CHAPTER 1

RECOVERY FROM LINK FAILURES IN A SMART GRID COMMUNICATION NETWORK

1.1 Introduction

In this chapter we continue our study of issues related to PMU sensors. In the previous chapter, we proposed and evaluated algorithms to ensure that PMU measurements are observed in the first place and are correct. Now we consider algorithms that take these (correct) PMU measurements and disseminate them quickly and reliably to power grid operators, utility companies, and power grid managing and monitoring entities.

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, power link failure), 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 [3]. 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 August 2003 blackout in North America [2] and the recent power grid failures in India [36].

As a result of this sensitivity, the communication network that disseminates PMU data must provide hard end-to-end data delivery guarantees [3]. 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 [5]. Instead, either a new network architecture or enhancements to the existing Internet architecture and its protocols are needed [3, 5, 6, 17] 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 [3].

Software-defined networking (SDN) provides a vehicle for 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 [27]. [1]

This separation of control and data planes is similar in spirit to the approach used by the Gridstat system [3] that also manages data plane actions through a separate control plane. Gridstat is a publish-subscribe system designed specifically for disseminating critical power grid data; however, because Gridstat is an overlay service built on top of existing network protocols, Gridstat alone cannot meet the delivery requirements of PMU applications. Rather, the underlying network protocols themselves must also be improved. This is the emphasis of our research here.

In this chapter, 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 (either due to excessive delay or actual packet loss) is considered failed. We propose, design, and evaluate solutions to all three aspects

[1]Protocols other than OpenFlow can be used. OpenFlow is the first and most popular protocol used to interface between SDN control and data planes.

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 chapter:

- **Design a link-failure detection algorithm**. We design a link-failure detection and reporting mechanism, PCOUNT, that uses OpenFlow [27] 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 [1, 8, 15] focuses on measuring end-to-end packet loss, resulting in slower detection times.

- **Formulate a novel optimization problem for computing backup multicast trees.** Inspired by MPLS fast-reroute algorithms that quickly reroute time-critical unicast IP flows over pre-computed backup paths [32], 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 [11, 13, 28, 30, 35] that use optimization criteria specified over a *single* multicast tree and typically emphasize maximizing node (link) disjointedness between the backup and primary path, while we consider conditions specified across *multiple* multicast trees.

- **Prove Multicast Recycling is at least NP-hard and propose an approximation algorithm, Bunchy, for Multicast Recycling.** BUNCHY uses an approximation to the directed Steiner Tree problem taken from the literature [9] to compute backup trees and modifies link weights to encourage backup trees to reuse existing forwarding rules installed in the network. Doing so reduces both the number of control messages the controller must send to install each backup tree and the number of forwarding rules maintained at each switch.

- **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 in cases where multiple multicast trees have common forwarding behavior.

- **Design two algorithms – Proactive and Reactive – for fast backup tree installation.** PROACTIVE pre-installs backup tree forwarding rules and activates these rules after a link failure is detected, while, REACTIVE installs backup trees *after* a link a failure is detected. We show how MERGER can be applied to PROACTIVE and REACTIVE to reduce the amount of PROACTIVE pre-installed forwarding state and decrease REACTIVE signaling overhead.

- **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 [26], an open-source OpenFlow controller.

  We use simulations based on the Mininet emulator [22] to evaluate our algorithms over synthetic graphs and actual IEEE bus systems. We find that PCOUNT provides fast and accurate link loss estimates: after sampling only 75 packets, the 95% confidence interval is within 15% of the true loss probability. Additionally, we find PROACTIVE yields faster recovery than REACTIVE (REACTIVE sends up to 10 times more control messages than PROACTIVE) but at the cost of storage overhead at each switch (in our simulations, pre-installed backup trees can account for as much as 35% of the capacity a conventional OpenFlow switch [12]). Lastly, we observe that MERGER reduces control plane messaging and the amount of pre-installed forwarding state by a factor of 2 to 2.5 when compared to a standard multicast implementation, resulting in faster installation and smaller flow table sizes.

The remainder of this chapter is structured as follows. In the following section (Section 1.2), we provide necessary background on PMU application requirements and OpenFlow, as well as introduce a running example used later to describe our algorithms. Then, we outline our algorithms in Section 1.3: Section 1.3.1 details our link-failure detection algorithm called PCOUNT; in Section 1.3.2, we outline our algorithms for computing backup multicast trees; and then describe algorithms for installation backup trees in Section 1.3.3; Section 1.3.5 presents MERGER, a fast multicast implementation when applied to our backup tree installation algorithms can significantly improve performance. Next, we briefly survey relevant literature (Section 1.4). Our simulation study is presented in Section 1.5. Section 1.6 concludes this chapter with a summary.

1.2 Preliminaries

In this section, we provide the necessary background to describe our algorithms in Section 1.3. First, we describe several PMU applications and their QoS requirements (Section 1.2.1). Then, we present a motivating example referenced throughout this chapter (Section 1.2.2). Section 1.2.3 defines terms and notation. We give an overview of OpenFlow in Section 1.2.4 and Section 1.2.5 details our OpenFlow multicast implementation.

1.2.1 PMU Applications and Their QoS Requirements

In this work, we consider the design of a communication network for disseminating critical Smart Grid data, principally data associated with PMU applications. Here we detail the QoS requirements of a few of these applications, with a particular emphasis on PMU applications with the most stringent end-to-end delay requirements. Table 1.1 shows packet delay and frequency requirements of three PMU applications, each described below.

PMU Application	**E2E Delay**	**Rate (Hz)**
SIPS	$8 - 16$ ms	$120 - 720+$
Wide Area Control	$5 - 50$ ms	$1 - 240$
Anti-Islanding	$5 - 50$ ms	$30 - 720+$

Table 1.1. PMU applications and their QoS requirements [3]. The end-to-end (E2E) delay requirement is *per-packet*, as advocated by Bakken et al. [3].

System Integrity Protection Scheme (SIPS) applications ensure that the entire power grid remains in a healthy state after local power grid components (e.g., relays) have taken mitigating actions to remedy local emergencies. To do so, SIPS applications require accurate and timely PMU measurements to identify system instability and to take correct mitigating actions (e.g., trip power generation) [3].

Islanding is a safety measure commonly used in power systems that separates entire sections (i.e., an island) of the power grid from the larger system in periods of voltage and frequency instability. Due to a lack of real-time situational awareness, power grid operators and oversight bodies mandate a conservative approach of disconnecting all distributed generation sources (e.g., wind and solar) from a locally islanded system. PMU measurements can provide the situational awareness to allow more narrowly targeted islands to be identified. Moreover, as distributed generation increases (as is the current trend), simultaneously disconnecting large number of generation sources may actually further destabilize the grid. PMUs are critical to future anti-islanding applications that will use PMU data to determine when distributed generation sources can safely remain online and thereby help preserve power grid stability [3].

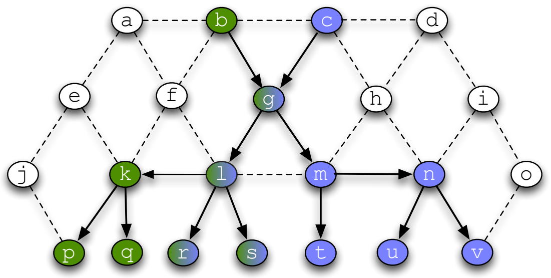
Wide-area control is a more general category of PMU applications, referring to applications that gather PMU data from disparate sources to determine the health of the power grid and initiate control actions, both in real-time. For example, Southern California Edison [19] measures (using PMUs) and controls voltage at remote locations to ensure voltage safety limits are met for its bulk energy transfers. Another wide-area control application uses PMUs to measure, detect, and dampen inter-area oscillations in real-time[3].

1.2.2 Motivating Example

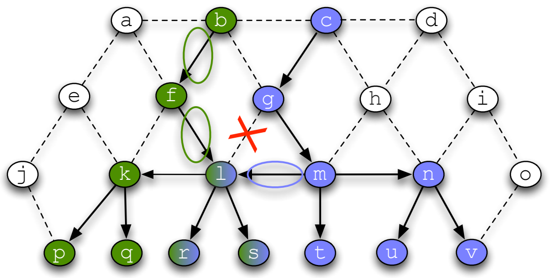
Here we present a motivating example to highlight different aspects of the challenges addressed in this work. This example is used throughout this chapter to help describe our algorithms.

Figure 1.1 shows two source-based multicast trees used to disseminate PMU measurement data produced by a PMU at b and another at c. The multicast tree, T_b, shown in green, is rooted at b and disseminates b's PMU data to its leaf nodes (i.e., data sinks) $\{p, q, r, s\}$. The blue multicast tree, T_c, multicasts c's PMU measurements to its leaf nodes $\{r, s, t, u, v\}$. Half-green/half-blue nodes are in both T_b and T_c. Dashed and solid lines indicate network links, with links in the multicast tree marked by arrows. Before any link failures occur, the original multicast trees (Figure 1.1(a)) meets the delay requirements specified by each data sink.

At some point, link (g, l) fails (e.g., its loss rate exceeds a threshold or it goes completely offline). This prevents p, q, r and s from receiving any packets from either T_b or T_c until each multicast tree is repaired, leaving the delay requirement of each these sink nodes unsatisfied. Figure 1.1(b) shows two backup multicast trees – one for T_b and the other for T_c – installed after it is detected that (g, l) has failed. Notice that each backup tree contains no path using the failed link, (g, l), and has a path between its root and each of its data sinks. In the following sections we present algorithms that detect these types of link failures, compute backup multicast trees such as those shown in Figure 1.1(b), and quickly install these backup trees in order to minimize packet loss and delay.



(a) Original multicast trees T_b (green) and T_c (blue).



(b) Backup multicast trees – $\hat{T}_b$ and $\hat{T}_c$ for T_b and T_c, respectively – installed after (g, l) fails. New links for each backup tree are circled.

Figure 1.1. Example problem scenarios with two source-based multicast trees, one rooted at b (in green), T_b, and the other at c (in blue), T_c. Half-blue/half-green nodes are members of both multicast trees. Let f_b and f_c denote the multicast flows corresponding to T_b and T_c, respectively. The multicast trees are shown before and after link (g, l) fails.

1.2.3 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 (typically via other switches). 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 T_i edges.

For convenience, denote $T_i^l = (V_i^l, E_i^l, r, S)$ as the *ith* directed tree with $l \in E_i^l$. For each link l in each directed tree i, T_i^l is a backup tree $\hat{T}_i^l = (\hat{V}_i^l, \hat{E}_i^l, r, S)$, a directed tree with root r, a directed path from r to each $s \in S$ such that $l \notin \hat{E}_i^l$. We refer to T_i^l and $\hat{T}_i^l$ as a *primary tree* and *backup tree*, respectively. In Figure 1.1(b) the two backup trees – one for T_b and the other for T_c – both route around the failed link,

(g, l), and have a directed path from its root (b and c) to their data sinks ($\{p, q, r, s\}$ and $\{r, s, t, u, v\}$).

Corresponding to each primary tree, $T_i = (V_i, E_i, r, S)$, is a multicast flow $f_i = (r, S)$ with source, r, and data sinks $S = \{d_1, d_2, ...d_k\}$. Each $d_i \in S$ has an end-to-end per-packet delay requirements and loss rate requirement (specified as the maximum tolerable loss rate of each $e \in E_i$). Let F be the set of all multicast flows in G.

Lastly, we make the following simplifying assumptions:

- Before any link fails, we assume that all packets (of PMU data) are correctly delivered such that each data sink's per-packet delay and loss requirements are satisfied.

- All sinks have the same same per-packet delay and loss rate requirements.

- We consider the case where multiple links fail over the lifetime of the network but assume that only a *single link fails at-a-time.*

1.2.4 OpenFlow

Our algorithms are built using OpenFlow abstractions and features. Here we provide a brief overview of OpenFlow, with a particular emphasis on the features used by our algorithms.

OpenFlow is an open standard that cleanly separates the control and data planes, and provides a programmable (and possibly centralized) control framework [27]. 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 exposes the flow tables of its switches, allowing the controller to add, remove, and delete flow table entries, which determine how switches forward, copy, or drop packets associated with a controller-managed flow. We will use the terms "flow table entry" and "forwarding rule" interchangeably.

OpenFlow switches follow a "match plus action" paradigm [27], in which each switch *matches* an incoming packet based on its header fields to a flow table table entry. *Actions* (e.g., forward packet, drop packet, copy packet, or modify packet header fields) are then applied to the packet as encoded in the flow table entry instructions. Switches maintains statistics for each flow table entry (e.g., packet counter, number of bytes received, time the flow was installed) that can can be queried by the controller. These statistics are key to our algorithm for detecting packet loss (Section 1.3.1).

Several of our algorithms use OpenFlow to modify packet headers to customize forwarding and other actions in parts of the network. We write identifiers in unused packet header fields to customize the set of actions applied to packets carrying specific identifiers. We refer to these identifiers as *tags*. Tags are an abstraction we use to measure packet loss rates (PCOUNT in Section 1.3.1), pre-install backup tree flow table entries (PROACTIVE in Section 1.3.3), and consolidate flow table entries that have common forwarding state (MERGER in Section 1.3.5).

Measurement studies from the literature [12, 33] have identified significant hardware limitations in OpenFlow switches. OpenFlow switches can only support a limited number of flow table entries because they rely on expensive TCAM memory to perform wildcard matching. For example, the HP5406zl switch supports approximately 1500 OpenFlow rules [12] and the Pronto 3290 switch can handle 1919 flow table entries [14]. Another major limitation is control plane bandwidth: OpenFlow switches have been found to have four orders of magnitude less control plane bandwidth than data plane forwarding bandwidth [12]. This limited bandwidth, along with their slow control/management CPUs, limits the rate in which flow table entries can be installed. Our simulation study (Section 1.5) shows the tangible effects these hardware limitations have on our algorithms (especially PCOUNT).

1.2.5 Multicast Implementation

In keeping with its role as a general framework that provides primitives for programmable networks, OpenFlow does not explicitly provide an implementation for multicast. Thus, 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. [2]

After the controller computes a multicast tree (described in Section 1.3.2), $T_i = (V_i, E_i, r, S)$, 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. If a switch in E_i is adjacent to a downstream host, h_j, in the multicast group, then the flow table entry rewrites the destination layer 2 and 3 addresses of the packet copy sent to h_j to h_j's layer 2 and 3 addresses. [3]

1.3 Algorithms

We propose a set of algorithms, collectively referred to as APPLESEED, that make multicast trees robust to link failures. [4] 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:

[2]Because multicast group membership is static for power grid applications (Section 1.2.1, we simply determine the members of each multicast group by reading their static assignment from a text file. Note that if dynamic group membership were to be required, we could replace this static policy using a protocol like IGMP.

[3]Our initial plan was to use the group table abstraction described in the OpenFlow 1.1 specification [29] to implement multicast but, unfortunately, as of the writing of this chapter, this feature is not yet supported by the POX controller [26] used to implement our algorithms and the Mininet emulator [22] used in our simulations.

[4]The name APPLESEED is inspired by Johnny Appleseed, the famous American pioneer and conservationist known for planting apple nurseries and caring for its trees.

1. **Pcount algorithm: monitor and quickly detect link failures** when and where they occur inside the network (Section 1.3.1).

2. **Precompute backup trees** that are amenable to fast installation. In Section 1.3.2 we formulate a new problem, Multicast Recycling, that aims to compute backup trees that reuse primary tree edges, prove Multicast Recycling is at least NP-hard, and provide an approximation algorithm for Multicast Recycling called Bunchy.

3. **Fast install of pre-computed backup trees** by reusing existing forwarding rules installed in the network, sharing forwarding rules among backup trees with common links, and in some cases pre-installing forwarding rules before link failures occur (Section 1.3.3). The backup trees are computed using Bunchy from part (2). Pcount, from part (1), triggers the installation of a set of backup trees.

1.3.1 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 [1, 8, 15] focuses on measuring end-to-end packet loss, resulting in slower detection times.

Pcount considers a link as failed when the rate of packet loss exceeds a threshold, given as input. For simplicity, our description of Pcount assumes it measures packet loss over a single link, (u, d). At the end of the section we comment on how Pcount easily generalizes to detect packet loss between multiple switches and non-adjacent switches.

PCOUNT estimates packet loss over a sampling window of length w. For each w, PCOUNT estimates packet loss along (u, d) by measuring the aggregate loss rate experienced by flows $M = \{f_1, f_2, ..., f_k\}$ across link (u, d), where M is given as input, using the following steps:

1. **Install rules (downstream) to count all tagged f_i packets received at d.** PCOUNT does so by installing a new flow table entry for each f_i at d, that matches packets using the identifier (i.e., the tag) applied at u in step (2). As noted in Section 1.2.4, for each flow table entry, OpenFlow automatically updates the packet counter each time a packet matches the flow table entry.

2. **Tag (upstream) all packets from each $f_i \in M$.** 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 a 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, turn tagging off at u** by installing a copy of each f_i's original flow table entry, e_i, but with a higher priority than e_i'.

4. **Query u and d for packet counts** in order to compute the packet loss. Each tagging rule is queried individually at u, while a single aggregate query (matching flows based on their the `vlan_id` field) is issued at d to retrieve the packet counts of total packet count of all $f_i \in M$. [5] Before querying d, PCOUNT waits time proportional to half the average RTT from u to d, starting from the time step (3) completes, to ensure all in-transit packets are considered at d.

[5]We are unable to issue an aggregate query at u because OpenFlow does not support query predicates specified over flow table entry actions. In our case, it would be convenient if we could to specify an aggregate query of the form "return statistics of all flow table entries that write identifier x in the `dl_vlan` field".

5. **Signal an alarm** if the estimated loss rate exceeds the input threshold.

6. **Delete tagging and counting flow table entries** created in steps (1) and (2).

Consider the example in Figure 1.1 and assume PCOUNT monitors packet loss of both multicast flows traversing (g, l); f_b for primary tree T_b (blue) and f_c for primary tree T_c (green). First, PCOUNT selects a unique `dl_vlan` value (the identifier) and installs two flow table entries at l, one for f_b and the other f_c. These flow table entries match packets based on the packet's multicast address and `dl_vlan` value. Next, a flow table entry for f_b and f_c is installed upstream at g that writes the `dl_vlan` identifier in each packet sent along the outgoing link to l. After w seconds, tagging is turned off at g and the flow statistics are read from g and l. Two individual flow statistic queries are sent to g, while a single aggregate query at l gathers packet counts of the two flow table entries installed in the first step. Lastly, the packet counts are used to compute the loss rate and, if necessary, PCOUNT raises an alarm if the measured loss rate during window w exceeds the given threshold. If not, a new PCOUNT session is initiated, repeating the above steps.

Tagging ensures that for monitored flow PCOUNT accounts for all packets dropped during w. However, in some cases PCOUNT may be configured to monitor a subset of flows traversing a monitored link because doing so can reduce the time required to compute packet loss, since $k + 1$ statistic queries are required when monitoring k flows. For example, in Figure 1.1 PCOUNT may only monitor f_b's packet loss along (g, l). As a result, only a single read statistics request needs to be sent to g rather than two statistic queries if f_c were also monitored. However, monitoring a subset of flows means that PCOUNT does not account for packet loss of unmonitored flows. In Section 1.5.1 we use simulations to explore how adjusting the number of monitored flows affects the speed and accuracy of packet loss estimates.

Although PCOUNT sends instructions simultaneously to start tagging each M flow (step 2) and, likewise, sends all k messages in parallel to stop tagging (step 3), in practice these actions are unlikely to be executed at the same time. The implication is that across each $f_i \in M$ the start and stop time of w is not perfectly synchronous. This does not affect the accuracy of PCOUNT packet loss measurements, provided that all tagged packets that will eventually reach the downstream node do so before that node is queried.

Pcount Extensions. No changes are required for PCOUNT to monitor packet loss between non-adjacent switches. Consider the case where PCOUNT measures packet loss between two non-adjacent switches a and b. The PCOUNT actions at a and b are the same as described above, while the forwarding at any switches along a path between a to b disregards (as they already do) any tag applied at a or b. In this scenario, PCOUNT measures packet loss of a path rather than that of a single link.

PCOUNT can also be used to monitor packet loss between multiple (possibly) non-adjacent switches. Consider the example topology in Figure 1.2, where PCOUNT measures packet loss between u and downstream nodes d_1 and d_2. For simplicity, we assume a single flow multicasts packets from u to d_1 and d_2. PCOUNT installs a rule to tag packets at u, leaves v is unchanged, and installs a rule at d_1 and d_2 to count packets tagged at u. Then, PCOUNT (as its only modification) queries u, d_1, and d_2 for their packet counts.

Notice that by comparing packet counts between u, d_1, and d_2, packet loss of links (u, v), (v, d_1), and (v, d_2) can all be estimated using network tomography techniques [7]. For example, if u and d_1 have the same packet counts but d_2 counts fewer packets than u, we can infer packet loss incurs along (v, d_2). This approach provides the same coverage (scope of packet loss measurements) as an alternative approach that creates three separate PCOUNT sessions between (u, v), (v, d_1), and (v, d_2), but does do so using fewer measurement points. We later find in our simulations (Section 1.5.1) that

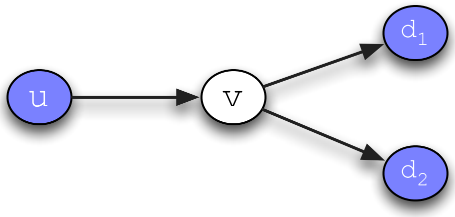


Figure 1.2. Example topology used to explain how PCOUNT can be used to monitor packet loss between multiple non-adjacent switches.

monitoring a large number of flows using a single link incurs high processing time at network switches, suggesting that the savings projected here of running a single PCOUNT session between multiple switches can provide significant savings.

1.3.2 Computing Backup Trees

APPLESEED pre-computes backup trees to install after PCOUNT detects a link failure. Here we present a new problem, MULTICAST RECYCLING, and an approximate solution to MULTICAST RECYCLING, called BUNCHY, that aim to facilitate fast recovery by computing backup trees that maximize the number of edges common between each backup tree and its primary tree. This reuse of primary tree edges speeds recovery from link failures because, in SDN, this reduces the number of new flow table entries that need to be installed in network routers in response to a link failure.

APPLESEED uses BUNCHY as a part of system initialization, where the set of backup trees are computed for each network link, l; APPLESEED computes a single backup tree for each primary tree using l. Additionally, BUNCHY is used after a set of backup trees, $\hat{T}^l$, are installed in response to a link failure. For each newly installed tree $\hat{T}^l_i \in \hat{T}^l$, APPLESEED computes a backup tree for each link in $\hat{T}^l_i$.

1.3.2.1 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, when generating the forwarding rules for multicasting packets using the backup tree, to use primary tree rules already installed in the network rather than install new ones. This speeds recovery in cases where backup trees are installed *after* a link failure is detected and reduces the number of flow table entries pre-installed at switches (control state) when backup trees are installed *before* a link failure occurs. Reducing control state is especially important with OpenFlow because OpenFlow switches can only store a limited number of flow table entries (see Section 1.2.4). [6]

For the primary tree $T_i^l = (V_i^l, E_i^l, r, S)$ and its backup $\hat{T}_i^l = (\hat{V}_i^l, \hat{E}_i^l, r, S)$, we define a binary variable c_v^l for all $v \in \hat{V}_i^l$. If v has exactly the same predecessors (outgoing edges) in T_i^l and $\hat{T}_i^l$, then c_v^l takes value 0. Otherwise, $c_v^l = 1$. For the T_i^l,$\hat{T}_i^l$ pair define:

$$C_i^l = \sum_{\forall v \in \hat{V}_i^l} c_v^l \tag{1.1}$$

For our purposes, C_i^l is the number of new rules (i.e., non-recycled primary tree rules) needed to install $\hat{T}_i^l$. Note that the primary tree rules not recycled by $\hat{T}_i^l$ should be deleted after l fails and $\hat{T}_i^l$ is installed, especially considering the limited size of OpenFlow switch flow tables (Section 1.2.4). As we describe in Section 1.3.4, these rules can be garbage collected in the background because stale primary tree forwarding rules do not affect how packets are (correctly) forwarded by $\hat{T}_i^l$. For this reason, Multicast Recycling aims to minimize the number of new rules needed

[6]Following from our assumption that a single link fails at-a-time, Multicast Recycling assumes that all other links besides the failed one, l, satisfy packet loss requirements.

to install a backup tree, rather than the number of primary tree rules that must be garbage collected after a link failure.

Consider the example in Figure 1.1 where (g, l) fails. The green backup tree, $\hat{T}_b$, shown in Figure 1.1(b), has $C_b = 2$ because a new forwarding rule is required at b, and f to account for the new outgoing links at each node. $\hat{T}_c$, in blue, has only one link, (m, l) not in $\hat{T}_c$'s primary tree. As a result, $C_c = 3$.

Our Multicast Recycling problem definition below references a modified version of the Steiner tree problem, called the Steiner-Arborescence problem [9]. As input, Steiner-Arborescence is given a directed graph $G = (V, A)$, a root vertex r, and a set of terminals, S. An arborescence is defined as a tree rooted at r that has directed edges spanning S. Steiner-Arborescence aims to find a minimum cost arborescence, called a Steiner arborescence or directed Steiner tree. We denote $SA_i(G) = (V, E, r, S)$ as the Steiner arborescence computed over directed graph, G, rooted at r, and spanning S such that $r, S \in T_i$.

We formulate the Multicast Recycling problem as follows:

- Input: (G, T^l, l, α) where $G = (V, E)$ is a directed graph, $T^l = \{T_1^l, T_2^l, \ldots T_k^l\}$ where each $T_i^l \in T^l$ is a primary tree that uses l, $l \in E$, and $\alpha \geq 1$.

- Output: A backup tree for each primary tree using l. This set of backup trees, $\hat{T}^l = \{\hat{T}_1^l, \hat{T}_2^l, \ldots, \hat{T}_k^l\}$:

$$\begin{aligned} &\text{minimize} \quad \sum_{1 \leq i \leq k} C_i^l \\ &\text{subject to} \quad w(\hat{T}_i^l) \leq \alpha \cdot w(SA_i(G')),\ \forall \hat{T}_i^l \in \hat{T}^l \end{aligned} \tag{1.2}$$

  where $G' = (V', E')$ such that $E' = E - \{l\}$ and $w(\hat{T}_i^l)$ is the sum of $\hat{T}_i^l$'s link weights. [7]

[7]We assume $\hat{T}_i^l$, satisfies all per-packet delay and loss requirements if $l \notin \hat{T}_i^l$ and $w(\hat{T}_i^l) \leq \alpha \cdot w(SA_i(G'))$

The objective function maximizes the reuse of primary tree edges, while α bounds how large the backup tree can grow as consequence of minimizing C_i^l. When applied to our problem scenario this formulation reduces the number of installation rules by reusing rules already installed in the network, under the constraint that the backup tree does not become too large to meet the end-to-end latency requirements. By defining G' as a copy of G with the failed link removed from G, we are assuming that all links in G besides l are operational. For our purposes, this amounts to assuming that all non-l links have packet loss rates less than their threshold.

Notice that we have defined C_i^l in Equation 1.1 on a per-backup tree basis where for backup tree $\hat{T}_i^l$, C_i^l is a relationship defined strictly between $\hat{T}_i^l$ and its primary tree T_i^l (there are no constraints specified across any other primary or backup tree). As a result, the globally optimal solution for Multicast Recycling (i.e., the optimal set of backup trees for a single link) can be found by computing the optimal backup for each primary tree in isolation and then taking the union of these solution We shall revisit this important property when describing our approximation algorithm for Multicast Recycling.

Theorem 1.1. Multicast Recycling *is at least NP-hard.*

Proof. The details of our proof can be found in Appendix **??**. This proof shows that Multicast Recycling is NP-hard even when considering just a single backup tree. The proof demonstrates that in some cases an optimal solution to Multicast Recycling requires a solution to Steiner-Arborescence, a problem known to be NP-hard. This proves Multicast Recycling is NP-hard when considering a single backup tree and therefore the general Multicast Recycling problem for k backup trees must at least be NP-hard. □

1.3.2.2 Bunchy Approximation Algorithm

Bunchy is a simple approximation algorithm for Multicast Recycling that manipulates link weights to encourage each backup tree to reuse primary tree edges. For each link l, Bunchy separately computes a backup tree for each primary tree using l and then returns the union of these computed trees.

Bunchy leverages the $\sqrt{s}$ Steiner-Arborescence approximation, where s is the number of terminal nodes, from Charikar et al. [9]. Their approximation algorithm computes bunches, where a *bunch* is a subgraph formed by taking the shortest path from the root to an intermediate vertex, i, and the union of shortest paths from i to the terminal nodes. The algorithm produces the bunch with best *density* – density is the average cost of connecting a terminal node with the root – as its approximation. The lowest density bunch can easily be computed in polynomial time: a brute-force approach that tries all possible nodes as the intermediate vertex yields an $O(ns^2 \log s)$ time algorithm.

Given (G, T^l, l, α), for each $T_i^l \in T^l$ Bunchy uses the following two-step procedure to compute $\hat{T}_i^l$:

1. Make a copy of G called $G' = (V', E')$ and remove l from E'. Set the link weight of each $e \in T_i^l$ to 0 and the link weight of $e \notin T_i^l$ to 1.

2. Run the Steiner-Arborescence approximation, using the brute-force approach described above, over G' and set $\hat{T}_i^l$ to be the result. If $\hat{T}_i^l$ satisfies the Equation 1.2 constraint, return $\hat{T}_i^l$ 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 penalty (i.e., adding cost to the backup tree) and so encourages reusing primary tree edges. If Bunchy returns False in Step (2) either α must be made larger or a new multicast tree should be computed from scratch that satisfies the tree-size constraint.

In Figure 1.1, Bunchy uses f as the the intermediate node for $\hat{T}_b$, yielding density of 2 (the cost of connecting terminals p,q,r, and s to the root is 2). Bunchy selects f as the intermediate node by iterating over all nodes and remembering the node with the smallest density. g or l could have been used as the intermediate node because both, like f, have density of 2 (f is selected arbitrarily using a tiebreaker). The bunch for $\hat{T}_c$ is formed using m as the intermediate node with density 0.4: the cost of connecting r and s to the root is 1 and t,u, and v connect with the root at 0 cost.

1.3.3 Installing Backup Trees

We are now ready to describe the last part of Appleseed, installing backup trees. Installing a backup tree is a two-step process. First, the flow table entries that forward packets along the backup tree 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 $l \in E^l$, $\hat{T}^l = (\hat{V}^l, \hat{E}^l, r, S)$.

Reactive Algorithm. Reactive first determines which nodes require a new forwarding rule. In cases where $\hat{T}^l$ and T^l use exactly the same outgoing links of a common node, u, we say $\hat{T}^l$ can "reuse" T^l's forwarding rule at u; since T^l's forwarding rule is already installed at u, no new forwarding rule (for $\hat{T}^l$) needs to be installed. Forwarding rules are only required at any $v \in \hat{V}^l \setminus V^l$ and at each $v \in V^l \cap \hat{V}^l$ with different outgoing links in $\hat{T}^l$ and T^l. We refer to this set of nodes as B^l.

Consider T_b and $\hat{T}_b$ in the Figure 1.1 example. Because T_b and $\hat{T}_b$ share the same outgoing links at l and k, $\hat{T}_b$ can reuse T_b's flow table entry at each of these nodes, whereas new forwarding rules are required at b and f.

REACTIVE then pre-computes a *basic* flow table entry for each $b \in B^l$. Like the flow table entries BASIC computes (see Section 1.2.5), a basic flow table entry, for a multicast tree T_i and $u \in V_i$, matches packets using T_i's multicast address and has instructions to forward matching packets out the correct ports at u. Lastly, when l fails, the REACTIVE signals each $b \in B^l$ to install the pre-computed basic flow rule.

Proactive Algorithm. PROACTIVE computes and installs backup tree flow table entries *before* a primary tree link, l, fails. After l fails, PROACTIVE signals the backup tree root to install a forwarding rule that activates the backup tree. We use the term "activate" to indicate that packets are multicasted using the backup tree rather than the primary tree. Note that the PROACTIVE algorithm can respond quickly to link failures, as only a single new flow table entry needs to be installed at the backup tree root.

PROACTIVE cannot, without modifications, pre-install basic flow table entries at all nodes because incorrect forwarding would result. Doing so at a node, d, common to the primary and backup tree, 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. We say that $d \in D^l$, where D^l contains each node with one or more outgoing links in T^l and one or more outgoing links in $\hat{T}^l \setminus T^l$. Revisiting $\hat{T}_c$ from Figure 1.1 example, $g, m \in D^l$ and so installing a forwarding rule at these two switches before (g, l) fails would be problematic for the reasons just described.

To circumvent this issue, PROACTIVE assigns a unique *backup tree id*, denoted `bid`, to each backup tree. For each $d \in D^l$, the flow table entry matches and forwards packets using the `bid` value written in the `dl_src` field. When the backup tree $\hat{T}^l$ is activated, PROACTIVE writes the `bid` in the `dl_src` packet header field, indicating that these packets should be disseminated by $\hat{T}^l$ rather than T^l. In more detail, PROACTIVE preinstalls and activates $\hat{T}^l$ using the following steps, where we assume $\hat{T}^l$ has `bid=AA`:

1. At each $d \in D^l$, PROACTIVE pre-installs a flow table entry matching packets using $\hat{T}^l$'s multicast address, `dl_src = AA`, and has wildcards for all other match fields. PROACTIVE preinstalls a basic flow table entry at each $b \in B^l \setminus D^l$ that matches packets using $\hat{T}^l$'s multicast address and has wildcards for all other match fields (including `dl_src`).

2. When it is detected that l fails, PROACTIVE installs a rule at the $\hat{T}^l$ root node that writes `AA` in the `dl_src` header field of each $\hat{T}^l$ packet.

For $\hat{T}_c$ in Figure 1.1, PROACTIVE pre-installs a forwarding rule at g and m that matches packets using $\hat{T}_c$'s `bid`, `CC`, and T_c's multicast address. After (g, l) fails, PROACTIVE signals c to write `CC` in the `dl_src` header field of each $\hat{T}_c$ packet. As a result, packets at g are correctly forwarded to m and, similarity, m correctly forwards packets to l, n, and t.

Comparing Proactive and Reactive. Since PROACTIVE must signal only a single node, as opposed to multiple nodes with REACTIVE, to install a backup tree, PROACTIVE is fast. However, PROACTIVE's speed comes at the cost of storing a potentially large number of flow table entries at the switches, especially since APPLESEED computes, for each primary tree, a backup tree for each primary tree link. REACTIVE, on the other hand, only installs backup tree flow table entries after a link failure is detected. These trade-offs are studied in Section 1.5 using simulations.

1.3.4 Garbage Collection

After a link fails, primary tree forwarding rules may become stale. APPLESEED's garbage collection routine identifies and deletes these stale flow table entries. Because garbage collection is not needed for correct data dissemination, garbage collection is run when necessary to free switch flow table space.

Garbage collection is straightforward, but more involved than simply deleting all flow table entries of each primary tree, T^l, using l. Doing so would be problematic because a backup tree may be reusing one of these flow table entries. To address this, APPLESEED maintains a dictionary, `rule_map`. For each node, v, `rule_map` records each flow table entry installed at v and the multicast trees using the flow table entry. When l fails, the garbage collection routine determines the set of stale forwarding rules for each $T_i^l \in T^l$ by consulting `rule_map`. A stale rule exists at each $v \in V_i \setminus \hat{V}_i^l$ (i.e., nodes unique to the primary tree) and each $d \in D_i^l$ (nodes where the backup tree diverges from the primary tree). Finally, each stale forwarding rule is either explicitly removed (if using a hard-state signaling protocol [18]) or APPLESEED allows the forwarding rule to timeout (if using a soft-state signaling algorithm [10]).

1.3.5 Optimized Multicast Implementation

As an optimization to the BASIC multicast implementation, described in Section 1.2.4, we present the MERGER algorithm. Given a set of directed trees, MERGER produces 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. 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.

In the next section (1.3.5.1), we motivate the need for MERGER by demonstrating several inefficiencies in BASIC. Next, Section 1.3.5.2 presents a simplified version of MERGER that considers only primary trees. Then, we extend MERGER in Section 1.3.5.3 to account for backup trees. Section 1.3.5.4 concludes the section with a discussion of how MERGER affects garbage collection and PCOUNT, along with informal commentary on its optimality.

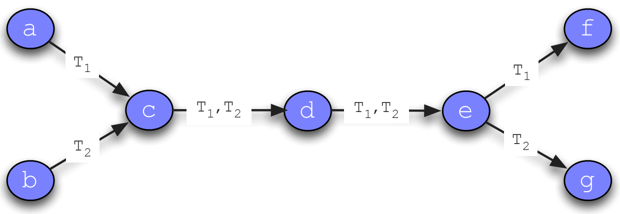


Figure 1.3. Example showing a subtree of two multicast trees, T_1 and T_2. The edges used by each multicast tree are marked.

1.3.5.1 Motivation: Basic Algorithm Inefficiencies

The Basic multicast implementation 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, v, may have multiple flow table entries executing the same forwarding actions. This occurs when multicast trees share the same outgoing links at v. [8] These inefficiencies are the motivation for developing the Merger algorithm, which replaces the set of flow table entries Basic would create at v with a single flow table entry. To do so, Merger writes a common identifier, or tag, in packet headers at the node immediately upstream from v. Using this tag, Merger creates a *single* rule at v to match and forward packets of *all* trees with the same outgoing links at v.

Consider the simple example shown in Figure 1.3 with two multicast trees T_1 and T_2. The directed links used by each tree are marked. Basic creates a flow table entry for T_1 at a, c, d, e, and f and a flow table entry at b, c, d, e, and g for T_2. Because T_1 and T_2 both use the same outgoing links at c and d, only a single forwarding rule is needed at each node. In the next two sections we describe how Merger finds duplicate forwarding actions and creates forwarding rules shared by multiple multicast trees.

[8]In cases where the tree can either be a primary tree or backup tree, we refer to the tree as a multicast tree.

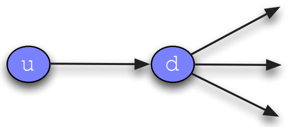


Figure 1.4. Subgraph used to describe MERGER in Section 1.3.5.2.

1.3.5.2 Merger Algorithm for Primary Trees

MERGER consolidates flow table entries at each node, v, where multiple primary trees share the same outgoing links. Upstream from v, MERGER writes an identifier, or *tag*, in packet headers and uses this tag to match packets at v using a single rule shared by each of these primary trees. The tag is removed downstream from v where the trees diverge.

A tag is a globally unique Ethernet address that MERGER writes in a packet header's `dl_dst` field (i.e., the Ethernet destination address). When possible, MERGER flow table entries use tags to match and forward packets, meaning that packets are matched solely on their `dl_dst` value. When the same Ethernet address is applied to the packets of more than one multicast tree, we refer to this as a *group tag*. A *single tag* is an Ethernet address used by only one tree. We use the term *tag* to generically refer to either a group or single tag. Note that, like BASIC, MERGER has each switch adjacent to a downstream host, h_j, overwrite the destination layer 2 (including the `dl_dst` field) and 3 fields in the packet forwarded to h_j, setting these fields to h_j's layer 2 and 3 addresses. This allows MERGER to safely modify the `dl_dst` field inside the multicast tree for its tagging purposes, while ensuring successful forwarding of packets to each multicast group host.

We are now ready to describe MERGER in more detail. First, MERGER marks the edges used by each primary tree. Then, MERGER executes a breadth first search of each primary tree, T_i, starting at its root. For each link $(u, d) \in T_i$ as shown in Figure

1.4, Merger determines the match pattern to create at d and the tagging actions to apply at u using the following steps: [9]

1. Finds the set of trees, S, using (u, d) that share the same outgoing links as T_i at d.

2. If $|S| \geq 1$, Merger creates an action to write a group tag at u. For each $T_j \in S \cup \{T_i\}$, Merger finds the rule at u used to forward T_j and appends an action to write a group tag to the rule's action list. Then, a single rule is created at d that matches packets using this group tag and has an initial action list forwarding packets out the appropriate ports.

3. If $S = \emptyset$, Merger looks upstream at u to determine whether to use a single tag or T_i's multicast destination address to match T_i's packets at d. When T_i is matched using either a single tag or its multicast destination address at u, Merger creates a rule at d to match packets using a single tag and writes this single tag at u. Otherwise, Merger creates a rule at d matching packets using T_i's multicast destination address (no action is needed at u).

In step (3), we aim to use single tags to match packets because they allow *any* backup tree to reuse T_i's rule at d by simply writing this tag at u. Whereas, if T_i's multicast address is used to match packets at d, only T_i's backup trees can reuse this forwarding rule (since T_i's multicast address is unique to its multicast group). We comment further on this design decision in the next section.

In the Figure 1.3 example, Merger creates an action at a and b to write a group tag in the packet headers of all packets traversing (a, c) and (b, c). Then, at c and d, Merger creates a single rule to match and forward packets based solely on this tag. With regard to the breadth-first search (BFS) described earlier, Merger executes

[9] Because T_i is a tree, (u, d) must be its only incoming link to d. Therefore, we can determine T_i's locally optimal tagging rule at d by only considering u and d.

the following steps in its breadth-first search (BFS) of T_1 at nodes c, d, and e. At c, $S = \{T_2\}$ so Merger finds T_1's rule at a and includes an action to write a group tag, `12`, in all packets sent out a's port to c. Then, Merger creates a flow table entry at c that matches packets with `dl_dst = 12` and forwards packets out the port to d. The same set of actions occur when the BFS reaches d. Merger creates a forwarding rule at e that matches packets using T_1's multicast address and forwards these packets to f.

1.3.5.3 Merger Algorithm for Backup Trees

Having discussed Merger for the primary tree case, we are now ready to extend Merger to generate merged forwarding rules for backup trees. Merger aims to reuse forwarding rules of primary trees because these rules are already installed in the network, allowing the installation algorithm (e.g., Proactive or Reactive) to avoid installation of redundant forwarding rules. In cases where primary tree rules cannot be reused, Merger consolidates flow table entries with other backup trees that have common forwarding behavior.

For a set of backup trees, $\hat{T}^l$, Merger generates backup-tree forwarding rules as follows. Merger executes two rounds of BFS, traversing all $\hat{T}^l_i \in \hat{T}^l$ in each round. In the first round, for each $\hat{T}^l_i$, Merger finds each node where $\hat{T}^l_i$ has the same outgoing links as a primary tree. If so, $\hat{T}^l_i$ reuses the primary tree flow table entry at this node, v: Merger writes the primary tree tag at v's parent node allowing $\hat{T}^l_i$ packets to be forwarded using the primary tree rule, and makes no changes at v.

In the second round of BFSs, Merger consolidates flow table entries among the other backup trees for l at nodes where primary tree tag reuse was not possible. To do so, the algorithm from Section 1.3.5.2 is executed but compares $\hat{T}^l_i$'s outgoing links with the outgoing links of each $\hat{T}^l_j \neq \hat{T}^l_i$ at nodes where $\hat{T}^l_i$ and $\hat{T}^l_j$ were unable to reuse primary tree tags.

When Merger is applied to Proactive and Reactive (referred to as Proactive+Merger and Reactive+Merger) the tag becomes the sole match criteria used by its flow table entry, with one exception. This occurs with Proactive+Merger when a `bid` is required to distinguish between a backup and primary tree, as described in Section 1.3.3. In those cases, the `bid` and `dl_dst` fields are both used as matching criteria.

1.3.5.4 Merger Discussion

We conclude our presentation of Merger by commenting on some of the properties of the algorithm along with the implications of using Merger on other important aspects of Appleseed.

Benefits. In comparison with Basic, Merger reduces the number of forwarding rules required to install each multicast tree. As a result, Merger is more space efficient and can yield faster backup tree installation; in Section 1.5.2 we quantitatively evaluate these gains. Recall that for each primary tree, Appleseed pre-computes a backup tree for each of its links. In this scenario Proactive+Merger can significantly reduce the number of pre-installed rules. These savings are especially important because, as noted in Section 1.2.4, OpenFlow switches can only support a limited number of flow table entries. With Reactive, Merger can yield faster recovery because fewer backup tree flow table entries translates into fewer control messages to activate backup trees.

Time Complexity. Like Basic, Merger's complexity is bounded by the breadth-first search (BFS) executed for each of the m multicast trees given as input. At each node, v, visited in the BFS, Merger compares the out-links used by all other multicast trees that use v. Since there can be at most m such trees, this takes $O(m)$ time.

If we let n be the number of graph nodes, each BFS takes $O(mn)$ time [10] and the total time complexity of Merger is $O(m^2n)$.

Garbage Collection. Appleseed's garbage collection algorithm remains unchanged from the description in Section 1.3.4. Recall that `rule_map` is a dictionary that maps the flow table entry installed at each node to the set of backup and primary trees using the flow table entry. The only difference between Basic and Merger garbage collection is the number of stale rules it identifies (likely fewer stale rules with Merger) not how the stale rules are found. As with Basic, Merger garbage collection can find any stale forwarding rules by consulting `rule_map`.

Do No Harm. We say that Merger is an algorithm that does "no harm" [11] because (a) Merger never creates more flow table entries than Basic and (b) when generating rules for backup trees, Merger makes no modifications to flow table entries of primary trees that do not use the failed link. We informally demonstrate each of these properties below.

Regarding (a), consider an arbitrary multicast tree (primary of backup tree), T_i. Merger creates at most one flow table entry at any $v \in T_i$. The flow table entry, e_i, either matches and forwards packets using a group tag, a single tag, or using the T_i's multicast address. Any e_i tagging actions are simply appended to e_i's action list (when Merger visits T_i's children of v), requiring the creation of no additional flow table entries. We conclude that in the worst case, Merger creates the same number of flow table entries as Basic.

Now consider property (b) where we let T_j refer to a primary tree not using l. By construction, Merger only creates flow table entries and new actions for the backup

[10] BFS has $O(|V| + |E|)$ complexity. In our case, each BFS is over a directed tree, meaning the number of edges traversed in each BFS is $O(n-1)$. Therefore, we can simplify BFS time complexity to $O(n)$.

[11] This is similar in spirit to the Hippocratic Oath taken by physicians that they will "never do harm [to patients]"

trees of link l (i.e., $\hat{T}^l$). These flow table entries have different match criteria than T_j's. If a backup tree reuses T_j's flow table entry at v, no changes are made to this flow table entry. Lastly, Appleseed's garbage collection algorithm ensures that no T_j flow table entry is removed.

Optimality. Because Merger makes tagging decisions locally at each node, v, based only on the multicast trees using v's outgoing links and the tags used at v's parent nodes, Merger does not always yield the minimum set of forwarding rules. Consider again Figure 1.3 but replace T_1 with S_1 and T_2 with S_2, where S_1 and S_2 are sets of multicast trees of size k. In this scenario, Merger writes the same group tag, denoted `12`, at a and b for each tree in S_1 and S_2. Then, at c and d Merger installs a single rule that matches and forwards using the `12` tag. Lastly, for each $T_i \in S_1 \cup S_2$, a rule is created at e that matches packets using each T_i's multicast address.

A better solution in this example would be to use a different group tag for S_1 and S_2. In this case, let `11` be the group tag applied to each tree in S_1 at a and `22` the tag written in the packet header of each tree in S_2. These group tags can be used to create two separate rules at c, d, and e to forward S_1 packets based on `11` and S_2 packets using `22`. By using different tags for S_1 and S_2, we avoid having to create $2k$ separate rules for each tree in $S_1 \cup S_2$ at e, clearly a better solution that the one produced by Merger.

This example suggests that an algorithm, $\mathcal{A}$, that finds the minimum number of forwarding rules for a set of multicast trees T must consider, for $S \subseteq T$ where each multicast tree $S_i \in S$ uses link (u, d), how each of S's subsets share links downstream from d. Since this requires computing the power set of S and there are an exponential number of ways S's subsets can use common links downstream from d, we conjecture that no polynomial time $\mathcal{A}$ exists.

Implications for Pcount. PCOUNT requires no changes to monitor the packet loss of flows forwarded using MERGER rules. However, we make the case here that MERGER can improve the accuracy of PCOUNT loss rate estimates and reduce the time to compute these estimates. In Section 1.5.1, we will find that our simulations bear out the qualitative argument made here.

Recall from Section 1.3.1 that with PCOUNT the number of monitored flows, k, can be tuned. Determining an appropriate value for k involves a trade-off between the accuracy of packet loss estimates and time: larger k yield more accurate packet loss estimates but at the cost of slower detection times (the time between when packet loss occurs and when it is detected). Detection times of a monitored link, (u, d), increase with larger k for two reasons. First, for each of the k flows, PCOUNT makes a copy of the flow's forwarding rule at u and d in order tag and count packets. Secondly, PCOUNT sends k queries to u to read the state of each flow table entry generated by PCOUNT.

With MERGER, the same flow table entry, e_i, can be used by r multiple flows. In these cases, PCOUNT only needs to explicitly monitor one of these flows to measure the packet loss of all r flows. That is, PCOUNT can monitor the loss of all r flows at the cost of monitoring a single flow: the one copy of e_i PCOUNT makes at u and d ensures that packets of any of these r flows are tagged and counted and thus only a single statistic query is needed to retrieve e_i's packet count.

Consider the example in Figure 1.1(a) and suppose that PCOUNT monitors link (g, l). Two multicast flows – f_b for primary tree T_b and f_c for primary tree T_c – traverse (g, l). BASIC creates a separate forwarding rule for f_b and f_c at g while MERGER generates a single forwarding rule at g used by both f_b and f_c. As a result, with MERGER, PCOUNT can track the packet loss of f_b and f_c by querying just the single shared MERGER forwarding rule at g (rather than interact with two separate BASIC forwarding rules). These savings are quantified via simulation in Section 1.5.1.

1.4 Related Work

Related work divides into the following categories: smart grid communication networks (Section 1.4.1), algorithms for detecting packet loss (Section 1.4.2), and link failure recovery algorithms (Section 1.4.3).

1.4.1 Smart Grid Communication Networks

The Gridstat project, started in 1999, was one of the first research projects to consider smart grid communication abstractions.[12] Our work has benefited from their detailed requirements specification [3]. Gridstat proposes a publish-subscribe architecture for PMU data dissemination. By design, subscription criteria are simple in order to ensure fast forwarding of PMU data.

Gridstat is separated into a data plane and a management plane. The management plane keeps track of subscriptions, monitors the quality of service provided by the data plane, and computes paths from subscribers to publishers. To increase reliability, each Gridstat publisher sends data over multiple paths to each subscriber. Each of these paths is a part of a different (edge-disjoint) multicast tree. Meanwhile, the data plane simply forwards data according to the paths and subscription criteria maintained by the management plane.

In North America, all PMU deployments are overseen by the North American SynchroPhasor Initiative (NASPI) [6]. NASPI has proposed and started (as of December 2012) to build the communication network used to deliver PMU data, called NASPInet. The interested reader can consult [6] for more details.

Although Gridstat [3] and NASPI [6] have similarities with Appleseed, these projects have a different focus than ours. Gridstat and NASPI are overlay networks built on top of existing network protocols (e.g., IP, MPLS), while the emphasis of

[12] `http://gridstat.net/`

our work is in making network protocols more robust to handle PMU application requirements.

Hopkinson et al [17] propose a Smart Grid communication architecture that handles heterogeneous traffic: traffic with strict timing requirements (e.g., protection systems), periodic traffic with greater tolerance for delay, and aperiodic traffic. They advocate a multi-tiered data dissemination that uses a technology such as MPLS to make hard bandwidth reservations for critical applications, the use of Gridstat to handle predictable traffic with less strict delivery requirements, and finally the use of Astrolab (which uses a gossip protocol) to manage aperiodic traffic sent over the remaining available bandwidth. They advocate hard bandwidth reservations – modeled as a multi-commodity flow problem – for critical Smart Grid applications.

1.4.2 Detecting Packet Loss

Most previous work for detecting packet loss [1, 8] is based on end-to-end measurements. These approaches require too many measurements (and therefore too much time between when the loss occurs and when it is detected) to accurately measure packet loss in our problem setting. For example, the loss model proposed by Càceres et al. [8] requires approximately 2000 end-to-end probe messages for packet loss estimates to converge on the true underlying packet loss rate. In our problem, where packet loss must be detected at small time scales, these 2000 probe messages would either need to be sent at a high rate to detect packet loss at small time scales (e.g., to detect packet loss at 1 second intervals, probe messages would need to be sent at a rate 30 times higher than PMU sending rates of 60 msgs/sec) [13] or require a prohibitively large window of time if probes were sent at a rate proportional to PMU measurement rates (e.g., over 30 seconds is required to send 2000 probes at a rate of 60 msgs/sec). PCOUNT provides faster and more accurate loss estimates of individ-

[13]The would likely lead to inaccurate results [4].

ual links than these approaches based on end-to-end measurements since it directly measures actual traffic *inside* the network.

Friedl et al. [15] propose a *passive* measurement algorithm that directly measures actual network traffic to determine application-level packet loss rates. Unfortunately, their approach can only measure packet loss after a flow is expired. Since PMU application flows are long lasting (running continuously for days, weeks, and even years), this makes their algorithm unsuitable for our purposes. PCOUNT has no such restriction that packet loss can only be measured over expired flows.

A standard Internet-based approach to passive monitoring of packet loss is to query the native Management Information Base (MIB) counters stored at each router using the Simple Network Management Protocol (SNMP) [4]. This approach is well-suited for course-grained packet loss measurements but not for the fine-grained packet loss detection required by critical PMU applications. Specifically, this approach cannot provide synchronized reads of packet counts across routers/switches, resulting inaccuracies too large for the applications we target.

Existing network protocols, such as BGP, send periodic keep-alive messages between routers to ensure network links are operational. Detecting down links is a different (but complementary) problem than the one we consider, estimating packet loss rates over small time scales.

PCOUNT's approach for ensuring consistent reads of packet counters bears strong resemblance to the idea of *per-packet consistency* introduced by Reitblatt et al. [31]. Per-packet consistency ensures that when a network of switches changes from an old policy to a new one, that each packet is guaranteed to be handled exclusively by one policy, rather than some combination of the two policies. In our case, we use per-packet consistency to ensure that when PCOUNT reads packet counters between an upstream node, u, and downstream node, d, that exactly the same set of packets are considered (excluding, of course, packets that are dropped at u or dropped along the path from u to d.)

1.4.3 Recovery from Link Failures

MPLS is commonly used to extend IP routing with traffic engineering capabilities and fast failure recovery [32]. MPLS pre-computes backup paths for link and router (node) failures and stores these paths at the node immediately upstream from the failed component. This allows for fast, localized recovery: the node detecting a link or node failure immediately reroutes packets along its pre-computed backup path. Unfortunately, MPLS cannot be directly applied to our multicast problem scenario because MPLS addresses unicast communication (a backup unicast path applied to a multicast tree may not result in a valid tree). However, PROACTIVE is in-part inspired by MPLS fast reroute's approach of storing pre-computed backup paths inside the network.

MULTICAST RECYCLING uses different optimization criteria to compute backup trees than prior work [11, 13, 20, 23, 24, 25, 28, 30, 35]. Past approaches use local/myopic 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). [14] In addition, none of these approaches seek to reuse already installed forwarding rules or minimize control signaling, as MULTICAST RECYCLING does. Instead, backup paths or trees are computed with one of the following objective functions: maximize node (link) disjointedness with the primary path [11, 13, 25, 28], minimize bandwidth used [35], minimize backup bandwidth reservations [20, 23, 24], minimize the number of group members that become disconnected after a link failure [30], or minimize path length [34].

[14] Li et al. [24] is an exception; they compute backup paths that aim to minimize the total bandwidth reserved by all backup paths.

Because these backup paths/trees are computed using distributed algorithms, the mechanisms to install these backup trees must navigate an inherent trade-off between high overhead (e.g., message complexity, storing large number of backup paths at routers) and fast recovery (i.e., the time between when the failure is detected and when the multicast tree is repaired should be small) [11]. Algorithms that compute and install backup paths on-demand (after a component failure is detected) scale well since forwarding state for backup paths is only installed after a failure is detected. However, on-demand solutions can be have slow convergence time.

Algorithms that pre-compute and pre-install backup path/trees are fast but scale poorly as significant forwarding state must be stored and maintained at routers. Scalability is particularly challenging because previous work [11, 13, 20, 23, 24, 25, 28, 30, 35] is tailored to support Internet-based applications that typically have a large number of short-lived multicast groups and dynamic group membership.

Our algorithms for installing backup trees avoid the scalability issues addressed by prior work [11, 13, 28]. SDN allows backup trees to be pre-computed offline and stored (in the case of Reactive) at the controller, thus introducing no extra forwarding state at the switches. In the case of Proactive, where backup trees are pre-installed in the network, managing extra forwarding state is tractable because the smart grid is many orders of magnitude smaller than the Internet and smart grid multicast group membership is mostly static [3] (for example, a utility company subscribing to a PMU data stream are likely to always want to receive updates from this PMU). As a result, the volume of pre-installed state is manageable and requires infrequent updates.

In the context of OpenFlow, Kotani et al. [21] propose an approach for fast switching between IP multicast trees, where each multicast group has two multicast trees, a primary tree and a backup tree. Each tree is assigned a unique tree ID and both trees are installed in the network, but only the primary tree is used during normal operation. After a link failure, the root node is signaled to write the backup tree ID in each packet header to force packets to be forwarded using the backup tree. Proactive uses a similar backup tree ID to quickly activate a pre-installed backup tree. However, Proactive takes advantage of common forwarding state between a backup tree and its primary tree to reduce the amount of pre-installed state, while Kotani et al. [21] wastefully pre-installs forwarding rules at each switch in the backup tree.

1.5 Evaluation

We implement each algorithm from Section 1.3 in the POX OpenFlow controller [26] and run simulations using the Mininet 2.0.0 virtualization environment [22]. Simulations 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 [15] virtual machine and is allocated 4GB RAM and a single CPU. All generated virtual networks use Mininet's default software switch, Open vSwitch [16]. Unless otherwise noted, the Appleseed controller algorithm runs inside the VirtualBox VM.

1.5.1 Link Failure Detection Simulations

We run two sets of Mininet-based simulations to evaluate Pcount. First, we measure the accuracy of Pcount loss probability estimates and quantify how accuracy improves as more flows are monitored. Then, we consider how controller and switch processing time increases as Pcount monitors more flows.

Accuracy of Loss Probability Estimates. For the dumbbell topology shown in Figure 1.5, we use Pcount to measure the 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 per second, the standard sampling rate

[15]`https://www.virtualbox.org/`

[16]`http://openvswitch.org/`

of PMUs. Basic is used to implement multicast, resulting in m separate flow table entries at u and d. At the end of the section we comment how our results apply to Merger. We let $m = \{10, 20, 30, 40, 50\}$ [17] and, using Mininet, drop packet traversing (u, d) using a Bernoulli process with loss probability $p = \{.01, .05, .10\}$.

In this simulation, we quantify how the accuracy of Pcount loss estimates – measured relative to the true underlying loss rate, p – as we modify the number of flows Pcount monitors. Recall from Section 1.3.1 that Pcount accounts for every dropped packet of a flow it monitors, meaning that the only error in Pcount estimates results from unmonitored flows. Because the same trends hold across all m and p values, we describe a single representative case, where $m = 10$ and $p = .05$, below.

Figure 1.6(a) compares the 95% confidence intervals of Pcount's link loss probability estimates – centered around the true loss probability (.05) for consistency – as a function of window size $w = \{0.5, 1, ..., 5\}$ seconds. Pcount is configured such that each measurement window starts only after the packet loss from the previous window has been computed. Results are shown where Pcount monitors $k = \{10\%, 40\%, 70\%, 100\%\}$ of (u, d) flows (each monitored flow is selected randomly). The confidence intervals for each w, k pair are computed over 100 simulation runs.

Pcount loss rate estimates are extremely accurate: the 95% confidence interval, across all w and k, lies within 15% of the true loss probability. This is the case even when Pcount's estimate is based on only 30 packets (occurs when $k = 10\%$ and $w = 0.5$). Figure 1.6(b), which plots link loss probability estimates as a function of the number of packets considered during each simulation run, shows that after Pcount considers 75 packets its mean loss probability estimate is within 2% of the

[17]Simulations run prohibitively slow for $m > 50$ due to CPU overload.

true loss probability. [18] As expected, PCOUNT accuracy increases with larger k. For each k, the standard deviation (of PCOUNT loss probability estimates) decreases as a function of the square root of w.

Processing Time. Next, we quantify how PCOUNT processing time increases when PCOUNT monitors additional flows. We measures packet loss over (u, d) from Figure 1.5. Processing time is measured as the time between when PCOUNT sends its first statistic query and when PCOUNT computes its packet loss estimate. Recall from Section 1.3.1 that if PCOUNT monitors packet loss of k flows traversing (u, d), PCOUNT sends k statistic queries to u and one aggregate query to d.

PCOUNT is configured such that each measurement window starts only after the packet loss from the previous window has been computed. Additionally, PCOUNT window size is fixed to 2 and the loss threshold is set to 0. Because Mininet multiplexes CPU resources using the default Linux scheduler, we found that running the constant rate PMU flows introduces unwanted CPU contention, adding noise to our results. For this reason, we create only a single multicast group (with source h_1 and a single sink s_1) but do not actually send any packets between the two hosts. To further reduce CPU contention, we run PCOUNT as a remote control application, outside of the VirtualBox VM.

As computed, processing time accounts for (a) the time at the controller to generate $k+1$ statistic queries, (b) the transmission delay associated with sending the $k+1$ statistic queries from the controller to u and d, (c) the network delay in sending each statistic query from the controller to switch, (d) total time to process the statistic query at u and d, (e) the delay in sending the $k + 1$ query results from switch to controller, and (f) the latency in receiving and recording statistic query replies at the controller. We subtract (c) and (e), the network delay between controller and

[18]This accuracy is not surprising since loss probability estimate is averaged over 100 simulation runs.

switches, from the measured processing times. Because the combined delay of (a), (b), and (f) accounts for less than 1% of the overall processing time, part (d), the time to process statistic queries at u and d, determines the overall processing times.

Figure 1.6(c) shows the processing time, computed as described in the previous paragraph, as a function of the number of flows PCOUNT monitors, k. [19] Each data point is the mean computed over 50 simulation runs. To measure the effect of flow table size on query processing time, we install r additional flow table entries at u and d.

We find that processing time increases quadratically with k and there is a significant gap in processing time between each r. In practice, we expect non-empty flow tables so the $r = 0$ curve is overly optimistic. Therefore, to reasonably achieve sub-second processing time, our results show that fewer than 75 flows can be monitored.

Because the switches are completely idle during each simulation run, except for the time to process the read state queries, and the software switches used have considerably more powerful CPUs relative to hardware switches [12, 33], these results likely underestimate processing time. Nonetheless, these results underscore the high cost in monitoring a large number of flows.

Summary. The slow processing times associated with monitoring large numbers of flows and the highly accurate loss estimates for even small k strongly suggest that k should be small. Because the software switch skews the processing time results in favor of PCOUNT, we expect that even a stronger case for using small k can be made using hardware switches. However, we caution that the (Bernoulli) loss process is biased in favor of PCOUNT because we found loss rates to be nearly uniform across all flows traversing (u, d).

[19]Fake multicast groups and corresponding flow table entries are generated and installed at u and d in cases where PCOUNT monitors more than the 1 multicast group.

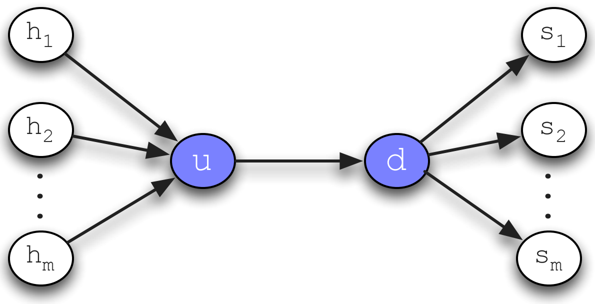


Figure 1.5. Dumbbell topology used in the Pcount evaluation.

1.5.2 Backup Tree Installation Simulations

In this section, we simulate the failure of a single link and then measure recovery time, control plane signaling, and garbage collection overhead for Proactive and Reactive both with and without the Merger optimization. We aim to answer the following questions with these simulations:

- How effective is Bunchy in reusing primary tree edges and in providing opportunities for backup tree installation algorithms to reuse primary tree forwarding rules?
- How much faster does Proactive recover from link failure than Reactive?
- How many fewer control messages are needed to install backup trees under Proactive versus Reactive?
- How much control state does Proactive pre-install?
- In terms of recovery time, control plane signaling, and garbage collection, how much does Merger improve performance relative to Basic?

Setup. We use IEEE bus systems 14, 30, 57, and 118 [20] and synthetic graphs based on these IEEE bus systems to evaluate our algorithms. Each bus system con-

[20]`http://www.ee.washington.edu/research/pstca/`.

sists of buses – electric substations, power generation centers, or aggregation points of electrical loads – and transmission lines connecting those buses. The IEEE bus systems are actual portions of the North American transmission network, where PMUs are being deployed. Note that the bus system number indicates the number of buses (nodes) in the graph (e.g., bus system 57 has 57 nodes). Synthetic graphs are generated using a procedure described in one of our previous papers [16] 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. It is assumed that an OpenFlow switch is co-located 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. Additionally, we connect each switch with a leaf host using a bidirectional communication link. 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, and lastly the first-hop switch multicasts the PMU measurements using the network of OpenFlow switches to a set subscribing hosts (terminals).

For each bus system n, we generate synthetic topologies with n switches, n hosts, and set all link weights to 1. Then, we randomly create $m = \{1, 2, ..., \frac{n}{2}\}$ multicast groups, each with $n/3$ random terminal hosts, and use the Steiner-Arborescence approximation proposed by Charikar et al. [9] to compute the m primary trees. Bunchy, with $\alpha = 1.1$, is then used to pre-compute, for each primary tree, a backup tree for each primary tree link. Next, a random communication link, l, that is used by at least one primary tree is chosen to fail (i.e., drop enough packets to trigger a Pcount alert), triggering the installation of backup trees using either Reactive or Proactive. For each m, we generate 35 different synthetic graphs and 3 random sets of multicast groups, yielding a total of 105 simulation runs per m.

The results described in the remainder of this section are those from synthetic topologies generated using IEEE bus system 57 as a template. The trends are consistent across all other networks. Switches in the networks generated using IEEE bus system 57 have an average diameter of 11.75 and average degree 3.74. For each of the m multicast groups, we initially attempted to multicast packets at a constant rate flow of 60 packets per second from the root host but this caused CPU overload. Instead, in each simulation run we only initiated the constant rate flows for the primary trees using the failed link.

1.5.2.1 Bunchy Results

The primary trees computed using the Steiner-Arborescence approximation described in Section 1.3.2.2 have an average root-to-terminal hop count of 7.54, while the Bunchy backup trees are slightly larger with an average end-to-end (E2E) length of 8.4. Based on the E2E latency requirements reported in Section 1.2.1, the per-link delay in our simulated topologies would need to be in the range of $0.6 - 1$ms to satisfy QoS requirements using these multicast trees. [21]

On average, the Bunchy backup trees have stretch of 1.17. Stretch is defined per multicast tree and is the ratio of path length from the root to terminal along the multicast tree to the length of the direct unicast path. For comparison with Bunchy results, we compute a second backup tree for each primary tree, T_i^l, by running the Steiner-Arborescence approximation described in Section 1.3.2.2 over the original graph with l removed. We denote the set backup trees for l computed using this algorithm as B^l.

[21] The average root-to-terminal path lengths were largest for IEEE bus system 57 and the synthetic graphs based on this bus system. The multicast trees computed for bus system 118 have an average E2E path length approximately 1 hop fewer than those for bus system 57, even though IEEE bus system 118 has more than twice as many nodes as bus system 57. This is mainly due to bus system 118's higher density (than bus system 57).

The B^l backup trees are marginally smaller than the $\hat{T}^l$ backup trees: $w(\hat{T}^l)/w(B^l) = 1.08$. This is expected because $\hat{T}^l$ is computed using a heuristic to guide Bunchy to reuse primary tree edges, while B^l trees are an approximation of the least cost directed tree (and so are computed independent of the edges used by the primary tree).

However, $\hat{T}^l$ reuses more primary tree edges ($\hat{T}^l$ reuses 59% of primary tree edges versus 41% under B^l). Most importantly, when comparing $\hat{T}^l$ and B^l with the primary tree, $\hat{T}^l$ has more common nodes with the primary tree that have the same children in $\hat{T}^l$ and the primary tree (55% of $\hat{T}^l$ nodes) than B^l (38% of B^l nodes). This last point is important because this allows $\hat{T}^l$ to reuse more primary tree rules once $\hat{T}^l$ is installed. In summary, these results suggest that Bunchy computes backup trees only slightly larger than an approximation of the least cost tree with a significant gain in primary tree edge and forwarding rule reuse.

1.5.2.2 Signaling Overhead

Next, 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 1.7(a) shows the results for Reactive running in Basic and Merger mode, referred to as Reactive+Basic and Reactive+Merger, respectively; a lower bound for Reactive (Reactive+LB); and Proactive in Basic mode, denoted as Proactive+Basic. Note that the results for Proactive are the same using Basic and Merger because Proactive requires that only the root node of each backup tree needs to be signaled to activate the backup. We shall later see how Basic and Merger affect the number of forwarding rules Proactive pre-installs.

Reactive+LB computes the lower bound at each switch, v, used by a backup tree for the failed link (l) and returns the sum of each node's lower bound. At v, the lower bound computation first finds the set of outports used by all primary and backup trees and then eliminates any backup tree with the same outports as a primary tree (these backup trees can reuse the primary tree forwarding rule rather than installing a new one). Among the remaining backup trees, the number of unique sets of outports, b, is equal to the minimum number of forwarding rules that must be installed at v: if fewer than b rules are installed at v, packets corresponding to at least one multicast group would not match with a rule that forwards packets out the correct set of ports.

As expected, we find that PROACTIVE requires less signaling overhead than REACTIVE, including even REACTIVE+LB. PROACTIVE activates the backup trees by sending a control message (to install a pre-computed forwarding rule) to the root switch of each of backup tree using the failed link, whereas REACTIVE must signal multiple switches to install each backup tree.

For REACTIVE, the gap between BASIC and MERGER increases as we introduce more primary trees. When $m = 1$ there are no opportunities for MERGER to consolidate forwarding rules so MERGER and BASIC require exactly the same number of control messages to install the backup tree.

As m grows, three factors contribute to an increasing gap between BASIC and MERGER. One, there are more primary tree forwarding rules (installed in the network) that MERGER can reuse. Our results show that for $m \geq 7$, 75% of MERGER savings (versus BASIC) are due to reusing primary tree forwarding rules. Two, as m increases more graph edges are used: when $m = 28$, 90% of all network links are used by at least one primary tree and is at least 80% for $m \geq 10$. This benefits MERGER because it increases the likelihood that at any switch a backup tree shares the same outgoing links as at least one primary tree. Three, as m increases more primary trees are affected by a link failure causing more backup trees to be installed for each link failure. This provides additional opportunities for MERGER to consolidate flow table entries with other backup trees. However, this third factor is less significant than the previous two, as only 25% of Merger savings are due to consolidating flows with other backup trees.

With Reactive, Merger does well compared with LB. On average, Merger requires 25% more control messages than LB, suggesting that Merger's local optimization does not miss many opportunities for consolidating flows.

1.5.2.3 Time to Install Backup Trees

Here we compare the time required by each of algorithms to install backup trees. Specifically, we measure the time between when the link failure is detected at the controller to when *all* pre-computed backup tree forwarding rules are installed at the network switches. We refer to this time duration as t_c. t_c is a function of the controller transmission delay (i.e., the time between when the first and last precomputed control messages are sent from the controller), the controller to switch RTT, and the time to install a forwarding rule at a switch.

We find the transmission delay to be negligible: on average, transmission delay is less than 2.8% and 0.9% of the time to install a *single* flow table entry at a switch, for Reactive+Basic and Reactive+Merger, respectively. Even if we conservatively assume that the inter-arrival time of installation messages at each switch is equal to the total transmission delay, it follows that each switch receives all control messages before completing the installation of its first backup tree rule. Because rules are installed in parallel across switches, the time to install all backup tree rules occurs when the switch with the most backup tree rules to install, s_x, installs its last rule. If we assume s_x has c_x rules to install, let t_d be the total transmission delay, and denote the average latency to install a single rule at an OpenFlow switch as t_i, then

$$t_c = \frac{1}{2}RTT + t_d + c_x(t_i)$$

Because Mininet lacks performance fidelity, we determine t_i values using measurement results from the literature [14], rather than measure rule installation times in Mininet. Specifically, we assume the mean installation time per rule is 7.12ms, as reported by Ferguson et al. [14] using the Pronto 3290 OpenFlow switch running the Indigo 2012.09.07 firmware.

Figure 1.7(b) shows the estimated elapsed time to install all backup trees as a function of m. We set $RTT = 0$ in this simulation. The trends for each algorithm are a function of their c_x values, the maximum number of rules any switch must install, found at each m. With PROACTIVE, c_x is always 1. The difference in total installation time between REACTIVE+BASIC and REACTIVE+MERGER is small in absolute terms (at most 25ms) because the install times depend on the amount of rule consolidation each algorithm is able to apply at a single switch, s_x, rather than the level of rule sharing possible at multiple switches (as we observed with signaling overhead results). Nonetheless, the extra milliseconds saved using MERGER can be valuable to critical PMU applications.

1.5.2.4 Switch Flow Table Size

In our Section 1.5.2.2 simulations we found that PROACTIVE incurs less signaling overhead than REACTIVE. Here we show that these savings come at a cost: PROACTIVE's pre-installed forwarding rules can account for a significant portion of limited OpenFlow switch capacity.

Using the same setup as the other simulations in this section, we record the number of pre-installed backup tree rules at each switch during each simulation run. Figure 1.7(c) shows the mean number of pre-installed rules per switch, r, as a function of m. The confidence intervals are omitted because of high variance (during each simulation run, individual counts range from 0 to 525.) REACTIVE is not included because it does not pre-install forwarding rules. The number of pre-installed rules for

Proactive+LB is computed using the same algorithm described in Section 1.5.2.2 for Reactive+LB.

Similar to our Reactive signaling overhead findings, Merger yields up to 2.5 times better performance than Basic. Merger and LB savings increase with m because more primary tree flows are reused and more backup tree rules are shared as m grows. As a result, the number of pre-installed forwarding rules per backup tree decreases linearly as m increases causing the rate at which r increases for Proactive+Merger to slow.

In contrast, r increases linearly with m using Proactive+Basic. For each backup tree, Basic is only able to avoid pre-installing a forwarding rule at a switch, v, if the backup tree uses the same outports as its primary tree at v. Because this condition depends only on the relationship between a backup tree and its primary tree, the number of pre-installed rules per backup tree is constant for all m. Since larger m implies more backup trees, r increases linearly with m.

Based on maximum flow table sizes of real OpenFlow switches (ranging from approximately 1500 to 1900 flow table entries [12, 14]), at most r occupies 19% and 6.7% of flow table space for Proactive+Basic and Proactive+Merger, respectively. We find that the switch with the most pre-installed forwarding rules (across all simulation runs) has at most has 525 pre-installed forwarding rules (occurs with Proactive+Basic when $m = 28$). At most, this accounts for 35% of the switch's flow table capacity.

1.5.2.5 Garbage Collection Overhead

After a link l fails, forwarding rules corresponding to primary trees using l may become stale (detailed in Section 1.3.4). These stale rules are deleted as part of Appleseed garbage collection. In this section, we first we describe garbage collection results for Reactive and then for Proactive using the same simulation setup described at the start of Section 1.5.2. Reactive+LB and Proactive+LB are both computed based on the LB algorithm described in Section 1.5.2.2.

Reactive Garbage Collection. Figure 1.7(d) shows a modest delta in the number of stale rules between Basic, Merger, and LB. For each of these algorithms, on average 55% of primary tree rules are reused by its backup tree (and are therefore not garbage collected). [22] This implies that Reactive+Merger and Reactive+LB are only able to reduce garbage collection over the remaining 45% of primary tree rules. Among these remaining primary tree rules, Reactive+Merger and Reactive+LB reduce garbage collection when any backup tree for l reuses a primary tree rule. Our results show that this yields small savings in garbage collection.

Proactive Garbage Collection. Compared with Reactive, we find a significant decrease in stale flows with Proactive because Proactive installs up to two orders of magnitude more backup trees, providing more opportunities for primary tree forwarding rules to be reused. Recall that Reactive only installs the backup trees for l, whereas Proactive pre-installs, for each primary tree, a backup tree for each primary tree link, amounting to approximately 32 backup trees per primary tree. As a result, we observe 2.5 times fewer stale rules with Proactive+Basic versus Reactive+Basic and Proactive+Merger has up to an order of magnitude decrease in garbage collection versus Reactive+Merger.

The number of stale Proactive+Merger forwarding rules actually decreases as m grows. Recall that for a primary tree using l, any of its rules is not stale if at least one other primary or backup tree uses this flow table entry. Because the number of pre-installed backup trees is so large (approximately $32m$), for large m, nearly all primary tree rules are still used after a link failure (resulting in a decrease in stale flow table entries as m grows).

[22] Because all three algorithms use Bunchy to compute backup trees this statistic is the same for Reactive+Basic, Reactive+Merger, and Reactive+LB.

1.5.2.6 Summary

In summary, we found that as more primary trees are installed in the network, the gap between Merger and Basic grows (for both Reactive and Proactive) in terms of signaling overhead, total backup tree installation time, number of pre-installed forwarding rules, and garbage collection overhead. The is the case because, with larger m, there are more opportunities for Merger to reuse primary tree rules and consolidate rules among other backup trees.

Additionally, we found Proactive yields fewer control messages and faster recovery than Reactive – Reactive sends up to 10 times more control messages than Proactive – but at the cost of storage overhead at each switch. Proactive's pre-installed backup trees can account for as much as 35% of the capacity reserved for wild-card matching rules of a conventional OpenFlow switches [12]. However, when applying Merger to Proactive, this statistic drops to 20% and, on average, Proactive+Merger accounts for only 6.7% of flow table capacity.

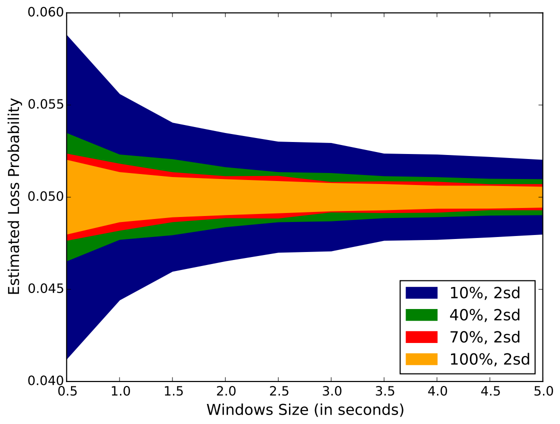
1.6 Conclusions

We addressed reliable multicasting of critical Smart Grid data. Towards this goal, 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 to execute these algorithms in the data plane.

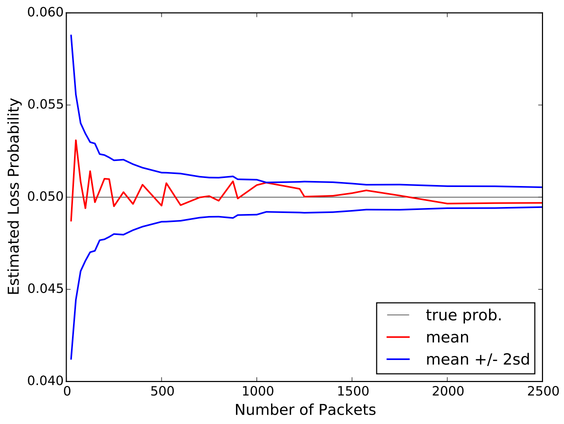
First, we presented Pcount, an algorithm that used OpenFlow primitives to accurately detect per-link packet loss inside the network rather than using slower E2E measurements. Next, we formulated a new problem, Multicast Recycling, that considered computing backup trees that reuse edges of already installed multicast trees as a means to reduce control plane signaling. Multicast Recycling was proved to be at least NP-hard so we designed an approximation algorithm called

Bunchy. Lastly, we presented two algorithms, Proactive and Reactive, that installed backup trees at OpenFlow controlled switches. As an optimization to Proactive and Reactive, we introduced Merger, an algorithm that consolidated forwarding rules at switches where multiple trees had common children.

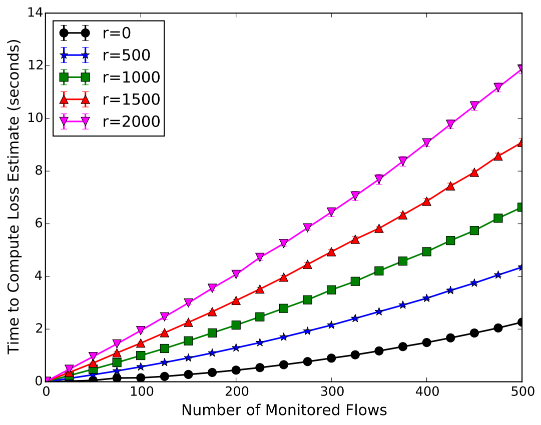
Mininet simulations were used to evaluate our algorithms over communication networks that mirrored the structure of IEEE bus systems (actual portions of the North American power grid). We found Pcount estimates were accurate when monitoring even a small number of flows over short time window: after sampling only 75 packets, the 95% confidence interval of Pcount loss estimates were within 15% of the true loss probability. By pre-installing backup trees, Proactive resulted in up to a ten-fold decrease in control messages compared with Reactive, which had to signal multiple switches to install each backup tree. However, in scenarios with many multicast groups, Proactive's pre-installed forwarding rules accounted for a significant portion of scarce OpenFlow switch table capacity (up to 35% of a standard OpenFlow switch). Fortunately, Merger reduced the amount of pre-installed forwarding state by a factor of $2 - 2.5$, to acceptable levels.



(a) Loss estimates as function of PCOUNT window size.

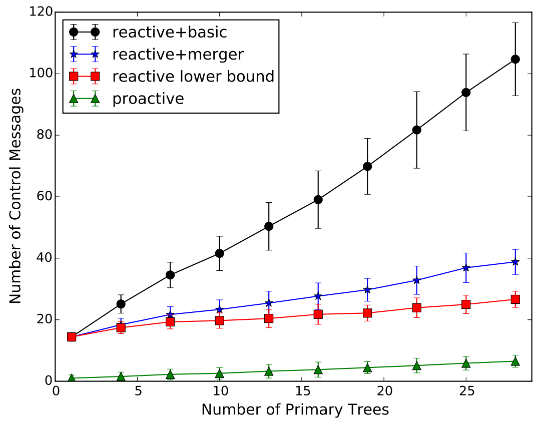


(b) Loss estimates as function of number of packets.

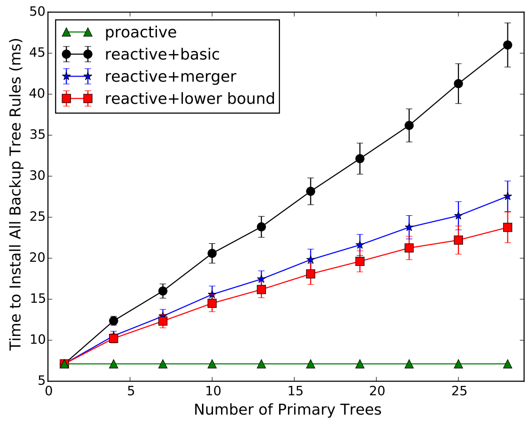


(c) Processing time, with 95% confidence intervals, as a function of number of monitored flows.

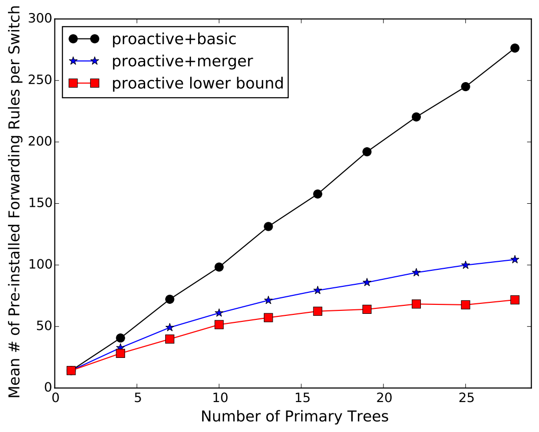
Figure 1.6. PCOUNT results monitoring a single link, (u, d), from Figure 1.5.



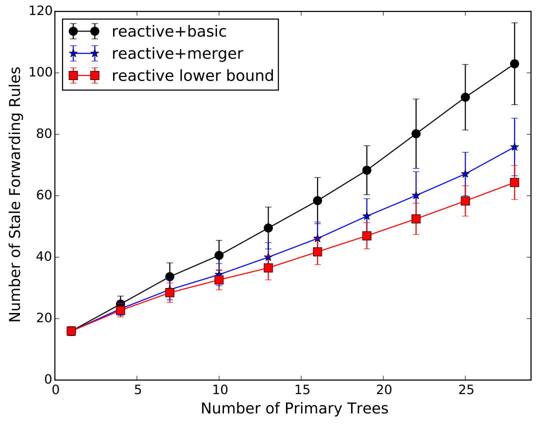
(a) Number of control messages to activate backup trees.



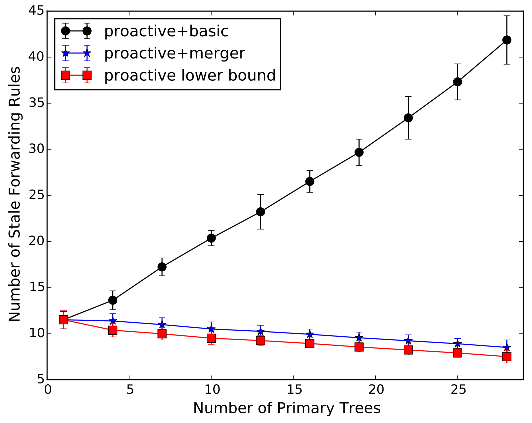
(b) Total time to install all backup tree flow table entries.



(c) PROACTIVE: mean number of pre-installed rules per switch.



(d) REACTIVE: number of stale flow table entries resulting from link failure.



(e) PROACTIVE: number of stale flow table entries resulting from link failure.

Figure 1.7. REACTIVE and PROACTIVE results for a single random link failure of synthetic topologies based on IEEE bus system 57. Each data point is the mean over 105 simulation runs and the 95% confidence interval is shown in all plots expect (c).

BIBLIOGRAPHY

[1] Almes, G., Kalidindi, S., and Zekauskas, M. A one-way packet loss metric for ippm. Tech. rep., RFC 2680, September, 1999.

[2] Andersson, G, Donalek, P, Farmer, R, Hatziargyriou, N, Kamwa, I, Kundur, P, Martins, N, Paserba, J, Pourbeik, P, Sanchez-Gasca, J, et al. Causes of the 2003 major grid blackouts in north america and europe, and recommended means to improve system dynamic performance. *Power Systems, IEEE Transactions on 20*, 4 (2005), 1922–1928.

[3] Bakken, D.E., Bose, A., Hauser, C.H., Whitehead, D.E., and Zweigle, G.C. Smart generation and transmission with coherent, real-time data. *Proceedings of the IEEE 99*, 6 (2011), 928–951.

[4] Barford, P., and Sommers, J. Comparing probe-and router-based packet-loss measurement. *Internet Computing, IEEE 8*, 5 (2004), 50–56.

[5] Birman, K.P., Chen, J., Hopkinson, E.M., Thomas, R.J., Thorp, J.S., Van Renesse, R., and Vogels, W. Overcoming communications challenges in software for monitoring and controlling power systems. *Proceedings of the IEEE 93*, 5 (2005), 1028–1041.

[6] Bobba, R., Heine, E., Khurana, H., and Yardley, T. Exploring a tiered architecture for NASPInet. In *Innovative Smart Grid Technologies (ISGT), 2010* (2010), IEEE, pp. 1–8.

[7] Bu, T., Duffield, N.G., Presti, F., and Towsley, D.F. Network tomography on general topologies. In *ACM SIGMETRICS Performance Evaluation Review* (2002), vol. 30, ACM, pp. 21–30.

[8] Cáceres, R., Duffield, N.G., Horowitz, J., and Towsley, D.F. Multicast-based inference of network-internal loss characteristics. *Information Theory, IEEE Transactions on 45*, 7 (1999), 2462–2480.

[9] Charikar, M., Chekuri, C., Cheung, T., Dai, Z., Goel, A., Guha, S., and Li, M. Approximation algorithms for directed steiner problems. In *Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms* (1998), Society for Industrial and Applied Mathematics, pp. 192–200.

[10] Clark, D. The design philosophy of the DARPA internet protocols. *ACM SIGCOMM Computer Communication Review 18*, 4 (1988), 106–114.

[11] Cui, J.H., Faloutsos, M., and Gerla, M. An architecture for scalable, efficient, and fast fault-tolerant multicast provisioning. *Network, IEEE 18*, 2 (2004), 26–34.

[12] Curtis, A., Mogul, J., Tourrilhes, J., Yalagandula, P., Sharma, Puneet, and Banerjee, S. Devoflow: Scaling flow management for high-performance networks. In *ACM SIGCOMM Computer Communication Review* (2011), vol. 41, ACM, pp. 254–265.

[13] Fei, A., Cui, J., Gerla, M., and Cavendish, D. A "dual-tree" scheme for fault-tolerant multicast. In *Communications, 2001. ICC 2001. IEEE International Conference on* (2001), vol. 3, IEEE, pp. 690–694.

[14] Ferguson, Andrew D., Guha, Arjun, Liang, Chen, Fonseca, Rodrigo, and Krishnamurthi, Shriram. Participatory Networking: An API for application control of SDNs. In *ACM SIGCOMM* (2013).

[15] Friedl, A., Ubik, S., Kapravelos, A., Polychronakis, M., and Markatos, E. Realistic passive packet loss measurement for high-speed networks. *Traffic Monitoring and Analysis* (2009), 1–7.

[16] Gyllstrom, D., Rosensweig, E., and Kurose, J. On the impact of pmu placement on observability and cross-validation. In *Proceedings of the 3rd International Conference on Future Energy Systems: Where Energy, Computing and Communication Meet* (2012), ACM, p. 20.

[17] Hopkinson, K., Roberts, G., Wang, X., and Thorp, J. Quality-of-service considerations in utility communication networks. *Power Delivery, IEEE Transactions on 24*, 3 (2009), 1465–1474.

[18] Ji, P., Ge, Z., Kurose, J., and Towsley, D. A comparison of hard-state and soft-state signaling protocols. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications* (2003), ACM, pp. 251–262.

[19] Johnson, Anthony, Tucker, Robert, Tran, Thuan, Paserba, John, Sullivan, Dan, Anderson, Chris, and Whitehead, Dave. Static var compensation controlled via synchrophasors. In *proceedings of the 34th Annual Western Protective Relay Conference, Spokane, WA* (2007).

[20] Kodialam, M., and Lakshman, T. Dynamic routing of bandwidth guaranteed multicasts with failure backup. In *Network Protocols, 2002. Proceedings. 10th IEEE International Conference on* (2002), IEEE, pp. 259–268.

[21] Kotani, D., Suzuki, K., and Shimonishi, H. A design and implementation of openflow controller handling ip multicast with fast tree switching. In *Applications and the Internet (SAINT), 2012 IEEE/IPSJ 12th International Symposium on* (2012), IEEE, pp. 60–67.

[22] Lantz, B., Heller, B., and McKeown, N. A network in a laptop: Rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks* (New York, NY, USA, 2010), Hotnets-IX, ACM, pp. 19:1–19:6.

[23] Lau, W., Jha, S., and Banerjee, S. Efficient bandwidth guaranteed restoration algorithms for multicast connections. *NETWORKING 2005. Networking Technologies, Services, and Protocols; Performance of Computer and Communication Networks; Mobile and Wireless Communications Systems* (2005), 237–243.

[24] Li, G., Wang, D., and Doverspike, R. Efficient distributed mpls p2mp fast reroute. In *Proc. of IEEE INFOCOM* (2006).

[25] Luebben, R., Li, G., Wang, D., Doverspike, R., and Fu, X. Fast rerouting for ip multicast in managed iptv networks. In *Quality of Service, 2009. IWQoS. 17th International Workshop on* (2009), IEEE, pp. 1–5.

[26] Mccauley, J. POX: A Python-based Openflow Controller. `http://www.noxrepo.org/pox/about-pox/`.

[27] McKeown, Nick, Anderson, Tom, Balakrishnan, Hari, Parulkar, Guru M., Peterson, Larry L., Rexford, Jennifer, Shenker, Scott, and Turner, Jonathan S. Openflow: enabling innovation in campus networks. *Computer Communication Review 38*, 2 (2008), 69–74.

[28] Médard, M., Finn, S.G., Barry, R.A., and Gallager, R.G. Redundant trees for preplanned recovery in arbitrary vertex-redundant or edge-redundant graphs. *IEEE/ACM Transactions on Networking (TON) 7*, 5 (1999), 641–652.

[29] Pfaff, B., et al. Openflow switch specification version 1.1.0 implemented (wire protocol 0x02), 2011.

[30] Pointurier, Y. Link failure recovery for mpls networks with multicasting. Master's thesis, University of Virginia, 2002.

[31] Reitblatt, M., Foster, N., Rexford, J., and Walker, D. Consistent updates for software-defined networks: Change you can believe in! In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks* (2011), ACM, p. 7.

[32] Rosen, E., Viswanathan, A., Callon, R., et al. Multiprotocol label switching architecture.

[33] Rotsos, C., Sarrar, N., Uhlig, S., Sherwood, R., and Moore, A. OFLOPS: An Open Framework for Openflow Switch Evaluation. In *Passive and Active Measurement* (2012), Springer, pp. 85–95.

[34] Tian, A.J., and Shen, N. Fast reroute using alternative shortest paths. draft-tian-frr-alt-shortest-path-01.txt, July 2004.

[35] Wu, C.S., Lee, S.W., and Hou, Y.T. Backup vp preplanning strategies for survivable multicast atm networks. In *Communications, 1997. ICC 97 Montreal,'Towards the Knowledge Millennium'. 1997 IEEE International Conference on* (1997), vol. 1, IEEE, pp. 267–271.

[36] Yardley, J., and Harris, G. 2nd day of power failures cripples wide swath of india, July 31, 2012. `http://www.nytimes.com/2012/08/01/world/asia/power-outages-hit-600-million-in-india.html?pagewanted=all&_r=1&`.