Ride: A Resilient IoT Data Exchange Middleware Leveraging SDN and Edge Cloud Resources

Kyle E. Benson, Guoxi Wang, Nalini Venkatasubramanian
Donald Bren School of Information and Computer Sciences
University of California, Irvine, USA
Email: {kebenson, guoxiw1, nalini}@ics.uci.edu

Young-Jin Kim
Nokia Bell Labs
Murray Hill, NJ, USA

***Abstract*—Internet of Things (IoT) deployments rely on data exchange middleware to manage communications between constrained devices and cloud resources that provide analytics, data storage, and serve user applications. In this paper, we propose the Resilient IoT Data Exchange (Ride) middleware that enables resilient operation of IoT applications despite prevalent network failures and congestion. It leverages programmable Software-Defined Networking (SDN)-enabled infrastructure along with both localized edge and cloud services. The two-phase Ride middleware extends existing publish-subscribe-oriented IoT data exchanges according to application-specified resilience requirements and without IoT device client modifications. The first phase, Ride-C, improves *IoT data collection* by gathering network-awareness via a novel resource-aware adaptive probing mechanism and dynamically redirecting IoT data flows across multiple public and local (edge) cloud data exchange connections. The second phase, Ride-D, uses this information to *disseminate time-critical alerts* via an intelligent network-aware resilient multicast mechanism. Results from our prototype smart campus testbed implementation, Mininet-based emulated experiments, and larger-scale simulations show that Ride enables network awareness for greater cloud connection up-times, timely fail-over to edge services, and more resilient local alert dissemination.**

***Keywords*-IoT; data exchange; resilience; SDN; multicast; seismic; alerting; pub-sub; failure-detection; edge cloud; fail-over;**

I. Motivation

Emerging global efforts in smart cities and communities aim to leverage the promise of the Internet of Things (IoT) to improve economic and living conditions for all. IoT applications seamlessly integrate sensors, actuators, communication, and analytics into our daily lives. Fig. 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. Leveraging *data exchange* platforms and protocols, they *publish* this data for transmission 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 detecting and publishing higher-level events.

Resource-constrained IoT devices and deployments keep system deployment 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 [32], 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 data exchanges and event-processing pipelines to generate actionable information in response to real-world events. However, infrastructure failures or resource limitations 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 [11], [40], [5], [12], [44], [4]. 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. Therefore, we propose a middleware for extending existing IoT data exchanges to more *resiliently collect, process, and disseminate* events of critical interest to humans (e.g. disaster alerts).

Our multiple experiences with real-world IoT deployments and mission-critical applications [9], [19], [31], [46] further motivated to us the need for a more resilient IoT data exchange. In particular, our recent project, the Safe Community Awareness and Alerting Network (SCALE) [19], which we extend with our prototype middleware to use as a test-bed in this paper, demonstrated the use of inexpensive off-the-shelf devices leveraging cloud services to improve safety in personal, home, and community environments. 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. Each participating organization must manage its independently-evolving deployment according to its own application domain, system, and policy requirements. Therefore, such deployments require both cloud and locally-managed *edge computing* solutions that capture and leverage **application and network awareness** to dynamically configure the data exchange in support of mission-critical applications.

Our contribution: In this paper, we advocate for a middleware-based approach to resilient timely data exchange for mission-critical applications without modifications to con-

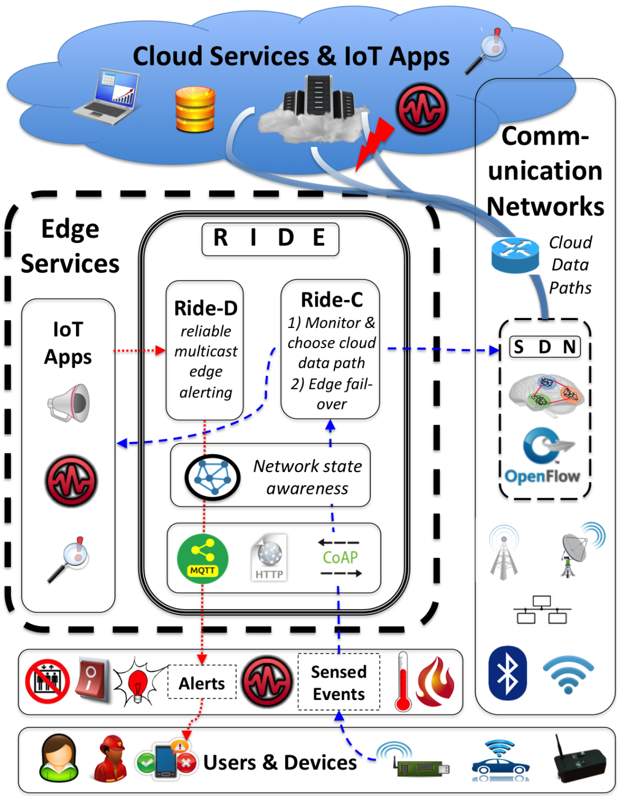


Fig. 1: The Ride middleware leverages edge cloud resources (without IoT device modifications) for network and application-aware resilient data exchange.

strained IoT devices or complete reliance on cloud platforms. We design and develop the **Resilient IoT Data Exchange (Ride)** middleware that gathers network awareness and application resilience requirements to leverage when it dynamically responds 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. Its novel path-selection scheme leverages network state information obtained from Ride-C and the network itself. By adapting information flow 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. Two key aspects of IoT deployments drive our design of Ride: edge computing and SDN.

First, 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.

Second, new networking technologies, such as **Software-Defined Networking (SDN)** [3], enable the accurate collection and maintenance of evolving network conditions in support of dynamically adapting IoT data exchanges without modifying constrained devices. The SDN philosophy exposes a logically-centralized control plane and unified software APIs (e.g. OpenFlow [3]). This enables the fully-automatable merging of network-and-application-awareness, derivation of unique communication requirements, and configuration of the underlying data plane switches. Ride utilizes these SDN APIs to create and maintain *resilient overlays* [15]; 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.

Related work: Techniques to handle end-to-end dependability due to infrastructure failures have been designed at different levels of the system stack. Resilient overlay networks [15] can account for physical geo-location [9] and route around failures/congestion at the application layer. The light-weight IoT protocol the Constrained Application Protocol (CoAP) [27] retransmits datagrams to overcome UDP's lack of transport layer reliability. Some systems exploit multiple access networks for redundancy [16], [31]. At the 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 [16], [43], [26]). Mesh networking solutions [7], [37] avoid the cloud and enable such approaches via direct communications between IoT devices. The authors in [34] propose moving IoT designs beyond the cloud using a distributed replicated append-only log for IoT data called the Global Data Plane. This edge approach also potentially improves QoS [50], [45] and performance over cloud-only architectures.

SDN approaches to resilience include: managing device mobility in distributed controller networks [33], controller placement for reliable operation [24], and leveraging *fast fail-over* to quickly recover from failed links using predefined backup paths [17]. Some techniques also use SDN for network and application awareness [35], including the PLEROMA system [29] that, like Ride-D, also supports multicast-based pub-sub. POSEIDON [39] aims to support different underlying pub-sub protocols, but it requires software agents running on the SDN switches. Some works exploring non-resilience aspects of SDN-based data exchange include real-time QoS guarantees regarding message timing [22] and leveraging a Value of Information metric to configure dissemination policies [30]. IoT systems also use SDN to leverage redundant routes for IoT devices [48] and manage heterogeneous networks [25]

In contrast to many of these works, 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.

II. 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.

A. A Driving Scenario: Smart Campus Disaster Response

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 scenario after the Community Seismic Network (CSN) [20]. CSN uses inexpensive accelerometers driven by small IoT computers (sensor-publishers) to measure ground motion and capture possible seismic sensed events. Consider a smart campus (e.g. university, downtown, industrial research park) instrumented with heterogeneous IoT devices to monitor the environment (e.g. seismic activity, air quality, occupancy assessment) as well as act on it (e.g. alarms, mobile device notifications, evacuation and sheltering services). IoT sensor-publisher devices upload *sensed events* to a cloud data exchange for further analysis, including detecting the location, scope, and severity of an earthquake. In collaboration with CSN, we incorporated CSN sensors on the UCI campus for use in the SCALE platform. This allowed us to study the challenges to an IoT-based seismic early-warning system that issues *alerts* containing situational awareness information to interested subscribers. Affected individuals can then take protective action (e.g.“duck-hold-and-cover”) while first responders assess damage, coordinate efforts, and direct evacuations.

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. Current techniques for disseminating early warnings/notifications/alerts at short notice to the public, i.e. flash dissemination[23], do not leverage IoT-based systems. Furthermore, existing efforts to recover from data exchange/communications failures are typically reactive, which may take several seconds or even minutes [23]. During recent hardware maintenance on our local campus data center we measured similar downtimes of $\approx$45secs.-5mins.

These reactive fail-over mechanisms include recovery of network paths, determination of alternate routes [15], and packet retransmission (e.g. TCP). To improve reliable seismic data accessibility and alerting during the critical initial seconds of an earthquake, we therefore 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.

B. 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.

In designing Ride's architecture (see Fig. 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 the Message Queue Telemetry Transport Protocol (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) that jointly enables a unified resilient framework while separating concerns.

Ride Data Collection (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 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 Cloud Data Path (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.

Ride Data Dissemination (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. 2b). At alert time, Ride-D leverages up-to-date local network awareness embedded in the data exchange workflow by Ride-C 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 [29], [6], 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; §IV 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).

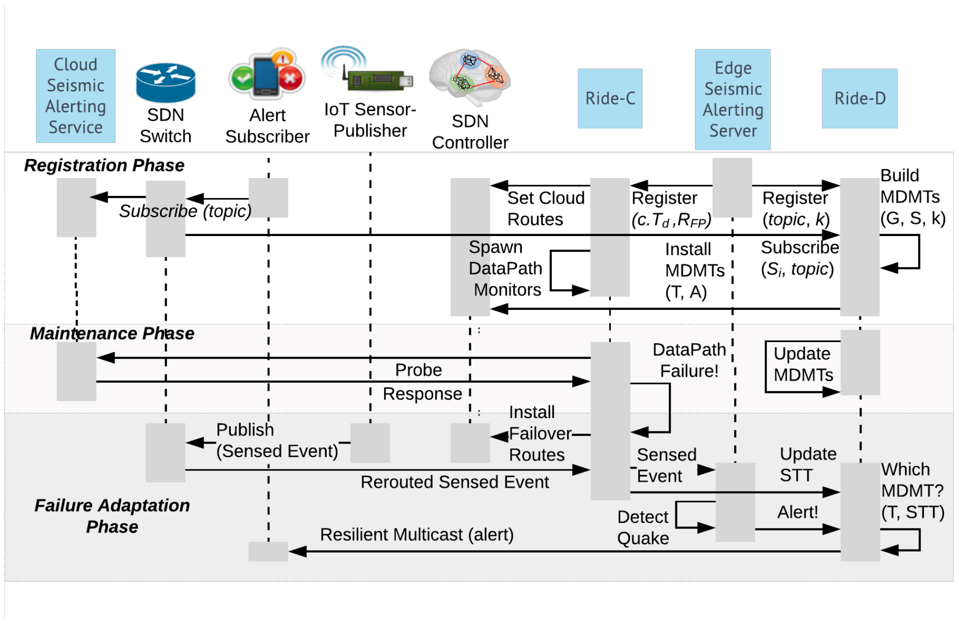
C. Ride Workflow

Ride's workflow executes at the network edge in three phases (see Fig. 2a): 1) a priori host registration and network configuration; 2) on-line network state analysis and maintenance; 3) event-time failure-detection, adaptation, and alerting.

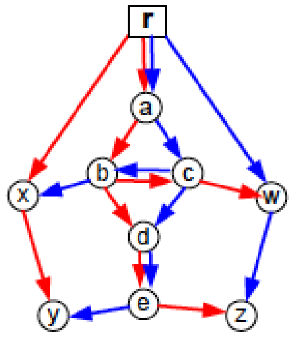
First, Ride **registers and configures** the participating hosts and network components. It exposes an API (e.g. as an SDN controller northbound API) for the Ride-enabled edge service to register: 1) its application resilience requirements and the available CDPs with Ride-C; 2) its time-critical alert 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 §III. 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 §III-A.

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.



(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 [8]'s *red-blue tree* algorithm.

Fig. 2: Ride's resilient IoT data exchange workflow and diverse multicast tree-based alert dissemination.

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. While our current implementation (see §IV) uses OpenFlow's [3] *flow rules* for packet forwarding/address translation and *group tables* for multicast, the Ride paradigm could utilize alternative SDN technologies, addressing schemes other than IPv4, and even incorporate non-SDN switches using tunneling.

III. 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 $y \in V(G)$.

A. Ride-C – Data Collection in Ride

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. This 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. 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. Because it is NP-Complete for $k > 2$ [10], we instead generate our diverse paths using the polynomial-time minimum cost flow-based algorithm proposed in [36]. It reformulates the problem to minimize *shareability*: the sum over all edges of the total number of paths (minus one) using that edge. 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 [36]'s algorithm.

Ride-C then configures the CDP monitoring mechanism (see Alg. 1) 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: $I = \frac{T_D}{\lceil \log_{P_l} R_{FP} \rceil}$.

On-line maintenance of network state awareness: As shown in Alg. 1, 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 [49]. 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 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.

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.

Algorithm 1: 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      $P_l, RTT_a \longleftarrow$ 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         $N_B \leftarrow \lceil \log_{P_l} R_{FP} \rceil, I \leftarrow \frac{T_D}{N_B}, T_o \leftarrow 2 * RTT_a$
12         Wait $I$
```

B. Ride-D – Data Dissemination in Ride

Configuring MDMTs a priori: 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. 2b. Determining even a single minimum-cost multicast tree is NP-Hard and referred to as the *Steiner tree problem* [41]. Hence, we compare several heuristic-based algorithms to pre-construct the k MDMTs based on network state information.

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

• ***steiner*** approximates the Steiner trees using the somewhat naïve method described in [41] that finds the minimum spanning tree of the metric closure subgraph. 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.

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 [10] as Ride-C's diverse path routing policy. It selectively adds each path to one of the k MDMTs with which it has maximal overlap, thereby maintaining lower total cost paths.

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

• ***red-blue*** incorporates the concept of *red-blue trees* shown in Fig. 2b that finds $k = 2$ edge-disjoint directed spanning trees in polynomial time [14], [8]. We adopt the *SkeletonList* data structure and algorithm proposed in [8], which red-blue colors *every* edge. This more efficiently handles topology updates as opposed to faster ($O(|V|+|E|)$ algorithms [14] that must be fully re-computed after topology updates since they color just those in the spanning trees. This coloring partitions G into two maximally-disjoint directed acyclic graphs (DAGs) that we recursively apply the procedure on (for $k > 2$; k a positive power of 2) to greedily further subdivide the graph.

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

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. We instead focus our contributions on intelligent MDMT-selection as described next.

Event-time failure response: Alg. 2 details Ride-D's alerting mechanism. We now describe how it's 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 §V-D. Note that the policies' objective functions break ties randomly.

• ***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: $|\{s \in S : \mathit{PathExists}(STT \cap T_i, r, s)\}|$

• ***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:

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

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 $\mathbf{O(k(|T_i| + |STT|)}$.

IV. 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 [19] IoT middleware to use as the edge alerting service. This complete prototype implements the proposed architecture (Fig. 1) by leveraging RESTful CoAP APIs to manage the workflow describe in §II-C, Fig. 2a. We invite the reader to try out Ride and find more details than we could fit below in our source code repository: https://github.com/KyleBenson/ride.

Our multi-sensing multi-network SCALE devices run an asynchronous Python framework that derives sensed events from abstract feeds of physical sensor readings, detected higher-level events, events received from networked devices, etc. It publishes them internally for storage, use by other

Algorithm 2: Ride-D network-aware multicast alerting algorithms for the configuration and alerting phases.

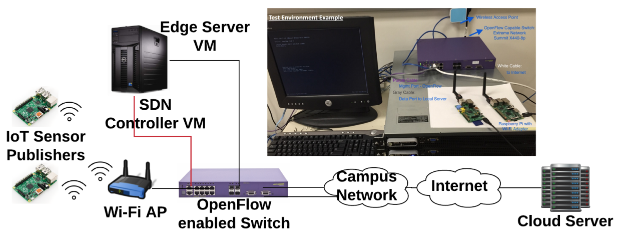
```
1  Function ConfigureMDMTs(S, topic, r, k, G, algorithm)
2      $T \leftarrow \mathit{BuildMDMTs}(\mathit{algorithm}, G, S, r, k)$
3      for $T_i \in T$ do
4          $\mathit{addresses} \leftarrow \mathit{InstallMulticastTreeFlowRules}(T_i)$
5      $\mathit{RegisterMDMTs}(T, \mathit{topic}, \mathit{addresses})$
6  Function SendAlert(msg, topic)
7      $\mathit{Metric} \leftarrow$ MDMT selection policy objective function
8      $S \leftarrow \mathit{GetSubscribers}(\mathit{topic})$
9      for $T_i \in \mathit{GetMDMTs}(\mathit{topic})$ do
10         $M_i \leftarrow \mathit{Metric}(S, \mathit{GetRoot}(T_i), T_i, \mathit{GetSTT}())$
11     $M^*, T^* \leftarrow \max\{(M_i, T_i) : i \in [1..|M|]\}$
12     $\mathit{SendMulticast}(\mathit{MakeAlert}(\mathit{msg}, \mathit{topic}), \mathit{GetAddress}(T^*))$
```

SCALE apps, or forwarding to interested devices. The earliest Ride 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 [28], 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 [42], 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.

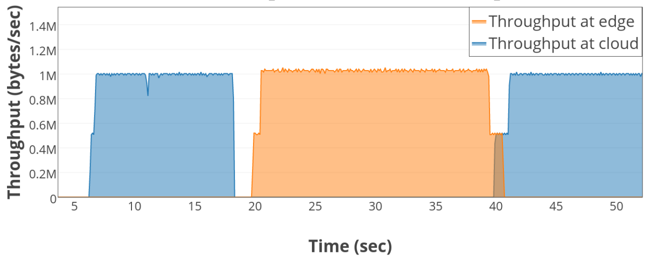
Fig. 3a depicts our lab's **real-world testbed** that we used in our initial proof-of-concept. *Unmodified* SCALE devices publish environmental sensed events to an MQTT [47] broker for visualization via our web-based dashboard or further processing by an analytics service. The SCALE devices associate with a Wi-Fi AP connected to the pictured switch, which routes data to either the edge or cloud broker instances. An ONOS [18] SDN controller connected to the SDN switch's management port controls its forwarding plane routing using the OpenFlow [3] protocol. We simulated a **broken link** by unplugging the Ethernet cable connecting the switch to our campus network. Fig. 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 (§V) 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 predefined 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 [38] 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



(a) Our experimental testbed setup.



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

Fig. 3: A prototype of Ride in our physical lab test-bed.

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 [21]. Ride-C spawns a local threaded client and simple cloud-based UDP echo server to monitor each registered CDP as described in §III-A. In response to congestion or failures, it re-routes IoT traffic through another available CDP 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. 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.

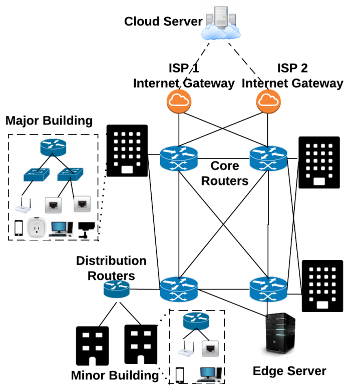
V. Experimental Evaluation

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.

A. 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) [2] 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 run our aforementioned Ride-enabled SCALE seismic clients. OVS switches connect via the SDN southbound protocol OpenFlow [3] to the distributed SDN controller platform ONOS [18] running on the same machine.

To lend a realistic setting to our experiments, we wrote a script to randomly generate a **synthetic campus network topology**, inspired by our university's network, with realistic link characteristics (e.g. bandwidth, latency). Fig. 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. A few buildings (e.g. two for the same department) connect directly together. 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.



(a) Smart campus network structure.

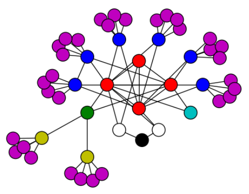

(b) Synthetically generated smart campus topology.

Fig. 4: The network topologies used in our experiments.

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 predetermined times; 5) and records results to determine Ride's performance. As indicated by the event flow captured in Fig. 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.

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.

B. 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. 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. 6a 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.

C. Ride-C Performance & Parameter Space Evaluation

We setup experiments to evaluate Ride-C's failure-detection-and-correction time and overhead (# probe packets) under varying parameters (e.g. the application-specified requirement T_D). Fig. 6b shows how Ride-C always meets the required T_D, which closely matches the observed failure detection time (linear trend). It also shows the trade-off 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. We also compared the two different routing policies (shortest/disjoint) given in §III-A, but found that they perform almost identically as shown in Fig. 6a. 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 [15] and [13]. 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) [15] 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 [13] is an adaptive version of the RON detector. It 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. 6c 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.

D. 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 [38] 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.

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.

Fig. 7a compares the different MDMT-construction algorithms. While the random MDMT choice performs worse than the unicast configuration, 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 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. 6a shows the results for *red-blue*), we found significant reachability improvements 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.

Fig. 7b 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. 7c 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. 7c that the number of subscribers vs. publishers has no effect on *reachability*. Similarly, the campus network topology's size and number of redundant connections (not pictured

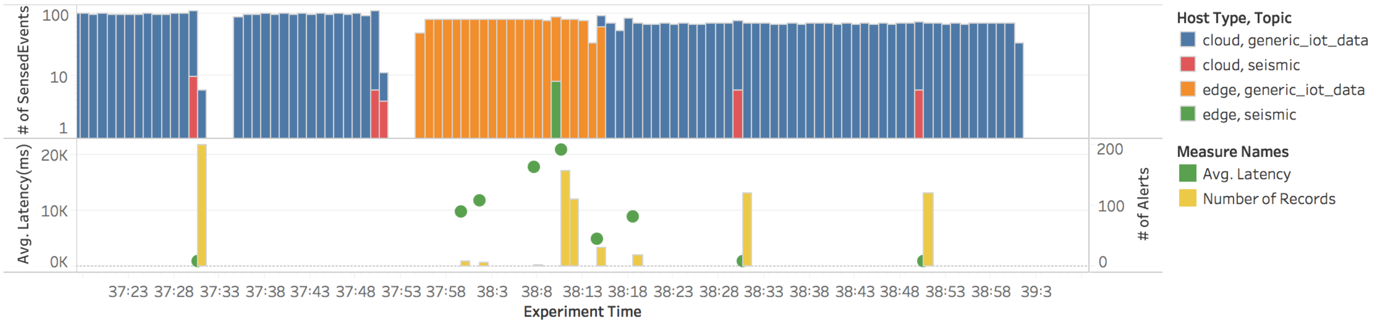
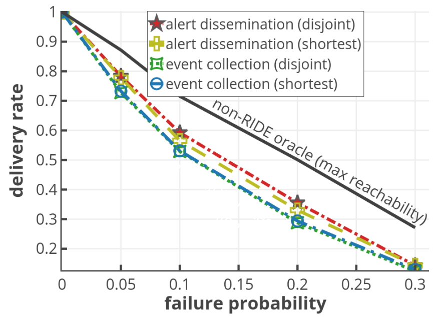
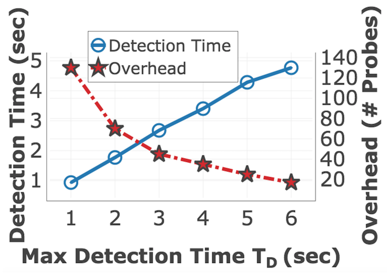


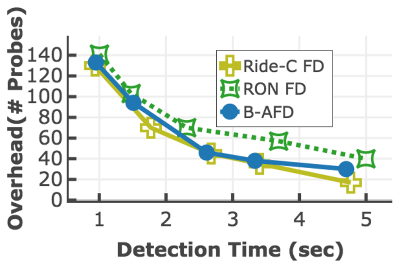
Fig. 5: Ride's failure adaptations during an example execution of our seismic alerting scenario.



(a) Varying network failure probability shows setting collection routes using disjoint method improves alerts' delivery rate

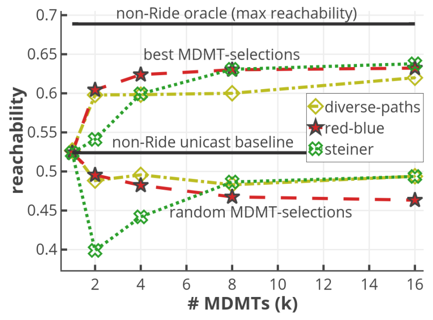


(b) Varying Maximum Detection Time T_D showcases Ride-C's resource-conserving adaptive probing.

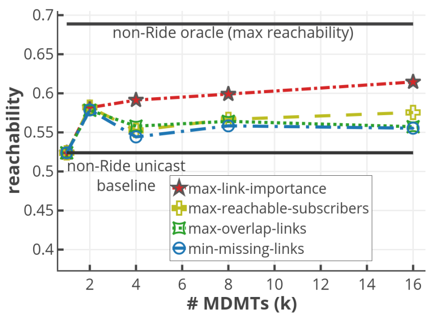


(c) Comparing failure detector (FD) overhead of Ride-C and two related works.

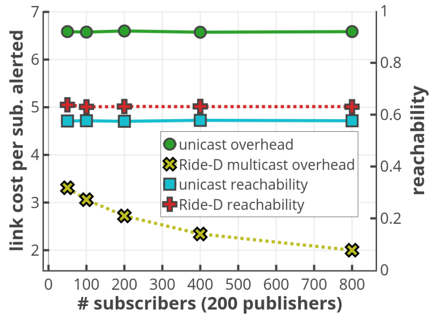
Fig. 6: Ride-C Performance & Parameter Space Evaluation



(a) Comparing MDMT-construction algorithms shows careful MDMT-selection performs better than unicast.



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



(c) Multicast-based dissemination improves overhead vs. unicast while reachability remains unaffected by increased # subscribers.

Fig. 7: Varying k (# MDMTs) and number of subscribers showcases Ride-D's resource-conscious resilient alert dissemination.

to save space) appears to have little effect on reachability.

VI. Concluding Remarks

This paper demonstrated Ride, an SDN-enabled edge service middleware for network and application-aware resilient IoT data exchange. Ride 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 our 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 at the edge deployment. The Ride middleware extends existing IoT data exchanges without modifying them, IoT devices, or 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 our seismic scenario. Furthermore, adaptive probing intervals and multicast actually conserve network resources as shown in §V. Hence, administrators must weigh the benefits of this conservation with the increased deployment complexity.

Moving forward, we plan to expand Ride by: improving data collection using multiple CDPs simultaneously, improving data dissemination via wireless ad-hoc networks and selective unicast, securing data transmission using tunneling (e.g. IPsec) for collection and secure multicast (e.g. DTLS-based multicast [51]) for dissemination, and considering different application scenarios. 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.

Acknowledgments

This work was supported by National Science Foundation award nos. CNS-1143705, CNS-0958520, CNS-1450768, CNS-1528995 and NIST Smart Fire Fighting award 70NANB17H285. The authors also thank the entire SCALE team; UCI's Office of IT for sharing information about its campus network topology; Extreme Networks for providing SDN testbed infrastructure; the CSN team for discussing the seismic scenario and lending us testbed devices.

References

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

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

[3] OpenFlow - Open Networking Foundation. https://www.opennetworking.org/sdn-resources/openflow.

[4] Summary of the Amazon S3 Service Disruption in the Northern Virginia (US-EAST-1) Region.

[5] Update on azure storage service interruption. https://azure.microsoft.com/en-us/blog/update-on-azure-storage-service-interruption/.

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

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

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

[9] K. E. Benson and N. Venkatasubramanian. Improving sensor data delivery during disaster scenarios with resilient overlay networks. In *PerNEM Workshop)*, 2013.

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

[11] 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.

[12] 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.

[13] 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.

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

[15] A. et al. Resilient overlay networks. In *Proceedings of SOSP '01*.

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

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

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

[19] B. et al. Scale: Safe community awareness and alerting leveraging the internet of things. *Communications Magazine, IEEE*, Dec 2015.

[20] C. et al. Community seismic network. *Annals of Geophysics*, 2012.

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

[22] K. et al. The MIDdleware Assurance Substrate: Enabling Strong Real-Time Guarantees in Open Systems with OpenFlow. In *ISORC '14*.

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

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

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

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

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

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

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

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

[31] U. et al. The scale2 multi-network architecture for iot-based resilient communities. In *SMARTCOMP '16*.

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

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

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

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

[36] 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.

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

[38] A. A. Hagberg, D. A. Schult, and P. J. Swart. Exploring network structure, dynamics, and function using NetworkX. In *SciPy2008*.

[39] A. Hakiri and A. Gokhale. Data-centric publish/subscribe routing middleware for realizing proactive overlay software-defined networking. DEBS 2016.

[40] J. Heidemann, L. Quan, and Y. Pradkin. *A preliminary analysis of network outages during hurricane sandy*. University of Southern California, Information Sciences Institute, 2012.

[41] F. Hwang, D. Richards, and P. Winter. *The Steiner tree problem*. North-Holland, 1992.

[42] IBM. *MQTT For Sensor Networks (MQTT-SN)*, 2013.

[43] I. Ku, Y. Lu, and M. Gerla. Software-defined mobile cloud: Architecture, services and use cases. In *IWCMC 2014*.

[44] 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.

[45] 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.

[46] 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.

[47] OASIS. *MQTT Version 3.1.1*, 2014.

[48] H. Sandor, B. Genge, and G. Sebestyen-Pal. Resilience in the Internet of Things: The Software Defined Networking approach. In *ICCP 2015*.

[49] M. Sargent, J. Chu, D. V. Paxson, and M. Allman. Computing TCP's Retransmission Timer. June 2011.

[50] 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.

[51] S. H. Shaheen and M. Yousaf. Security analysis of dtls structure and its application to secure multicast communication. In *IEEE FIT '14*.