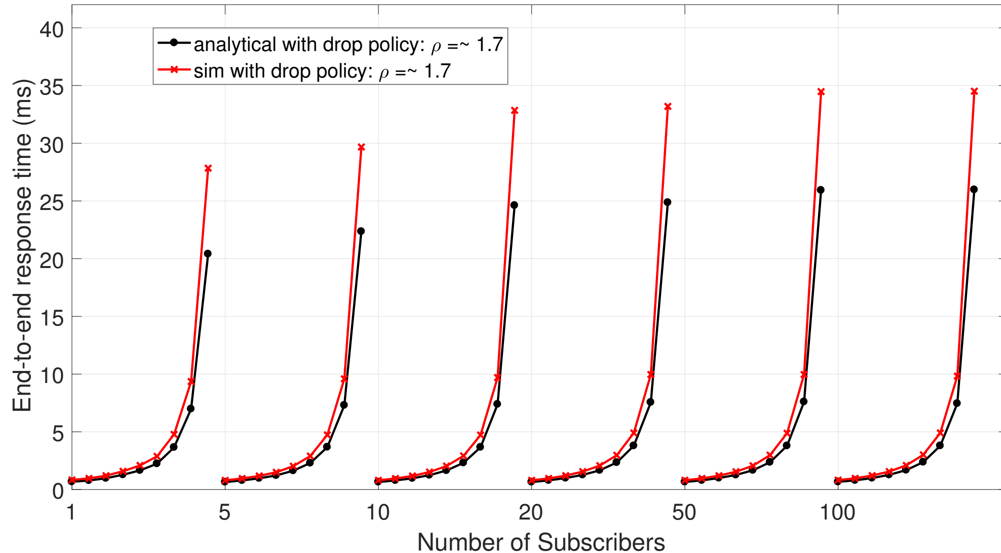


Figure 7.12: Analytical vs. simulation end-to-end response times for varying numbers of subscribers.

7.6.3 Evaluating the FireDeX Approach

We now compare our approach's efficacy with that of an unprioritized system and a system without preemptive packet drops. We will evaluate the trade-off between response times and success rates. However, we discuss in more detail the concept of network switch buffers and their limited capacity within the context of FireDeX. Recall from §7.4.3 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. Fig. 7.13 depicts the detrimental effects on response time of selecting too low a value for $\tilde{\rho}$ (i.e. the buffers grow too large). As evidenced by the linear decrease in success rate, we must carefully analyze the trade-off in expected queue size with achievable utility when tuning the data exchange. These results helped determine the default of $\tilde{\rho} = 0.1$ that we 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

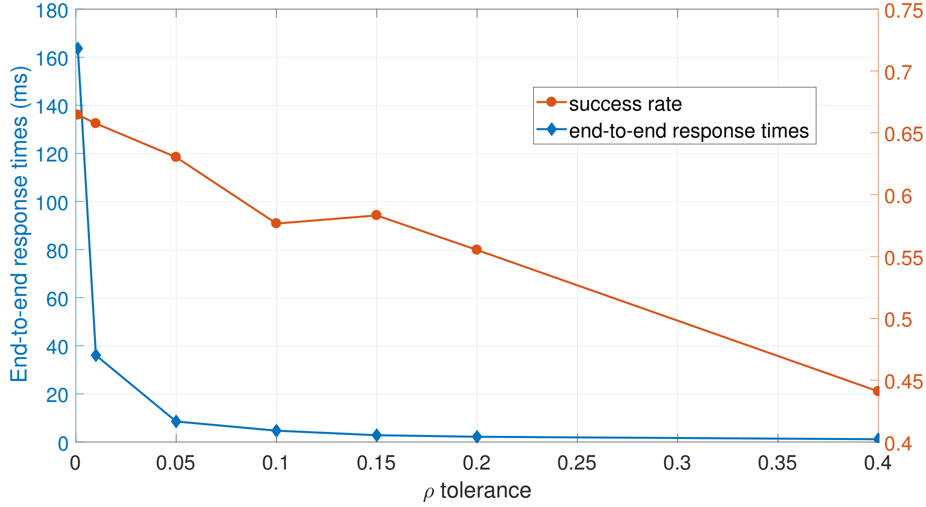


Figure 7.13: Comparing various $\tilde{\rho}$ values.

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[43]. 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 FireDeX approach)

These experiments use the parameters given in Table 7.2. Figs. 7.14 and 7.15 show the success rates and end-to-end response times, respectively. Configuration (i) results in a 58% average success rate and 0.9 sec average response time regardless of assigned priority.

The configuration (ii) experiments used the algorithm we proposed in §7.4.2, which we call *bw-adjusted-prio*, for assigning priorities to each network flow (i.e. to their contained

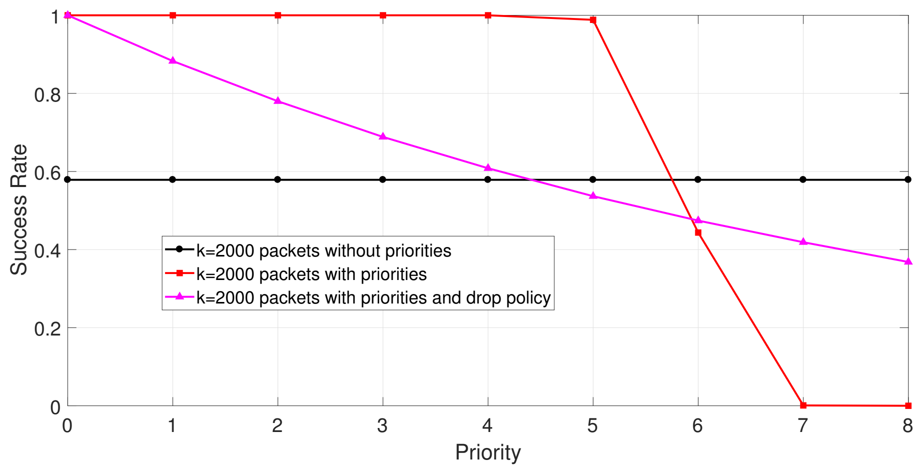


Figure 7.14: Comparing success rates for no priorities (i.e. single switch buffer), priorities only, and an added drop policy.

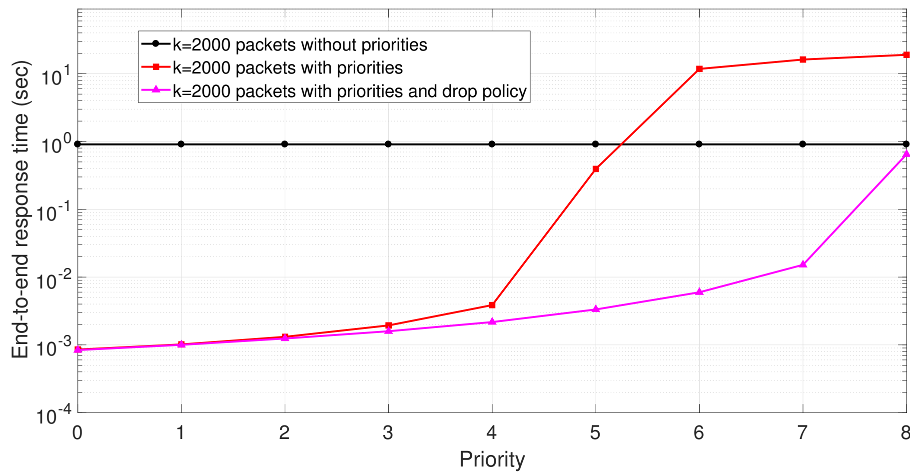


Figure 7.15: Comparing response times for no priorities (i.e. single switch buffer), priorities only, and an added drop policy.

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 FireDeX *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.

7.6.4 Comparing Prioritization & Drop Rate Algorithms for Situational Awareness

We now compare different 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.

Fig. 7.16 compares our proposed priority-assignment algorithm (named *bw-adjusted-prio*) with: no priorities (i.e. a single queue; named *no-prio*), randomly-assigned priorities (i.e. to

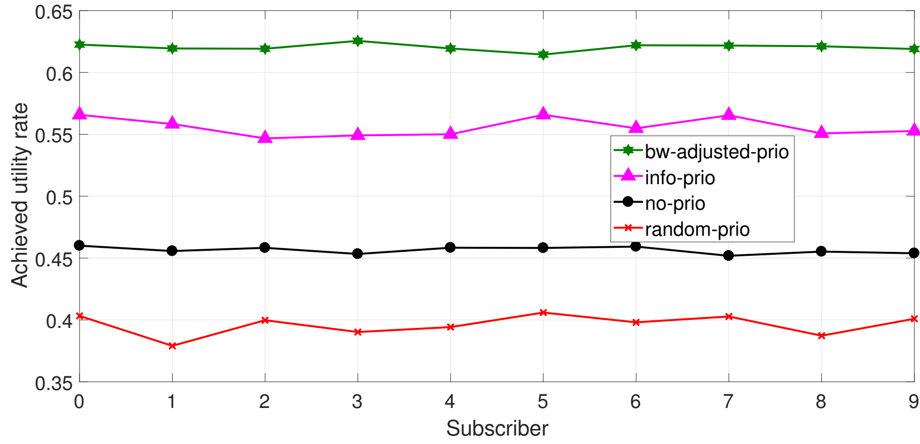


Figure 7.16: Comparing priority-assignment algorithms with other naive approaches.

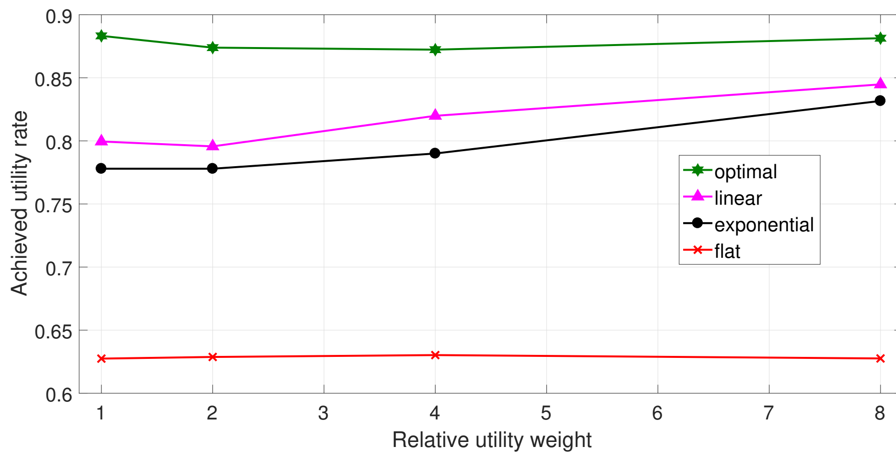


Figure 7.17: Comparing drop rate policies by varying the utility of one subscription class relative to the other.

show the poor performance resulting from inaccurate priority assignment; named *random-prio*), and a naive version of the proposed greedy algorithm that, when ranking subscriptions, considers only max utility without scaling by the bandwidth requirement (named *info-prio*). This figure shows how leveraging network awareness when assigning priorities improves the achieved utility rate by 12% vs. the naive version and 36% vs. no prioritization. Note that we do not assign drop rates for the priority algorithms comparison. Instead, we configure the simulator to drop packets once the buffers fill up in order to compare the priority-assignment algorithms only. With drop rates, FireDeX improves the value of exchanged data by 42% vs. prioritization only and 94% vs. no prioritization.

We then compare the four drop rate-assignment algorithms outlined in §7.4.3 in conjunction with the best-performing *bw-adjusted-prio* algorithm. To demonstrate FireDeX's ability to improve situational awareness for heterogeneous data and information requirements, our experiments varied the utilities of each topic class relative to the other. We increase the random variable distribution parameters used to generate the utility weights (α_{r_j}) of one topic class relative to the other (i.e. telemetry data vs. the higher-utility higher-bandwidth lower-publication frequency asynchronous events). The x-axis in Fig. 7.17 shows this relative difference in terms of the constant factor we scale up the asynchronous event class's α_{r_j} by compared with that of the telemetry data class. Note that we maintain α_{r_j} for the telemetry data as the constant default value given in Table 7.2. These results demonstrate how the optimization-based algorithm captures the most overall utility (i.e. it maximizes situational awareness) under a particular set of network conditions.

7.6.5 Assessing the FireDeX Prototype

We now assess the performance of our prototype implementation and compare its results with those predicted by our analytical and simulation models. We proceed as in §7.6.2:

Table 7.3: Default parameters for our experimental configurations used with the prototype.

Data Exchange Parameters					
	#topics ($\|V\|$)	**pub rate** ($\lambda^{pub}_{p_i,v_j}$)	**event size** (G_{v_j})	**#subscriptions** ($\|R_{s_i}\|$)	**utility weight** (α_{r_j})
Async. events	7	1	100	70	Exp(5) $\in$ [0.01,100]

Network Parameters					
#subscribers ($\|S\|$)	**#publishers** ($\|P\|$)	**#flows** ($\|F_{s_i}\|$)	**#priorities** ($\|Y\|$)	**bandwidth** (w_{s_i})	ρ **tolerance** ($\tilde{\rho}$)
10	10	7	7	320 Kbps	0.1

we first show the results obtained for different traffic loads and then scale up the number of subscribers. Once again, we omit the trivial validation of drop rates as those results exactly matched that we intuitively expected. We change the experiments' configuration parameters to those in Table 7.3 to overcome practical scaling issues in Mininet and the challenges described in §7.5.4. To this end, we reduce the number of subscriptions to have only one subscription per network flow. 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 because of the smaller number of total events published as compared to in the queueing simulator. If we varied the publication rate and event size the same way as in the simulation, it would take longer for the results to converge on the expected value of the exponential distributions. Also, to more closely match the simulation framework and Mininet environment we:

i) Use only asynchronously-published events.

ii) Set the SDN output queue's service rate in the simulation framework to deterministic rather than exponential. This better represents the concept of a switch's bandwidth, which essentially has a constant service rate measured in bytes/second.

We test FireDeX under different network load conditions: *a)* medium-load conditions ($\rho_{Q^{out}_{x_k}}$

$= 0.5$); *b)* high-load conditions (i.e. close to saturation – $\rho_{Q^{out}_{x_k}} = 0.9$); *c)* overloaded conditions (i.e. saturated – $\rho_{Q^{out}_{x_k}} = 1.7$). Fig. 7.18 shows the end-to-end response time obtained from an experiment with 10 subscribers and the aforementioned load conditions.

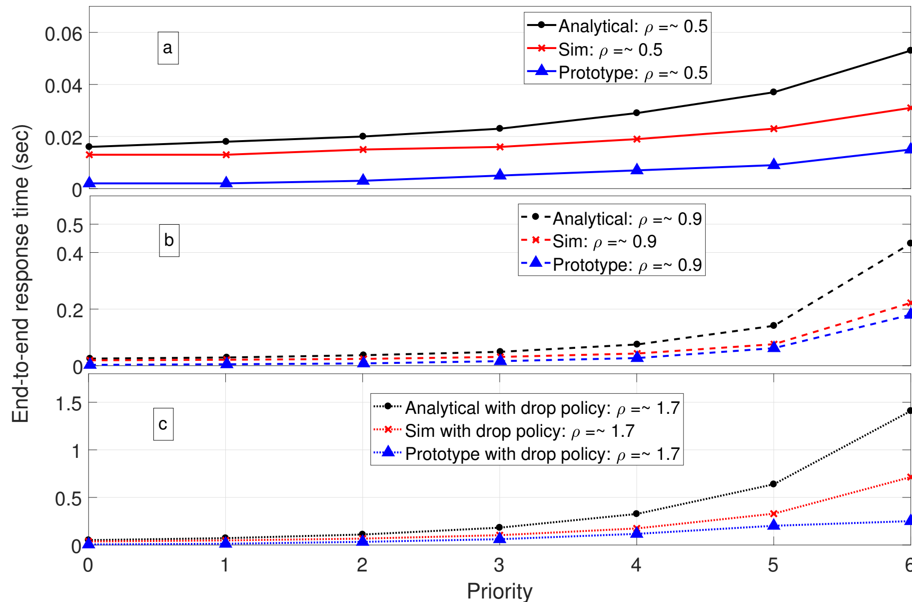


Figure 7.18: Analytical vs. simulation vs. prototype end-to-end response times for varying traffic loads ($\rho_{Q^{out}_{x_k}}$).

We see from these results that the prototype's performance closely matches that of the simulator and that predicted by the analytical model. However, we identified two reasons for the slight differences. First, the prototype and simulation results match each other closely but differ from the analytical model's predictions due to the deterministic service rates vs. the latter's exponential service rates. Second, we still see significant difference in response times for the simulator vs. prototype due to the emulator's lack of proper queueing delay. Recall that, as discussed in §7.6.1.2, transmission delay and queueing delay are not correctly applied by Linux TC. Hence, we expected to observe the same trends for end-to-end response time, but with lower values. This matches the results we see here in Fig. 7.18.

Note that just as in §7.6.2 the lowest priority events experience the largest difference in response time under saturated conditions. However, we see a difference in response time

that remains constant across priority classes in the unsaturated setting (Fig. 7.18a). For the saturated and overloaded conditions (Fig. 7.18b & 7.18c), we see larger differences for the lowest priorities in part due to the above issue regarding transmission delay. Because these messages are waiting in the queues longer, this difference is compounded further by the lack of transmission delay for each queued message in front of it. Hence, we expect to see the gap between the analytical model and the simulated/emulated results, but a more accurate emulated setting should further reduce the difference between simulated vs. emulated. Even with these slight differences, we consider these results acceptable.

We then scaled up the number of subscribers as shown in Fig. 7.19. We test the system with numbers 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. 7.19, the implementation again produces results closely matching that of the simulation, even with an increased number of subscribers.

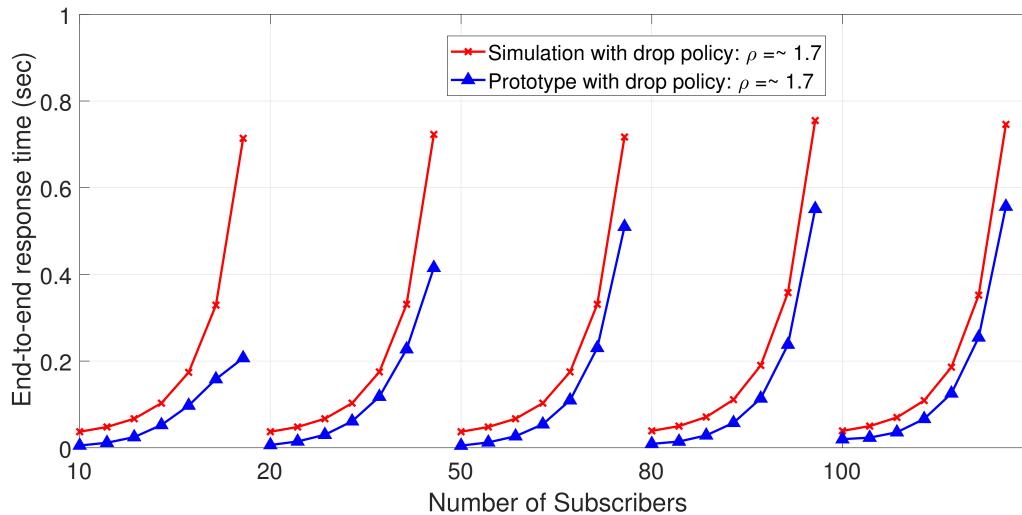


Figure 7.19: Simulation vs. prototype end-to-end response times for varying numbers of subscribers.

7.7 Chapter Summary and Discussion

In this chapter we explored the challenge of managing multiple heterogeneous mission-critical information flows at the edge. We added to the previous chapters' approaches by balancing the needs of different stakeholders (i.e. subscribers and IoT emergency response applications) through a rigorous treatment of data prioritization in a constrained networking environment. To this end, we developed the FireDeX extensible integration middleware. Our proposed SDN-enabled three-layer approach bridges application-specified information requirements, generic data exchange capabilities, and physical network characteristics. Its algorithms assign priorities to subscriptions and tune their bandwidth allocation (i.e. via packet drop rates) to maximize overall situational awareness. Our experimental results showed that this approach greatly improves the performance in terms of information value captured as well as end-to-end delays. We also expanded on our previous work with FireDeX [26] by developing a prototype implementation and comparing its performance with our analytical model and simulation-based studies.

The cross-layer queueing model serves as a sound theoretical framework underpinning the FireDeX middleware and enables the analysis used to drive its algorithms. Its modular design supports composition of alternative queueing models. Hence it lays the groundwork for many potential extensions and alterations, some of which we will address in the future work discussed later in Chap. 8.

7.7.1 Integrating FireDeX Into Our Proposed Middleware

FireDeX builds on our overall middleware approach proposed in this thesis by balancing the needs of multiple mission-critical applications and data consumers. While our proof-of-concept prioritizes events within a smart building during an active fire response, this

technique fits within our greater vision of resilient IoT data exchange. Through our proposed prioritization approach, queueing theoretic model, and algorithms, our complete system can now leverage edge resources to enhance the resilience of multiple mission-critical IoT applications. The prototype implementation described previously incorporates this data exchange logic into the consumer device and a coordinator service as shown in Fig. 7.20 and Fig. 7.5.

FireDeX also expands on the previous two chapters' approaches by potentially incorporating their techniques in support of various mission-critical IoT applications. The highly-generalized FireDeX SDN abstraction (i.e. big switch) can deploy the GeoCRON and Ride techniques as well without requiring coordination with the FireDeX modules. Alternatively, our extensible queueing model and analysis could also incorporate these previous techniques to further improve awareness of and control over the entire data exchange process.

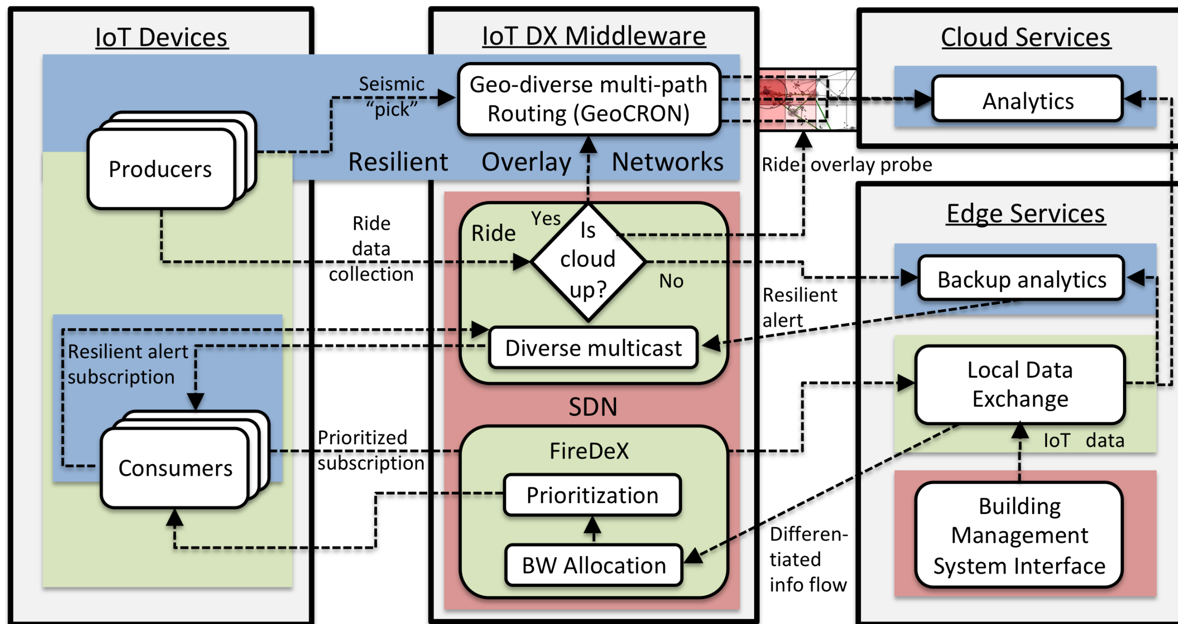


Figure 7.20: FireDeX adds data exchange logic to the consumers and middleware service, but enforces prioritization in the network.

Chapter 8

Conclusion

In this thesis we proposed a cross-layer middleware approach to enabling resilient communications in support of IoT data exchange. We identified research challenges during the SCALE project that served as a proof-of-concept IoT deployment and experimental testbed. We explored this approach within two different IoT-enhanced mission-critical scenarios (i.e. seismic and structure fire response). Each of the core chapters detailed our work in three projects that successively built upon each other to implement this approach. We first constructed geo-diverse resilient overlays for cloud-centric data collection in Chap. 5. We then ensured an application-specified maximum cloud connection downtime in Chap. 6 by monitoring these overlays for challenges, failing over to the edge in response, and improving dissemination of local alerts through a network-aware resilient multicast mechanism. In Chap. 7, we considered the information requirements of multiple mission-critical apps and ensured delivery of the most important messages first based on a formal model of the data exchange system, its resource constraints, and application requirements.

8.1 Future Directions

As we consider developing a comprehensive middleware solution that incorporates the techniques proposed in this thesis, many other challenges to IoT data exchange remain open. Here we outline the future directions for each of the core chapters as well as general challenges to IoT data exchange.

GeoCRON future directions:

- More realistically modeling correlated failures due to shared link bundles and cascading failures (e.g. power grid dependencies).
- Incorporating wireless ad-hoc networking so that physically hyper-close nodes can work closely on data delivery and event detection.
- Considering an even more distributed architecture for GeoCRON in which devices nearby each other gossip about their sensor readings before uploading them. This can reduce the volume of data uploaded through data compression as well as avoiding upload of false positive readings.
- Modeling and testing against dynamic failures in which failures happen over a period of time, rather than instantaneously, and also recover over time. The overlay heuristics and implementation should not negatively impact the network during recovery, and the peers should coordinate overlay maintenance so none become disconnected. Furthermore, overlay peers can exchange information about perceived failures in the network to improve this dynamic adaptation.
- Incorporating additional network technologies (e.g. cellular), infrastructures (e.g. transportation), and failure models (e.g. flood, fire).

- Studying different strategies for overlay usage according to the application requirements of different IoT systems (e.g. quality-of-service levels).
- Studying whether GeoCRON also generalizes to mobile nodes (e.g. smartphones).

Ride future directions:

- Improving data collection using multiple CDPs simultaneously and determining which data should traverse each overlay link.
- Improving data dissemination via wireless ad-hoc networks that facilitate forwarding alerts to other nearby subscribers that were unable to receive the multicast alert directly.
- Completing and exploring the Ride-D retransmission mechanism as well as its selective unicast extension.
- Securing data transmission using tunneling (e.g. IPsec) for collection and secure multicast (e.g. DTLS-based multicast [167]) for dissemination,
- Experimenting with additional network failure models (e.g. shared-risk groups) and application scenarios (e.g. rapidly spreading fire).

FireDeX future directions include expanding on the theoretical model and the prototype system. For example, we aim to extend the theoretical model by relaxing some of the Assumptions in §7.3:

- Considering multiple switches and physical network routes in managing subscriber data flows.
- Considering non-Poisson arrival and service rates by using e.g. G/G/1 queues.

- Converting larger events into many packets (or many events into one packet) by applying the queueing theoretic concept of *batch arrivals* [168].
- Configuring an entire broker network rather than just the BMS's local broker.

Moving forward, we will also build on the FireDeX prototype to explore further IoT middleware challenges in emergency response settings. We plan to deploy it in a smart building on the UCI campus and integrate it with our existing IoT testbed privacy-preserving smart building system [27, 132] named Tippers. Tippers will facilitate an *E-Vault* system for granting emergency responders access to and control over building infrastructure (i.e. devices and network connections) and data, similar to key vaults commonly-used now to help unlock doors during emergencies. We are also developing a SFF situational awareness dashboard. This inter-disciplinary collaborative project will both set the stage for future SFF investigations as well as cutting-edge research in the intersection of emergency response and privacy-preserving cyber-spaces. Armed with this prototype system and application, we plan to conduct drills with fire fighters aimed at identifying, developing, and testing IoT-enabled workflows and technologies for SFF. Specific expected research questions include:

- How users can define utility functions, without understanding the underlying implementation details of FireDeX, for enhancing situational awareness in real-world settings.
- Mobility and multi-network management in hostile communications environments as FFs enter buildings with damaged or otherwise suspect communications infrastructure.
- How to collate heterogeneous data from various sources for targeted timely decision-making.
- Managing dynamic conditions such as failing publisher devices, subscriber churn, 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.

Aside from those challenges listed previously in §1.3.3 and above for each of the three core chapters, future research in the area of general IoT data exchange should also consider:

- **Interoperability** of network and data exchange protocols. Rather than classic approaches of implementing translators for each protocol pair, dynamically-configurable software artifacts can improve flexibility and ease of IoT deployments.
- **Data semantics** can be leveraged to configure network elements for improved operation. As networking technologies mature, programmable Layer 7 switches may support actions based on data contents exposed by protocols such as DDS.
- **Distributed data exchange brokers** must coordinate among each other as well as between physically-dispersed edge deployments.
- **Synchronizing cloud and edge** services to ensure they remain in agreement about various state information, especially system configurations. This also includes the question of how to best recover from a cloud service outage when switching back from edge operation.
- **Mobility** of IoT devices requires tight synchronization between controller entities when a mobile node passes into a region managed by a different controller. This remains an open area of research for SDN especially.

- **Resilient SDN controller placement** to ensure all switches remain connected with their corresponding master controller for no downtime in management, which is especially crucial for resilient operations.

8.2 Towards the Future of Resilient IoT Data Exchange

Altogether, the middleware approach proposed in this thesis enables a more holistic view of and control over the data exchange process. As demonstrated in the individual projects, this occurs at different network and IoT deployment scales. Fig. 8.1 depicts the combined system components that make up our entire middleware and address the challenges set forth by these differing requirements. By converting GeoCRON's overlay network to a direct overlay service offered as an SDN abstraction by network service providers, our entire middleware would comprise very little additional logic necessary on the IoT devices themselves. Rather, it deploys the necessary software artifacts predominately as edge services with some public cloud counterparts. This strategy keeps in line with thin IoT client designs and enables more rapid development and deployment of IoT services.

Unifying the data exchange process through such a middleware further enables modular extensions to be dynamically deployed without requiring modifications to the IoT devices themselves. Furthermore, the nature of SDN allows future iterations to borrow on previous network abstractions just as we did in the main chapters of this thesis. Thus future research can easily explore other SDN capabilities not covered here. The formal queueing theory-based model presented in FireDeX (Ch. 7) can serve as the sound theoretical underpinning through which these extensions might be modeled for developing mathematically-sound configuration algorithms. As such, our work represents a step in the direction of supporting plug-and-play operation in future dynamic IoT-based applications through flexible, efficient, reliable, and timely methods for information exchange.

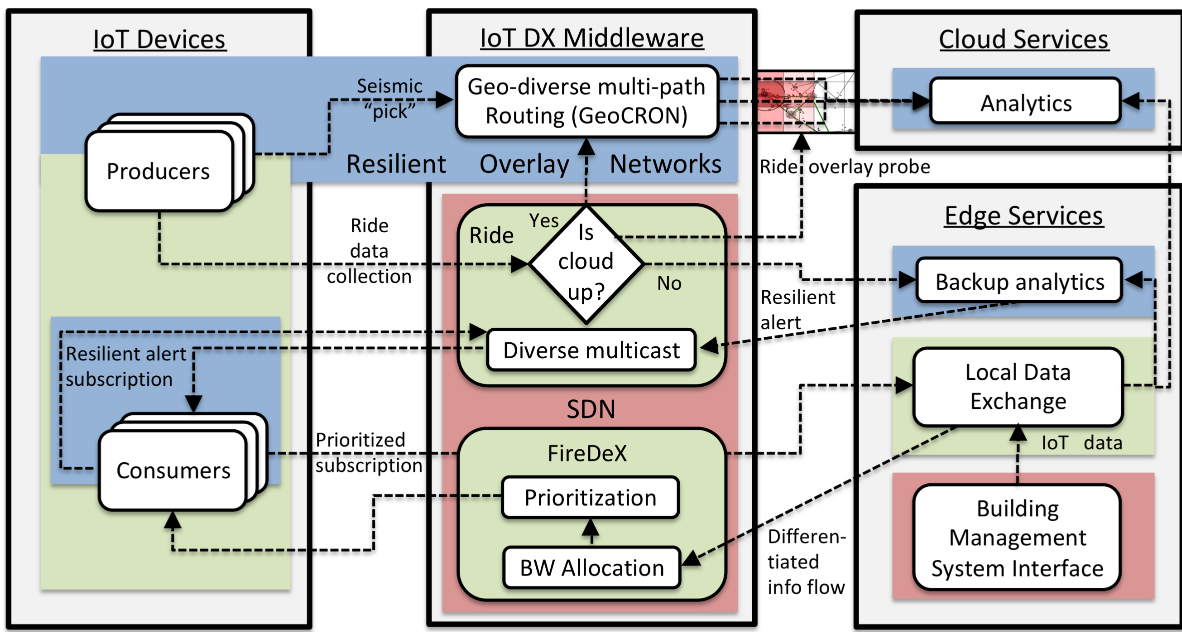


Figure 8.1: Our proposed middleware unifies the techniques described throughout this thesis for a more holistic IoT data exchange.

Bibliography

[1] Mininet: An Instant Virtual Network on your Laptop (or other PC) - Mininet. http://mininet.org/.

[2] Onos: Open network operating system. https://onosproject.org/.

[3] Open vSwitch. http://openvswitch.org/.

[4] Protocol Buffers - Google's data interchange format. https://github.com/google/protobuf.

[5] Pervasive computing for disaster response. http://www.cacr.caltech.edu/projects/PerDis/, Aug 2012.

[6] Community resilience planning guide for buildings and infrastructure systems. *National Institute of Standards and Technology*, 1, July 2015.

[7] Community Seismic Network. http://www.communityseismicnetwork.org/, May 2015.

[8] ns-3. http://www.nsnam.org/, May 2015.

[9] Sentilo. http://www.sentilo.io, Feb 2016.

[10] S. D. Akshay Agrawal, Robin Verschueren and S. Boyd. A rewriting system for convex optimization problems. *Journal of Control and Decision*, 5(1):42–60, 2018.

[11] W. al. SDNPS: A Load-Balanced Topic-Based Publish/Subscribe System in Software-Defined Networking. *Applied Sciences*, 6(4):91, mar 2016.

[12] A. Al-Shabibi, M. De Leenheer, M. Gerola, A. Koshibe, W. Snow, G. M. Parulkar, et al. Openvirtex: A network hypervisor. In *ONS*, 2014.

[13] N. S. Alhassoun, M. Y. S. Uddin, and N. Venkatasubramanian. Safer: An iot-based perpetual safe community awareness and alerting network. In *2017 Eighth International Green and Sustainable Computing Conference (IGSC)*, pages 1–8, Oct 2017.

[14] AMQP Working Group 0-9-1. http://www.amqp.org/specification/0-9-1/amqp-org-download, 2008.

[15] N. An, T. Ha, K. Park, and H. Lim. Dynamic priority-adjustment for real-time flows in software-defined networks. In *2016 17th International Telecommunications Network Strategy and Planning Symposium (Networks)*, pages 144–149, Sept 2016.

[16] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient overlay networks. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, pages 131–145, New York, NY, USA, 2001. ACM.

[17] A. P. Athreya and P. Tague. Network self-organization in the Internet of Things. In *SECON '13*.

[18] T. Bakhshi. State of the Art and Recent Research Advances in Software Defined Networking. *Wireless Communications and Mobile Computing*, 2017:35, 2017.

[19] B. Balaji et al. Brick: Towards a unified metadata schema for buildings. In *BuildSys*, 2016.

[20] C. C. Beard and V. S. Frost. Prioritization of emergency network traffic using ticket servers: A performance analysis. *Simulation*, 2004.

[21] S. Behnel, L. Fiege, and G. Muhl. On quality-of-service and publish-subscribe. In *ICDCS Workshops*. IEEE, 2006.

[22] Y. Bejerano and P. V. Koppol. Link-coloring based scheme for multicast and unicast protection. In *HPSR '13*, 2013.

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

[24] K. Benson, T. Estrada, M. Taufer, J. Lawrence, and E. Cochran. On the powerful use of simulations in the quake-catcher network to efficiently position low-cost earthquake sensors. In *Proceedings of the 2011 IEEE Seventh International Conference on eScience*, ESCIENCE '11, pages 77–84, Washington, DC, USA, 2011. IEEE Computer Society.

[25] K. Benson, S. Schlachter, T. Estrada, M. Taufer, J. Lawrence, and E. Cochran. On the powerful use of simulations in the quake-catcher network to efficiently position low-cost earthquake sensors. *Future Generation Computer Systems*, 29(8):2128–2142, 2013.

[26] K. E. Benson, G. Bouloukakis, C. Grant, V. Issarny, I. Moscholios, S. Mehrotra, and N. Venkatasubramanian. FireDeX: a Prioritized IoT Data Exchange Middleware for Emergency Response. In *Proceedings of the 19th International Middleware Conference*, Middleware '18. ACM, December 2018.

[27] K. E. Benson, C. Fracchia, G. Wang, Q. Zhu, S. Almomen, J. Cohn, L. Darcy, D. Hoffman, M. Makai, J. Stamatakis, et al. Scale: Safe community awareness and alerting leveraging the internet of things. *IEEE Communications Magazine*, 53(12):27–34, 2015.

[28] K. E. Benson, Q. Han, K. Kim, P. Nguyen, and N. Venkatasubramanian. Resilient overlays for iot-based community infrastructure communications. In *2016 IEEE First International Conference on Internet-of-Things Design and Implementation (IoTDI)*, pages 152–163. IEEE, 2016.

[29] K. E. Benson and N. Venkatasubramanian. Improving sensor data delivery during disaster scenarios with resilient overlay networks. In *Third International Workshop on Pervasive Networks for Emergency Management 2013 (PerNEM 2013) as part of the 2013 IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops)*, pages 547–552. IEEE, 2013.

[30] K. E. Benson, G. Wang, Y.-J. Kim, and N. Venkatasubramanian. Ride: A Resilient IoT Data Exchange Middleware Leveraging SDN and Edge Cloud Resources. In *2018 IEEE Second International Conference on Internet-of-Things Design and Implementation (IoTDI)*. IEEE, 2018.

[31] L. Bertaux, A. Hakiri, S. Medjiah, P. Berthou, and S. Abdellatif. A DDS/SDN Based Communication System for Efficient Support of Dynamic Distributed Real-Time Applications. In *2014 IEEE/ACM 18th International Symposium on Distributed Simulation and Real Time Applications*, pages 77–84. IEEE, oct 2014.

[32] S. Bhowmik, M. A. Tariq, J. Grunert, and K. Rothermel. Bandwidth-efficient content-based routing on software-defined networks. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, DEBS '16, pages 137–144, New York, NY, USA, 2016. ACM.

[33] S. Bhowmik, M. A. Tariq, L. Hegazy, and K. Rothermel. Hybrid content-based routing using network and application layer filtering. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pages 221–231, June 2016.

[34] M. Blackstock and R. Lea. Iot interoperability: A hub-based approach. In *2014 International Conference on the Internet of Things (IOT)*, pages 79–84, Oct 2014.

[35] A. Blenk, A. Basta, M. Reisslein, and W. Kellerer. Survey on network virtualization hypervisors for software defined networking. *IEEE Communications Surveys Tutorials*, 18(1):655–685, Firstquarter 2016.

[36] A. Bley and J. Neto. Approximability of 3- and 4-Hop Bounded Disjoint Paths Problems. In *Proceedings of IPCO '10*.

[37] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[38] M. Botts and A. Robin. Opengis sensor model language (sensorml) implementation specification. *OpenGIS Implementation Specification OGC*, 7(000), 2007.

[39] G. Bouloukakis, N. Georgantas, A. Kattepur, and V. Issarny. Timeliness evaluation of intermittent mobile connectivity over pub/sub systems. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, L'Aquila, Italy, Apr. 2017.

[40] G. Bouloukakis, I. Moscholios, N. Georgantas, and V. Issarny. Performance modeling of the middleware overlay infrastructure of mobile things. In *IEEE International Conference on Communications*, Paris, France, May 2017.

[41] A. Bourke, J. OBrien, and G. Lyons. Evaluation of a threshold-based tri-axial accelerometer fall detection algorithm. *Gait & Posture*, 26(2):194 – 199, 2007.

[42] I. N. Bozkurt and T. Benson. Contextual router: Advancing experience oriented networking to the home. In *Proceedings of the Symposium on SDN Research*, SOSR '16, pages 15:1–15:7, New York, NY, USA, 2016. ACM.

[43] Buffer requirements. `https://people.ucsc.edu/~warner/buffer.html`, 2008.

[44] S. Bushby, H. M. Newman, and M. A. Applebaum. *NISTIR 6392 GSA Guide to Specifying Interoperable Building Automation and Control Systems Using ANSI/ASHRAE Standard 135-1995, BACnet*. National Institute Of Standards and Technology, 1999.

[45] A. L. Caro, P. D. Amer, and R. R. Stewart. Transport layer multihoming for fault tolerance in fcs networks. In *MILCOM*, volume 2, pages 949–953. Citeseer, 2003.

[46] A. Chakrabarti and G. Manimaran. Reliability constrained routing in qos networks. *IEEE/ACM Transactions on Networking*, 13(3):662–675, June 2005.

[47] K. Chen, S. He, B. Chen, J. Kolb, R. H. Katz, and D. E. Culler. Bearloc: A composable distributed framework for indoor localization systems. In *Workshop on IoT-Sys*, 2015.

[48] K. Cho, C. Pelsser, R. Bush, and Y. Won. The Japan Earthquake: The Impact on Traffic and Routing Observed by a Local ISP. In *Proceedings of the Special Workshop on Internet and Disasters*, SWID '11, pages 2:1–2:8. ACM, 2011.

[49] H.-Y. Choi, A. L. King, and I. Lee. Making DDS really real-time with openflow. In *Proceedings of the 13th International Conference on Embedded Software - EMSOFT '16*, pages 1–10, New York, New York, USA, 2016. ACM Press.

[50] R. Clayton, T. Heaton, M. Chandy, A. Krause, M. Kohler, J. Bunn, R. Guy, M. Olson, M. Faulkner, M. Cheng, L. Strand, R. Chandy, D. Obenshain, A. Liu, and M. Aivazis. Community seismic network. *Annals of Geophysics*, 54(6), 2012.

[51] E. Cochran, J. Lawrence, C. Christensen, and A. Chung. A novel strong-motion seismic network for community participation in earthquake monitoring. *IEEE Instrumentation Measurement Magazine*, 12(6):8–15, December 2009.

[52] A. Dainotti, C. Squarcella, E. Aben, K. C. Claffy, M. Chiesa, M. Russo, and A. Pescapé. Analysis of country-wide internet outages caused by censorship. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, pages 1–18. ACM, 2011.

[53] A. Das, C. U. Martel, and B. Mukherjee. A partial-protection approach using multipath provisioning. In *2009 IEEE International Conference on Communications*, pages 1–5, June 2009.

[54] S. Das, K. Yamada, H. Yu, S. S. Lee, and M. Gerla. A qos network management system for robust and reliable multimedia services. In K. C. Almeroth and M. Hasan, editors, *Management of Multimedia on the Internet*, pages 1–11, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[55] S. Dawson-Haggerty, A. Krioukov, J. Taneja, S. Karandikar, G. Fierro, N. Kitaev, and D. Culler. Boss: Building operating system services. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 443–458, Berkeley, CA, USA, 2013. USENIX Association.

[56] S. Diamond and S. Boyd. CVXPY: A Python-embedded modeling language for convex optimization. *Journal of Machine Learning Research*, 17(83):1–5, 2016.

[57] C. Dixon, R. Mahajan, S. Agarwal, A. J. Brush, B. Lee, S. Saroiu, and P. Bahl. An operating system for the home. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 25–25, Berkeley, CA, USA, 2012. USENIX Association.

[58] J. Dong, X. Ren, D. Zuo, and H. Liu. An adaptive failure detector based on quality of service in peer-to-peer networks. *Sensors (Basel, Switzerland)*, 14(9):16617–16629, 2014.

[59] R. Durner, A. Blenk, and W. Kellerer. Performance study of dynamic qos management for openflow-enabled sdn switches. In *2015 IEEE 23rd International Symposium on Quality of Service (IWQoS)*, pages 177–182, June 2015.

[60] A. El-Mougy, M. Ibnkahla, and L. Hegazy. Software-defined wireless network architectures for the internet-of-things. In *2015 IEEE 40th Local Computer Networks Conference Workshops (LCN Workshops)*, pages 804–811, Oct 2015.

[61] G. Enyedi and G. Rétvári. Finding multiple maximally redundant trees in linear time. *Periodica Polytechnica Electrical Engineering*, 54(1-2):29, 2010.

[62] A. et al. Increasing network resilience through edge diversity in nebula. *SIGMOBILE Mob. Comput. Commun. Rev.*, 16(3):14–20, Dec. 2012.

[63] B. et al. Provable data plane connectivity with local fast failover: Introducing openflow graph algorithms. In *Proceedings of HotSDN '14*.

[64] B. et al. ONOS. In *Proceedings of HotSDN '14*, 2014.

[65] F. et al. *RFC 7761 - Protocol Independent Multicast - Sparse Mode (PIM-SM).*

[66] G. et al. Recovery from link failures in a Smart Grid communication network using OpenFlow. In *SmartGridComm '14*.

[67] K. et al. Efficient and reliable application layer multicast for flash dissemination. *IEEE TPDS*, 25(10):2571–2582, Oct 2014.

[68] M. et al. Survivor: An enhanced controller placement strategy for improving sdn survivability. In *Globecom '14*.

[69] Q. et al. A Software Defined Networking architecture for the Internet-of-Things. In *NOMS 2014*.

[70] R. et al. MAPCloud: Mobile Applications on an Elastic and Scalable 2-Tier Cloud Architecture. In *UCC 2012*.

[71] S. et al. *RFC 7252 - The Constrained Application Protocol (CoAP).*

[72] T. et al. Coapthon: Easy development of coap-based iot applications with python. In *WF-IoT '15*.

[73] T. et al. Pleroma: A sdn-based high performance publish/subscribe middleware. In *Middleware '14*.

[74] T. et al. Software-defined and value-based information processing and dissemination in iot applications. In *NOMS '16*.

[75] W. et al. Ubiflow: Mobility management in urban-scale software defined iot. In *INFOCOM '15*.

[76] W. et al. Iproiot: An in-network processing framework for iot using information centric networking. In *ICUFN 2017*, 2017.

[77] Z. et al. Dynamic application-aware resource management using software-defined networking: Implementation prospects and challenges. In *NOMS '14*.

[78] Z. et al. Minimum-Cost Multiple Paths Subject to Minimum Link and Node Sharing in a Network. *IEEE/ACM Transactions on Networking*, 18(5):1436–1449, 10 2010.

[79] Z. et al. The cloud is not enough: Saving iot from the cloud. In *Proceedings of HotCloud'15*, 2015.

[80] G. Faraci, A. Lombardo, and G. Schembra. A building block to model an sdn/nfv network. In *2017 IEEE International Conference on Communications (ICC)*, pages 1–7, May, Paris, France 2017.

[81] M. Y. Fathany and T. Adiono. Wireless protocol design for smart home on mesh wireless sensor network. In *ISPACS '15*, 2015.

[82] N. Feamster, D. G. Andersen, H. Balakrishnan, and M. F. Kaashoek. Measuring the effects of internet path faults on reactive routing. *SIGMETRICS Perform. Eval. Rev.*, 31(1):126–137, Jun 2003.

[83] T. Field. Jinqs: An extensible library for simulating multiclass queueing networks, v1. 0 user guide, 2006.

[84] A. Fischer, J. F. Botero, M. T. Beck, H. de Meer, and X. Hesselbach. Virtual network embedding: A survey. *IEEE Communications Surveys Tutorials*, 15(4):1888–1906, Fourth 2013.

[85] Flask web framework. `http://flask.pocoo.org/`, 2010.

[86] C. H. Foh, Y. Zhang, Z. Ni, J. Cai, and K. N. Ngan. Optimized cross-layer design for scalable video transmission over the ieee 802.11 e networks. *IEEE Transactions on Circuits and Systems for Video Technology*, 2007.

[87] I. L. Freire and J. A. Apolinario Jr. Gunshot detection in noisy environments. In *Proceeding of the 7th International Telecommunications Symposium, Manaus, Brazil*, pages 1–4, 2010.

[88] P. Gill, Z. Li, A. Mahanti, J. Luo, and C. Williamson. Network information flow in network of queues. In *MASCOTS 2008*. IEEE, 2008.

[89] D. Gross, J. Shortle, J. Thompson, and C. Harris. *Fundamentals of queueing theory*. John Wiley & Sons, 4th edition, 2008.

[90] A. A. Hagberg, D. A. Schult, and P. J. Swart. Exploring network structure, dynamics, and function using NetworkX. In *SciPy2008*.

[91] A. Hakiri, P. Berthou, A. Gokhale, and S. Abdellatif. Publish/subscribe-enabled software defined networking for efficient and scalable iot communications. *IEEE Communications Magazine*, 53(9):48–54, September 2015.

[92] A. Hakiri and A. Gokhale. Data-centric publish/subscribe routing middleware for realizing proactive overlay software-defined networking. DEBS 2016.

[93] H. Halabian, I. Lambadaris, and C.-H. Lung. Network capacity region of multi-queue multi-server queueing system with time varying connectivities. In *ISIT*. IEEE, 2010.

[94] A. Hamins, C. Grant, N. Bryner, A. Jones, and G. Koepke. *NIST Special Publication 1191 Research Roadmap for Smart Fire Fighting*. National Institute Of Standards and Technology, 2015.

[95] J. Han, D. Watson, and F. Jahanian. Topology aware overlay networks. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, volume 4, pages 2554 – 2565 vol. 4, march 2005.

[96] Q. Han, P. Nguyen, R. T. Eguchi, K. Hsu, and N. Venkatasubramanian. Toward an integrated approach to localizing failures in community water networks. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 1250–1260, June 2017.

[97] A. F. Hansen, A. Kvalbein, T. Čičić, and S. Gjessing. Resilient routing layers for network disaster planning. In *Proceedings of the 4th international conference on Networking - Volume Part II*, ICN'05, pages 1097–1105. Springer-Verlag, 2005.

[98] J. Hartigan and M. Wong. Algorithm as 136: A k-means clustering algorithm. *Royal Statistical Society, Series C*, 28(1):100–108, 1979.

[99] F. He, L. Baresi, C. Ghezzi, and P. Spoletini. Formal analysis of publish-subscribe systems by probabilistic timed automata. In *International Conference on Formal Techniques for Networked and Distributed Systems*, Tallinn, Estonia, June 2007.

[100] J. Heidemann, L. Quan, and Y. Pradkin. *A preliminary analysis of network outages during hurricane sandy.* University of Southern California, Information Sciences Institute, 2012.

[101] B. Heller, R. Sherwood, and N. McKeown. The controller placement problem. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 7–12. ACM, 2012.

[102] B. Hore, H. Jafarpour, R. Jain, S. Ji, D. Massaguer, S. Mehrotra, N. Venkatasubramanian, and U. Westermann. Design and Implementation of a Middleware for Sentient Spaces. In *2007 IEEE Intelligence and Security Informatics*, pages 137–144. IEEE, May 2007.

[103] R. A. Horn and C. R. Johnson. Matrix analysis. *Cambridge, UK: Cambridge University Press*, pages 146 – 147, 1999.

[104] F. Hwang, D. Richards, and P. Winter. *The Steiner tree problem.* North-Holland, 1992.

[105] IBM. *MQTT For Sensor Networks (MQTT-SN)*, Nov. 2013.

[106] U. o. S. C. Information Sciences Institute. *RFC 791 - INTERNET PROTOCOL.*

[107] M. Inoue, Y. Owada, K. Hamaguti, and R. Miura. Nerve net: A regional-area network for resilient local information sharing and communications. In *2014 Second International Symposium on Computing and Networking*, pages 3–6, Dec 2014.

[108] G. Jain, S. Babu, R. Raj, K. Benson, B. Manoj, and N. Venkatasubramanian. On disaster information gathering in a complex shanty town terrain. In *Global Humanitarian Technology Conference-South Asia Satellite (GHTC-SAS)*, pages 147–153. IEEE, 2014.

[109] Y. Jararweh, M. Al-Ayyoub, A. Darabseh, E. Benkhelifa, M. Vouk, and A. Rindos. Sdiot: a software defined based internet of things framework. *Journal of Ambient Intelligence and Humanized Computing*, 6(4):453–461, Aug 2015.

[110] P. Jokela, A. Zahemszky, C. Esteve Rothenberg, S. Arianfar, and P. Nikander. Lipsin: Line speed publish/subscribe inter-networking. *SIGCOMM Comput. Commun. Rev.*, 39(4):195–206, Aug. 2009.

[111] P. Kathiravelu, L. Sharifi, and L. Veiga. Cassowary: Middleware platform for context-aware smart buildings with software-defined sensor networks. In *Proceedings of the 2Nd Workshop on Middleware for Context-Aware Applications in the IoT*, M4IoT 2015, pages 1–6, New York, NY, USA, 2015. ACM.

[112] K. Kim, S. Min, and Y. Han. A programmable data plane to support in-network data processing in software-defined iot. In *2017 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 855–860, Oct 2017.

[113] K. Kim and N. Venkatasubramanian. Assessing the Impact of Geographically Correlated Failures on Overlay-Based Data Dissemination. In *2010 IEEE Global Telecommunications Conference GLOBECOM 2010*, pages 1–5. IEEE, Dec. 2010.

[114] K. Kim, Y. Zhao, and N. Venkatasubramanian. GSFord: Towards a Reliable Geo-social Notification System. In *2012 IEEE 31st Symposium on Reliable Distributed Systems*, pages 267–272. IEEE, Oct. 2012.

[115] S. Kounev, K. Sachs, J. Bacon, and A. Buchmann. A methodology for performance modeling of distributed event-based systems. In *11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*, Orlando, FL, USA, May 2008.

[116] H. Krishna, N. L. M. van Adrichem, and F. A. Kuipers. Providing bandwidth guarantees with openflow. In *2016 Symposium on Communications and Vehicular Technologies (SCVT)*, pages 1–6, Nov 2016.

[117] I. Ku, Y. Lu, and M. Gerla. Software-defined mobile cloud: Architecture, services and use cases. In *IWCMC 2014*.

[118] R. Kumar, M. Hasan, S. Padhy, K. Evchenko, L. Piramanayagam, S. Mohan, and R. B. Bobba. End-to-end network delay guarantees for real-time systems using sdn. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 231–242, Dec 2017.

[119] C. Labovitz, A. Ahuja, A. Bose, and F. Jahanian. Delayed internet routing convergence. *SIGCOMM Comput. Commun. Rev.*, 30(4):175–187, Aug 2000.

[120] E. Lazowska, J. Zahorjan, S. Graham, and K. Sevcik. *Quantitative system performance: computer system analysis using queueing network models.* Prentice-Hall, Inc., 1984.

[121] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann. logically centralized?: State distribution trade-offs in software defined networks.

[122] X. Li, Z. Qin, and T. Yu. Optimizing the qos performance of fast rerouting. In *2009 Ninth International Conference on Hybrid Intelligent Systems*, volume 3, pages 313–318, Aug 2009.

[123] Z. Li, M. Liang, L. O'Brien, and H. Zhang. The cloud's cloudy moment: A systematic survey of public cloud service outage. *CoRR*, abs/1312.6485, 2013.

[124] C. Lin, K. Wang, and G. Deng. A qos-aware routing in sdn hybrid networks. *Procedia Computer Science*, 110:242 – 249, 2017. 14th International Conference on Mobile Systems and Pervasive Computing (MobiSPC 2017) / 12th International Conference on Future Networks and Communications (FNC 2017) / Affiliated Workshops.

[125] Linux TC. `http://lartc.org/manpages/tc.txt`, 2001.

[126] H. Liu, J. Li, Z. Xie, S. Lin, K. Whitehouse, J. A. Stankovic, and D. Siu. Automatic and robust breadcrumb system deployment for indoor firefighter applications. In *Proceedings of the 8th international conference on Mobile systems, applications, and services - MobiSys '10*, page 21, New York, New York, USA, jun 2010. ACM Press.

[127] P. Liu, D. Willis, and S. Banerjee. ParaDrop: Enabling Lightweight Multi-tenancy at the Network's Extreme Edge. In *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 1–13. IEEE, oct 2016.

[128] L. Massoulie, A.-M. Kermarrec, and A. Ganesh. Network awareness and failure resilience in self-organizing overlay networks. In *Reliable Distributed Systems, 2003. Proceedings. 22nd International Symposium on*, pages 47 – 55, oct. 2003.

[129] R. Mayer, B. Koldehofe, and K. Rothermel. Predictable low-latency event detection with parallel complex event processing. *IEEE Internet of Things Journal*, pages 274–286, 2015.

[130] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, Mar. 2008.

[131] A. Medina, A. Lakhina, I. Matta, and J. Byers. BRITE: An Approach to Universal Topology Generation. page 346, Aug. 2001.

[132] S. Mehrotra, A. Kobsa, N. Venkatasubramanian, and S. R. Rajagopalan. Tippers: A privacy cognizant iot environment. In *2016 IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops)*, March 2016.

[133] Moquette broker. `https://github.com/andsel/moquette/`, 2014.

[134] Mqtt-sn transparent gateway. `https://www.eclipse.org/paho/components/mqtt-sn-transparent-gateway/`, 2016.

[135] Mqtt-sn udp client. `https://github.com/jsaak/mqtt-sn-gateway`, 2016.

[136] K. Nakayama, K. E. Benson, V. Avagyan, M. B. Dillencourt, L. F. Bic, and N. Venkatasubramanian. Tie-set based fault tolerance for autonomous recovery of double-link failures. In *2013 IEEE Symposium on Computers and Communications (ISCC)*, pages 000391–000397. IEEE, 2013.

[137] S. Neumayer and E. Modiano. Network reliability with geographically correlated failures. In *INFOCOM, 2010 Proceedings IEEE*, pages 1 –9, march 2010.

[138] H. A. Nguyen, T. V. Nguyen, and D. Choi. How to maximize user satisfaction degree in multi-service ip networks. In *2009 First Asian Conference on Intelligent Information and Database Systems*, pages 471–476, April 2009.

[139] Node-RED. `http://nodered.org/`.

[140] NumPy. `http://www.numpy.org/`, 1995.

[141] OASIS. *MQTT Version 3.1.1*, Oct. 2014.

[142] Object Management Group. *Data Distribution Service*, Mar. 2015.

[143] S. OVS. `https://github.com/saeenali/openvswitch/wiki/Stochastic-Switching-using-Open-vSwitch-in-Mininet`, 2014.

[144] Paho Java Client. `https://www.eclipse.org/paho/clients/java/`, 2008.

[145] P. R. Pereira, A. Casaca, J. J. Rodrigues, V. N. Soares, J. Triay, and C. Cervello-Pastor. From delay-tolerant networks to vehicular delay-tolerant networks. *IEEE Communications Surveys & Tutorials*, 14(4):1166–1182, 2012.

[146] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The design and implementation of open vswitch. In *NSDI*, 2015.

[147] Pivotal, "RabbitMQ". `https://www.rabbitmq.com/`, 2018.

[148] Ponte - Bringing Things to REST Developers. `http://eclipse.org/ponte/`.

[149] S. Qazi and T. Moors. Finding Alternate Paths in the Internet:A Survey of Techniques for End-to-End Path Discovery. page 13, aug 2013.

[150] B. Quoitin, V. V. Schrieck, P. Francois, and O. Bonaventure. Igen: Generation of router-level internet topologies through network design heuristics. *in Proceedings of the 21st International Teletraffic Congress*, Sep. 2009.

[151] R. Raj, S. Babu, K. Benson, G. Jain, B. Manoj, and N. Venkatasubramanian. Efficient path rescheduling of heterogeneous mobile data collectors for dynamic events in shanty town emergency response. In *Global Communications Conference (GLOBECOM)*, pages 1–7. IEEE, 2015.

[152] Y. Ren, L. Liu, L. Cui, Y. Shi, and D. Zheng. Qos evaluation of prioritized data plane service employing queueing model. In *2017 IEEE/ACM 25th International Symposium on Quality of Service (IWQoS)*, pages 1–6, June 2017.

[153] J. P. Rohrer, A. Jabbar, and J. P. G. Sterbenz. Path diversification for future internet end-to-end resilience and survivability. *Telecommunication Systems*, 56(1):49–67, Aug. 2014.

[154] L. A. Rossman. Epanet users manual. *United States Water Supply and Water Resources Division, National Risk Management Research Laboratory*, September 2000.

[155] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware '01, pages 329–350, London, UK, UK, 2001. Springer-Verlag.

[156] Ryu SDN controller. `https://osrg.github.io/ryu/`, 2011.

[157] T. V. P. S, S. S. Prasad, and K. Kataoka. AMPF: application-aware multipath packet forwarding using machine learning and SDN. *CoRR*, abs/1606.05743, 2016.

[158] K. Sachs, S. Kounev, and A. Buchmann. Performance modeling and analysis of message-oriented event-driven systems. *Software & Systems Modeling*, 2013.

[159] P. Saint-Andre. Extensible messaging and presence protocol (xmpp): Core. Technical report, 2011.

[160] P. Salehi, K. Zhang, and H.-A. Jacobsen. Popsub: Improving resource utilization in distributed content-based publish/subscribe systems. In *ACM DEBS*, 2017.

[161] F. Samie, V. Tsoutsouras, S. Xydis, L. Bauer, D. Soudris, and J. Henkel. Distributed qos management for internet of things under resource constraints. In *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES '16, pages 9:1–9:10. ACM, 2016.

[162] H. Sandor, B. Genge, and G. Sebestyen-Pal. Resilience in the Internet of Things: The Software Defined Networking approach. In *ICCP 2015*.

[163] M. Sargent, J. Chu, D. V. Paxson, and M. Allman. Computing TCP's Retransmission Timer. (RFC-6298), June 2011.

[164] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The Case for VM-Based Cloudlets in Mobile Computing. *IEEE Pervasive Computing*, 8(4):14–23, oct 2009.

[165] C. Schlesinger, H. Ballani, T. Karagiannis, and D. Vytiniotis. Quality of service abstractions for software-defined networks. Technical report, March 2015.

[166] A. Schröter, G. Mühl, S. Kounev, H. Parzyjegla, and J. Richling. Stochastic performance analysis and capacity planning of publish/subscribe systems. In *DEBS*, pages 258–269. ACM, 2010.

[167] S. H. Shaheen and M. Yousaf. Security analysis of dtls structure and its application to secure multicast communication. In *IEEE FIT '14*.

[168] D. Shanbhag. On infinite server queues with batch arrivals. *Journal of Applied Probability*, 3(1):274–279, 1966.

[169] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Flowvisor: A network virtualization layer. *OpenFlow Switch Consortium, Tech. Rep*, 1:132, 2009.

[170] D. Singh, B. Ng, Y.-C. Lai, Y.-D. Lin, and W. K. Seah. Modelling software-defined networking: Switch design with finite buffer and priority queueing. In *LCN*. IEEE, 2017.

[171] Smart emergency response system (sers). `http://smartamerica.org/teams/smart-emergency-response-system-sers/`, 2014.

[172] SmartAmerica. `http://smartamerica.org/`.

[173] P. Smith, A. Schaeffer-Filho, D. Hutchison, and A. Mauthe. Management patterns: SDN-enabled network resilience management. In *2014 IEEE Network Operations and Management Symposium (NOMS)*, pages 1–9. IEEE, may 2014.

[174] K. Sood, S. Yu, and Y. Xiang. Performance analysis of software-defined network switch using $m/geo/1$ model. *IEEE Communications Letters*, pages 2522–2525, 2016.

[175] N. Spring, R. Mahajan, D. Wetherall, and T. Anderson. Measuring ISP topologies with Rocketfuel. *Networking, IEEE/ACM Transactions on*, 12(1):2 – 16, Feb. 2004.

[176] J. Sterbenz, E. C andetinkaya, M. Hameed, A. Jabbar, and J. Rohrer. Modelling and analysis of network resilience. In *Communication Systems and Networks (COMSNETS), 2011 Third International Conference on*, pages 1 –10, jan. 2011.

[177] J. P. G. Sterbenz, D. Hutchison, E. K. Çetinkaya, A. Jabbar, J. P. Rohrer, M. Schöller, and P. Smith. Resilience and survivability in communication networks: Strategies, principles, and survey of disciplines. *Comput. Netw.*, 54(8):1245–1265, jun 2010.

[178] Summary of the Amazon S3 Service Disruption in the Northern Virginia (US-EAST-1) Region. `https://aws.amazon.com/message/41926/`.

[179] U. S. W. Supply and N. R. M. R. L. Water Resources Division. Epanet. *[Online] Available: http://www.epa.gov/nrmrl/wswrd/dw/epanet.html*, April 2009.

[180] J. Swetina, G. Lu, P. Jacobs, F. Ennesser, and J. Song. Toward a standardized common m2m service layer platform: Introduction to onem2m. *IEEE Wireless Communications*, 21(3):20–26, June 2014.

[181] N. Taft-Plotkin, B. Bellur, and R. Ogier. Quality-of-service routing using maximally disjoint paths. In *1999 Seventh International Workshop on Quality of Service. IWQoS'99. (Cat. No.98EX354)*, pages 119–128, May 1999.

[182] S. K. Tayyaba and M. A. Shah. Resource allocation in sdn based 5g cellular networks. *Peer-to-Peer Networking and Applications*, 2018.

[183] J. Thalheim, A. Rodrigues, I. E. Akkus, P. Bhatotia, R. Chen, B. Viswanath, L. Jiao, and C. Fetzer. Sieve: actionable insights from monitored metrics in distributed systems. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. ACM, 2017.

[184] S. Tomovic, I. Radusinovic, and N. Prasad. Performance comparison of qos routing algorithms applicable to large-scale sdn networks. In *IEEE EUROCON 2015 - International Conference on Computer as a Tool (EUROCON)*, pages 1–6, Sept 2015.

[185] M. Uddin, S. Mukherjee, H. Chang, and T. V. Lakshman. Sdn-based multi-protocol edge switching for iot service automation. *IEEE Journal on Selected Areas in Communications*, pages 1–1, 2018.

[186] M. Y. S. Uddin, A. Nelson, K. Benson, G. Wang, Q. Zhu, Q. Han, N. Alhassoun, P. Chakravarthi, J. Stamatakis, D. Hoffman, et al. The scale2 multi-network architecture for iot-based resilient communities. In *IEEE SMARTCOMP '16*, pages 1–8. IEEE, 2016.

[187] Update on azure storage service interruption. `https://azure.microsoft.com/en-us/blog/update-on-azure-storage-service-interruption/`.

[188] M. Vernon, J. Zahorjan, and E. D. Lazowska. *A comparison of performance Petri nets and queueing network models*. University of Wisconsin-Madison, Computer Sciences Department, 1986.

[189] Y. Wang, Y. Zhang, and J. Chen. Pursuing differentiated services in a sdn-based iot-oriented pub/sub system. pages 906–909, 06 2017.

[190] T. Weng, A. Nwokafor, and Y. Agarwal. Buildingdepot 2.0: An integrated management system for building analysis and control. In *ACM BuildSys'13*, 2013.

[191] J. Wu, Y. Zhang, Z. M. Mao, and K. G. Shin. Internet routing resilience to failures: analysis and implications. In *Proceedings of the 2007 ACM CoNEXT conference*, CoNEXT '07, pages 25:1–25:12, New York, NY, USA, 2007. ACM.

[192] X. Wu, R. Dunne, Q. Zhang, and W. Shi. Edge computing enabled smart firefighting: Opportunities and challenges. In *Proceedings of the Fifth ACM/IEEE Workshop on Hot Topics in Web Systems and Technologies*, HotWeb '17, pages 11:1–11:6. ACM, 2017.

[193] P. Xiong, H. Hacigumus, and J. F. Naughton. A software-defined networking based approach for performance management of analytical queries on distributed data stores. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 955–966, New York, NY, USA, 2014. ACM.

[194] Y. Xu, V. Mahendran, and S. Radhakrishnan. Towards sdn-based fog computing: Mqtt broker virtualization for effective and reliable delivery. In *2016 8th International Conference on Communication Systems and Networks (COMSNETS)*, pages 1–6, Jan 2016.

[195] Y. Yi and M. Chiang. Stochastic network utility maximisation - a tribute to kelly's paper published in this journal a decade ago. 19:421–442, 06 2008.

[196] H. Yoon, S. Kim, T. Nam, and J. Kim. Dynamic flow steering for iot monitoring data in sdn-coordinated iot-cloud services. In *2017 International Conference on Information Networking (ICOIN)*, pages 625–627, Jan 2017.

[197] B. Zhang, N. Mor, J. Kolb, D. S. Chan, K. Lutz, E. Allman, J. Wawrzynek, E. Lee, and J. Kubiatowicz. The cloud is not enough: Saving iot from the cloud. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, Santa Clara, CA, July 2015. USENIX Association.

[198] K. Zhang and H. Jacobsen. Sdn-like: The next generation of pub/sub. *CoRR*, abs/1308.0056, 2013.

[199] K. Zhang, V. Muthusamy, M. Sadoghi, and H.-A. Jacobsen. Subscription covering for relevance-based filtering in content-based publish/subscribe systems. In *IEEE ICDCS*, 2017.

[200] K. Zhang, M. Sadoghi, V. Muthusamy, and H.-A. Jacobsen. Efficient covering for top-k filtering in content-based publish/subscribe systems. In *ACM/IFIP/USENIX Middleware Conference*, 2017.

[201] Q. Zhu, M. Y. S. Uddin, Z. Qin, and N. Venkatasubramanian. Upload planning for mobile data collection in smart community internet-of-things deployments. In *2016 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 1–8, May 2016.

Appendices

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 (7.6), 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:

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

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

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

$$y_c = \Phi \circ \Psi(r_k) \tag{A.1}$$

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

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

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

Because one or more r_j can be mapped to a y_c, by (A.1) we can estimate the arrival rate λ_{y_c} of events with assigned priority y_c as follows:

$$\lambda_{y_c} = \sum_{\{r_j \in R : y_c = \Phi \circ \Psi(r_j)\}} \lambda_{r_j} \tag{A.2}$$

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

$$\mu_{y_c} = \left[\sum_{\{r_j \in R : y_c = \Phi \circ \Psi(r_j)\}} \frac{\lambda_{r_j}}{\lambda_{y_c}} \frac{1}{\mu_{r_j}} \right]^{-1} \tag{A.3}$$

Similarly, we can estimate arrival and processing rates for any priority y_j. We now rely on (A.2),(A.3), and the analysis in Section 3.4.2 of [89] to estimate the waiting time (delay only in the queue) $\Delta_q^{y_c}$ for a given y_c as follows:

$$\Delta_q^{y_c} = \frac{\sum_{y_j \in Y} \frac{\rho_{y_j}}{\mu_{y_j}}}{(1 - \sigma_{y_c - 1})(1 - \sigma_{y_c})} \tag{A.4}$$

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 (A.4), Little's formula then gives:

$$L_q^{y_c} = \Delta_q^{y_c} \lambda_{y_c} \tag{A.5}$$

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:

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

Let $L_q^{r_k}$ be the average number of events in the queue matching subscription r_k with priority y_c. Using (A.2) and (A.5) this can be estimated by:

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

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

$$L^{r_k} = L_q^{r_k} \frac{\lambda_{r_k}}{\mu_{r_k}} \tag{A.6}$$

Finally, by relying on little's law formula and (A.6), the average response time of events matching a given subscription r_k in the multi-class priority queueing system (Q_{mclpr}) is

given by:

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

B Efficiently Computing Drop Rate Policies

We now detail efficiently computing drop rate policies for the FireDeX middleware by solving Eq. 7.12 for the flat, linear and exponential drop rate policies. Considering Eq. 7.12 and Eq. 7.13 we aim to find:

$$\rho_{Q^{out}_{x_k}} = \sum_{r_j \in R_{x_k}} \frac{\lambda^{thru}_{x_k,r_j}}{\mu^{out}_{x_k,r_j}} = 1 - \tilde{\rho}$$

We can expand the denominator to rewrite the previous equation considering Eq. 7.1 and Eq. 7.2:

$$\rho_{Q^{out}_{x_k}} = \sum_{r_j \in R_{x_k}} \frac{\lambda^{notify}_{b_k,r_j} G_{v_j} \Big(1 - \Omega \circ \Psi\big(r_j\big)\Big)}{w_{x_k,s_i}} = 1 - \tilde{\rho} \tag{B.7}$$

where $\Omega \circ \Psi\big(r_j\big)$ represents the drop rate for the subscription r_j. Eq. B.7 is the starting point for each of the following derivations.

Flat drop rate policy. This policy sets all network flows' drop rates equal. The drop rate for subscription r_j is equal to the drop rate assigned to its network flow f_j. Hence, considering Eq. 7.14 we have each drop rate equal to β:

$$\Omega \circ \Psi\big(r_j\big) = \Omega(f_j) = \beta \tag{B.8}$$

Substituting Eq. B.8 into Eq. B.7 we obtain:

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

The bandwidth w_{x_k,s_i} is constant across every subscription r_j. Therefore we can write:

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

Now, we isolate the constant term β:

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

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

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

$$\beta = 1 - \frac{w_{x_k,s_i}(1-\tilde{\rho})}{\sum_{r_j \in R_{x_k}} \lambda_{b_k,r_j}^{notify} G_{v_j}} \tag{B.9}$$

Hence, we can use Eq. (B.9) to efficiently compute the flat drop rates.

Linear drop rate policy. This policy sets each network flow's drop rate as proportional 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 Eq. 7.15 we have:

$$\Omega \circ \Psi\big(r_j\big) = \Omega(f_j) = \beta\Phi(f_j) \tag{B.10}$$

Substituting Eq. B.10 into Eq. B.7 we obtain:

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

Similarly to the flat drop rate policy, we isolate the constant term β:

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

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

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

$$\beta = \frac{\sum_{r_j \in R_{x_k}} \lambda_{b_k,r_j}^{notify} G_{v_j} - w_{x_k,s_i}(1 - \tilde{\rho})}{\sum_{r_j \in R_{x_k}} \lambda_{b_k,r_j}^{notify} G_{v_j} \Phi(f_j)} \tag{B.11}$$

Hence, we can use Eq. (B.11) to efficiently compute the linear drop rates.

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 Eq. 7.16 we have:

$$\Omega \circ \Psi\big(r_j\big) = \Omega(f_j) = 1 - \beta^{-\Phi(f_j)} \tag{B.12}$$

Substituting Eq. B.12 into Eq. B.7 we obtain:

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

Similarly to the previous cases we isolate the constant term β:

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

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

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

$$\sum_{y \in Y} \beta^{-y} \; (\sum_{r_j \in R_{x_k}, \Phi(f_j)=y} \lambda_{b_k,r_j}^{notify} G_{v_j})$$

$$\sum_{y \in Y} (\frac{1}{\beta})^{y} \; (\sum_{r_j \in R_{x_k}, \Phi(f_j)=y} \lambda_{b_k,r_j}^{notify} G_{v_j})$$

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

$$\sum_{y \in Y} \alpha^{y} \; (\sum_{r_j \in R_{x_k}, \Phi(f_j)=y} \lambda_{b_k,r_j}^{notify} G_{v_j}) \tag{B.13}$$

We can therefore solve the (N-1)-order polynomial given in Eq. (B.13) to efficiently compute the exponential drop rates. We can solve this polynomial using the algorithm described in [103]. It relies on computing the eigenvalues of the companion matrix. The commonly-used NumPy Python library [140] implements this algorithm.