Fast Detection and Recovery from Communication Link Failures in a Smart Grid using OpenFlow

Daniel Gyllstrom, Nicholas Braga, and Jim Kurose
School of Computer Science. University of Massachusetts Amherst USA

Abstract—In this paper, we design and evaluate algorithms for fast recovery from link failures in a smart grid communication network, addressing all three aspects of link failure recovery: (a) link failure detection, (b) algorithms for computing backup multicast trees, and (c) fast backup tree installation.

To address (a), we design link-failure detection and reporting mechanisms that use OpenFlow to detect link failures when and where they occur inside the network. OpenFlow is an open source framework that cleanly separates the control and data planes for use in network management and control. For part (b), we formulate a new problem, MULTICAST RECYCLING, that computes backup multicast trees that aim to minimize control plane signaling overhead. We prove MULTICAST RECYCLING is at least NP-hard and present a corresponding approximation algorithm. Lastly, two control plane algorithms are proposed that signal data plane switches to install pre-computed backup trees. An optimized version of each installation algorithm is designed that finds a near minimum set of forwarding rules by sharing forwarding rules across multicast groups, thereby reducing backup tree install time and associated control state. We implement these algorithms in the POX OpenFlow controller and evaluate them using the Mininet emulator, quantifying control plane signaling and installation time.

I. INTRODUCTION

An electric power grid consists of a set of buses – electric substations, power generation centers, or aggregation points of electrical loads – and transmission lines connecting those buses. The operation of the power grid can be greatly improved using high-frequency voltage and current measurements provided by Phasor Measurement Units (PMUs).

PMU applications have stringent, and in many cases ultra-low, per-packet delay and loss requirements. If these per-packet delay requirements are not met, PMU applications can miss a critical power grid event (e.g., lightning strike), potentially leading to a cascade of incorrect decisions and corresponding actions. For example, closed-loop control applications require delays of 8–16 ms per-packet [1]. If any packet is not received within this time window, the closed-loop control application may take a wrong control action. In the worst case, this can lead to a cascade of power grid failures similar to the recent power grid failures in India [2].

As a result of this sensitivity, the communication network that disseminates PMU data must provide hard end-to-end data delivery guarantees [1]. For this reason, the Internet’s best-effort service model alone is unable to meet the stringent packet delay and loss requirements of PMU applications [3]. Instead, either a new network architecture or enhancements to the existing protocols are needed [1], [3], [4] to provide efficient, in-network forwarding and fast recovery from link and switch failures. Additionally, multicast should figure prominently in data delivery, since PMUs disseminate data to applications across many locations [1].

Software-defined networking (SDN) can facilitate this type of innovation by providing programmable access to the forwarding plane of network switches and routers. New network services are defined in a programmable control plane, which SDN cleanly separates from the data plane (e.g., forwarding), and are instantiated as forwarding rules installed at network switches. The communication between the control and data planes, including the messaging to install forwarding rules, are typically managed by the OpenFlow protocol [5].

In this paper, we use OpenFlow to define and implement new control plane algorithms, tailored specifically for disseminating critical power grid data, that program data plane forwarding by installing forwarding rules at network switches. We focus on algorithms for fast recovery from link failures. Informally, a link that does not meet its packet delivery requirement is considered failed. We propose, design, and evaluate solutions to all three aspects of link failure recovery: link failure detection, algorithms for pre-computing backup multicast trees, and fast backup tree installation.

We make the following contributions in this paper:

- Design a link-failure detection and reporting algorithm, PCOUNT, that uses OpenFlow [5] to detect link failures when and where they occur, inside the network. In-network detection is used to reduce the time between when the loss occurs and when it is detected. In contrast, most previous work [6], [7] focuses on measuring end-to-end packet loss, resulting in slower detection times.
- Formulate a novel optimization problem for computing backup multicast trees. We formulate a new problem, MULTICAST RECYCLING, that pre-computes backup multicast trees, to be used after a link failure, with the aim of minimizing the control overhead required to install the backup trees. This optimization criteria differs from those proposed in the literature [8], [9], [10], [11] that typically emphasize maximizing node (link) disjointedness between the backup and primary path.
- Prove MULTICAST RECYCLING is at least NP-hard and propose an approximation algorithm, BUNCHY.
- Propose MERGER, an OpenFlow implementation of multicast that aims to reduce forwarding state. MERGER uses local optimization to create a near minimal set of forwarding rules by “merging” forwarding rules where multiple multicast trees have common children.
PMU Application | E2E Delay
---|---
System Integrity Protection Scheme (SIPS) | 8 – 16 ms
Wide Area Control, Anti-Islanding | 5 – 50 ms

**TABLE I**

PMU APPLICATIONS AND THEIR QoS REQUIREMENTS [1].

- Design two algorithms – **PROACTIVE** and **REACTIVE** – for fast backup tree installation. **PROACTIVE** preinstalls backup tree forwarding rules and activates these rules after a link failure is detected, while, **REACTIVE** installs backup trees after a link is failed is detected.
- Provide a prototype implementation of our algorithms, **APPLESEED**, using POX and evaluate each algorithm using Mininet. **PCOUNT**, **BUNCHY**, **MERGER**, **PROACTIVE**, and **REACTIVE** are implemented in POX [12], an open-source OpenFlow controller, and are each evaluated using Mininet emulations [13].

The remainder of this paper is structured as follows. In Section II, we specify PMU application requirements and give an overview of OpenFlow. Then, we outline our algorithms in Section III. Section IV briefly surveys related work and our emulation study is presented in Section V. Section VI concludes this paper with a short summary.

II. PRELIMINARIES

In this section, we describe QoS requirements of three PMU applications (Section II-A), introduce notation and terms (Section II-B), and briefly overview OpenFlow (Section II-C).

A. PMU Applications and Their QoS Requirements

We consider the design of a communication network for disseminating critical Smart Grid data, principally data associated with PMU applications with the most challenging QoS requirements. Table I shows end-to-end per packet delay requirements of three PMU applications that we aim to support. Reference [14] gives a detailed description of these PMU applications.

B. Notation and Assumptions

We model the communication network as a directed graph $G = (V, E)$, where $(u, d) \in E$ denotes a directed edge from $u$ to $d$ and $V$ consists of three types of nodes: ones that send PMU data (PMU nodes), nodes that receive PMU data (data sinks), and switches connecting PMU nodes and data sinks. We assume $G$ has $m \geq 1$ source-based multicast trees to disseminate PMU data. Let $T = \{T_1, T_2, \ldots, T_m\}$ refer to the set of $m$ source-based multicast trees in $G$ such that each $T_i = (V_i, E_i, r, S)$ is a tree rooted at $r$ with directed edges $E_i$, vertices $V_i$, and a directed path from $r$ to each $s \in S$. Let $w(T_i)$ be the total weight of all $E_i$ edges.

For convenience, denote $T_i^l = (V_i^l, E_i^l, r, S)$ as the $i$th directed tree with $l \in E_i^l$. For each link $l$ in each directed tree, $T_i^l$, we define a backup tree, $\hat{T}_i^l$, as follows. $\hat{T}_i^l = (V_i^l, E_i^l, r, S)$ is a directed tree with root $r$ and has a directed path from $r$ to each $s \in S$ such that $l \notin E_i^l$. We refer to $T_i^l$ and $\hat{T}_i^l$ as a primary tree and backup tree, respectively.

C. OpenFlow

OpenFlow is an open standard that cleanly separates the control and data planes, and provides a programmable (and possibly centralized) control framework [5]. All OpenFlow algorithms and protocols are managed by a (logically) centralized controller, while network switches (as their only task) forward packets according to the local forwarding rules installed by the controller at that switch.

OpenFlow provides access to its switch flow tables, where each switch processes packets using a “match plus action” paradigm [5]: flow table table entries match incoming packets based on packet header fields and actions (e.g., forward, drop) are then applied to the packet. Switches maintains statistics for each flow table entry (e.g., packet counter) that can be queried by the controller. We use the terms “flow table entry” and “forwarding rule” interchangeably.

OpenFlow does not explicitly provide an implementation for multicast so we design our own multicast implementation, called **BASIC**. **BASIC** assigns a multicast IP address to each multicast group and uses this address to setup the flow tables at the multicast tree switches. For each multicast tree $T_i = (V_i, E_i, r, S)$, **BASIC** installs a flow table entry at each switch in $V_i$. The flow table entry matches packets using the group’s multicast address (all other field are left as wildcards) and forwards a copy of each packet out the ports corresponding to the switch’s outgoing links in $E_i$.

III. ALGORITHMS

We propose a set of algorithms, collectively referred to as **APPLESEED**, that make multicast trees robust to link failures. **APPLESEED** runs at the OpenFlow controller with the goal of minimizing packet loss associated with link failures while ensuring that end-to-end delay requirements are satisfied. **APPLESEED** divides into three parts: fast link-failure detection using **PCOUNT** (Section III-A), computing backup trees that are amenable to fast installation (Section III-B), and fast installation of pre-computed backup trees (Section III-C).

A. Link Failure Detection Using OpenFlow

We present **PCOUNT**, an algorithm that uses OpenFlow to detect link failures inside the network. In-network detection is used to reduce the time between when packet loss occurs and when it is detected. Fast packet loss detection is crucial to the critical PMU applications that we target in this work, as they are particularly sensitive to packet loss. Most previous work [6], [7] focuses on using end-to-end measurements to estimate packet loss, resulting in slower detection times.

**PCOUNT** considers a link as failed when the rate of packet loss exceeds a threshold. Our definition of a “link” here is

1The name **APPLESEED** is inspired by Johnny Appleseed, the famous conservationist known for planting apple nurseries and caring for its trees.
broad: we consider a link as a path where the two endpoints are OpenFlow switches. In [14], we detail how PCOUNT easily generalizes to detect packet loss between multiple switches.

For a sampling window of length \( w \), PCOUNT estimates packet loss along a link, \((u, d)\), by measuring the aggregate loss rate experienced by flows \( M = \{f_1, f_2, \ldots, f_k\} \) along \((u, d)\) using the following steps:

1. **Install rules (downstream) to count all tagged \( f_i \) packets received at \( d \).** A new flow table entry for each \( f_i \in M \) is installed at \( d \) that matches packets using the identifier (i.e., the tag) applied at \( u \) in step (2).

2. **Tag (upstream) all packets from each \( f_i \in M \) at \( u \).** Suppose \( u \) uses flow table entry \( e_i \) to match and forward flow \( f_i \) packets. First, PCOUNT generates a unique identifier (tag). Then, for each \( f_i \), PCOUNT creates a new flow table entry, \( e'_i \), that is an exact copy of \( e_i \) except that \( e'_i \) embeds the tag in the packet’s dl_vlan field. \( e'_i \) is installed with a higher OpenFlow priority than \( e_i \), ensuring that all flow \( f_i \) packets are tagged.

3. **After \( w \) time units, stop tagging at \( u \) and query \( u \) and \( d \) packet counts.** Each tagging rule is queried separately at \( u \) and a single aggregate query is issued at \( d \).


### B. Computing Backup Trees

APPLESEED pre-computes backup trees to install after PCOUNT detects a link failure. Here we present a new problem, **MULTICAST RECYCLING**, for computing backup trees and a corresponding approximation algorithm, **BUNCHY**.

1. **Multicast Recycling Problem:** The goal of the **MULTICAST RECYCLING** problem is to compute backup trees that maximize reuse of primary tree edges. Recycling primary tree edges allows the SDN controller to use primary tree rules to handle new incoming traffic (i.e., the tag) applied at \( u \) in step (2). This speeds recovery in cases where backup trees are installed after a link failure is detected and reduces the number of flow table entries installed at switches (control state).

For primary tree \( T^l_1 = (V^l_1, E^l_1, r, S) \) and its backup \( T^l_i = (V^l_i, E^l_i, r, S) \), we define a binary variable \( c^l_v \) for all \( v \in V^l_i \). If \( v \) has exactly the same predecessors in \( T^l_1 \) and \( T^l_i \), then \( c^l_v \) takes value 0. Otherwise, \( c^l_v = 1 \). For the \( T^l_1, T^l_i \) pair define:

\[
C^l_i = \sum_{v \in V^l_i} c^l_v \tag{1}
\]

For our purposes, \( C^l_i \) is the number of new rules (i.e., non-recycled primary tree rules) needed to install \( T^l_i \).

Our **MULTICAST RECYCLING** problem definition below uses a modified version of the Steiner tree problem, called **STEINER-ARBORESCENCE** [15]. For root \( r \) and a set of terminals \( S \), **STEINER-ARBORESCENCE** computes a minimum weight directed tree, rooted at \( r \) and spanning \( S \). We denote \( SA_i(G) = (V, E, r, S) \) as the Steiner arborescence computed over \( G \), rooted at \( r \), and spanning \( S \) such that \( r, S \in T_i \).

The input to **MULTICAST RECYCLING** is \((G, T^l, l, \alpha)\) where \( G = (V, E) \) is a directed graph, \( T^l = \{T^l_1, T^l_2, \ldots, T^l_k\} \) where each \( T^l_i \in T^l \) is a primary tree that uses \( l \), \( l \in E \), and \( \alpha \geq 1 \). The output is a backup tree for each primary tree using \( l \). This set of backup trees, \( T^l = \{T^l_1, T^l_2, \ldots, T^l_k\} \):

\[
\text{minimize } \sum_{1 \leq i \leq k} C^l_i \tag{2}
\]

subject to \( w(T^l_i) \leq \alpha \cdot w(SA_i(G')) \), \( \forall T^l_i \in T^l \)

where \( G' = (V', E') \) such that \( E' = E - \{l\} \) and \( w(T^l_i) \) is the sum of \( T^l_i \)'s link weights. \( \alpha \) The objective function maximizes the reuse of primary tree edges, while \( \alpha \) bounds how large the backup tree can grow as consequence of minimizing \( C^l_i \).

**Theorem 3.1:** **MULTICAST RECYCLING** is at least NP-hard.

**Proof:** See reference [14].

2. **Bunchy Approximation Algorithm:** **BUNCHY** is a simple approximation algorithm for **MULTICAST RECYCLING** that modifies link weights to encourage each backup tree to reuse primary tree edges. For each link \( l \), **APPLESEED** uses **BATCH** to compute a backup tree for each primary tree using \( l \).

Given \((G, T^l, l, \alpha)\), for each \( T^l_i \in T^l \)** **BUNCHY** uses the following two-step procedure to compute \( T^l_i \). First, make a copy of \( G \) called \( G' = (V', E') \) and remove \( l \) from \( E' \). Set link weights of each \( e \in T^l_i \) to 0 and each \( e \notin T^l_i \) to 1. Second, run the **STEINER-ARBORESCENCE** approximation from [15] over \( G' \) and use \( T^l_i \) as the result. If \( T^l_i \) satisfies the Equation 2, return \( T^l_i \) as the solution. Otherwise, return False.

Setting the primary tree link weights to 0 in step (1) allows the **STEINER-ARBORESCENCE** approximation algorithm to use any primary tree edge without cost, encouraging the reuse of primary tree edges. If **BUNCHY** returns False in step (2) either \( \alpha \) must be made larger or a new multicast tree should be computed from scratch that satisfies the tree-size constraint.

### C. Installing Backup Trees

We are now ready to describe the last part of **APPLESEED**, installing backup trees. This is a two-step process. First, we compute the switches requiring new forwarding rules and corresponding OpenFlow flow table entries are generated. Second, the controller signals the necessary switches to install the generated forwarding rules. Here we introduce two such installation algorithms, **PROACTIVE** and **REACTIVE**. Both algorithms compute forwarding rules for a single backup tree at-a-time and so our description of each algorithm (with some abuse of notation) refers to a generic primary tree, \( T^l = (V^l, E^l, r, S) \), and its backup tree for \( l \in E^l \). \( T^l_i = (V^l_i, E^l_i, r, S) \).

**REACTIVE Algorithm.** **REACTIVE** first determines which nodes require a new forwarding rule. In cases where \( T^l \) and \( T^l_i \) use exactly the same outgoing links at \( u \), no new forwarding rule (for \( T^l_i \)) at \( u \) needs to be installed because \( T^l \) can reuse \( T^l_i \)'s forwarding rule already installed at \( u \). Forwarding rules are only required at any \( v \in V^l \setminus V^l_i \) and

\[^3\]We assume \( T^l_i \), satisfies all per-packet delay and loss requirements if \( l \notin T^l_i \) and \( w(T^l_i) \leq \alpha \cdot w(SA_i(G')) \)
at each \( v \in V^i \cap \hat{V}^i \) with different outgoing links in \( \hat{T}^i \) and \( T^i \). At each of these nodes, \textsc{Proactive} pre-computes a flow table entry that matches packets using \( T^i \)'s multicast address. Lastly, when \( l \) fails, \textsc{Proactive} signals the switches to install the pre-computed forwarding rules.

**Proactive Algorithm.** \textsc{Proactive} computes and installs backup tree flow table entries before a primary tree link, \( l \), fails. Recovery is fast with \textsc{Proactive} because, after \( l \) fails, \textsc{Proactive} only needs to signal a single node, the backup tree root, to activate each backup tree.

\textsc{Proactive} cannot, without modifications, pre-install flow table entries at all nodes because incorrect forwarding would result. Doing so at a node, \( d \), where the backup and primary tree have different outgoing links, would either result in packets erroneously forwarded at \( d \) using the backup tree before a link failure occurs or incorrectly forwarding packets using the primary tree after the link failure.

To circumvent this issue, \textsc{Proactive} assigns a unique backup tree id, denoted \( \text{bid} \), to each backup tree. Each backup tree flow table entry matches and forwards packets using the \( \text{bid} \) value written in the \( \text{dl}_\text{src} \) field. When the backup tree \( \hat{T}^i \) is activated, \( \hat{T}^i \)'s root writes the \( \text{bid} \) in the \( \text{dl}_\text{src} \) packet header field, indicating that these packets should be disseminated by \( \hat{T}^i \) rather than \( T^i \). See [14] for more details.

**D. Optimized Multicast Implementation**

Recall that the \textsc{Basic} multicast implementation (Section II-C), creates a flow table entry at each node of a multicast tree that matches incoming packets using the tree’s multicast address. As a result, a switch may have multiple flow table entries executing the same forwarding actions. As an optimization to \textsc{Basic}, we propose \textsc{Merger}, an algorithm that computes a near-minimum set of OpenFlow forwarding rules by consolidating flow table entries at each node where multiple trees use the same set of out-links. \textsc{Merger} reduces the control state (i.e., number of forwarding rules) necessary to multicast packets and, when applied to installing backup trees, can yield faster recovery since fewer control messages are needed to activate backup trees. Due to space constraints, we refer the interested reader to [14] for more details.

**IV. RELATED WORK**

Gridstat [1], a publish-subscribe system, was one of the first research projects to consider smart grid data dissemination. Although Gridstat has similarities with \textsc{Appleseed}, Gridstat has a different focus than ours. Gridstat is an overlay networks built over existing network protocols (e.g., IP, MPLS), while the emphasis of our work is in making network protocols more robust to handle PMU application requirements.

Most previous work for detecting packet loss [6], [7] is based on end-to-end measurements. These approaches require too many measurements (and therefore time) to accurately measure packet loss in our problem setting. \textsc{Pcount} provides faster and more accurate loss estimates of individual links by directly measuring actual traffic inside the network.

Our approach for pre-computing backup trees is partly inspired by MPLS fast-reroute algorithms for rerouting time critical unicast IP flows [16] but differs because we consider multicast. Prior work on computing backup trees and paths use local optimization criteria (i.e., constraints specified over a single multicast tree), while we consider global (network-wide) criteria (i.e., constraints specified across multiple multicast trees). In addition, no prior work seeks to minimize control plane signaling, as \textsc{Multicast Recycling} does. Instead, backup paths or trees are computed that maximize node (link) disjointedness with the primary path [8], [9], [17], [10], minimize backup bandwidth reservations [18], [19], or minimize path length [20].

**V. EVALUATION**

We implement each algorithm from Section III in the POX OpenFlow controller [12] and run emulations using Mininet 2.0.0 [13]. Emulations run on a Linux machine with four 2.33GHz Intel(R) Xeon(R) CPUs and 15GB of RAM. Mininet is configured to run inside Oracle’s VirtualBox virtual machine and is allocated 4GB RAM and a single CPU. We use Mininet’s default software switch, Open vSwitch, and \textsc{Appleseed} runs inside the VirtualBox VM.

**A. Link Failure Detection Emulations**

We run two sets of emulations to evaluate \textsc{PCOUNT}. First, we measure the accuracy of \textsc{PCOUNT} loss probability estimates and, then, quantify how controller and switch processing time increases as \textsc{PCOUNT} monitors more flows.

**Accuracy of Loss Probability Estimates.** For the dumbbell topology shown in Figure 1, \textsc{PCOUNT} measures packet loss over link \((u,d)\). We generate \( m \) multicast groups where each \( h_1, h_2, ..., h_m \) multicasts packets to terminal nodes \( s_1, s_2, ..., s_m \) at a constant rate of 60 packets/second. \textsc{Basic} is used to implement multicast, resulting in \( m \) separate flow table entries at \( u \) and \( d \). We let \( m = \{10, 20, 30, 40, 50\} \) and, using Mininet, drop packet traversing \((u,d)\) following a Bernoulli process with loss probability \( p = \{.01, .05, .10\} \).

We quantify the accuracy of \textsc{PCOUNT} loss estimates, measured relative to the true underlying loss rate, as we modify the number of flows \textsc{PCOUNT} monitors. Recall that \textsc{PCOUNT} accounts for every dropped packet of a flow that it monitors, meaning that the only error in \textsc{PCOUNT} estimates results from unmonitored flows. Here we describe the results from a single representative case, where \( m = 10 \) and \( p = .05 \).

Figure 2(a) compares the 95% confidence intervals of \textsc{PCOUNT}'s link loss probability estimates as a function of...


Fig. 2. Section V emulation results.

B. Backup Tree Installation Emulations

In this section, we emulate the failure of a single link and then measure recovery time, switch storage overhead, control plane signaling, and garbage collection overhead for PROACTIVE and REACTIVE both with and without the MERGER optimization. Due to space constraints, we only present control plane signaling results and briefly summarize additional results. Citation [14] details our complete set of results.

Setup. We use IEEE bus systems 14, 30, 57, and 118 4 and synthetic graphs based on these IEEE bus systems. Synthetic graphs are generated using a procedure described in [14] that uses an IEEE bus system as a template to generate graphs with the same degree distribution as the template bus system.

We assume that the communication network mirrors the physical bus system topology, that an OpenFlow switch is colocated at each bus, and that two unidirectional communication links, one in each direction, connects these switches following the same configuration as the bus system’s transmission lines. In this setup, the PMUs measure voltage and current phasors at the buses, then these measurements are sent by the bus’s attached host to its first-hop switch, which then multicasts the PMU measurements using the network of OpenFlow switches.

For each bus system, we generate synthetic topologies with \( n \) switches, \( n \) hosts, and set all link weights to 1. Then, we randomly create \( m = \{1, 2, ..., \frac{n}{3}\} \) multicast groups, each with \( n/3 \) random terminal hosts, and use the STEINER-ARBORESCENCE approximation from [15] to compute \( m \) primary trees. BUNCHY, with \( \alpha = 1.1 \), is then used to precompute, for each primary tree, a backup tree for each primary tree link. Next, we fail a random communication link, \( l \), triggering either REACTIVE or PROACTIVE to install backup trees. For each \( m \), we generate 35 different synthetic graphs and 3 random sets of multicast groups.

We compare the number of control messages required to install backup trees as function of the number of primary trees \((m)\) installed in the network. Figure 2(c) shows the results for REACTIVE running in BASIC and MERGER mode, a lower

\[ w = \{0.5, 1, ..., 5\} \text{ seconds. Results are shown where PCOUNT monitors } k = \{10\%, 40\%, 70\%, 100\%\} \text{ of } (u, d) \text{ flows (each monitored flow is selected randomly). The confidence intervals for each } w, k \text{ pair are computed over 100 emulation runs.} \]

**Processing Time.** Next, we quantify how PCOUNT processing time increases as PCOUNT monitors more flows. Again, we measure packet loss over \((u, d)\) from Figure 1 but fix the window size to 2 seconds. Processing time is measured as the time between when PCOUNT sends its first statistic query and when PCOUNT computes its packet loss estimate. Recall that if PCOUNT monitors aggregate packet loss of \( k \) flows over \((u, d)\), then PCOUNT sends \( k \) statistic queries to \( u \) and one aggregate query to \( d \). Because Mininet multiplexes CPU resources using the default Linux scheduler, we ensure that the switches are idle during emulations, except for processing PCOUNT requests.

Figure 2(b) shows the total processing time as a function of the number of flows PCOUNT monitors, \( k \), where each data point is the mean computed over 50 emulation runs. To measure the effect of flow table size on query processing time, we install \( r \) additional flow table entries at \( u \) and \( d \). The processing time — where we found 99% of the total processing time is spent processing the \( k + 1 \) statistic queries at the switches — increases roughly quadratically with \( k \) and there is a significant gap in processing time between each \( r \). Our results show that to reasonably achieve sub-second processing time, fewer than 75 flows must be monitored.

Because the switches are idle during emulations and software switches have considerably more powerful CPUs than hardware switches [21], our results likely underestimate processing time. Also note that the (Bernoulli) loss process favors PCOUNT because loss rates are uniform across all flows and loss events are i.i.d. Nonetheless, these results underscore the high cost in monitoring and querying a large number of flows.

\[ \text{http://www.ee.washington.edu/research/pirstca/} \]
bound for \textsc{Reactive} (\textsc{Reactive+LB}), and \textsc{Proactive} in \textsc{Basic} mode. The results are those from synthetic topologies based on IEEEBus system 57. The trends are consistent across all other networks.

\textsc{Reactive+LB} computes the lower bound on the number of new rules required to install backup trees \( T_l \) after \( l \) fails. Informally, \textsc{Reactive+LB} finds the number of unique sets of outgoing links used by \( T_l \) at each node, \( v \), since at least this many rules are required to forward \( T_l \) packets at \( v \). These values are summed across all nodes used by \( T_l \) and is returned as the lower bound. A more detailed description of \textsc{Reactive+LB} can be found in [14].

\textbf{Signaling Overhead Results.} As expected, we find that \textsc{Proactive} requires less signaling overhead than \textsc{Reactive}, including even \textsc{Reactive+LB}. \textsc{Proactive} activates the backup trees by sending a single control message to the root of each of backup tree using the failed link, whereas \textsc{Reactive} activates backup trees by sending a single control message to the root of each backup tree. The remaining \( 25\% \) of savings are due to merging rules with other backup trees. On average, \textsc{Reactive} requires only \( 25\% \) more control messages than LB, suggesting that \textsc{Merger}’s local optimization does not miss many opportunities for consolidating flows.

\textbf{Additional Emulation Results} Using the same setup, we quantified the following as a function of \( m \): the number of pre-installed forwarding rules required by \textsc{Proactive} (the control state), the time to install backup trees, and garbage collection overhead. We find that these three emulations follow the same trends observed with signaling overhead and a full description of each can be found in [14].

\section{VI. Conclusions}

In this paper we have addressed an important challenge in reliable multicasting of critical Smart Grid data. We designed, implemented, and evaluated a suite of algorithms that collectively provide fast packet loss detection and fast rerouting using pre-computed backup multicast trees. Because this required making changes to network switches, we used OpenFlow to modify switch forwarding tables.

First, we presented \textsc{PCount}, an algorithm that used OpenFlow primitives to accurately detect per-link packet loss inside the network. Next, we formulated a new problem, \textsc{Multicast Recycling}, that computes backup trees that re-use edges of already-installed multicast trees to reduce control plane signaling. \textsc{Multicast Recycling} was proved to be at least NP-hard, so we designed an approximation algorithm. Then, we presented \textsc{Proactive} and \textsc{Reactive}, algorithms that installed backup trees at OpenFlow-controlled switches. As an optimization to \textsc{Proactive} and \textsc{Reactive}, we introduced \textsc{Merger}, an algorithm that consolidated forwarding rules at switches where multiple trees had common children.

Mininet emulations showed that \textsc{PCount} estimates were accurate when monitoring even a small number of flows over short time window. By pre-installing backup trees, \textsc{Proactive} resulted in up to a ten-fold decrease in control messages compared with \textsc{Reactive}, which had to signal multiple switches to install each backup tree. Lastly, we found that \textsc{Merger} reduced control plane signaling by \( 2 \)-\( 2.5 \) when applied to \textsc{Reactive}.

\textbf{References}