FAULT INJECTIONS ON MISSION-CRITICAL COMPUTER SYSTEMS

BY

LAVIN R. DEVNANI

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

Advisers:

Professor Ravishankar K. Iyer
Professor Zbigniew T. Kalbarczyk

ABSTRACT

This thesis presents two unique sets of fault injections on mission-critical computer systems with the goal of (1) understanding the impact of faults, errors and failures, and (2) evaluating fault-tolerance and resilience of the targeted systems in the presence of failures.

Our first fault injection campaign studies the effects of failures on high-performance computing (HPC) systems. We target the Cray XE Blue Waters JYC testbed at the National Center for Supercomputing Applications, with the goal of improving the understanding of failure causes and propagation observed in the field failure data analysis of Blue Waters. We use data collected from system logs and network performance counters to (1) characterize fault-error-failure sequences and recovery mechanisms in Gemini interconnection networks and in Cray compute elements, (2) understand the impact of failures on the system and user applications at different scales, and (3) identify and recreate fault scenarios that induce unrecoverable failures, to create new tests for system and application design. We utilize *HPCArrow*, a newly developed software-implemented fault injection tool with the ability to disable and restore user-specified network links, directional connections, compute nodes and blades. We observe failures manifesting in the form of applications not making forward progress and network quiescence operations causing extended system recovery times.

Our second fault injection campaign studies the effects of faults, attacks and failures on a smart power grid utilizing software-defined networking (SDN) to orchestrate its data acquisition network. We evaluate our fault models on a smart power grid simulation running Raincoat, an SDN application that reroutes and spoofs network traffic to thwart attackers. Additionally, we propose an application- and data plane-based solution to pro-actively monitor system state and enforce user defined policies. We show that under certain faults, (1) applications orchestrating the network become ineffective,

and (2) periodically monitoring the state of the network can identify faults or attacks before they manifest as failures. The results obtained from this work can aid in enhancing the resiliency of future SDN applications.

To my parents, for their love and support.

ACKNOWLEDGMENTS

I would like to thank my research advisers Ravi Iyer and Zbigniew Kalbarczyk for their encouragement and support during my years as a graduate student. I would also like to thank my colleagues in the DEPEND research group, particularly Ching Yang Tan, Sharon Tang, Fei Deng, Hui Lin, and Saurabh Jha for their help and encouragement with my research projects.

Additionally, I would like to thank Mike Showerman, Greg Bauer, Bill Kramer (NCSA), Jim Brandt, and Ann Gentile (SNL) for their constant and prompt help with my Blue Waters fault injection work.

Material included in this thesis is partially based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under Award Number 2015-02674. This work is partially supported by NSF CNS 13-14891. This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (awards OCI-0725070 and ACI-1238993) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications.

Material included in this thesis is partially based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under Award Number DOE DE-OE 0000780.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AMR	Adaptive Mesh Refinement
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
AWP	Anelastic Wave Propagation
BC	Blade Controller
BTE	Block Transfer Engine
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
FFT	Fast Fourier Transform
FI	Fault Injection
FMA	Fast Memory Access
GNI	Generic Network Interface
HPC	High-Performance Computing
HSN	High-Speed Network
HSS	Health Supervisory System
IP	Internet Protocol
LDMS	Lightweight Distributed Metric Service
MCE	Memory Check Error
MILC	MIMD Lattice Computation
MIMD	Multiple Instruction, Multiple Data

MPI	Message Passing Interface
NID	Node IDentifier
NMI	Non-Maskable Interrupt
ORB	Output Request Buffer
OVS	OpenVSwitch
PE	Processing Element
PGAS	Partitioned Global Address Space
PGI	Portland Group, Inc.
PSDNS	Pseudo-Spectral Direct Numerical Simulations
RDMA	Remote Direct Memory Access
RMA	Remote Memory Access
RMT	Receive Message Table
SCADA	Supervisory Control and Data Acquisition
SDN	Software-Defined Networking
SEC-DED	Single Error Correction-Double Error Detection
SMP	Symmetric MultiProcessing
SMSG	Short MeSsaGe
SMW	System Management Workstation
SSID	Synchronization Sequence IDentifier
SWIFI	SoftWare-Implemented Fault Injection
TCP	Transmission Control Protocol
UPC	Unified Parallel C
WAN	Wide Area Network

CHAPTER 1

INTRODUCTION

Large-scale computing clusters, high-performance computer systems, and software-defined flexible and programmable networks are an important computing enterprise in a wide variety of application domains ranging from artificial intelligence and machine learning to traditional high-performance computing applications such as weather forecasting and molecular dynamics. Partial or complete failures in any component of these systems in mission-critical environments can have significant social and societal implications. An increasing reliance on highly critical computer systems underscores the need for robust and effective techniques to evaluate their resiliency and mitigate attacks and failures. A broad approach for assessing the resilience of large-scale systems is fault injection [1].

In this thesis, we present two unique sets of fault injections on mission-critical computer systems: (1) the Blue Waters JYC testbed, a Cray XE high-performance computing (HPC) system, and (2) a software-defined networking (SDN) enabled smart power grid. We summarize our fault injection campaigns in Table 1.1. While our underlying approach to evaluating the resiliency of our target systems is the same, we develop separate injection tools and analysis methods which enable us to adapt to specific intricacies presented by the different systems.

Table 1.1: Summary of Fault Injection Campaigns

System	Area	**Target Components**	**Failure Scenarios**
Blue Waters JYC testbed	High-performance computing (HPC)	Network components: Links, Directional connections Compute components: Nodes, Blades	Premature termination and hangs of application workloads
SDN-enabled power grid simulation	Software-defined networking (SDN)	SDN network switches: OpenFlow table entries, Packet processing pipelines, PacketIn flooding attacks	SCADA Timeouts, Measurement Obfuscation Failures, Denial of service

1.1 Fault Injection

Fault injection (FI) is a reliability evaluation technique used to study system behaviors by deliberately and systematically introducing faults into various levels or components of a target system. Deliberately introducing faults allows system designers to (1) evaluate the correctness of fault-tolerance mechanisms employed by the system, (2) understand fault-error-failure propagation paths, and (3) assess system vulnerability to resulting failure scenarios.

Fault injection methods have been widely used to investigate fault-to-failure propagation and to quantify the impact of failures on applications and systems. As the complexity of newer systems increases, more rigorous and formal techniques like model checking, state-space searching and theorem proving become infeasible due to the immense effort required. Fault injection methods remain feasible as they allow researchers to control fault conditions, workload executions and instrumentation on target systems. Many past works have been successful in evaluating the resiliency of complex systems, such as computer processors [2], [3], software programs [4], operating systems [5], dynamic memory [6], stream processing [7], genomic sequencing [8], and surgical robots [9].

1.2 Challenges in Assessing System Resilience

A number of challenges exist when studying the resiliency of large and complex systems. System modeling approaches, while useful in the early design stage, often fall short of providing the ability to perform in-depth analysis of a system or its interacting components. While analysis of field data is usually the preferred approach, researchers often fail in connecting failure events with their root causes or precursor faults. Additionally, existing logging mechanisms may not account for all influencing factors and may not provide a complete view of the system being studied. Furthermore, as the complexity of newer systems increases, more rigorous and formal techniques like model checking, state-space searching and theorem proving require an increasing amount of effort, both computationally and on part of researchers.

Fault injection methods provide researchers the ability to study the system under isolated faults, with complete control over experiment parameters, tar-

get components, workloads, and type of injected faults. This degree of control also allows researchers to verify deterministic system behavior under faults and failures, with the ability to repeat experiments and reproduce scenarios observed in production systems. However, fault injection also requires direct access to target systems in isolation, which may be prohibitive for those in mission-critical spaces. Additionally, poorly designed fault injection campaigns with low system coverage may not uncover all possible failure scenarios. Nonetheless, fault injection methods remain effective as they offer researchers the unique ability to study the impact of faults, errors and failures under controlled conditions on the target system and connect observed events to root causes.

1.3 Contributions

The work presented in this thesis focuses on utilizing fault injection methods to improve our understanding of the impact of faults, errors and failures on (1) high-performance computers and (2) software-defined networking enabled power grids. The key research contributions of this thesis are:

- A hierarchical understanding of systems targeted in this work, in order to guide component selection during fault injection experiment design and preparation.
- Newly developed fault injection tools and approaches, to enable execution of fault injection experiments. We present toolkits for fault injections on HPC systems (*HPCArrow*) and SDN-enabled power grids.
- Analyses methodologies and tools, to collect and analyze system, network, and application-level logs after injection. We build analysis tools and scripts around existing applications, such as LogDiver [10] and Wireshark [11].
- A smart power grid network simulator, composed of Mininet [12], the Pox SDN controller [13], OpenVSwitch (OVS) switches [14], and Automatak DNP3 Applications [15], to provide a simulation environment for fault injection experiments.

- Resiliency recommendations based on our findings, to improve the fault-tolerance of our target systems and mitigate attacks and failure scenarios discovered in this work.

1.4 Lessons Learned

Over the course of the work presented in this thesis, we encountered several challenges that required new approaches and methodologies. We summarize our observations and experiences in this section.

- On large proprietary systems, fault-error-failure propagation paths are often complex and not well documented. System fault-tolerance and recovery mechanisms are not well understood and have a high degree of uncertainty.

- To ensure reasonable coverage when studying complex systems, fault injection experiments must either (1) account for all parameters in an experiment, or (2) intelligently select experiment parameters to maximize the number of impactful injections.

- A significant number of system, network, and application logs are generated for each fault injection experiment. Analyzing the collected logs requires specialized tools and much human involvement. Additionally, existing tools often prove to be insufficient and require extensions to be developed for use with specific systems.

- Complex systems usually contain a large number of highly individualized components, each requiring a specialized understanding to work with. Often, the implementation of a component may deviate from a manufacturer's documentation or specification, requiring additional efforts to integrate into our overall study.

- Fault injection tools and analysis methods need to be constantly updated as software and systems evolve over time. Updates to individual system components may require entire toolkits to be redeveloped to remain current.

1.5 Future Work

An important continuation of the work presented in this thesis is evaluating the feasibility and trade-offs of resiliency recommendations made in Chapters 2 and 3, especially as systems grow in scale and features. Looking beyond the target systems studies, efforts could be made to generalize tools and analysis methodologies developed in this work. Future directions of specific projects are further documented in Chapters 2 and 3.

1.6 Terminology

In this section, we define terms specific to fault injection methods and approaches. These terms are further used in Chapters 2 and 3. Definitions of dependability terms below are derived from [16].

- **Reliability**: Probability that a system will continue functioning correctly over time.
- **Error**: Deviation from correct functioning of a system.
- **Fault**: Hypothesized cause of an error in a system.
- **Malicious Fault or Attack**: Faults deliberately introduced with the objective of causing harm to a system.
- **Failure**: A transition from correct functioning of a system to incorrect functioning, due to an error.
- **Fault Tolerance**: Ability of a system to avoid failures in the presence of faults.
- **Failover**: A fault tolerance strategy whereby a system utilizes a secondary or redundant component when a primary component fails.
- **Injection experiment**: An experiment consisting of an introduction of one or more faults in a deliberate and controlled manner, with the intention of studying resulting effects of the introduced fault(s) on the system.

- **Baseline experiment**: An experiment performed without a fault injection, with the intention of studying the behavior of an error-free system.
- **Workload**: An application or system execution during an experiment to mimic real-world operation of a system.
- **Application set**: A set of applications with specified configurations and input parameters.
- **Fault injection campaign**: A set of one or more fault injection and baseline experiments with varying parameters for each experiment.
- **Recovery**: Handling of an error or failure to maintain reliable operation of a system. Errors and failures can be handled by masking their effects or restoring the system to an error-free state.
- **Restoration**: Restoration of a system to an error or failure-free state. In Chapter 2, we distinguish between automatic recoveries invoked by the system in response to injected faults, and manual restoration of target components by a user after an experiment has concluded.

1.7 Thesis Organization

The remainder of this thesis is organized as follows: Chapter 2 presents our work on understanding the impact of failures on the Cray Gemini platform via fault injections on JYC, a Blue Waters testbed system at the National Center for Supercomputing Applications. Chapter 3 presents fault injections and failure scenarios of the Raincoat application on an SDN-enabled smart electric power grid simulation. Additionally, we propose several resiliency recommendations to mitigate failure scenarios discovered in our analysis.

CHAPTER 2

UNDERSTANDING THE IMPACT OF FAILURES THROUGH FAULT INJECTIONS ON CRAY GEMINI SYSTEMS

This chapter presents a set of fault injection experiments performed on the Cray XE Blue Waters JYC testbed at the National Center for Supercomputing Applications (NCSA). We use this experimental campaign to improve the understanding of failure causes and propagation observed in the field failure data analysis of NCSA Blue Waters. We use data collected from system logs and network performance counters to (1) characterize fault-error-failure sequences and recovery mechanisms in Gemini interconnection networks and in Cray compute elements, (2) understand the impact of failures on the system and user applications at different scales, and (3) identify and recreate fault scenarios that induce unrecoverable failures, to create new tests for system and application design. In this work, we injected faults through our newly developed tool, *HPCArrow*, with the ability to disable and restore user-specified network links, directional connections, compute nodes and blades. We observe failures manifesting in the form of applications not making forward progress (i.e. crashes and hangs) and network quiesces causing extended system recovery times.

The remainder of this chapter is organized as follows: Section 2.1 introduces the resiliency challenges associated with HPC interconnection networks. Section 2.2 summarizes related work that addresses HPC resilience and fault injection methods applied to large scale systems. Section 2.3 presents a motivation for this work. Section 2.4 provides an overview of the Cray XE system- and network-level architecture, including fault tolerance, detection and recovery mechanisms. Section 2.5 describes the network and compute related fault models studied in this work. Section 2.6 describes the features and functionality of our developed HPC fault injection and recovery tool, and Section 2.7 describes our analysis of collected application, system and performance logs. Sections 2.8 and 2.9 present details and results of our fault injection campaign on the JYC testbed, and make recommendations to

address resiliency issues uncovered in this project. Finally, we conclude in Sections 2.10 and 2.11.

2.1 Introduction

As HPC systems evolve beyond petascale computing, several resiliency concerns at the system, network and application levels remain unaddressed. With exascale systems around the corner, we expect error rates to continue increasing to a point where traditional application-level checkpointing approaches will become unsustainable [17], [18]. To overcome the upper bound of reliability of an HPC system (i.e. the "reliability wall", as discussed in [18]), the HPC community needs a better understanding of fault-to-failure scenarios and improved system instrumentation for detecting and mitigating errors.

Past analyses of recovery mechanisms of interconnection systems have shown the criticality of network-related failures in Cray XE platforms by providing empirical evidence of the impact of those failures on applications and systems [19]. However, understanding of fault-to-failure scenarios and their impact based on production data is limited. Analyses are ultimately constrained to naturally occurring events while the logging and monitoring capabilities of these systems do not provide enough information to completely map the fault-propagation path leading up to failures. In the case of multiple errors and failures, it becomes even more difficult to diagnose such fault-to-failure propagation paths.

In this work, we focus on improving our understanding of faults, errors and failures on interconnection networks of HPC systems by conducting a fault injection campaign on the Cray XE Blue Waters JYC testbed at the National Center for Supercomputing Applications (NCSA). As part of this project, we developed *HPCArrow*, a software-implemented fault injection (SWIFI)[20] tool to inject faults and control recoveries of system components at various levels. Our injection campaign consisted of 84 unique experiments (15 baseline and 69 injection experiments), spanning 5 months and requiring 4,656 node hours on the target system. Over the course of our campaign, 9 unique applications were executed a total of 462 times to provide realistic workload scenarios as we injected faults into system components. In total,

69 faults were injected into link, connection, node and blade components to study the effects of faults, errors and failures across the system.

The key contributions of this project are:

- **Network and compute fault injection tool for large-scale HPC systems:** We designed and developed *HPCArrow*, a fault injection tool and methodology that can inject one or more faults into user-specified links, connections, nodes, and blades on an HPC system. *HPCArrow* also handles workload generation and automated fault injections with user-defined timing requirements. Injected faults are usually accompanied by restore procedures to return the target system to the state prior to fault injection. The tool currently supports failure injections on Cray platforms with Gemini and Aries interconnects. *HPCArrow* was successfully used to investigate and validate failure scenarios presented in [19], [21], [10] and establish in-depth fault-to-failure propagation and delays. Specific modules of this tool and their functionality are discussed in Section 2.6.

- **Assessing susceptibility of HPC runtime frameworks to faults, failures and errors:** Given the distributed nature of HPC applications, application runtime frameworks are commonly used to improve communication efficiency and reduce development complexity. Runtime frameworks provide varying degrees of fault tolerance, load balancing, power awareness, and automatic overlap of communication with computation [22]. In this work, we consider workloads based on variations of the message passing interface (MPI), Charm++ and partitioned global address space (PGAS) runtime frameworks. We observe that some frameworks are more susceptible to application hangs and crashes than others. We attempt to correlate abnormal behaviors such as premature terminations and hangs in application executions to hardware and network-level errors. These results are further discussed in Section 2.9.

- **Recommendation for notification of errors at application and system levels:** Results from our experiments revealed a lack of reporting of network-related errors, resulting in a lack of real-time feedback to applications. Extended component and system recovery dura-

tions present an opportunity to report information to an application or system monitoring service, which could improve their response to network-related failures. Placing additional detectors and/or a notification system on the health supervisory system (HSS) could be used to trigger higher-level recovery mechanisms and transmit low-level fault information to the system management workstation (SMW).

- **Identification of critical errors and conditions:** The analyses of error data obtained from our fault injection campaign identified critical errors and conditions that can be used to provide real-time feedback to applications and resource management services. For example, (1) at the system level, one can detect and send notifications of application hang conditions, and (2) at the application level, one can send notifications of critical errors that can lead to corruption or unexpected terminations of applications.

2.2 Literature Review

This project is motivated by a number of past studies on the reliability of HPC systems, especially those on Cray platforms and interconnects. In this section, we outline existing literature and work related to this study. Table 2.1 summarizes the literature reviewed as part of this study.

The fault models, tools and analysis methodologies used in this work are based on past fault injection campaigns on Cielo, a Cray XE supercomputer at Los Alamos (LANL) and Sandia (SNL) National Laboratories [23]. The work presented in this chapter is part of the Holistic, Measurement-Driven Resilience (HMDR) project, a collaboration between the University of Illinois at Urbana-Champaign (UIUC); Sandia (SNL), Los Alamos (LANL), and Lawrence Berkeley (LBNL) National Laboratories; and Cray Inc. Other fault injection campaigns on HPC systems have injected faults at the memory [24], [25], processor [25], [26], [27], and application [4], [28] levels. A number of narrow studies [29], [30] have investigated faults and failures on the message-passing interface (MPI). The work presented in this chapter takes a holistic approach to studying faults and failures across various levels of system components and application runtime frameworks.

Table 2.1: Summary of Literature Reviewed for Work Presented in Chapter 2

	Title	**Comments**
[10]	LogDiver: A Tool for Measuring Resilience of Extreme-Scale Systems and Applications	Presents LogDiver for analyzing application-level resiliency on HPC systems.
[17]	Addressing Failures in Exascale Computing	Presents a summary of system and application resilience in HPCs, establishes taxonomy and discusses error prevention, detection, and recovery.
[18]	The Reliability Wall for Exascale Supercomputing	Quantifies the effects of reliability on performance and generalizes a "reliability wall" for exascale systems.
[19]	Analysis of Gemini Interconnect Recovery Mechanisms	Characterizes recovery mechanisms of Gemini interconnects from raw system logs.
[21]	Lessons Learned from the Analysis of System Failures at Petascale	Analyzes failures and impact on Blue Waters. Concludes with software being the main cause of failures.
[31]	Measuring and Understanding Extreme-Scale Application Resilience	Characterizes resiliency of HPC application runs on Blue Waters by analyzing system- and application-level logs.
[32]	A Survey of Fault Tolerance Mechanisms and Checkpoint Restart Implementations for High Performance Computing Systems	Reviews the failure rates of HPCs and surveys fault tolerance approaches like rollback-recovery techniques.
[33]	A Large-Scale Study of Failures in High-Performance Computing Systems	Analyzes failure data of two HPC systems and reports time and rates of failures and repairs.

2.3 Motivation

Application resilience in current HPC systems is primarily achieved through checkpointing to mitigate software and hardware failures [32]. This brute-force approach allows applications to revert back to a previous error-free state upon encountering system-level failures. Application-level checkpointing is the preferred resilience mechanism due to difficulties with designing and implementing fault tolerance at the programming framework level (e.g., the MPI User Level Failure Mitigation approach [34]). However, this checkpoint-and-restart approach comes with high overheads and performance penalties, especially as we move to exascale systems with a larger number of components. Because of the evolutionary nature of HPC technologies, it is expected that systems, for the foreseeable future, will continue to have fault mechanisms and behaviors similar to those found in current deployments [35]. Thus, comparisons of well-explored failure scenarios across multiple generations of systems should enable identification of persistent high impact fault scenarios. Tailoring instrumentation and resilience techniques to enhance system and application resilience characteristics in these high impact scenarios can enhance the efficiency and throughput of both current and future platform architectures.

System recovery mechanisms that are defined and implemented by HPC platform vendors are typically not well understood or characterized by their signatures in log files and platform measurables in terms of durations, impacts, and success rates, particularly for complex fault scenarios. A number of studies have explored system logs from large-scale HPC systems ([21], [33], [36]), but connecting failures with their root causes or precursor faults has proven difficult. The resulting fault-to-failure path models are rarely complete, and there is a significant amount of associated uncertainty. In addition, built-in, automatically triggered recovery mechanisms can further obscure failure paths and may leave no trace in the log files typically used by system administrators and made available to researchers. The research community needs a way to verify, and possibly augment, failure models through testing in a controlled environment. In particular, the community needs tools to enable documented and repeatable HPC environment configurations, including instrumentation and applications placement, and injection of known faults in a repeatable non-destructive manner on large scale HPC systems.

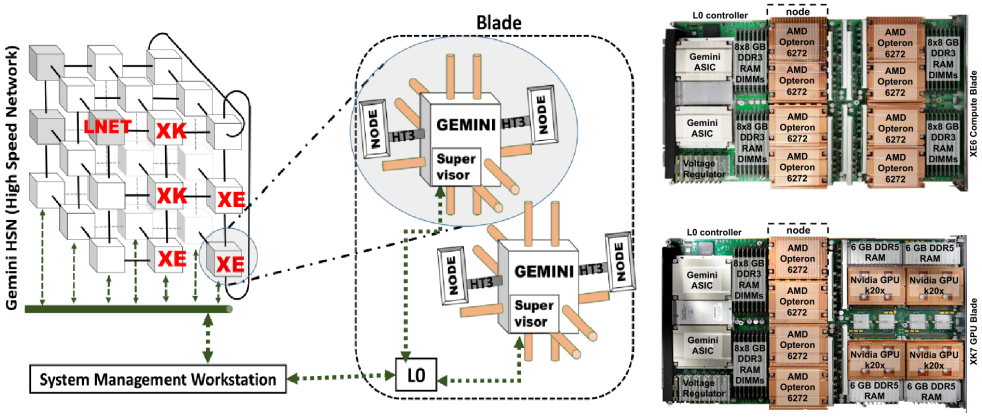

Figure 2.1: Cray XE platform architecture [21]

2.4 Architecture Overview

This section provides an architecture overview of the Cray XE platform using Gemini interconnection networks. We also document the fault tolerance mechanisms present in Cray XE systems as well as fault detection and recovery processes in later subsections.

2.4.1 System Architecture

Cray XE systems are hierarchically organized into cabinets, chassis and blades, as shown in Fig. 2.1. The high-speed network (HSN) of a Cray XE system is laid out as a anisotropic 3-D torus. Each Cray XE6/XK7 blade consists of four compute nodes, two Gemini Application Specific Integrated Circuits (ASICs), each housing two Network Interface Controllers (NICs) and a 48-port router. NICs are attached to nodes using a HyperTransport 3 host interface. Each Gemini ASIC is connected to the network by means of 10 torus connections, two each in X+, X–, Z+, Z– and one each in Y+ and Y–. An ASIC also connects two nodes internally using NICs. Each connection is composed of four links and each link is composed of three single-bit bidirectional lanes. Thus, each connection consists of 12 lanes, and ASICs connect to one another on the network via 24 lanes in the X and Z dimensions, and 12 lanes in the Y dimension. Further details of the Cray XE platform architecture may be found in [19].

2.4.2 Fault Tolerance and Resiliency

Cray XE systems provide several levels of fault tolerance via software-level supervisory services, hardware-level error corrections and network-level redundancy.

2.4.2.1 Hardware Supervisor System (HSS) and System Resiliency Features

The Hardware Supervisor System (HSS) is a collection of hardware components responsible for monitoring compute nodes. The HSS consists of:

(i) Blade-level (L0) and cabinet-level (L1) controllers (see Fig. 2.1), which monitor their housed nodes, reply to heartbeat signal requests and collect data on temperature, voltage, power, network performance counters, and runtime software exceptions,

(ii) the HSS manager, which collects node health data and executes management software, and

(iii) the HSS network, which connects blade and cabinet-level controllers to the HSS manager.

The behavior of the HSS upon detection of failures is presented in Section 2.4.3. Further details of the HSS and system resiliency features can be found in [21].

2.4.2.2 Hardware-Level Error Correction

Network traffic passing through Gemini ASICs is protected by a 16-bit cyclic redundancy check (CRC). For each network packet, the CRC computation is performed upon arrival at a Gemini ASIC and between the transition from NIC to router. Similar to the common transport control protocol (TCP), Gemini ensures reliable delivery of packets by using the sliding window protocol. Further, memory regions (except router table buffers) are protected via single error correction-double error detection (SEC-DED) [37].

2.4.2.3 Network-Level Redundancy

To ensure successful delivery of packets, Gemini ASICs connect to one another via two redundant connections in the X and Z dimensions, and one connection in the Y dimension. Each connection is composed of two redundant links, each with three redundant lanes. These layers of redundancy allow network communication to continue in degraded mode with just one active link with active one lane.

2.4.3 Fault Detection and Recovery

In this work, we investigate failures in compute nodes and blades and network links, ASICs and connections in isolation as well as in combination. Cray XE systems handle such failures by triggering automatic recovery processes. Failures, depending on where they occur, are detected by a supervisory block on the router ASIC, a blade controller (BC) on a blade, or a system management workstation (SMW). Each BC is locally connected to a supervisory block on the router ASIC, and remotely connected to the SMW through the Cray hardware supervisory system (HSS) network. Blade controllers detect failed links and power loss to mezzanine cards housing Gemini ASICs, and deliver information about critical failures to the SMW in order to initiate associated recovery procedures.

Upon detection of a network-related failure, the SMW initiates a system-wide recovery. Actions taken by the SMW during the recovery process depend on the type of failures described in the following sections.

2.4.3.1 Lane Failure

Each link in a Gemini network consists of three single-bit bidirectional lanes, allowing it to tolerate up to two lane failures and continue to operate in degraded mode. For each lane failure, the L0 blade controller logs a failure event and triggers a lane recovery up to a fixed number of times (defined by the system administrator). If a lane recovery is unsuccessful, the lane is marked "permanently failed". Upon failure of all three lanes, the L0 blade controller marks the link as inactive and triggers a link failover instead of individual lane recoveries [19]. The lane recovery process is summarized as a

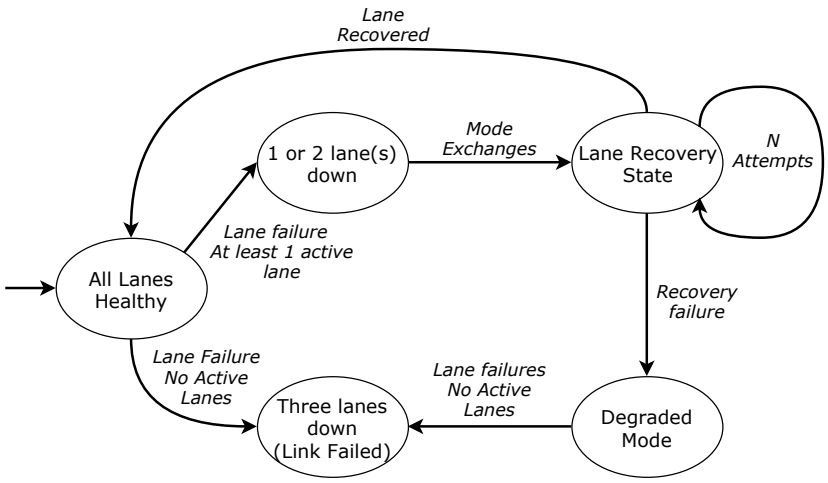

Figure 2.2: State transition diagram for lane recovery procedure, derived from [19]

state transition diagram in Fig. 2.2.

2.4.3.2 Link Failover

A link failover is triggered by the L0 blade controller when a link becomes inactive, due to failures of all three lanes. A link failure could be caused by a more widespread failure in the cabinet, blade or mezzanine. It could also be caused by corruptions in routing tables or faults in physical cables. Gemini ASICs continue maintaining connectivity through remaining functional links.

The link failover procedure waits to aggregate failures from other components, determines which compute blades are alive, quiesces network traffic and attempts to find a new route. This entire procedure varies from 30 to 600 seconds, depending on the size of the system. If the link failover process is successful, the failed link is masked and communication paths in the network are restored. A failure of the link failover process, however, causes the HSN to fail leading to a system-wide outage [19]. The link failure process is summarized as a state transition diagram in Fig. 2.3.

2.4.3.3 Warm Swap

The warm swap process allows system administrators to manually add and remove compute nodes and blades on the system without impacting other compute elements. The warm swap process initializes or disables links and

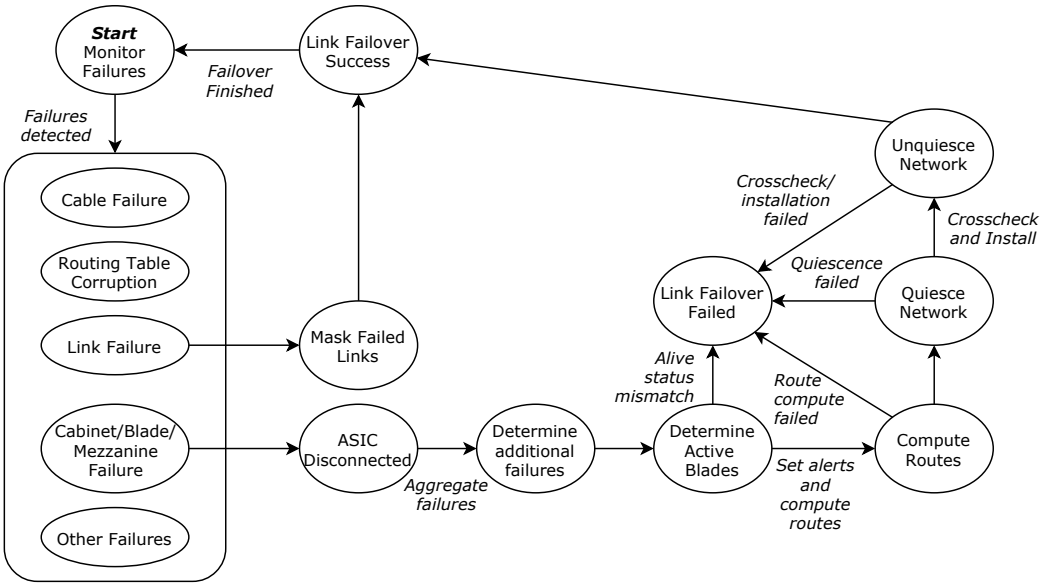

Figure 2.3: State transition diagram for link failover procedure, derived from [19]

connections, computes new routes, quiesces the network, installs computed routes and unquiesces the network. During network quiescence, traffic is suspended across all links on the network. This process is summarized as a state transition diagram in Fig. 2.4 .

In some cases, recovery mechanisms can mask failures without causing major system interruptions. Analyses of field failure data indicate that: (1) recovery mechanisms handling complex failure scenarios may not always succeed, and (2) protracted recoveries that eventually succeed may still have a significant impact on the system and applications [21].

2.5 Fault Models

In this section, we present the fault models studied in this work. We investigate failures in compute nodes and blades and network links, ASICs and connections. We targeted failures of these compute and network components since they occur frequently enough in production systems to be responsible for performance degradation [21].

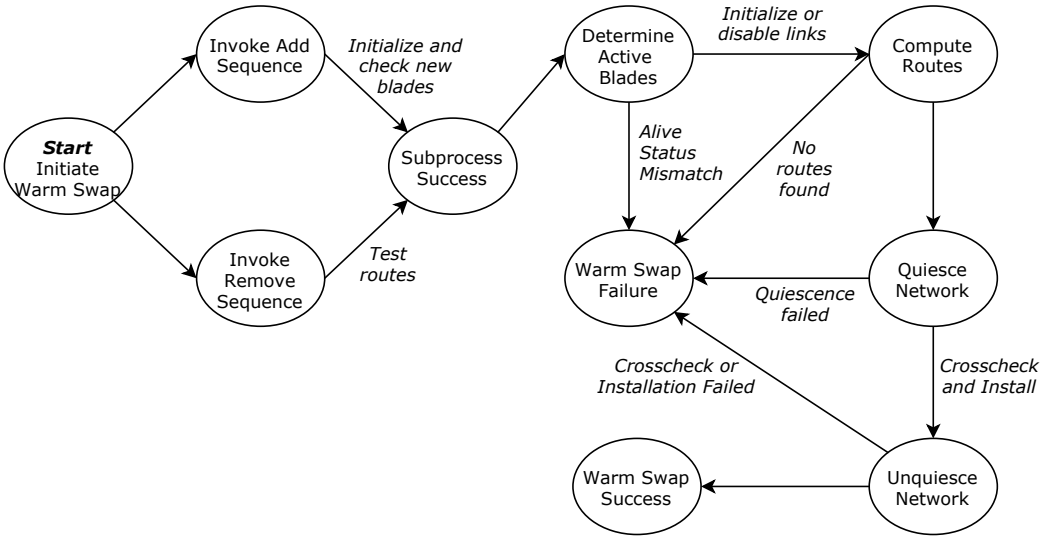

Figure 2.4: State transition diagram of warm swap operation, derived from [19]

2.5.1 Link Fault Model

The link fault model involves the failure of a network link between two Gemini ASIC routers, which causes packets in flight to be dropped (see Fig. 2.5). We recreate a link failure by deactivating a connection's links by modifying a status flag on either end of the target link. This emulates a scenario where the link is intentionally deactivated by system administration software, to prevent use of a physically damaged link or where maintenance is required. When the status flag on one end of the target link is modified, the Gemini ASIC on the opposite end is also affected. The link failure is detected by the Hardware Supervisory System (HSS) which responds by masking the target link. This causes traffic to be routed on other links on the same connection. After the automated recovery procedure associated with link failures completes, the link is marked as disabled on the SMW.

2.5.2 Connection Fault Model

The connection fault model considers a scenario where all network links of a connection between two Gemini ASICs are deactivated, thereby causing a hole in the routable topology of the system (see Fig. 2.6). The automated recovery process responds by rerouting all traffic around the hole, via connections in other directions. For example, a failure in the Z+ direction may cause

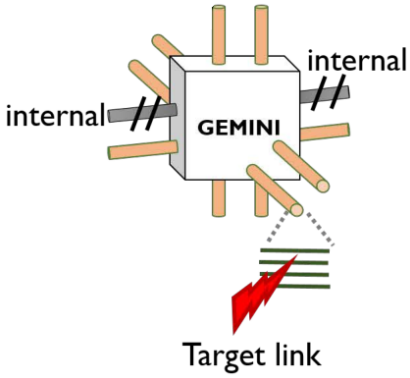

Figure 2.5: Link failure fault model

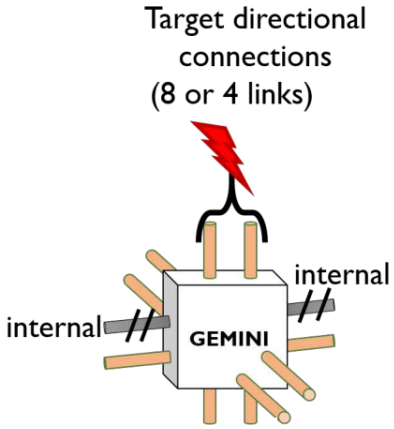

Figure 2.6: Connection failure fault model

all traffic to be routed via the X+, Z+, X- connections after the automated recovery procedure completes.

2.5.3 Node Fault Model

The node fault model considers a scenario where a compute node fails and causes an application running on the node to terminate (see Fig. 2.7). We emulate a node failure by sending a non-maskable interrupt (NMI) to the CPU on the target node, which causes the CPU to hang and not make forward progress. A node failure or hang does not cause an automated recovery procedure to be initiated as routing paths in the network remain unaffected. However, traffic balance across surrounding nodes could be affected.

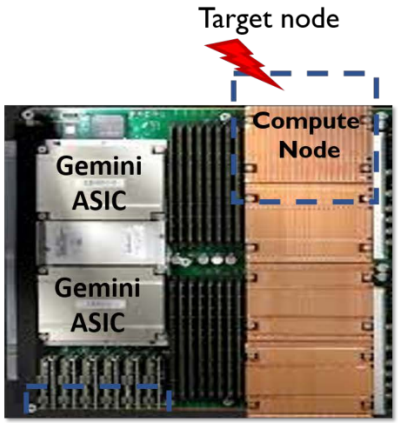

Figure 2.7: Node failure fault model

2.5.4 Blade Fault Model

The blade fault model considers a scenario where an entire blade consisting of four compute nodes, two Gemini ASICs and 40 network links is turned off. We emulate this scenario by turning off the voltage regulator of the mezzanine in the target blade via an administrative command (see Fig. 2.8). When the target blade is powered off, an automated recovery process is initiated to handle unavailable links. We expect the automated recovery process to route around the failed blade. Similar to the node failure scenario described in Section 2.5.3, we expect any applications running on the nodes within the target blade to terminate. Additionally, traffic from other blades passing through the target blade may be affected.

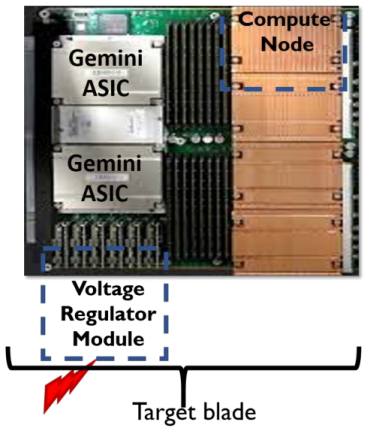

Figure 2.8: Blade failure fault model

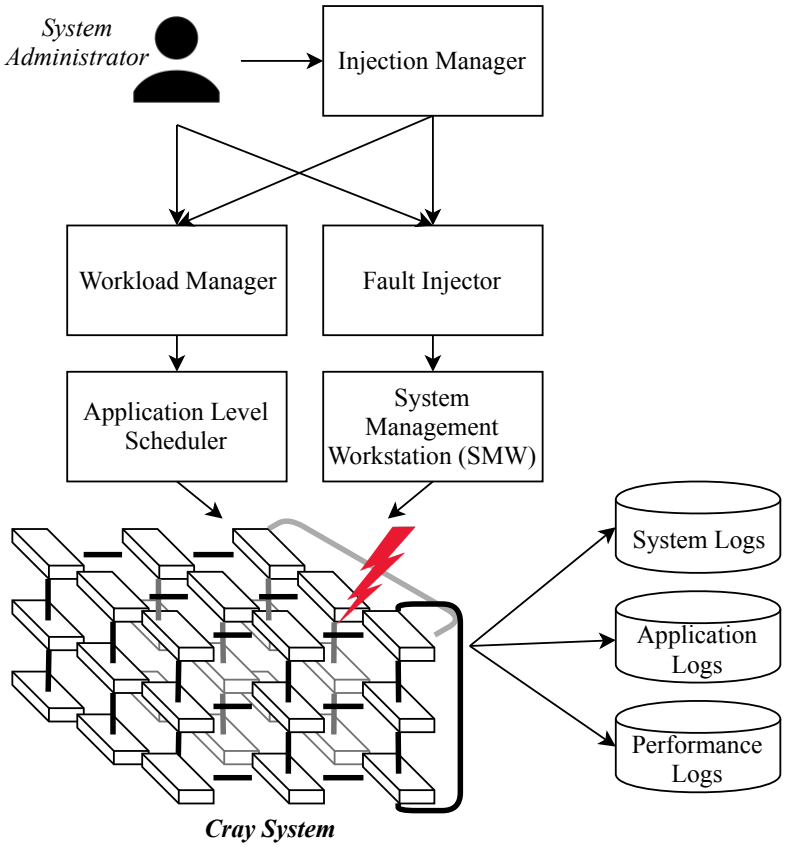

Figure 2.9: *HPCArrow* toolkit and components

2.6 Fault Injection Tool - *HPCArrow*

A major contribution of this work is the development of *HPCArrow*,[1] a software implemented fault-injection (SWIFI) [20] tool and methodology that can inject one or more faults into user-specified links, connections, nodes, and blades on an HPC system. Injected faults are usually accompanied by restore procedures to return the target system to the state prior to fault injection.

HPCArrow supports execution of arbitrary failure scenarios on network and compute components. The tool currently supports failure injections on Cray machines with Gemini and Aries interconnects. *HPCArrow* consists of three major modules (see Fig. 2.9) to systematically study the effects of

[1] *HPCArrow* was developed by Lavin Devnani (author) and Sharon Tang. *HPCArrow* is based on past work by Fei Deng presented in [23] and is part of the Holistic, Measurement-Driven Resilience (HMDR) project, a collaboration between the University of Illinois at Urbana-Champaign (UIUC); Sandia (SNL), Los Alamos (LANL), and Lawrence Berkeley (LBNL) National Laboratories; and Cray Inc.

faults and failures on HPC systems and applications:

- A **Workload Manager** that launches applications of varying scales at user defined locations (nodes and blades) on the target system and collects output logs from each application run.
- A **Fault Injector** that injects user-specified faults into selected network or compute components and manages restoration of impacted ones.
- An **Injection Manager** that automates injection experiments without requiring user intervention between each experiment run.

We expand on the specifics of these modules in Sections 2.6.1–2.6.3.

2.6.1 Workload Manager

The Workload Manager module of *HPCArrow* is responsible for the execution of application workloads during fault injection, failure recovery and restore operations for each experiment run. This module allows users to define system resources (size, position, number and types of nodes) and application configuration (input parameters and output logging) required for each workload run in an *HPCArrow* specific format. Specifying system and application configurations in a custom format allows *HPCArrow* to execute workloads on systems with different application-level schedulers. Currently, the Workload Manager module of *HPCArrow* supports MOAB/Torque [38] and Slurm [39] application-level schedulers.

Beyond specifying configurations for individual applications, the Workload Manager module supports invocations of multiple workloads to run simultaneously. Users can select a 'set' of applications to launch at the same time, which allows for studying the impact of fault injections across multiple workloads running on different compute components.

2.6.2 Fault Injector

The fault injector module executes commands that inject faults into user-specified system components and initiate restore processes to re-enable impacted components. For multiple and sequential injections, this module is

responsible for timing each fault injection and any subsequent recovery invocations. Faults and restoration procedures are invoked via administrative commands supplied by Cray. This module (1) translates user-selected target components into appropriate parameters for injection and recovery commands, (2) executes aforementioned commands with translated parameters, and (3) collects outputs and errors from executed commands to a centralized injection experiment log. A summary of commands used by *HPCArrow* for targeting Cray Gemini systems is provided in Appendix A.

2.6.3 Injection Manager

The Injection Manager module allows a fault injection campaign (consisting of one or more experiments) to run without user intervention between each experiment execution. Users can specify a series of experiments as a fault injection campaign consisting of applications workloads, target components, and injection/restore delays. Upon invocation of the campaign, the injection manager executes the workflow illustrated in Fig. 2.10 for each experiment.

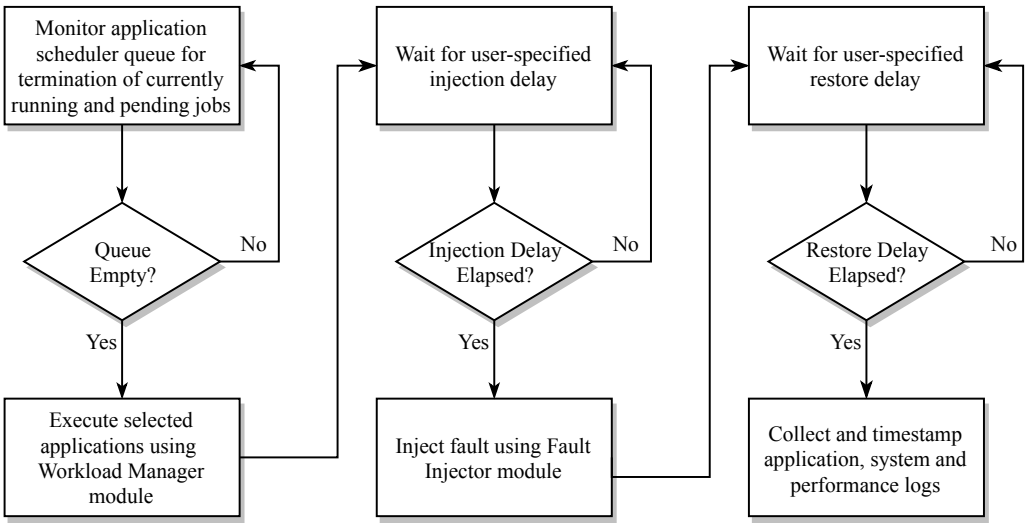

Figure 2.10: Injection manager workflow

2.7 Analysis Methodology

In this work, we analyze system-generated logs as well as performance and monitoring data provided by collection and aggregation services running at various levels of the system. We use results from our analysis to (1) identify appropriate components to inject upon during our preparation stage, and (2) identify recovery status and quantify metrics after injection.

2.7.1 Component Selection

The fault injection, automated recovery and manual restore procedures described in Section 2.5 vary in execution time on our target system from approximately 5 minutes for single link failures to approximately 20 minutes for blade failures. Such an extended turnaround time, combined with limited availability of our target system, prohibits us from performing a statistically significant number of injections on randomly selected components.

In our initial experiments, consisting of application sets 1 to 5, we targeted workloads and components at random from all running applications for each experiment. In the next phase, our experiments targeted application sets 6 to 8. We selected components with maximum utilization over the course of an experiment's lifetime to ensure that our injections are impactful. The selected components were connections between Gemini routers with the maximum amount of traffic (in bytes) or nodes and blades connected to Geminis with maximum traffic.

The Lightweight Distributed Metric System (LDMS) service, described in [40], logs traffic throughput (in bytes/second) in the X+, X–, Y+, Y–, Z+ and Z– directions for each Gemini on the target system. To identify components with maximum utilization over an experiment run, we first profile an application without injecting faults. A smoothed time series plot of traffic data obtained from the LDMS service allows us to visually identify connections with high throughput and select suitable components for fault injections.

As an example, we demonstrate the traffic throughput of an execution of the Kripke Charm++ application running on 8 nodes. Figure 2.11 shows smoothened traffic plots over the duration of the workload run. Applying our selection methodology, we would inject on links or connections of either

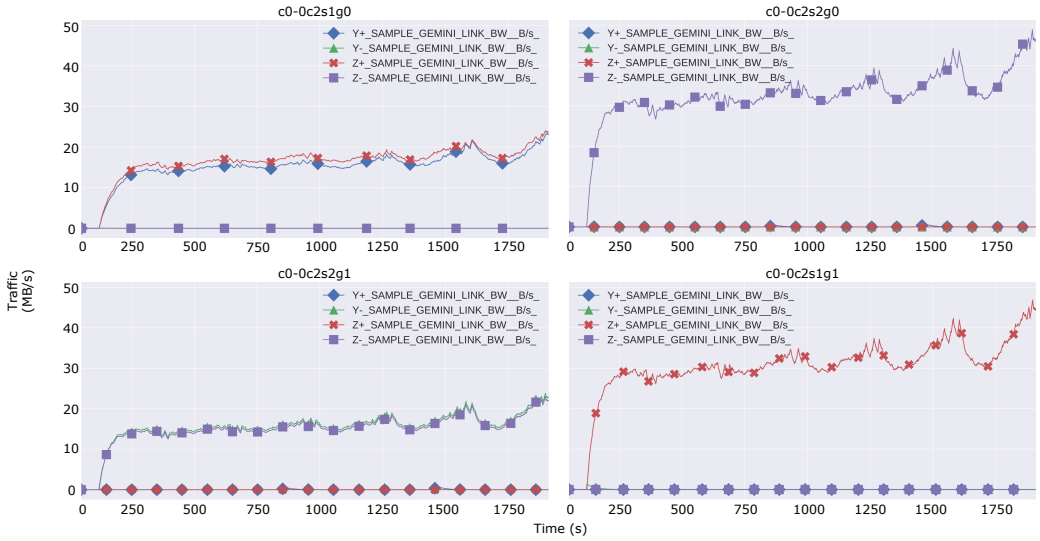

Figure 2.11: Component selection example for Kripke Charm++ applications. Each plot represents network traffic for a given Gemini and connection over application execution.

Gemini `c0-0c2s1g1` in the Z+ direction or Gemini `c0-0c2s2g0` in the Z– direction, as these connections handle the maximum amount of traffic.

2.7.2 Event Analysis

To identify the occurrence and duration of injection, recovery, and restore events, we collect and analyze the system log files provided in Table 2.2. These system-level log files are generated by Cray logging daemons running at multiple levels of the system, from the SMW and component controllers to each individual compute node. We use LogDiver [10], a tool for the analysis of system and application-level resiliency in extreme-scale environments to identify events of interest in the collected system logs. In this study, we use LogDiver to (1) extract network-recovery operations, determine the completion status of recoveries, and diagnose the cause of recovery failures, and (2) identify application termination status and reasons behind abnormal terminations (crashes and hangs). Prior to execution, LogDiver is configured with regular expressions (regex) that match with events of interest in collected system logs. These regular expressions are constructed from information available in Cray documentation and from manual inspection of events in collected logs [41].

Table 2.2: System Logs Analyzed in this Study, Containing Log Names and Content Descriptions

Log Name	Content of Log File
`smwmessages`	System Management Workstation (SMW) hardware and environmental history.
`xtdiscover`	Output from xtdiscover command, used to discover hardware components and respond to changing hardware configurations.
`events`	Generic system-level events, heartbeats, sequence identifiers.
`commands`	Start and end timestamps of commands executed on SMW.
`netwatch`	Timestamps of when network links become inactive.
`nlrd`	Timestamps and details of hardware-level errors, network-level failures, automated recoveries and warm swaps. All phases of automatic failure recoveries and warm swaps are logged.

2.7.3 Network Performance Counters

On Cray systems, the Lightweight Distributed Metric Service (LDMS) is responsible for logging network performance for each Gemini connection [40]. In this study, LDMS is configured to sample traffic data (in bytes/second) at 1-second intervals. However, LDMS logging is susceptible to node and network failures. Node failures cause data collected by on-node daemons to be lost if memory is overwritten or lost. Failures in the HSN (e.g. during a network quiescence) also cause data points to be dropped. We overcome potential data loss by exponentially smoothing traffic samples to one minute moving averages, as described in Section 2.7.1.

2.7.4 Application Data

To analyze the impact of failures and recoveries on applications, we redirect information reported on *stdout* and *stderr* to application logs for each workload run. Applications report timestamps, computation steps, exit reasons, and critical errors to varying degrees in their logs. Additionally, the application-level scheduler reports global network quiescence and throttling events when automatic failure recoveries or manual warm swap procedures are invoked. Analysis of application-level output allows us to correlate system-level failures and recoveries to events that occur during workload runs, and identify causes of abnormal application behavior (e.g. premature terminations and hangs).

2.8 Experiment Setup

The preceding sections of this chapter describe our general fault injection approach and methodology, applicable to any Cray XE system. This section presents details of a fault injection campaign conducted on the Cray XE Blue Waters JYC testbed at the National Center for Supercomputing Applications (NCSA). Our injection campaign consisted of 84 unique experiments, spanning 5 months and requiring 4,656 node hours on the target system. Out of the 84 experiments, 15 consisted of baseline runs with no injections and the remaining 69 were fault injection ones. The baseline results were later used to compare system and application behavior to results obtained from fault injection experiments. Over the course of our campaign, 9 unique applications were executed a total of 462 times to provide realistic workload scenarios as we injected faults into system components. In total, 69 faults were injected into link, connection, node and blade system components.

2.8.1 Target System Description

Our experimental campaign targeted JYC, a 96-node Cray XE/XK testbed at the National Center for Supercomputing Applications (NCSA). JYC is a 1-cabinet, 3 chassis machine consisting of 56 XE nodes, 28 XK nodes and 14 service nodes.

The 56 XE compute nodes are spread across 14 Cray XE6 blades, with up to four nodes being housed per blade. Each XE node consists of 2 $\times$ 16-core AMD Opteron 6276 processes @ 2.3 GHz. An Opteron processor is composed of 8 dual-core AMD Bulldozer modules, with each module having a 8$\times$64 KB L1 instruction cache, 16$\times$16 KB L1 data cache. The processor also includes 8$\times$2 MB L2 caches (shared between cores of each Bulldozer module), and a 2$\times$8 MB L3 cache (shared among all cores). Memory-wise, 64 GB of DDR3 RAM (8 $\times$ 8GB DIMMs) are installed for each compute node. The installed memory modules are protected with x8 Chipkill code that uses eighteen 8-bit symbols to make a 144-bit ECC word (128 data bits + 16 check bits) [37], [42]. The L1 data, L2 and L3 data caches are also protected with ECC, while other caches are protected with parity [21].

The 28 XK nodes are spread across 7 Cray XK7 blades with 4 nodes housed on each blade. On each node, one socket is occupied by a 16-core

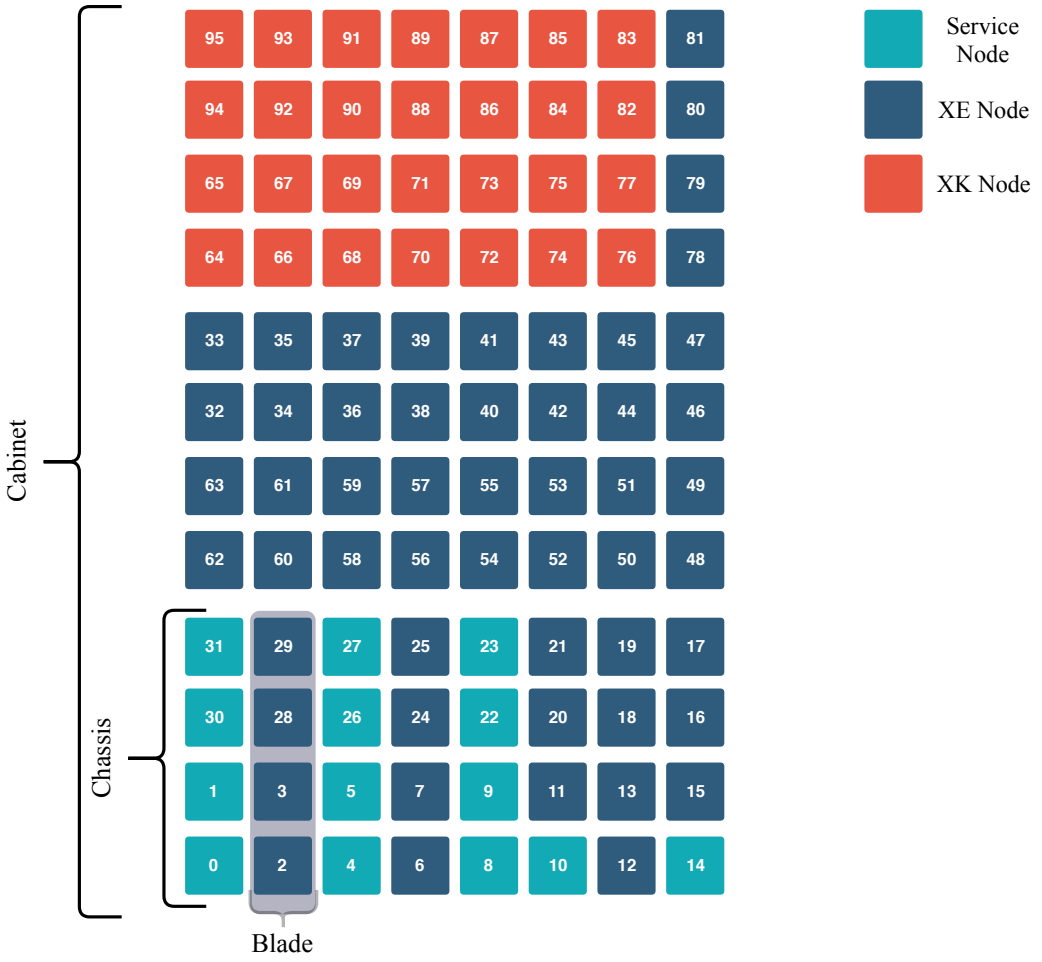

Figure 2.12: JYC system overview with node types and identifiers.
A detailed representation of the system is provided in Appendix B.

Opteron 6276 processor described above. The second socket is occupied by a NVIDIA K20X accelerator, with 2,880 single-precision CUDA cores, 64 KB of L1 cache, 1,536 KB of dedicated L2 cache, and 6 GB of ECC protected DDR5 RAM. Since only one socket is occupied by a general purpose CPU, the amount of memory available to CPU tasks is halved to 32 GB (4×8 GB DIMMs) when compared to XE nodes [21].

The remaining fourteen nodes are reserved for servicing the system. Twelve out of fourteen nodes are spread across three Cray XIO blades with four nodes per blade. These nodes consist 6-core AMD Opteron 2435 Istanbul processors @ 2.6 GHz and x4 Chipkill 16 GB of DDR2 memory (4×4 GB DIMMs). The other two service nodes contain AMD Opteron 6276 processors and are housed alongside XE compute nodes in Cray XE6 blades [21].

A high-level system map of JYC is provided in Fig. 2.12. We expand on this system map by providing node identifiers and component names in Appendix B.

2.8.2 Application Workloads

To generate sufficient network and compute activity during our experiments, we executed several HPC benchmark applications at various scales and observed their behavior before, during and after our fault injections and recovery invocations. The benchmark applications were chosen from different programming frameworks and represent characteristics of real-world HPC workloads. Each workload was tuned to run for approximately 30 minutes, to allow for injection, automatic recovery and manual restoration operations to execute with sufficient time.

One of the many goals of this project is to assess the susceptibility of HPC runtime frameworks to faults, failures and errors. To this end, we select workloads from the following frameworks:

1. **Message Passing Interface (MPI)** Message Passing Interface is a communication protocol that supports point-to-point and collective communication between compute nodes in a distributed computing system. The MPI layer handles synchronization and communication between processes mapped onto compute nodes in a language-independent way, with language-specific bindings. MPI remains the dominant model used in HPCs today [43]. We specifically look at Cray, Intel and PGI implementations of the MPI standard in our workloads.

2. **Charm++**

   Charm++ is a C++-based parallel programming system that implements a message-driven migratable objects programming model, supported by an adaptive runtime system. Charm++ is based on a message-driven migratable objects programming model, and consists of a C++-based parallel notation, an adaptive runtime system that automates resource management, a collection of debugging and performance analysis tools, and an associated family of higher level languages [44]. We consider applications that utilize the native uGNI communication li-

Table 2.3: Application Descriptions and Characteristics

Ref.	Application	Discipline	Programming Model	Characteristics
[46]	Anelastic Wave Propagation (AWP-ODC)	Seismic	MPI (PGI)	Structured Grid, Sparse Matrix
[47]	MIMD Lattice Computation (MILC)	Particle Physics	MPI (Intel)	Structured Grid, Dense Matrix
[48]	Pseudo-Spectral Direct Numerical Simulations (PSDNS)	Fluid Dynamics	MPI (Cray)	Structured Grid, FFT
[49]	NAMD	Molecular Dynamics	Charm++ (SMP over uGNI)	N-body, FFT
[50]	Adaptive Mesh Refinement (AMR)	Numerical Analysis	Charm++ (SMP, HugePages over uGNI)	-
[51]	LeanMD	Molecular Dynamics	Charm++ (HugePages over uGNI)	N-body, FFT
[52]	Kripke	Particle Physics	Charm++ (HugePages over uGNI)	Structured Grid, Dense Matrix
[53]	Unified Parallel C Fourier Transform (UPC-FT)	Numerical Analysis	PGAS Unified Parallel C	FFT

brary available on Cray XE systems, with shared-memory optimizations using *pthreads* and Cray huge pages modules.

3. **Partitioned Global Address Space (PGAS)**

   The PGAS programming model shares data between compute elements by creating a global address space for shared memory. PGAS assumes a global memory address space that is logically partitioned and a portion of it is local to each compute element [45]. We specifically look at applications using the Unified Parallel C (UPC) programming model in this work.

The application benchmarks selected in this study are highly parallel workloads and span several scientific disciplines, from seismic simulations to molecular dynamics. We summarize applications and their characteristics in Table 2.3.

In order to study the impact of injected faults and recoveries across the entire system, we simultaneously launched a set of workloads for each experiment. Launching workloads across the entire system allows us to identify failures propagating or cascading through the system and impacting applications running further away from the injected component. Applications were chosen such that each set had a mix of runtime frameworks, application sizes, and overall system utilization. In our injection campaign, the *HPCArrow* workload generator was configured with eight unique application sets. Application sets and their runtime parameters are defined in Appendix C.

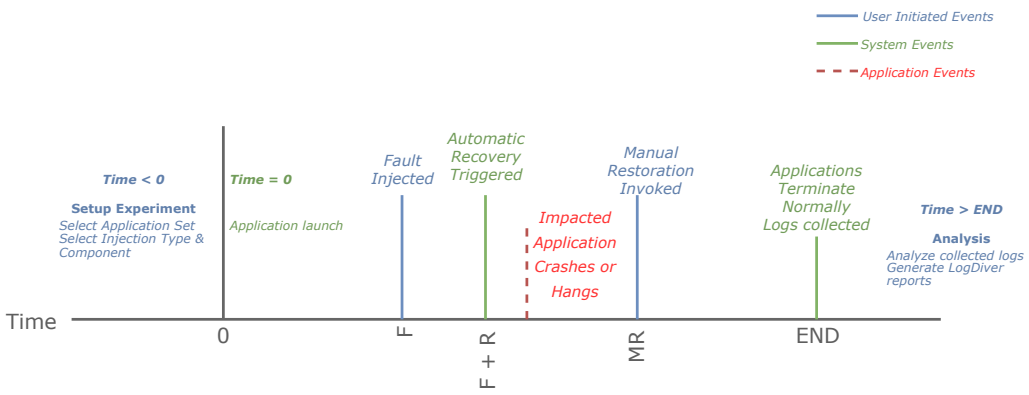

Figure 2.13: Fault injection timeline demonstrating user, system and application events

2.8.3 Experiment Timeline

Experiments in our fault injection campaign consist of specifying an application workload set, an injection type and a target component. The timeline presented in Fig. 2.13 summarizes an example experiment run. Prior to execution, a user configures *HPCArrow* with (1) the selected application set using the Workload Manager module, and (2) the injection type, component and delay using the Fault Injector module. At time $T = 0$, the selected application set is executed by submitting workload jobs to the application-level scheduler. The Fault Injector module waits until F seconds have elapsed before injecting a fault into the selected component. If the injected fault causes a component to fail, an automatic recovery procedure associated with the component is invoked by the SMW after R seconds have elapsed. This procedure computes new routes, quiesces the network, installs new routes, masks failures, and finally unquiesces the network. Depending on the application impacted, a job may crash or hang due to operations performed during the recovery phase. At time $T = MR$, the manual restore procedure is invoked by the Fault Injector module, which re-enables the failed component and warm-swaps it back into service. During the warm swap, the network is once again quiesced, new routes installed and unquiesced. At the end of our experiment (when applications terminate naturally), system-, network- and application-level logs are collected to be analyzed offline. LogDiver is used to generate a report containing timestamps and frequency of observed events. In this work, we select a constant value of F= 100 seconds, which

allows workloads to complete their initial setup and enter a steady state of computation. The manual restoration delay (MR) is dependent on the component type, but is configured to invoke 200 seconds after automatic recovery is completed.

2.9 Results

This section provides a comprehensive overview of the results obtained from our fault injection campaign. For each experiment, we apply our analysis methodology described in Section 2.7 to characterize failures and recoveries at the system level and analyze impact at the application level. In Section 2.9.1, we provide a description of common hardware errors observed in system logs collected from our experiments. In Section 2.9.2, we describe errors reported by applications in response to injected faults and identify potential causes for these errors. We summarize our fault injection campaign in Section 2.9.3 by classifying each experiment by runtime framework and application. Additionally, we provide timelines demonstrating network and system activity of selected experiments for network and compute related failures. Finally, we conclude with a discussion of our observations and present resiliency recommendations.

2.9.1 Hardware Error Descriptions

During fault injection and recovery, hardware error codes are logged by the `xtnlrd` daemon running on the System Management Workstation (SMW). The resulting `nlrd` system log contains a component-based history of hardware errors in chronological order. We configure *LogDiver* with hardware error codes to identify events of interest in our analysis. We summarize the description of hardware errors identified in our fault injection campaign in Table 2.4. While these errors are not critical at the system level (i.e., they do not cause network deadlocks or system crashes), they contextualize application-level events observed during workload executions.

Table 2.4: Hardware Errors Observed in Fault Injection Analysis

Error	**Description**
SSID Response Request-Timeout	Logged due to a failure in the HSN, due to a failed link, connection, or node. Also logged due to severe network congestion if congestion protection mechanisms fail.
SSID Response Protocol	Logged during network re-routing phase of quiescence. Also logged when unregistered memory (e.g. memory of terminated application) is used for network transfers.
SSID Detected Misrouted Packet	Logged during blade warm swap procedures. Routing algorithm deliberately misroutes packets destined for a blade being warm-swapped.
ORB RAM Scrubbed Upper/Lower Entry	Logged when a network request times out and is removed from the output request buffer (ORB). Indicates a problem with the Gemini HSN. Transient error which indicates a network deadlock if continuously generated.
BTE Descriptor Invalid	Logged when a block transfer engine descriptor (BTE) is invalid. This error is seen when a node is being rebooted and the Gemini is targeted before boot completion.
RMT Request for Invalid Descriptor	Logged when a receive message table (RMT) request descriptor is invalid.

2.9.2 Application Error Descriptions

In our fault injection experiments, the *HPCArrow* Workload Manager is configured to redirect application output and errors to application logs for analysis offline. The verbosity of these logs varies depending on the application and runtime framework used. In our analysis, we observed several critical errors that prevented an application from making forward progress. We summarize the observed errors in this section.

- `Assertion msg_nbytes > 0 failed`: This message is logged by the Charm++ runtime framework upon encountering a 0-byte sized fast memory access (FMA) short message (SMSG) in the uGNI layer. The uGNI layer handles low-level communication between network ASICs and user-space software [54]. Such an error indicates that a network packet was dropped in-flight to its destination Gemini due to a failure in the communication path. In our experiments, we encounter this error upon link and connection injections.

- `DMAPP_RC_TRANSACTION_ERROR`: DMAPP is a communication library which supports a logically shared, distributed memory programming model. DMAPP provides remote memory access (RMA) between processes within a job in a one-sided manner [54].

  The `DMAPP_RC_TRANSACTION_ERROR` indicates that a network transaction completed with an error state, either a non-recoverable transaction error or a transient error such as network error, uncorrectable memory error or resource shortage [54]. The DMAPP library is used by the Unified Parallel C (UPC) runtime framework built on the PGAS model. In our experiments, we encounter this error upon link and connection injections.

- `RCA ec_node_failed event`: This error is generated when a node or blade fails with a hardware error such as a memory check error (MCE). Typically, the node is automatically marked down by the node health checker [55]. In our experiments, this error was observed when faults were injected into compute nodes and blades.

- `ioctl(GNI_IOC_POST_RDMA) returned error`: This message is logged by the Charm++ runtime framework upon encountering an invalid

argument to a remote direct memory access (RDMA) transaction. This error leads to application crashes and hangs. In our experiments, we encounter this error upon link and connection injections.

Besides critical errors preventing applications from making forward progress, we also observed network related messages logged to application outputs:

- *Network quiesced*: This message is logged when a network quiescence is in progress, in response to a link, connection or blade being taken out of service or added back in (i.e. during a warm swap). We observe this message logged in response to link, connection and blade injections.
- *Network throttled*: This message is logged when network congestion occurs and the network is throttled to prevent further congestion. We observe this condition when a link, connection or blade is taken out of service or added back in (i.e., during a warm swap).

2.9.3 Summary of Fault Injection Experiments

Our injection campaign consists of 84 fault injection and baseline experiments. Over the course of our campaign, 9 unique applications were executed a total of 462 times to provide realistic workload scenarios as we injected faults into system components. In total, 69 faults were injected into link, connection, node and blade system components. Table 2.5 summarizes fault injection experiments classified by runtime frameworks, applications, fault models and their outcome scenarios. Of particular interest is the PS-DNS application, which terminated prematurely when not targeted. This observation is discussed further in Section 2.9.5. Additionally, it is worth noting that we expect applications to terminate prematurely upon node and blade failures. Thus, the node and blade failure results reported are not out of the ordinary.

We also summarize events observed during each experiment at the system, network and application levels in a tabular format. As an example, we provide Table 2.6 generated by LogDiver that summarizes events observed during a link injection experiment. The numbers of occurrences of system events and errors are reported to identify any missing recovery phases during our experiments.

Table 2.5: Summary of Fault Injection Campaign by Target Application, Fault Model and Outcome Scenarios. Descriptions of Runtime Frameworks and Fault Models may be found in Sections 2.8 and 2.5 respectively.

Application		**Workload Size(s)**	**Scenarios**			
			Completed Successfully	**Crash**	**Hang**	**Total**
MPI	PSDNS	8, 32	3 Link	0	0	3
	MILC	16	2 Link	1 Link	0	3
	AWP-ODC	4, 32	3 Link 3 Connection	3 Node/Blade	0	9
Charm++ (SMP)	NAMD	16	0	0	3 Link	3
	AMR	4, 32, 64	0	4 Link 4 Connection 5 Node/Blade	0	13
Charm++ (HugePages)	Kripke	4, 8	5 Link	1 Node/Blade	1 Link	7
	LeanMD	8, 16	1 Link	2 Connection 3 Node/Blade	6 Link	12
	AMR	2, 4	3 Link 2 Connection	3 Link 2 Node/Blade	1 Link	11
PGAS	UPC-FT	8, 32	1 Connection	3 Link 2 Connection 2 Node/Blade	0	8
		Total	23	35	11	**69**

Table 2.6: Example Report Containing Event Counts for a Link Injection Experiment

Entry	**Value**
Experiment ID	3
Experiment Type	Link
Target Component	`c0-0c2s7g0l52`
Application Set	5
Impacted Application	AMR SMP (64 XE/XK)
Application Status	Crash
Link Failed	4
Link Recovery Successful	4
SSID Request Timeout	635
SSID Response Protocol	62
Set Throttle Mask	4
Network Quiesced	2
Network Unquiesced	2
Network Throttled	4
Network Unthrottled	4
Warm Swap Successful	6
Link Auto Recovery Duration	37 seconds
Warm Swap Duration	38 seconds

2.9.4 Failure Scenarios

In our injection campaign, we study faults and recoveries targeting links, connections, nodes and blades. We discuss details of experiments with interesting outcomes for each fault model in the following sections.

2.9.4.1 Single Link and Single Connection Failures

For the single link failure and single connection failure fault models, we observed three scenarios for application resilience: (1) impacted applications making forward progress and terminating naturally despite link or connection failures, (2) applications utilizing impacted links or connections terminating prematurely, and (3) applications utilizing impacted links or connections resulting in hangs (i.e. not making forward progress).

In this section, we discuss system- and network-level events that occur during a connection failure experiment where an application utilizing the failed connection terminates prematurely. While we do not discuss the remaining scenarios, we apply the same analysis methodology to other experiments.

Figure 2.14 demonstrates the timeline of a connection failure experiment, where an impacted application crashes upon a connection becoming unavailable. In this experiment, application set 4 was executed (see Appendix C) and connection `c0-0c1s1g0` in the Y+ direction was targeted. This experiment specifically impacts the `UPC-FT` application running on 8 nodes, as this workload utilizes the targeted connection during injection and recovery.

In the top plot of Fig. 2.14, three traffic curves are presented – one for the connection that a failure was injected into, a second for remaining connections utilized by the impacted application, and third for all other connections on the system. From time $T = 0$ to $T = 335$, we observe consistent traffic flow on all connections. At time $T = 333$ seconds, a connection failure is injected causing connection `c0-0c1s1g0` to fail in the Y+ direction. Next, at time $T = 335$ seconds, the automatic recovery process is triggered. This recovery process requires a route recalculation and network quiescence, which temporarily suspends network traffic flow on all connections until successful completion. At time $T = 372$ seconds, the automatic recovery completes by successfully masking the impacted connection (i.e. removing the connection from service). After completion of the automatic recovery process, traffic

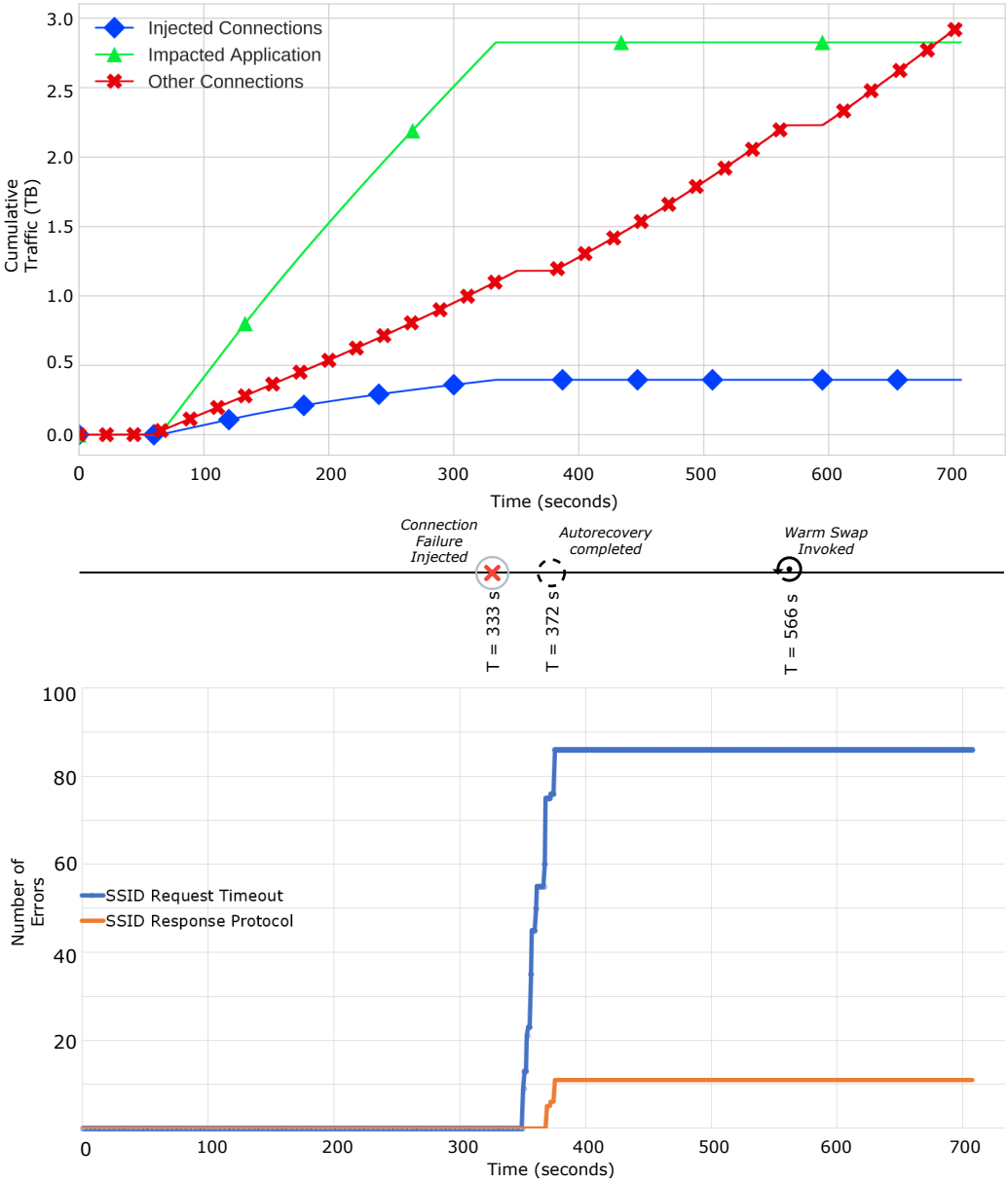

Figure 2.14: Single connection failure with impacted application terminating prematurely. Traffic plots (top), fault injection timeline (middle), and hardware errors (bottom) are displayed.

begins flowing only on connections utilized by non-impacted applications. Traffic does not flow on connections being utilized by the impacted application, indicating a network- or application-level error. At time $T = 566$, a manual connection restoration is invoked by the fault injector module of *HPCArrow*. This manual restoration process triggers a warm swap, which initializes all links on the masked connection, recalculates new routes and quiesces the network. Once again, the network quiescence suspends traffic until the warm swap procedure is successful. Upon a successful warm swap, traffic begins flowing on connections utilized by non-impacted applications.

Analysis of hardware error logs reveals 86 `SSID Request Timeouts` and 11 `SSID Response Protocol` errors, indicating dropped network packets. In the bottom plot of Fig. 2.14, we observe a sudden spike of errors when the connection failure is injected at $T = 333$ seconds. Analysis of application output logs reveals network quiescence due to automatic connection recovery and manual warm swap procedures. During network quiescence, additional `SSID Response Protocol` errors occur due to the impacted application not being able to send packets between its assigned compute nodes. Additionally, the `UPC-FT` workload reports a `DMAPP_RC_TRANSACTION_ERROR` and terminates upon failure injection. We discuss the cause behind this error in Section 2.9.5.3.

2.9.4.2 Node and Blade Failures

For the node failure and blade failure fault models, we observed applications placed on impacted nodes and blades terminating prematurely. This behavior is expected as applications cannot make forward progress upon failure of a compute component. In the case of node failures, no automatic recovery process is invoked as the HSN is unaffected. This allows non-impacted applications to continue without disruptions in traffic flow. Upon a blade failure, however, two Gemini ASICs housed on the impacted blade are taken out of service. This causes failures in links connected to these Geminis as well as links connected to other ends of physical links on the failed blade. An automatic recovery process is triggered to mask the impacted blade and links, which causes the entire network to be quiesced, new routes installed and network finally unquiesced. Meanwhile, traffic on other connections is paused until the automatic recovery process completes successfully. In our

experiments on the JYC testbed, we observed blade recoveries executing for approximately 30 seconds. However, we expect this number to increase significantly for larger systems, as demonstrated in a similar fault injection campaign on a larger Cray XE system at LANL [23].

We present an example blade failure experiment in Fig. 2.15 to highlight traffic patterns and system-level events during injection. In this experiment, we target blade `c0-0c2s5` while running application set 1 (see Appendix C). This experiment specifically impacts the `AWP-ODC` application running on 32 nodes, as this workload was placed on the targeted blade during injection and recovery. The impacted application terminates prematurely upon blade failure and automatic recovery.

In Fig. 2.15, we observe a series of cascading link and connection failures due to a blade being taken out of service. Two Gemini ASICs (`c0-0c2s5g0` and `c0-0c2s5g1`) housed on the targeted blade are taken offline, causing outgoing links to fail. Link failures cascade to links connected to other ends of physical links on the failed blade. In total, 80 link failures across eight Geminis and four blades are observed.

In the top plot of Fig. 2.15, three traffic curves are presented – one for the blade that a failure was injected into, a second for remaining blades utilized by the impacted application, and third for all other blades on the system. From time $T = 0$ to $T = 355$, we observe consistent traffic flow through all blades. At time $T = 341$ seconds, a blade failure is injected causing Geminis `c0-0c2s5g0` and `c0-0c2s5g1` to fail in all directions. Next, at time $T = 355$ seconds, the automatic recovery process is triggered. This recovery process requires a route recalculation and network quiescence, which temporarily suspends network traffic flow on all connections until successful completion. At time $T = 408$ seconds, the automatic recovery completes by successfully masking the impacted blade (i.e. removing it from service). After completion of the automatic recovery process, traffic begins flowing only on blades utilized by non-impacted applications. Traffic does not flow on connections being utilized by the impacted application, indicating a network- or application-level error. At time $T = 865$, a manual blade restoration is invoked by the fault injector module of *HPCArrow*. This manual restoration process triggers a warm swap, which initializes all links on the masked blade, recalculates new routes and quiesces the network. Once again, the network quiescence suspends traffic until the warm swap procedure is suc-

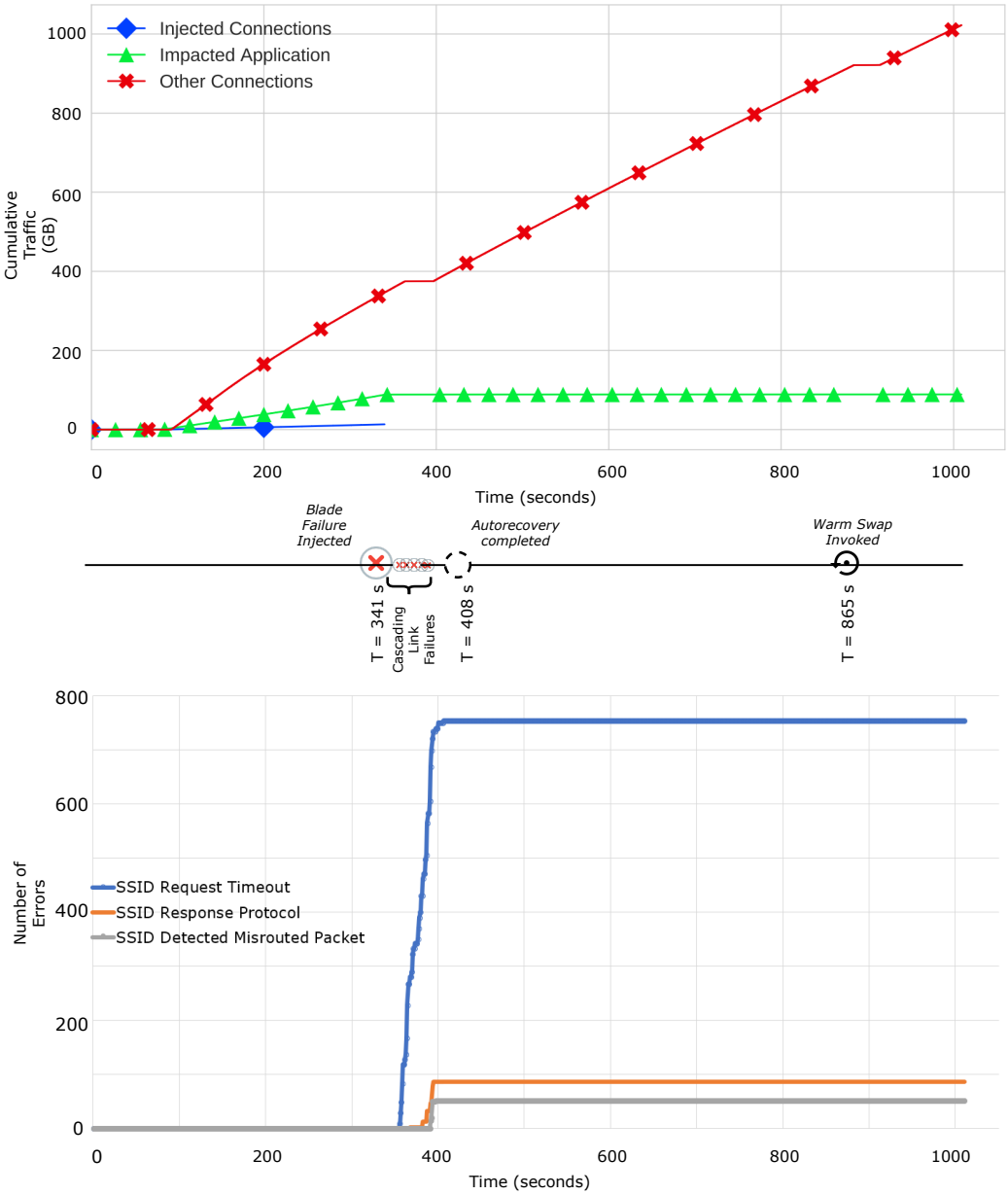

Figure 2.15: Blade failure with impacted application terminating prematurely. Traffic plots (top), fault injection timeline (middle), and hardware errors (bottom) are displayed.

cessful. Upon a successful warm swap, traffic begins flowing only on blades utilized by non-impacted applications.

Analysis of hardware error logs reveals 754 `SSID Request Timeouts`, 87 `SSID Response Protocol`, and 58 `SSID Detected Misrouted Packet` errors, indicating dropped and misrouted network packets. In the bottom plot of Fig. 2.15, we observe a sudden spike of errors when the blade failure is injected at $T = 341$ seconds. The number of errors continuously increases until the impacted blade and network links are masked out of service by the automatic recovery procedure. In output logs of the `AWP-ODC`, an `RCA ec_node_failed event` is reported due to the workload not being able to communicate with compute nodes on the failed blade, causing it to terminate immediately. Applications not impacted by the blade failure continue to execute and terminate naturally at the end of their runs.

2.9.5 Application-Level Resilience

The results obtained from our fault injection campaign have helped in providing a better understanding of failure scenarios previously observed in [21]. While all system and network recovery procedures completed successfully within expected time windows, in some cases application workloads were severely affected due to crashes and hangs. We summarize our observations for the MPI, Charm++ and PGAS runtime frameworks in this section.

2.9.5.1 Message Passing Interface (MPI)

For MPI applications, we observed premature terminations for the MILC and PSDNS applications. Upon link and connection failures, MILC reported a number of `SOURCE_SSID_SRSP:REQUEST_TIMEOUT` messages due to a `GNI_RC_TRANSACTION_ERROR`. This message indicates that network packets were dropped or there was an error in processing after data transaction [54], which prevented the application from making forward progress.

The PSDNS application terminated prematurely when we targeted links on chassis where the job was not placed. In the output logs of PSDNS, we observed `UNRECOVERABLE library` errors, where the program indicated that it was unable to request more memory space. It is unclear why this error is generated in response to a failure in a link not being utilized by

PSDNS. This behavior is reproducible, as we observed it in multiple runs of the same experiment. At the same time, we did not observe any premature terminations in baseline runs of the PSDNS application.

2.9.5.2 Charm++

For Charm++ applications, we observed jobs terminating prematurely and hanging due to link and connection failures. The LeanMD application terminated prematurely when `ioctl(GNI_IOC_POST_RDMA)` returned an error due to encountering an invalid argument to a remote direct memory access (RDMA) transaction. In some experiments, we observed the LeanMD application not making forward progress upon link and connection failures. The `ioctl(GNI_IOC_POST_RDMA)` message was continuously logged until the job was terminated by the application-level scheduler when its maximum allowed time (walltime) had elapsed.

The AMR application terminated prematurely upon a link or connection failure, reporting a `Assertion msg_nbytes > 0 failed` error due to an SMSG send failure. This error indicates that a network packet was dropped in-flight to its destination Gemini caused by a failure in the communication path.

The Kripke application did not make forward progress upon injection of a link or connection failure. No error message was reported in its output logs. The application-level scheduler continued reporting the job as "running", with no indication to the user that forward progress was not being made. The job was eventually terminated by the application-level scheduler when its maximum allowed time (walltime) had elapsed.

2.9.5.3 Partitioned Global Address Space (PGAS)

Upon a link or connection failure, the UPC-FT application reported a `DMAPP_RC_TRANSACTION_ERROR` and terminated prematurely. This message indicates that network packets were dropped or there was an error in processing after data transaction [54]. This scenario is expected as PGAS applications are known to be vulnerable to network failures and recoveries [56]. PGAS applications rely on ordered delivery capabilities and atomic memory

operations, and any disruption to packet delivery and ordering will not allow the application to make forward progress.

2.9.5.4 Discussion

From our fault injection analysis, we identified that application behavior in the face of network-level failures is based on the runtime framework used. Across all frameworks, we observe applications terminating prematurely when link or connection failures are injected. Moreover, applications utilizing the Charm++ runtime framework appear to hang upon network-level failures, with no indication to application-level schedulers. Such a lack of notifications to services running throughout the system stack leads to waste of system resources and reduces the overall efficiency of HPC systems.

Premature terminations of applications across all frameworks are caused due to drops of incoming packets, causing applications to immediately throw errors upon failed assertions. This scenario can be avoided by a network-level broadcast mechanism that causes application threads to pause computation upon network failures. An impacted thread could broadcast an emergency stop/pause message to other threads in the current application, causing them to temporarily halt computation until network connectivity is restored. This will allow applications to recover from failures immediately, without having to revert to previous checkpoints and lose application state. Additionally, application designers could implement a packet retransmission mechanism. Gemini ASICs already handle in-flight packets with a sliding window protocol. Identifying missing or dropped packets in the sliding window could allow applications to re-request packets from their source, allowing them to make forward progress when failures are masked and network connectivity is restored.

At the network level, a two-phase commit protocol could be implemented to handle lost transactions [56]. With the two-phase commit protocol, nodes can identify if network transactions have completed before attempting to make forward progress. At the system level, instrumentation and detection mechanisms could be implemented to detect processors not making forward progress. Such detection could be used to notify application-level schedulers of job hangs, which would allow affected workloads to be terminated early and free up system resources.

2.10 Future Work - Cray XC Platform and Aries Interconnects

The next phase of this project is to execute a similar fault injection campaign on Voltrino and Mutrino, two Cray XC machines utilizing Aries interconnects at Sandia National Laboratories. However, due to differences between Gemini and Aries interconnects, our injection tools and analysis methodologies would require modifications. While system and network architectures may differ, our general approach to validating network failures and recovery operations on HPC systems remains applicable to newer generations of Cray platforms.

2.11 Conclusion

In this chapter, we presented a fault injection campaign on the Blue Waters JYC testbed at NCSA, with a newly developed software fault injection tool to induce failures on link, connection, node and blade components on the Cray XE platform. Analysis of results from our fault injection experiments reveals a lack of fault tolerance at the application layer for MPI, Charm++ and PGAS runtime frameworks. We also observe that the assumption of a lossless Gemini network does not hold under failures. Additionally, we note a lack of notification reporting to applications and application-level schedulers in the event of system and network failures. In order to improve the fault-tolerance of their applications and make forward progress in the face of system- and network-level failures, application designers must take into account software-level resiliency mechanisms and not rely on hardware and network guarantees.

CHAPTER 3

FAULT INJECTIONS ON SMART POWER GRID NETWORK ENVIRONMENTS

The programmability of software-defined systems, like smart electric power grids, has allowed for the widespread adoption of software-defined networking (SDN) in industrial control networks. However, the increased complexity of such networks gives rise to previously unconsidered resiliency issues. This chapter investigates the impact of faults and attacks on SDN infrastructures when deployed in mission-critical environments and proposes an application- and data plane-based solution to pro-actively monitor system state and enforce user defined policies. We evaluate our fault models on a smart power grid simulation running Raincoat, an SDN application that reroutes and spoofs network traffic to thwart attackers. We show that under certain faults, (1) applications orchestrating the network become ineffective and (2) periodically monitoring the state of the network can identify faults or attacks before they manifest as failures. The results obtained from this work can aid in enhancing the resiliency of future SDN applications.

3.1 Introduction

The communication network is a fundamental component of a smart power grid infrastructure, connecting grid devices that span a wide geographic area. Traditionally, grid communication networks have used the standard Internet Protocol (IP) networking paradigm, but with the increasing complexity and need for more efficient utilization and distribution of power in today's world, this traditional, non-adaptive network model cannot provide the desired programmability and reconfigurability. This need for more dynamic and reactive network functionality has led to considerations of utilizing the software-defined networking (SDN) paradigm in place of the traditional IP stack [57].

Software-defined networking is an architectural approach to manipulating computer networks, enabling direct programmability and dynamic reconfigurability of network control and resources. The key to the programmability of SDN networks is the decoupling of the packet routing process from the forwarding functions of network packets, which are typically performed together on a single network component. This decoupling ultimately allows network programmers to design and deploy network control algorithms more easily and not be constrained by proprietary implementations of network switches and controllers. While still a nascent approach, SDN has seen commercial deployment in more general network environments. For example, in the early 2010s, Google deployed B4, a private SDN-based wide area network to globally connect its data centers and efficiently improve utilization [58].

While SDN provides the agility and reconfigurability demanded by modern smart grid technologies, system reliability is also a key consideration and challenge. Critical infrastructures like power grids must be able to maintain their most crucial services in the face of natural failures and malicious attacks and be able to recover quickly. Because of the physical separation between control decision making and forwarding operations, SDN network intelligence is centralized in a single network component, thus posing a single point of failure. In order for the power grid industry to consider adopting SDN into its technologies, understanding system resilience and risks that come with applying SDN to smart grids is crucial and necessary.

This chapter investigates the impact of faults and attacks on the data acquisition network of a smart grid managed by SDN. Unlike previous fault injection studies, we specifically focus on the faults that introduce "silent errors" (i.e. errors that allow the executions of SDN applications to continue, but incorrectly change their functionality). In order to identify and mitigate silent errors, we propose an application- and data plane-based monitoring solution (refer to Section 3.2.1 for details on SDN architectures) that periodically verifies data plane integrity and detects violations of user defined policies. While we evaluate our solution with a case study on the smart power grid, the ideas presented in this work can be extended to software-defined networks deployed in other mission-critical environments.

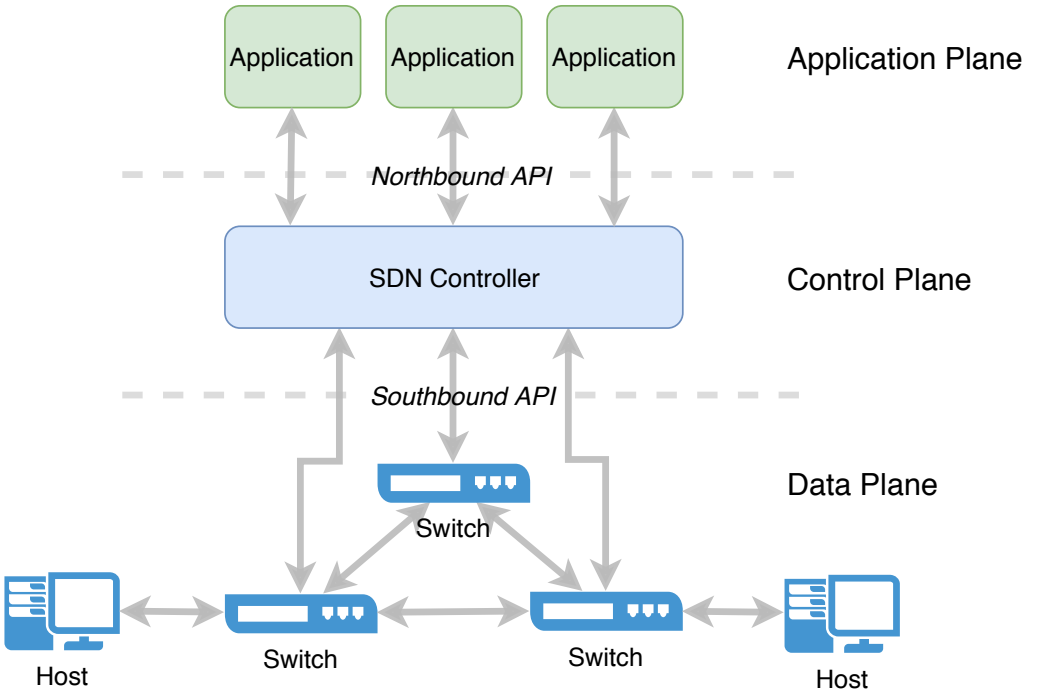

Figure 3.1: SDN architecture overview, showing the application plane (top), control plane (middle), and data plane (bottom).

3.2 Background

In this section, we introduce concepts related to software-defined networking, power grid data acquisition networks and how SDNs are integrated into smart power grids.

3.2.1 Software-Defined Networking

The key principles of software-defined networking are the programmability and reconfigurability that it provides. These aspects are achieved by a simple architectural change: unlike traditional IP networks, the control layer of a software-defined network is decoupled from the data layer.

The general architecture of an SDN network, as shown in Fig. 3.1, can be divided into three planes or layers: the data plane, the control plane, and the application plane [59]. The data plane consists of network switches, the devices responsible for forwarding traffic. The control plane is the central entity of an SDN network, consisting of one or more controllers that maintain a global view of the overall network and centrally decide how to handle net-

work traffic. The application plane consists of various user applications (e.g. network management, quality of service optimization like load balancing, and system resilience enhancement) which need information about the network state. These applications also define the network policies which are enforced by the control plane and executed by the data plane. A benefit of the application plane is that applications can be written and provided by third parties, who are distinct from the routing or controller vendor. This allows network behavior to be programmed independently from the hardware switches or software controller. The application plane and the control plane communicate via a northbound application programming interface (API) while the the data plane and the control plane communicate via a southbound API.

The northbound API contributes to the SDN principle of programmability. Through this API, controllers expose information about the network state, enabling network-aware applications that can dynamically react to the live state of the network. In traditional networks, this information is not shared with network applications. Additionally, applications can specify their networking needs to the controllers. In traditional networks, this communication is limited and static. At worst, traditional networks have to infer application needs via traffic analysis and predict the sufficient resources to allocate. In SDN, this information can be communicated easily and directly.

The southbound API contributes to the SDN principle of reconfigurability. The API is commonly standardized by the OpenFlow communications protocol [60], one of the first standards for software-defined networking. Like traditional switches, an OpenFlow switch uses one or more flow tables to route incoming packet sequences. Each packet is matched to an entry in these flow tables, called a flow rule, which determines how packets should be processed and forwarded. These flow rules are dynamically created, modified, or removed by the controller and network applications. In turn, the switch communicates information about traffic flow it tracks within its tables. This ability to dynamically generate and reconfigure flow rules proves to be a very attractive feature of SDN.

3.2.2 Power Grid Data Acquisition Network

Supervisory control and data acquisition (SCADA) systems are commonly used to transfer information between devices connected to networks of power grids, gas pipelines and waste water control systems [61]. The SCADA system of a power grid network, shown in Fig. 3.2, consists of:

- A control center, responsible for communicating and acquiring measurement data from remote devices on the network. The control center is staffed by human operators and consists of computers and networking equipment.
- Substations at remote sites, responsible for collecting and aggregating measurement information of their respective sites and responding to collection requests from the control center.
- Wide-area network(s), to connect substations at remote sites to the control center. The wide-area network (WAN) consists of network switches and carries information using the DNP3 and Modbus protocols over TCP/IP [62, 63].

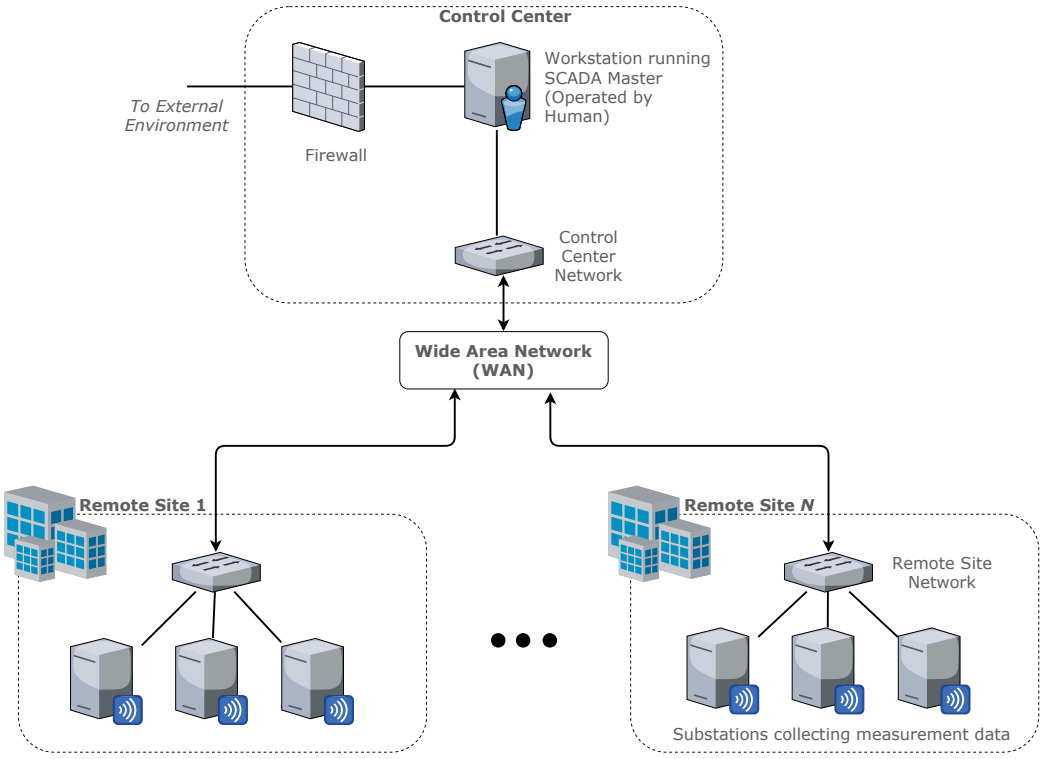

Figure 3.2: SCADA system of a power grid network

3.2.3 SDNs in Smart Power Grids

The programmability of software-defined networking provides several benefits in smart grid communication environments. By orchestrating network switches with software applications, grid operators can improve functionality and enhance resilience with more complex algorithms. The adoption of SDN in smart grid would require augmenting the existing network infrastructure with SDN controllers and applications specifically developed for smart grid operation. Figure 3.3 demonstrates an example smart grid communication network augmented with SDN controllers. Multiple SDN controllers may be set up at remote sites to manage their respective local area networks. Applications running on multiple SDN controllers across the entire grid are usually synchronized with a fixed initialization parameter (e.g. a deterministic seed like nearest hour). Further details of SDN adoption and implementation on smart grids may be found in [64].

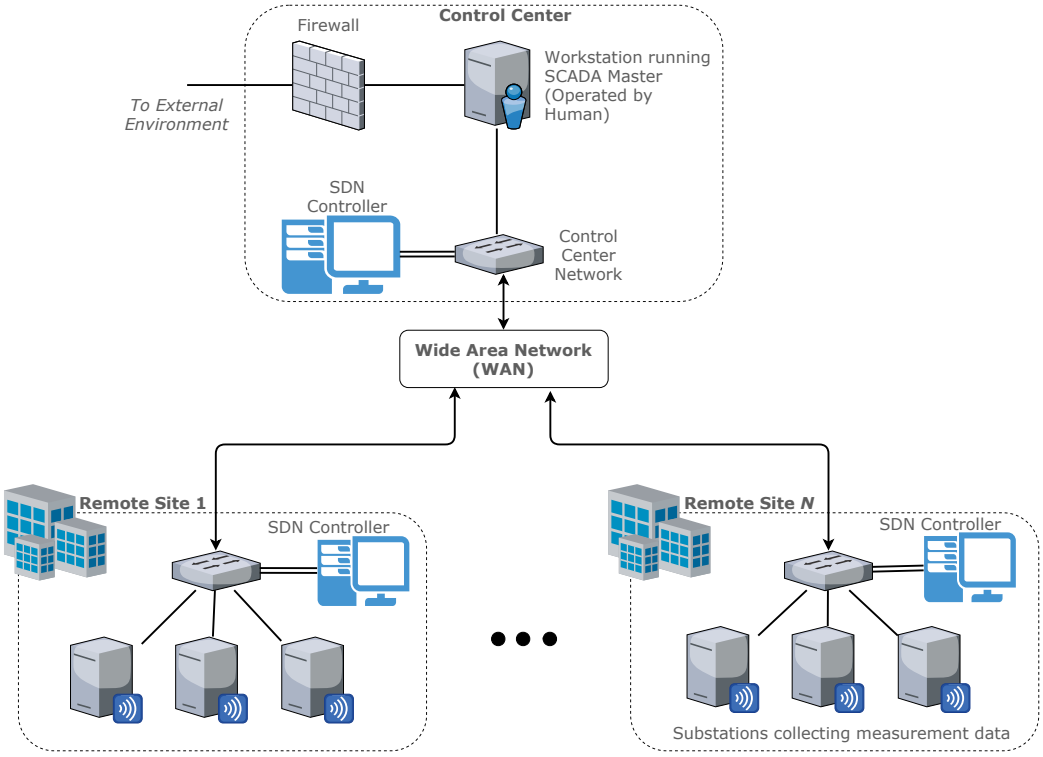

Figure 3.3: SCADA system of an SDN-managed power grid network. SDN controllers manage local-area networks at control center and remote sites.

3.3 Fault Models

In this study, we focus on failures which may be caused by accidental events (e.g. single event upsets at the bit-level) or induced by malicious actors on-purpose (e.g. denial of service attacks). Such failures are silent in nature and are difficult to detect in a timely fashion.

Our fault models target the data plane of a software-defined network and are described in the following subsections. We specifically study faults that occur in switches following the OpenFlow switch specification v1.5.1 established in [65], and involve corruptions within fields of flow table entries (Section 3.3.1), packet processing pipelines (Section 3.3.2), and malicious actions against the network infrastructure (Section 3.3.3). In this work, we do not inject faults or replay attacks within the SDN controller, as different implementations of controller software exist. From a security viewpoint, if an adversary has compromised a central controller or an OpenFlow link between controller and switches, he or she gains complete visibility into the data plane.

3.3.1 Flow Table Entry Corruptions

OpenFlow switches process incoming packets by matching packet headers against installed flow table entries. Packets that do not have entries installed in any flow tables within a switch are forwarded to the SDN controller via `PacketIn` requests for further processing or dropped entirely. Each flow table entry consists of:

- Header (Match) fields, to match incoming packets against
- Priority, to establish matching precedence of the flow entry
- Counters, to track information about the flow and update with every matching packet
- Instructions (Actions), to apply to and handle matching packets
- Timeouts, to evict an entry after a fixed or idle duration
- Cookie, to identify a flow to a querying controller

Table 3.1: Flow Table Entry Action Fields [65]

Instruction (Action)	**Behavior**
Output all ports	Send the packet out all interfaces, not including the incoming interface.
Output Controller	Encapsulate and send the packet to the controller.
Output Local	Send the packet to the switch's local networking stack.
Output Table	Perform actions specified in flow table. Only for packet-out messages.
Output Ingress Port	Send the packet out the input port.
Output Normal	Process the packet using the traditional forwarding path supported by the switch.
Output Flood	Flood the packet along the minimum spanning tree, not including the incoming interface.
Enqueue	Forward a packet through a queue attached to a port.
Drop	Drop the incoming packet.

A complete list of instructions (actions) targeted in this study is specified in Table 3.1. While switch designers may encode these actions at the hardware level with different bit-level representations, we assume that bit corruptions to these fields cause one action to be interpreted as another. Furthermore, due to our fault injection mechanism, a corrupted flow always contains fields with valid interpretations. In hardware switches, however, corrupt flows may contain fields with invalid representations. In such cases, the behavior of the hardware switch logic is undefined.

Bit-level corruptions of flow table entry fields could cause unintended packet routing behavior. For example, corruptions in header fields could cause packets to be matched with a flow not intended for them. Corruptions in action fields could cause packets that should be dropped or forwarded to the controller to be routed out on network ports. Additionally, attackers could exploit potential vulnerabilities at the OS/controller layer to manipulate OpenFlow switches and add, drop or modify entire flow table entries.

In this work, we specifically inject corruptions into action fields to change their interpretation to another valid action or cause the entire flow entry to be dropped (see Fig 3.4). We specifically target action fields as corruptions would have adverse effects on packet routing. Such corruptions are silent and not detected until either (1) an error in the switch manifests as a failure elsewhere, or (2) installed flows are verified against a known set of valid flows

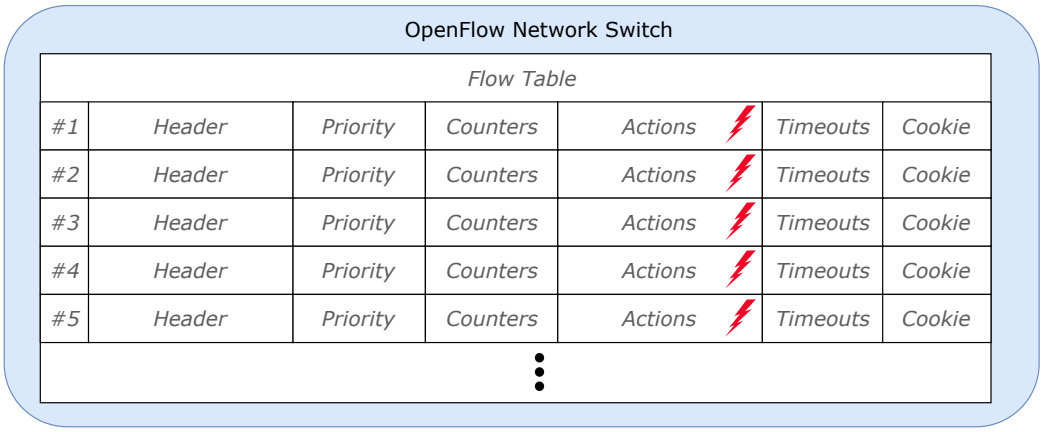

Figure 3.4: Flow table entry corruptions to action fields

by the SDN controller or an application.

3.3.2 Pipeline Processing Manipulations

OpenFlow switches process packets through one or more flow tables in a pipeline, with each flow table consisting of one or more entries. Packet processing occurs in two phases - the *ingress* pipeline, and *egress* pipeline (see Fig. 3.5). Upon matching with flow entries in one or more tables, actions defined with matching entries are added to an "action set" to be executed at the end of the current pipeline. Beyond executing actions in Table 3.1, a flow table entry can also direct matching packets to tables further down in the pipeline. Any further flow table entries matching with the packet being processed will have their actions added to the action set as well. All actions added to the action set are executed at the end of their corresponding pipeline, i.e. when a flow table entry does not redirect a packet further into the pipeline or when the last flow table has processed a packet.

However, if an incoming packet does not match with a flow table entry, actions associated with the table-miss flow entry (configured by the SDN controller) are executed. This could involve dropping the packet, forwarding the packet to the SDN controller or redirecting the packet to a flow table further down the pipeline.

A disruption in either of these pipelines could cause unintended packet routing behavior. Corruptions of pipeline forwarding actions could cause unnecessary redirects of network packets to other flow tables, where they

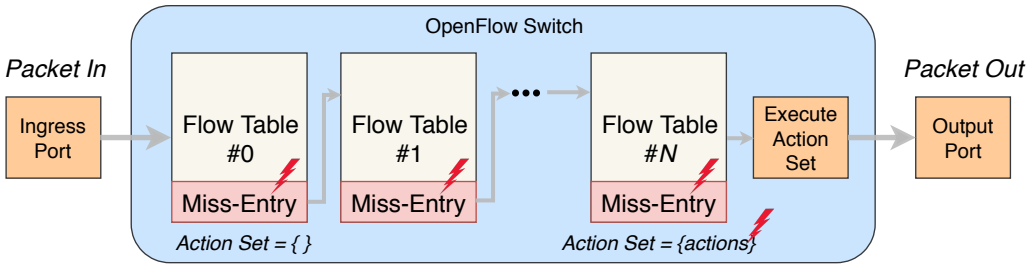

Figure 3.5: Simplified OpenFlow packet processing pipeline. Corruptions are injected as (1) packets traverse through pipelines, and (2) into action sets.

may not be handled correctly. Additionally, corruptions in the table-miss flow entry could cause packets to be dropped instead of continuing through the pipeline or being forwarded to the SDN controller.

In this work, we target corruptions of pipeline action sets and table-miss entry actions. We inject faults such that values contained in action sets or the table-miss entry take on valid representations different from the original (e.g. a "forward to controller" action is modified to "drop" packet instead).

3.3.3 Network Overloading

In the network overloading scenario, an adversary attempts to disable or cause degraded performance on OpenFlow switches by flooding them with an excessive number of requests from a connected host (see Fig. 3.6). Excessive requests forwarded to the controller saturate the southbound OpenFlow link between the switch and SDN controller or overload the controller itself. This type of attack is commonly known as `PacketIn` flooding [66]. Depending on controller and switch policies to handle an increased traffic load, this attack could lead to a denial-of-service scenario across the network.

In this work, we emulate a network overloading scenario by sending an excessive number of packets from multiple hosts connected to SDN managed switches. Network packets are generated from arbitrary network ports on hosts to ensure that flow table entries for these packets do not exist. This approach guarantees that `PacketIn` requests are sent to the SDN controller for each network packet arriving on a target switch.

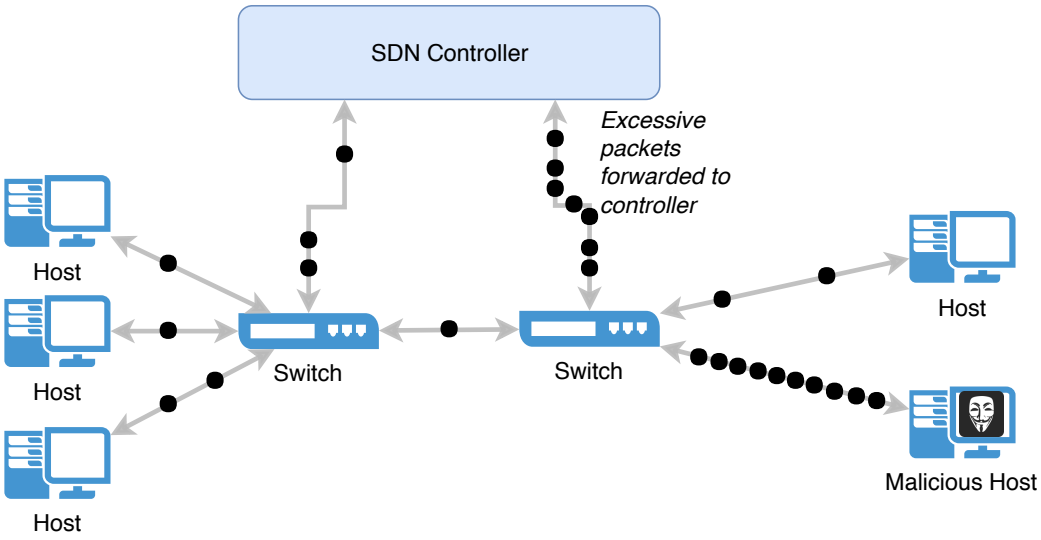

Figure 3.6: Network overload attack

3.4 Fault Injection and Analysis Framework

In order to systematically study the effects of faults, errors and failures on SDN infrastructures, we have developed an SDN fault injection work flow and analysis framework as part of this work. In this section, we describe our fault injector, analysis tools, and overall work flow applied to our fault injection experiments.

3.4.1 SDN Fault Injector

The SDN fault injector module runs as an independent entity on the simulation system and is composed of wrappers around the OpenVSwitch (OVS) administrative interface and traffic generation scripts. This module is responsible for the selection, timing and injection of faults into target network switches being managed by a centralized SDN controller. The injection mechanisms employed in our fault injector module are summarized in Table 3.2.

The fault injector is executed with superuser privileges on the simulation system. A user selects the fault model to execute, target components (i.e. network switches, hosts, flow tables and entries) and time delay. The injector is written in Python and can easily be integrated into larger applications for testing and development of SDN systems.

Table 3.2: SDN Fault Injection Mechanisms

Fault Model	**Injection Method**
Flow Table Entry Corruptions	`ovs-ofctl dump-flows` to read existing flow entries
	`ovs-ofctl del-flow` to delete existing flow entries
	`ovs-ofctl mod-flow` to update action fields of existing flow entries
Pipeline Processing Corruptions	`ovs-ofctl mod-flow` to update action field of table-miss entry
	`ovs-ofctl add-flow` to add flow with "drop" action to clear action set in pipeline
Network Overloading	Python traffic generation scripts to send UDP messages to target switches

3.4.2 Network Testbed

Our networking testbed is composed of Mininet [12], the Pox SDN controller [13], OpenVSwitch (OVS) switches [14], and Automatak DNP3 Applications [15]. Mininet is a network emulator that emulates hosts, links, switches, and controllers on a single machine by utilizing lightweight virtualization to enable a single system to host a complete network, running the same kernel, system, and user code. The emulated hosts behave like real machines and can run arbitrary programs. The virtual links, switches, and hosts behave like real hardware, i.e. packets are sent out on interfaces visible to an end user. Automatak DNP3 applications are used to generate network traffic between virtual hosts provided by Mininet. Figure 3.7 describes how the fault injector and simulated environment are set up. The fault injection and packet capture tools run as independent entities in the simulation environment. After fault injection is complete, packet captures are saved and analyzed offline. We describe our analysis framework in Section 3.4.3.

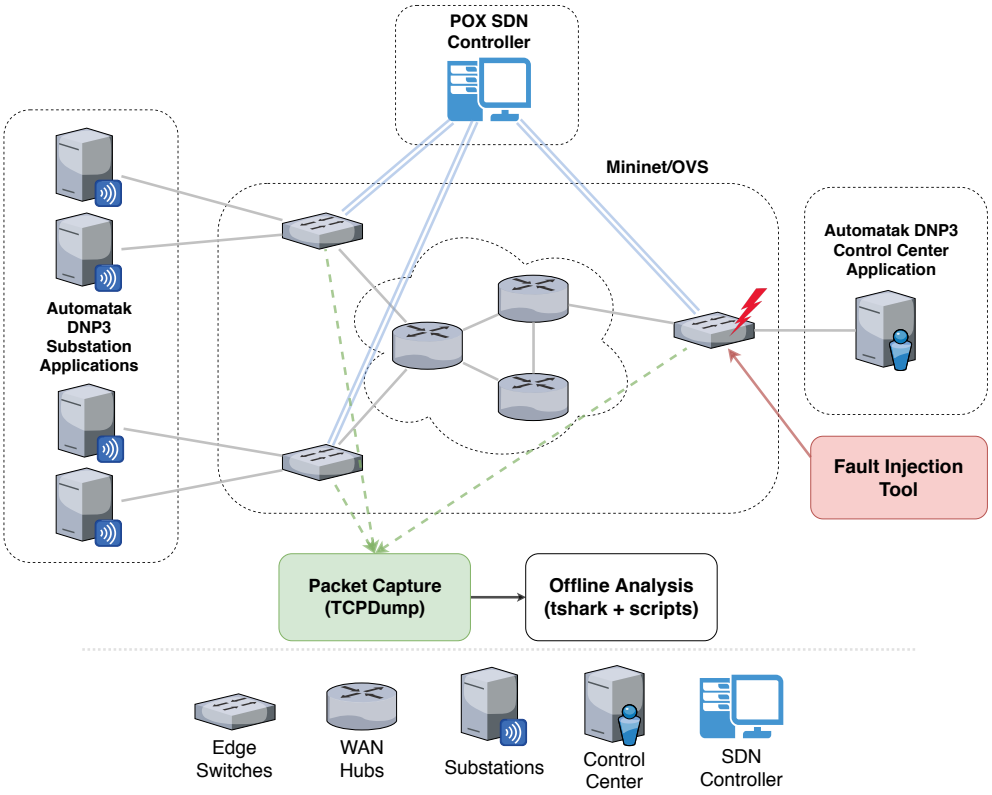

Figure 3.7: Fault injection and analysis framework

3.4.3 Analysis Framework

To study the effects of faults on network traffic, we capture packet traces for output ports on each network switch being managed by the centralized SDN controller using tcpdump [67]. Packet captures are analyzed offline using tshark (Wireshark's command line utility) [11], with additional logic written in Python. Analysis scripts are used to identify anomalies like misrouted or dropped packets, delays in transmission, and other interesting events in captured traffic flows (see Fig. 3.8).

3.4.4 Experiment Workflow

For each experiment, our fault injection workflow involves launching a simulated network topology using Mininet and a central SDN controller using Pox. Next, we launch DNP3 applications to generate traffic between virtual hosts on the network. We also launch the tcpdump utility to capture network traces across all switches managed by the SDN controller. The fault

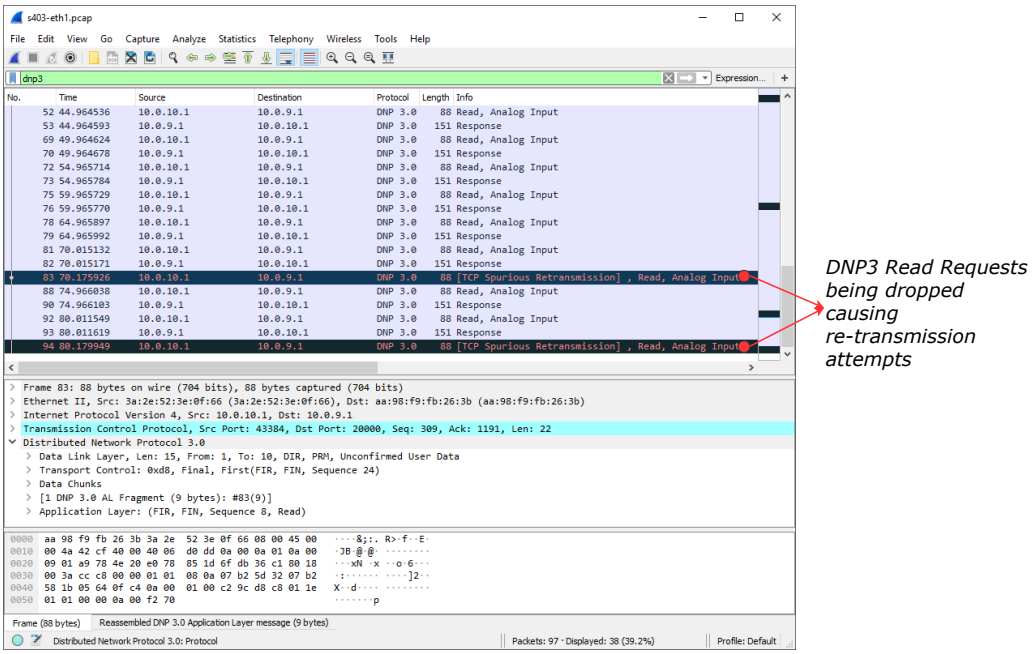

Figure 3.8: Analysis example, identifying events of interest in Wireshark

injection tool is used to manipulate the state of flow tables and inject faults into network switches. Once faults have been injected and failures observed, we shut down network captures and the rest of the simulation environment to prepare for the next experiment. Captured packet traces are analyzed offline using tools described in Section 3.4.3 to identify possible failure scenarios. Figure 3.9 summarizes our experiment workflow.

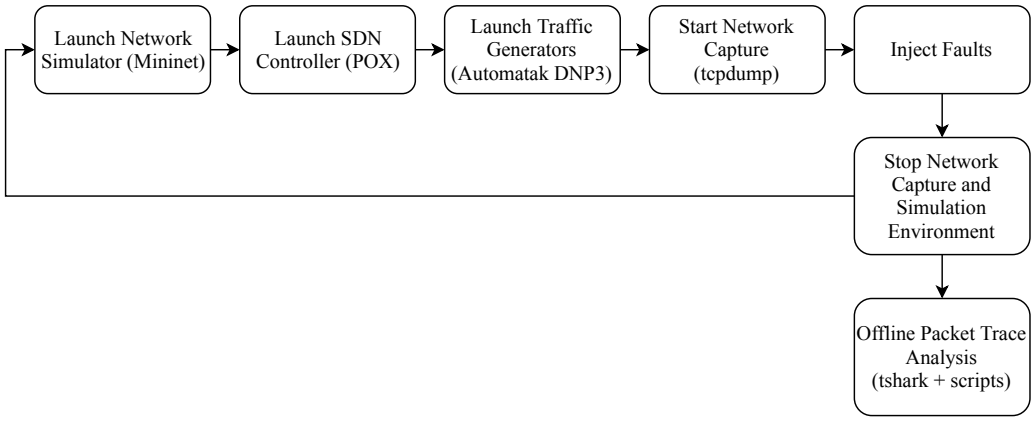

Figure 3.9: Experiment workflow for SDN fault injection

3.5 Case Study - Raincoat for Smart Power Grids

To demonstrate the impact on real world SDN infrastructures, we evaluate our fault models on a simulation of a data acquisition network of a smart electric power grid. The data acquisition network consists of a control center and multiple substations (end devices) connected over a wide-area network (WAN). Network switches at the first hop from the control center and end devices are designated as edge switches. The control center runs a supervisory control and data acquisition (SCADA) system which periodically queries end devices for measurements using the Distributed Network Protocol v3 (DNP3). Upon receiving a SCADA DNP3 read request, substations send their response containing power measurements over the network.

When designing such systems, the remote insider threat model is commonly considered. Attackers can bypass standard security measures (such as firewalls) and establish themselves on core computing devices on the network. Such a level of access allows attackers to inspect and potentially modify network traffic from the penetrated computing device. For example, the cyber attack on the Ukrainian power grid in December 2015 involved malicious actors establishing a foothold within the power grid network and remotely controlling operator stations to hijack SCADA systems. The attackers gained access to operator stations by harvesting credentials through spear phishing methods and caused a blackout that affected more than 225,000 residents [68].

In this case study, we inject faults into network switches administered by an SDN controller running the Raincoat application. Raincoat, proposed in [69], is an algorithm that randomizes data acquisitions in power systems with the goal of exposing and misleading attackers as they observe network traffic to prepare their attack strategies. At runtime, a single data acquisition operation issued by the control center to all substation devices is transformed into multiple rounds of data delivery. An example execution run of the Raincoat algorithm is demonstrated in Fig. 3.10.

Edge switches are configured to route DNP3 read requests from the SCADA master to the Raincoat application, which forwards these requests to each substation multiple times. For each round of responses, Raincoat randomly designates a subset of responding substations as "online" and allows their measurements to be forwarded to the controller. The remainder of responses

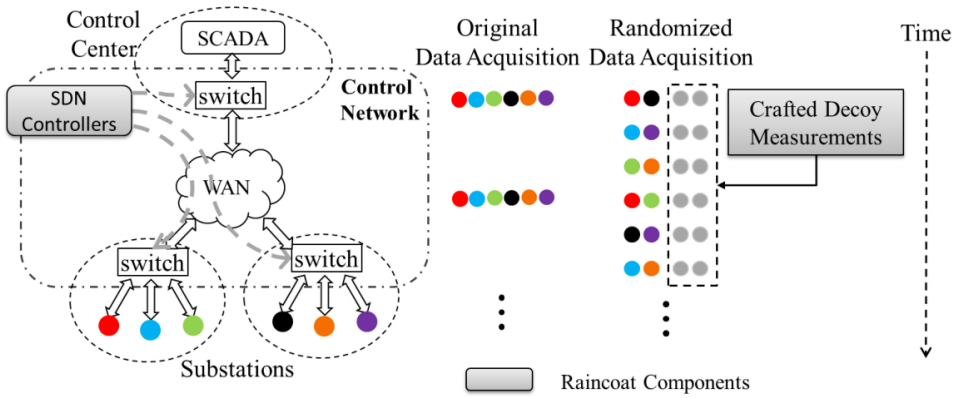

Figure 3.10: Example execution of the Raincoat algorithm [69]

from "offline" substations are intercepted and substituted with decoy measurements instead. Over the course of all responses to a single read request, Raincoat guarantees that at least one response per substation will be genuine. The SCADA system in the control center is synchronized with the Raincoat algorithm to extract genuine measurements, while the attacker observes both genuine and decoy measurements which are indistinguishable on the network. Further details of the Raincoat framework can be found in [69].

3.5.1 Fault Injection Campaign

We conducted our fault injection experiments on the network topology shown in Fig. 3.11. The topology consists of 9 substations, 8 edge switches, and one control center. Edge switches are managed by a single SDN controller (switch–controller links omitted from figure) running the Raincoat application. As described in Section 3.4, the network topology is simulated with Mininet, OpenVSwitch and the Pox SDN controller. The control center and substations are Automatak DNP3 applications running on virtualized Mininet hosts. In total, our campaign consists of 235 unique faults injections. We repeat our fault injection campaign three times to verify deterministic behavior of the simulated system under failures.

We summarize our experiment setup by fault model in Sections 3.5.1.1 – 3.5.1.3.

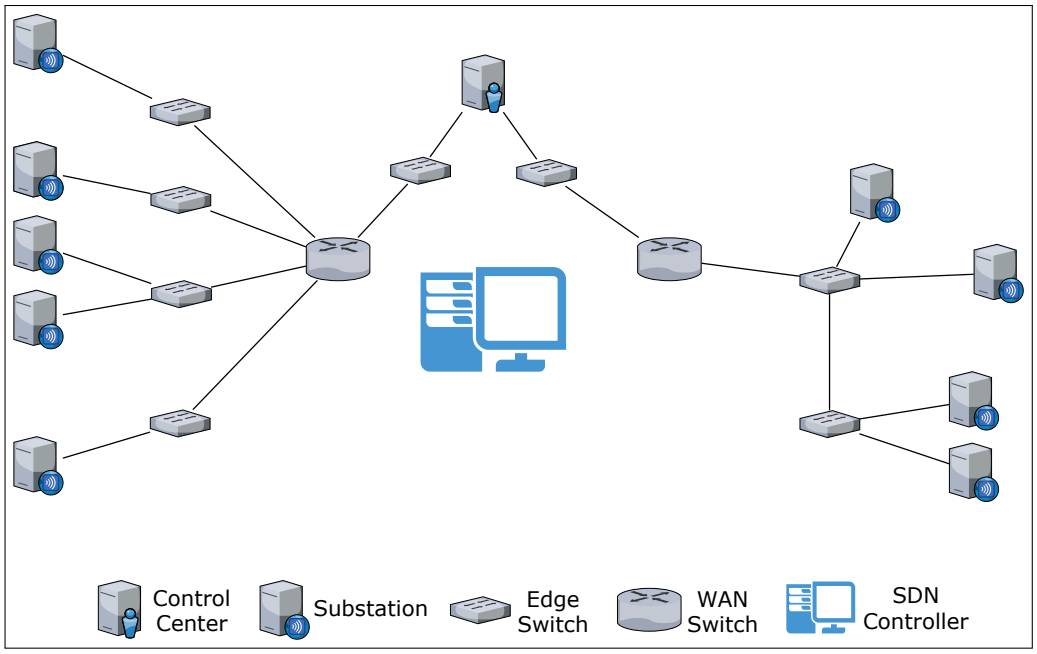

Figure 3.11: Network topology used in fault injection experiments

3.5.1.1 Flow Table Entry Corruptions

The Raincoat application installs three sets of flow table entries for each edge switch managed by the SDN controller (see Fig. 3.12). Packets from the SCADA master to substations are forwarded to the SDN controller at all times via a "Output to Controller" entry on all edge switches. Upon receiving these packets from edge switches, the Raincoat application selectively forwards DNP3 read requests to "online" substations. Additionally, responses from "online" substations are handled with flow table entries that simply forward packets out of the intended port, i.e. to the SCADA master in the control center. Responses from "offline" substations are handled by forwarding to the Raincoat application running on top of the SDN controller. The Raincoat application then obfuscates original measurements within an acceptable tolerance before forwarding to the SCADA master.

Fault injection experiments to study the flow table entry corruption fault model are summarized in Table 3.3. Descriptions of flow table entry actions may be found in Table 3.1. Each action is injected once per edge switch, giving us a total of 208 unique fault injections for the table entry corruption fault model.

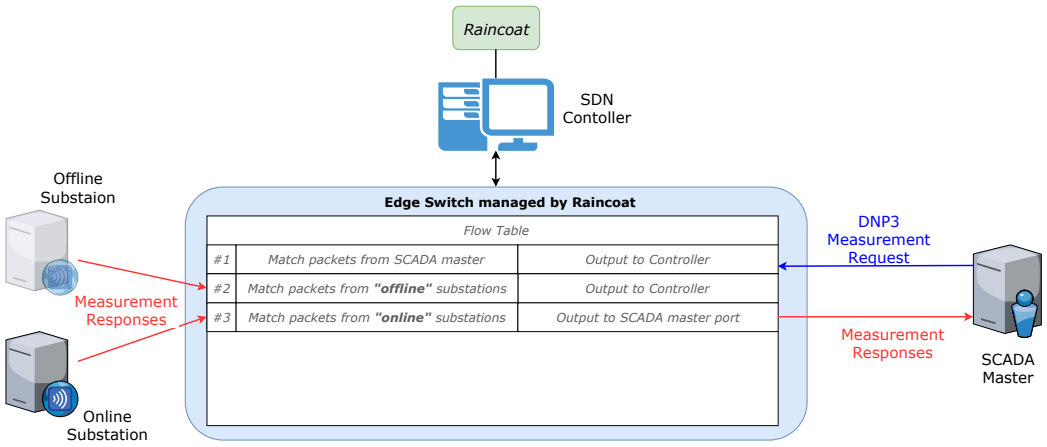

Figure 3.12: Flow table entries installed in a Raincoat managed edge switch. Requests from the SCADA master are shown in blue, whereas responses from substations are shown in red.

Table 3.3: Flow Table Entry Corruption Experiments

Flow Table Entry	**Original Action**	**Injected Actions**
SCADA Master to Substation	Output to controller	Drop
		Enqueue
		Output to {ALL, Local, Table, Ingress Port, Normal, Flood}
"Online" Substation to Master	Output to port	Modified port number
		Drop
		Enqueue
		Output to {ALL, Controller, Local, Table, Ingress Port, Normal, Flood}
"Offline" Substation to Master	Output to controller	Drop
		Enqueue
		Output to {ALL, Local, Table, Ingress Port, Normal, Flood}

Table 3.4: Pipeline Processing Corruption Experiments

Component	Initial Value	Injected Value
Action Set	Output Action(s), i.e. forward to controller OR output to SCADA master	Empty Set via "drop" action
Table-Miss Entry Action	Goto Table	Forward to controller
		Drop packet

3.5.1.2 Pipeline Processing Corruptions

While Raincoat does not utilize multiple flow tables in packet processing pipelines, we modify the flow table entry installation procedure to allow us to study corruptions in pipeline processing. Instead of installing flow entries in the first available flow table, we deliberately install entries in a table further down in the pipeline. Additionally, we configure table-miss entries to forward packets to the next flow table in the pipeline sequence via "Goto" actions. This ensures ingress packets are processed through multiple flow tables before the intended action is applied to them and allows us to inject corruptions into the pipeline. Our injection experiments related to the pipeline processing corruption fault model are summarized in Table 3.4. Each fault is injected once per edge switch, giving us 24 unique fault injections for this fault model.

3.5.1.3 Network Overloading

To simulate a network overloading scenario, we launch traffic generators on a subset of connected substations. These traffic generators continuously send an excessive number of network packets (as fast as the simulator allows) to connected edge switches. Depending on whether substations are considered "online" or "offline" by Raincoat and network switches, these packets are forwarded to the SDN controller or to the SCADA master in the control center. We conduct the network overloading experiment three times to verify deterministic and repeatable behavior.

3.5.2 Failure Scenarios

In our analysis of captured network traces, we identify faults manifesting as failures in the form of (1) SCADA timeouts, (2) measurement obfuscation failures, and (3) denial-of-service. These failure scenarios and their causes are described in Sections 3.5.2.1 – 3.5.2.3.

3.5.2.1 SCADA Timeout

In the SCADA timeout scenario, the SCADA master running in the control center receives an incomplete set of measurements within an acceptable period from substations responding to a data acquisition command. This happens when requests from the SCADA master are dropped en route to a substation, or when responses from a substation are dropped en route to the master. In some cases, packets from a substation eventually arrive at the master due to retransmissions, but beyond our expected time limit from when the read request is sent out.

3.5.2.2 Measurement Obfuscation Failure

In the measurement obfuscation failure scenario, packets from substations that should be treated as "offline" by the Raincoat algorithm are transmitted beyond edge switches into the wide area network instead. This occurs when packets are incorrectly routed through the network instead of being forwarded up to the SDN controller and applications. This causes genuine measurements from "offline" substations to become visible to an adversary observing network traffic, rendering the Raincoat algorithm ineffective.

3.5.2.3 Denial-of-Service

The Raincoat algorithm requires edge switches to forward SCADA read requests up to the SDN controller via OpenFlow `PacketIn` requests. However, flooding a large number of these requests saturates the OpenFlow channel between edge switches and the SDN controller. This causes other requests to remain pending or be dropped, leading to a denial-of-service scenario in which power grid data acquisition cannot continue. Normal functionality

of the network is only restored when a system administrator intervenes and disables the malicious host that is flooding the network.

3.5.3 Results

Failure scenarios that resulted from faults injected during our campaign are summarized in Table 3.5.

Raincoat installs flow table entries that redirect incoming packets to (1) the SDN controller or (2) out on a specific port (see Fig. 3.12). Upon injecting faults that modify flow table entries to forward packets out on all ports (or when flooding packets across the spanning tree), we observe traffic from DNP3 substations considered "offline" by Raincoat being forwarded to the control center, instead of being forwarded to the SDN controller for obfuscation. Modifying actions to output to "Table", "Ingress Port", "Enqueue", or "Drop" causes DNP3 packets to be dropped or sent back to the sender, instead of being forwarded to the control center or SDN controller. This eventually causes SCADA timeouts, where a full set of measurements is not received within an acceptable window of time (10 seconds in our experiment).

Additionally, we observe a number of SCADA timeouts (i.e. data acquisition requests pending for longer than 10 seconds) upon corrupting table-miss actions and action sets in processing pipelines. By clearing action sets as packets progressed through their ingress pipeline, packets are immediately dropped. By corrupting table-miss actions to "Drop", we are able to drop packets instead of continuing through the pipeline. We also observe a measurement obfuscation failure when a table-miss action is corrupted to "Forward to controller". The corrupt table-miss action sends packets from "online" substations to the Raincoat application instead of the SCADA master. Raincoat responds by obfuscating measurements contained in these packets before forwarding them to the SCADA master. However, this scenario can be avoided by distinguishing packets from "online" and "offline" substations within the Raincoat algorithm's control logic.

Finally, upon overloading the network, we are able to create a denial-of-service scenario where data acquisition could not continue due to the SDN controller being non-responsive. While the network is being overloaded, the SDN controller does not respond to `PacketIn` requests from edge switches.

Additionally, no new flow table entries are installed by the SDN controller. This causes DNP3 requests and responses to be dropped upon arriving at an edge switch once existing flows have expired.

Table 3.5: Results from Smart Power Grid Case Study

Injected Fault			**SCADA Timeout**	**Measurement Obfuscation Failure**	**Denial of Service**
Corrupt Flow Action	Output	All Ports		✔	
		Controller			
		Local			
		Table	✔		
		Ingress Port	✔		
		Normal			
		Flood		✔	
	Enqueue		✔		
	Drop		✔		
Corrupt Pipeline Action Set	Clear action set		✔		
Corrupt Table-Miss Entry Action	Forward to controller			✔	
	Drop packet		✔		
Table Overflow Attack					✔
✔indicates injected fault manifesting as failure scenario faults					

3.6 Resiliency Recommendations

The failure scenarios described in Section 3.5.2 can be overcome by leveraging data plane monitoring and metering features provided in newer versions of OpenFlow. In this section, we propose three approaches to mitigate failures described in our power grid case study.

3.6.1 Continuous Monitoring via SDN Applications

To identify silent corruptions before they manifest as failures, we propose an application-based solution that continuously monitors the state of the data plane and detects corruptions of flow table state. Our solution leverages features made possible by SDNs – network visibility and programmability – that are not achievable with traditional networking. This approach to network monitoring requires only minor changes to the northbound interface

between SDN applications and the controller. Instead of directly interacting with the controller, applications route their requests via an auxiliary monitoring application, as shown in Fig. 3.13.

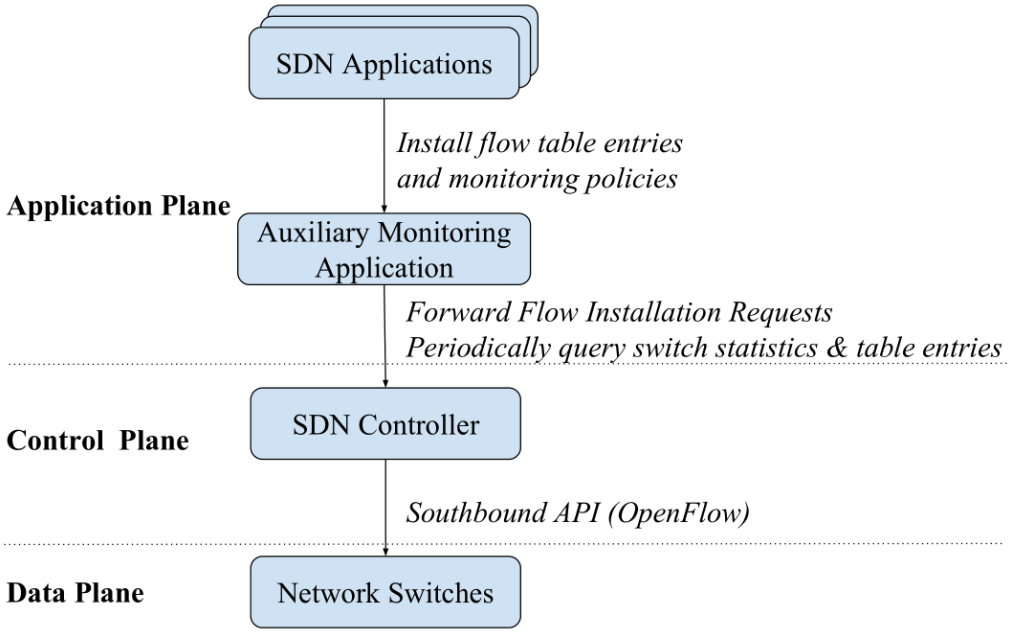

Figure 3.13: Data plane monitoring via auxiliary SDN application

Upon receiving flow installation requests from other SDN applications, the monitoring application forwards these requests to the controller. At the same time, the application caches flow information contained in these requests into a "golden" set. The integrity of flow table entries is periodically verified for each network switch. Any deviation from the cached "golden" set causes an alarm to be raised within the monitoring application. This periodic verification approach detects bit-level corruptions as well as malicious modifications of flow tables.

Inconsistencies in flow table states cause an alarm to be triggered within the monitoring application. A system administrator can respond to alarms raised by taking precautionary measures like scanning the network for unknown hosts or identifying faulty switches. While we implement our monitoring solution as a separate SDN application, its functionality can be integrated into existing applications without significant additional overhead.

3.6.2 Rate-Limiting via Metering Flows

Beyond monitoring flow table states, we can mitigate denial-of-service failure scenarios by metering traffic flows installed in flow tables. OpenFlow v1.3.0 and beyond provide a meter table on each network switch, which allows traffic to be managed based on the rate of incoming packets. We can configure per-flow meters to implement rate-limiting by constraining bandwidth and limiting excessive requests to controllers and other hosts on the network.

For example, we can define a simple meter flow rule for flow table entries on each network switch to limit the number of requests and responses to 1 packet per second. In OpenFlow, such a metering policy would be defined with the following parameters:

- Band Type: `drop`, which drops packets beyond the specified rate
- Rate: 1 packet (per second)
- Burst (granularity): 1 packet

This rate limit metering policy will limit SCADA requests to 10 per data collection round (10 seconds) by dropping packets in violation. While this approach will not prevent a malicious host from sending network packets, it can be used to protect against spamming switches and the SDN controller, potentially avoiding a denial-of-service attack.

OpenFlow v1.5.1 supports an additional band type called `dscp remark`. Specifying `dscp remark` in place of `drop` as the band type in a meter entry would allow the network switch to differentiate between senders and de-prioritize or drop packets from malicious or unknown hosts, while allowing genuine hosts to continue sending and receiving packets. However, the `dscp remark` band type is not supported in OpenVSwitch's implementation of OpenFlow at the time of this publication, and hence not evaluated in this work. Furthermore, there is limited support of meter tables on hardware implementations of OpenFlow network switches. Thus, this approach must be carefully evaluated with the available hardware and software stack when deployed in a mission-critical environment.

3.6.3 Failover Flow Table Entries

We can also leverage multiple flow tables in the OpenFlow processing pipeline to provide a failover or redundant flow table entry, if the original entry is corrupted or evicted (see Fig. 3.14). Installing a failover flow table entry further down in the pipeline allows switches to process packets when the original flow table entry is corrupted. When a network packet arrives at a switch with a corrupt flow entry, it will not be matched in the original table causing the packet to rely on the table-miss entry for routing decisions. The table-miss entry will force the packet to a table further down in the pipeline where it can be matched with a redundant failover entry instead and continue processing as intended by the application. This approach mitigates incorrect routing decisions by corrupt flow table entries on SDN-managed network switches.

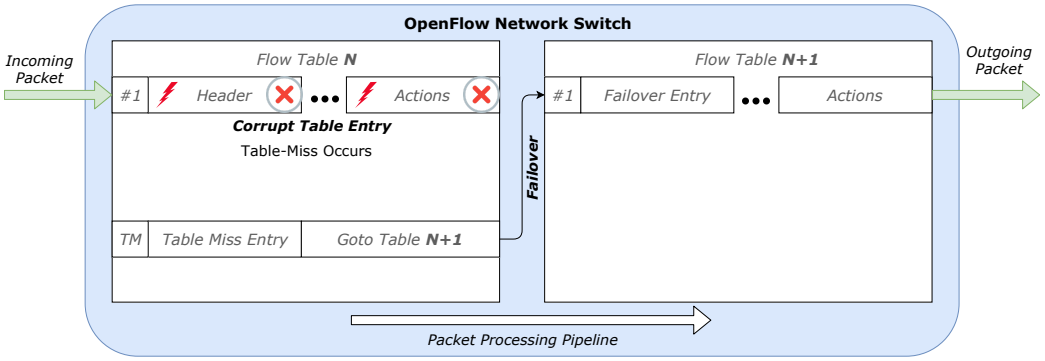

Figure 3.14: Pipeline processing with failover flow table entry

To implement this failover strategy, application designers will have to install redundant flows in separate tables on a network switch. Additionally, the table-miss entry in the flow table containing the primary flow will have to be configured to forward requests to a table containing the redundant flow. We tested our implementation of failover entries by evicting the primary flow in a target network switch. In all tests, the table-miss entry was activated which forwarded the ingress packet to the table containing the redundant flow. The ingress packet was processed as expected in all cases.

While this is a quick solution to handling corrupt flows, it comes with several caveats:

- The failover table entry is only utilized if the original flow table entry does not match with the incoming packet, i.e. this solution does not

apply to corrupt original entries with valid representations of other actions.

- The table-miss entry can only forward packets to tables further down in the pipeline. This failover strategy cannot be applied to the final table in a packet processing pipeline.

- This solution relies on the table-miss entry forwarding to the correct flow table in the processing pipeline. This may not occur for switches with widespread faults or deliberate attacks that target multiple flow tables or entries, where the table-miss entry is also corrupted.

3.7 Related Work

In this chapter, we covered faults and attacks against SDN infrastructures in mission-critical systems. A number of past works have evaluated the reliability and security of SDN systems. [70] introduces the idea of using SDNs in industrial networks to improve security. Existing works [71, 72, 73, 74, 75, 76] cover the impact of specific attacks against SDN systems. Additionally, [77], [78] cover attacks and fault injections in the control plane. However, unlike our work, [77] focuses on injecting attacks in the Southbound API between network hubs and SDN controllers, and [78] performs fault injections within specific implementations of SDN controllers. To the best of our knowledge, our work is unique as it considers failures at the OpenFlow networking switch level in industrial control systems, independent of controller or application.

3.8 Future Work

While this project lays the groundwork for understanding failures in mission-critical SDN infrastructures, several aspects are yet to be addressed:

- **Validating fault models with hardware testbeds**
  Our experimental approach made use of software simulators to emulate networking environments. Consequently, some of our assumptions may not necessarily hold true with industrial grade systems. Ideally,

we would like to validate our fault models with a physical networking testbed comprising of network components from various vendors.

- **Validating fault models with additional applications**
  In this work, we covered the case study of Raincoat on the smart power grid. Future work could validate our proposed fault models with more complex applications and identify additional failure scenarios.

- **Singular point of failure**
  In SDN architectures, the controller is a singular point of failure. Using the results of this work, we would like to ascertain whether automatic failures could cause the controller to fail and render the network unusable.

- **Attacks against SDN systems**
  Future work could identify whether an adversary could leverage the results presented in this work to cause attacks against SDN systems to succeed.

3.9 Conclusion

In this work, we demonstrated that our fault models can create an inconsistent state between the decoupled data and control planes on SDN systems. Such inconsistencies often lead to errors which manifest as failures in the environment that render the SDN controller and application ineffective. As demonstrated in the smart power grid case study, even applications with minimal interaction with the control plane are susceptible errors and failures in the data plane. However, equipped with results from this project, developers could integrate recommendations proposed in Section 3.6 to enhance the resiliency of their SDN applications and mitigate automatic and malicious failures.

REFERENCES

[1] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, Apr 1997.

[2] G. P. Saggese, N. J. Wang, Z. T. Kalbarczyk, S. J. Patel, and R. K. Iyer, "An experimental study of soft errors in microprocessors," *IEEE Micro*, vol. 25, no. 6, pp. 30–39, Nov. 2005. [Online]. Available: http://dx.doi.org/10.1109/MM.2005.104

[3] D. Chen, G. Jacques-Silva, Z. Kalbarczyk, R. K. Iyer, and B. Mealey, "Error behavior comparison of multiple computing systems: A case study using linux on pentium, solaris on sparc, and aix on power," in *2008 14th IEEE Pacific Rim International Symposium on Dependable Computing*, Dec. 2008, pp. 339–346.

[4] K. Pattabiraman, N. M. Nakka, Z. Kalbarczyk, and R. K. Iyer, "SymPLFIED: Symbolic program-level fault injection and error detection framework," *IEEE Trans. Comput.*, vol. 62, no. 11, pp. 2292–2307, Nov. 2013. [Online]. Available: http://dx.doi.org/10.1109/TC.2012.219

[5] R. K. Iyer, Z. Kalbarczyk, and W. Gu, *Benchmarking the Operating System against Faults Impacting Operating System Functions*. Wiley-Blackwell, 2008, ch. 15, pp. 311–339. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470370506.ch15

[6] K. S. Yim, Z. Kalbarczyk, and R. K. Iyer, "Measurement-based analysis of fault and error sensitivities of dynamic memory," in *2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*, June 2010, pp. 431–436.

[7] G. Jacques-Silva, B. Gedik, H. Andrade, K.-L. Wu, and R. K. Iyer, "Fault injection-based assessment of partial fault tolerance in stream processing applications," in *Proceedings of the 5th ACM International Conference on Distributed Event-based System*, ser. DEBS '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/2002259.2002292 pp. 231–242.

[8] S. Y. Lee, "Analysis of the impact of sequencing errors on blast using fault injection," M.S. thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois, 2013.

[9] H. Alemzadeh, D. Chen, X. Li, T. Kesavadas, Z. T. Kalbarczyk, and R. K. Iyer, "Targeted attacks on teleoperated surgical robots: Dynamic model-based detection and mitigation," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2016, pp. 395–406.

[10] C. D. Martino, S. Jha, W. Kramer, Z. Kalbarczyk, and R. K. Iyer, "Logdiver: A tool for measuring resilience of extreme-scale systems and applications," in *Proceedings of the 5th Workshop on Fault Tolerance for HPC at eXtreme Scale*, ser. FTXS '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2751504.2751511 pp. 11–18.

[11] "Wireshark," accessed: 2018-06-18. [Online]. Available: https://www.wireshark.org/

[12] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, ser. Hotnets-IX. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1868447.1868466 pp. 19:1–19:6.

[13] "POX wiki," Mar. 2015, Website, Open Networking Lab, accessed: 2018-06-18. [Online]. Available: https://openflow.stanford.edu/display/ONL/POX+Wiki

[14] "Production quality, multilayer open virtual switch," accessed: 2018-06-18. [Online]. Available: http://www.openvswitch.org/

[15] "OpenDNP3," accessed: 2018-06-18. [Online]. Available: http://www.automatak.com/opendnp3/

[16] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. Dependable Secur. Comput.*, vol. 1, no. 1, pp. 11–33, Jan. 2004. [Online]. Available: http://dx.doi.org/10.1109/TDSC.2004.2

[17] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, A. A. Chien, P. Coteus, N. A. Debardeleben, P. C. Diniz, C. Engelmann, M. Erez, S. Fazzari, A. Geist, R. Gupta, F. Johnson, S. Krishnamoorthy, S. Leyffer, D. Liberty, S. Mitra, T. Munson, R. Schreiber, J. Stearley, and E. V. Hensbergen, "Addressing failures in exascale computing," *Int. J. High Perform. Comput. Appl.*, vol. 28, no. 2, pp. 129–173, May 2014. [Online]. Available: http://dx.doi.org/10.1177/1094342014522573

[18] X. Yang, Z. Wang, J. Xue, and Y. Zhou, "The reliability wall for exascale supercomputing," *IEEE Trans. Comput.*, vol. 61, no. 6, pp. 767–779, June 2012. [Online]. Available: http://dx.doi.org/10.1109/TC.2011.106

[19] S. Jha, V. Formicola, Z. Kalbarczyk, C. Di Martino, W. T. Kramer, and R. K. Iyer, "Analysis of Gemini interconnect recovery mechanisms: Methods and observations," *Cray User Group*, pp. 8–12, 2016.

[20] J. H. Barton, E. W. Czeck, Z. Z. Segall, and D. P. Siewiorek, "Fault injection experiments using FIAT," *IEEE Trans. Comput.*, vol. 39, no. 4, pp. 575–582, Apr. 1990. [Online]. Available: http://dx.doi.org/10.1109/12.54853

[21] C. D. Martino, Z. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and W. Kramer, "Lessons learned from the analysis of system failures at petascale: The case of Blue Waters," in *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, ser. DSN '14. Washington, DC, USA: IEEE Computer Society, 2014. [Online]. Available: https://doi.org/10.1109/DSN.2014.62 pp. 610–621.

[22] "Charm++: Parallel programming framework." [Online]. Available: http://charmplusplus.org/

[23] V. Formicola, S. Jha, D. Chen, F. Deng, A. Bonnie, M. Mason, J. Brandt, A. Gentile, L. Kaplan, J. Repik, J. Enos, M. Showerman, A. Greiner, Z. Kalbarczyk, R. Iyer, and W. Kramer, "Understanding fault scenarios and impacts through fault injection experiments in Cielo," in *Cray User Group*, May 2017.

[24] T. Naughton, W. Bland, G. Vallee, C. Engelmann, and S. L. Scott, "Fault injection framework for system resilience evaluation: Fake faults for finding future failures," in *Proceedings of the 2009 Workshop on Resiliency in High Performance*, ser. Resilience '09. New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1552526.1552530 pp. 23–28.

[25] D. T. Stott, B. Floering, Z. Kalbarczyk, and R. K. Iyer, "A framework for assessing dependability in distributed systems with lightweight fault injectors," in *Proceedings of the 4th International Computer Performance and Dependability Symposium*, ser. IPDS '00. Washington, DC, USA: IEEE Computer Society, 2000. [Online]. Available: http://dl.acm.org/citation.cfm?id=857196.857849 p. 91.

[26] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "GPU-Qin: A methodology for evaluating the error resilience of GPGPU applications," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Mar. 2014, pp. 221–230.

[27] C. Constantinescu, "Teraflops supercomputer: Architecture and validation of the fault tolerance mechanisms," *IEEE Trans. Comput.*, vol. 49, no. 9, pp. 886–894, Sep. 2000. [Online]. Available: https://doi.org/10.1109/12.869320

[28] D. M. Blough and P. Liu, "FIMD-MPI: a tool for injecting faults into MPI applications," in *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*, 2000, pp. 241–247.

[29] T. Naughton, C. Engelmann, G. Vallée, and S. Böhm, "Supporting the development of resilient message passing applications using simulation," in *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, Feb. 2014, pp. 271–278.

[30] K. Feng, M. G. Venkata, D. Li, and X. H. Sun, "Fast fault injection and sensitivity analysis for collective communications," in *2015 IEEE International Conference on Cluster Computing*, Sep. 2015, pp. 148–157.

[31] C. D. Martino, W. Kramer, Z. Kalbarczyk, and R. Iyer, "Measuring and understanding extreme-scale application resilience: A field study of 5,000,000 HPC application runs," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2015, pp. 25–36.

[32] I. P. Egwutuoha, D. Levy, B. Selic, and S. Chen, "A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems," *J. Supercomput.*, vol. 65, no. 3, pp. 1302–1326, Sep. 2013. [Online]. Available: http://dx.doi.org/10.1007/s11227-013-0884-0

[33] B. Schroeder and G. Gibson, "A large-scale study of failures in high-performance computing systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp. 337–351, Oct. 2010. [Online]. Available: http://dx.doi.org/10.1109/TDSC.2009.4

[34] W. Bland, A. Bouteiller, T. Herault, J. Hursey, G. Bosilca, and J. J. Dongarra, "An evaluation of user-level failure mitigation support in MPI," in *Proceedings of the 19th European Conference on Recent Advances in the Message Passing Interface*, ser. EuroMPI'12. Berlin, Heidelberg: Springer-Verlag, 2012. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33518-1_24 pp. 193–203.

[35] U.S. Department of Energy Office of Science, "Resilience for extreme scale supercomputing systems," DOE National Laboratory Announcement Number LAB 14-1059, July 2014.

[36] L. Bautista-Gomez, A. Gainaru, S. Perarnau, D. Tiwari, S. Gupta, C. Engelmann, F. Cappello, and M. Snir, "Reducing waste in extreme scale systems through introspective analysis," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016, pp. 212–221.

[37] T. J. Dell, "A white paper on the benefits of chipkill-correct ECC for PC server main memory." IBM Microelectronics Division, Nov. 1997.

[38] "Torque resource manager." [Online]. Available: http://www.adaptivecomputing.com/products/open-source/torque/

[39] "Slurm workload manager." [Online]. Available: https://slurm.schedmd.com/

[40] A. Agelastos, B. Allan, J. Brandt, P. Cassella, J. Enos, J. Fullop, A. Gentile, S. Monk, N. Naksinehaboon, J. Ogden, M. Rajan, M. Showerman, J. Stevenson, N. Taerat, and T. Tucker, "The lightweight distributed metric service: A scalable infrastructure for continuous monitoring of large scale computing systems and applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. Piscataway, NJ, USA: IEEE Press, 2014. [Online]. Available: https://doi.org/10.1109/SC.2014.18 pp. 154–165.

[41] Cray, "Gemini network resiliency guide." [Online]. Available: http://docs.cray.com/books/S-0032-E/

[42] *AMD Inc. BIOS and Kernel Developers Guide, for AMD Family 16h*, Advanced Micro Devices, Inc., Feb. 2015.

[43] S. Sur, M. J. Koop, and D. K. Panda, "High-performance and scalable MPI over infiniband with reduced memory usage: An in-depth performance analysis," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, ser. SC '06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1188455.1188565

[44] L. Kale, "Charm++," in *Encyclopedia of Parallel Computing*, D. Padua, Ed. Springer Verlag, 2011.

[45] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao, and D. Chavarría-Miranda, "An evaluation of global address space languages: Co-array fortran and unified parallel C," in *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2005, pp. 36–47.

[46] P. Chen and E.-J. Lee, *Anelastic Wave Propagation (AWP)*. Cham: Springer International Publishing, 2015, pp. 15–90. [Online]. Available: https://doi.org/10.1007/978-3-319-16604-9_2

[47] A. Bazavov et al., "MIMD lattice computation (MILC) collaboration." [Online]. Available: http://physics.indiana.edu/s̃g/milc.html

[48] D. A. Donzis, P. Yeung, and D. Pekurovsky, "Turbulence simulations on $O(10^4)$ processors," in *TeraGrid '08 Conf*, 2008.

[49] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kale, and K. Schulten, "Scalable molecular dynamics with NAMD," *Journal of Computational Chemistry*, vol. 26, no. 16, pp. 1781–1802. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/jcc.20289

[50] A. Langer, J. Lifflander, P. Miller, K.-C. Pan, L. Kale, and P. Ricker, "Scalable algorithms for distributed-memory adaptive mesh refinement," in *Proceedings of the 24th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2012)*, New York, USA, October 2012.

[51] V. Mehta, "LeanMD: A Charm++ framework for high performance molecular dynamics simulation on large parallel machines," M.S. thesis, University of Illinois at Urbana-Champaign, 2004.

[52] A. J. Kunen, T. S. Bailey, and P. N. Brown, "KRIPKE–a massively parallel transport mini-app," 2015.

[53] "NPB UPC-FT." [Online]. Available: http://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks/npb-upc-ft/

[54] Cray, "XC series GNI and DMAPP API user guide (CLE 6.0.UP05) S-2446." [Online]. Available: https://pubs.cray.com/content/S-2446/

[55] "Blue Waters user portal — FAQ." [Online]. Available: https://bluewaters.ncsa.illinois.edu/faq

[56] S. Jha, V. Formicola, C. D. Martino, M. Dalton, W. T. Kramer, Z. Kalbarczyk, and R. K. Iyer, "Resiliency of HPC interconnects: A case study of interconnect failures and recovery in Blue Waters," *IEEE Transactions on Dependable and Secure Computing*, pp. 1–1, 2017.

[57] J. Zhang, B.-C. Seet, T.-T. Lie, and C. H. Foh, "Opportunities for software-defined networking in smart grid," in *2013 9th International Conference on Information, Communications Signal Processing*, Dec. 2013, pp. 1–5.

[58] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4: Experience with a globally-deployed software defined wan," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, ser. SIGCOMM '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2486001.2486019 pp. 3–14.

[59] *SDN Architecture Overview*, 1st ed., Open Networking Foundation, Dec. 2013.

[60] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1355734.1355746

[61] K. A. Stouffer, J. A. Falco, and K. A. Scarfone, "Sp 800-82. guide to industrial control systems (ICS) security: Supervisory control and data acquisition (SCADA) systems, distributed control systems (DCS), and other control system configurations such as programmable logic controllers (PLC)," National Institute of Standards & Technology, Gaithersburg, MD, United States, Tech. Rep., 2011.

[62] *IEEE Standard for Electric Power Systems Communications-Distributed Network Protocol (DNP3)*, IEEE Std. 1815-2012, Oct. 2012.

[63] "Modbus messaging on TCP/IP implementation guide v1.0b," Idaho National Laboratory, Tech. Rep., 2006.

[64] X. Dong, H. Lin, R. Tan, R. K. Iyer, and Z. Kalbarczyk, "Software-defined networking for smart grid resilience: Opportunities and challenges," in *Proceedings of the 1st ACM Workshop on Cyber-Physical System Security*, ser. CPSS '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2732198.2732203 pp. 61–68.

[65] *OpenFlow Switch Specification v1.5.1*, Open Networking Foundation, Mar. 2015.

[66] "Attacks - sdnsecurity.org," accessed: 2018-06-18. [Online]. Available: http://www.sdnsecurity.org/vulnerability/attacks/

[67] "TCPDUMP/LIBPCAP public repository," accessed: 2018-06-18. [Online]. Available: http://www.tcpdump.org

[68] R. M. Lee, M. J. Assante, and T. Conway, "Analysis of the cyber attack on the Ukrainian power grid," Electricity Information Sharing and Analysis Center (E-ISAC), Washington, DC, USA, Tech. Rep., 2016.

[69] H. Lin, "Detection and prevention of intrusions in power systems cyber-physical infrastructure," Ph.D. dissertation, University of Illinois at Urbana-Champaign, Urbana, Illinois, 2017.

[70] M. Cheminod, L. Durante, L. Seno, F. Valenza, A. Valenzano, and C. Zunino, "Leveraging SDN to improve security in industrial networks," in *2017 IEEE 13th International Workshop on Factory Communication Systems (WFCS)*, May 2017, pp. 1–7.

[71] S. Shin and G. Gu, "Attacking software-defined networks: A first feasibility study," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2491185.2491220 pp. 165–166.

[72] L. Dridi and M. F. Zhani, "SDN-Guard: DoS attacks mitigation in SDN networks," in *2016 5th IEEE International Conference on Cloud Networking (Cloudnet)*, Oct. 2016, pp. 212–217.

[73] P. Zhang, H. Wang, C. Hu, and C. Lin, "On denial of service attacks in software defined networks," *IEEE Network*, vol. 30, no. 6, pp. 28–33, Nov. 2016.

[74] S. M. Mousavi and M. St-Hilaire, "Early detection of DDoS attacks against SDN controllers," in *2015 International Conference on Computing, Networking and Communications (ICNC)*, Feb. 2015, pp. 77–81.

[75] Q. Yan, F. R. Yu, Q. Gong, and J. Li, "Software-defined networking (SDN) and distributed denial of service (DDoS) attacks in cloud computing environments: A survey, some research issues, and challenges," *IEEE Communications Surveys Tutorials*, vol. 18, no. 1, pp. 602–622, 2016.

[76] I. Ahmad, S. Namal, M. Ylianttila, and A. Gurtov, "Security in software defined networks: A survey," *IEEE Communications Surveys Tutorials*, vol. 17, no. 4, pp. 2317–2346, 2015.

[77] B. E. Ujcich, U. Thakore, and W. H. Sanders, "Attain: An attack injection framework for software-defined networking," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2017, pp. 567–578.

[78] U. Ghosh, X. Dong, R. Tan, Z. Kalbarczyk, D. K. Yau, and R. K. Iyer, "A simulation study on smart grid resilience under software-defined networking controller failures," in *Proceedings of the 2Nd ACM International Workshop on Cyber-Physical System Security*, ser. CPSS '16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2899015.2899020 pp. 52–58.

APPENDIX A

FAULT INJECTION COMMANDS FOR CRAY XE PLATFORM

The fault injector module of *HPCArrow* supports execution of arbitrary failures and restorations on link, connection, node and blade components of a Cray XE machine with Gemini ASIC routers. Commands are executed on a system management workstation (SMW) by an administrator. Fault injection commands and their parameters are summarized in Table A.1, and restoration commands are summarized in Table A.2. Note that all restoration commands must be run as the 'crayadm' user on the SMW.

Table A.1: Fault Injection Commands for Cray XE Platform

Component(s)	**Command**	**Comments**
Link Connection	`xtmemio -w ::{gemini} {0x0006000128 | \ (link_row << 22) | (link_col << 19)} 2 0`	{gemini}: Component name of targeted Gemini. {link_row} and {link_col}: Row and column of link. For example, a target link "c0-0c0s1g0l45" should be specified as: {gemini} = "c0-0c0s1g0" {link_row} = 4 {link_col} = 5 For connection failures, the command must be repeated for each link in the target connection.
Node	`xtnmi {node}`	{node}: Component name of targeted node. For example, {node} = "c0-0c0s3n2" Note: Must be run as 'crayadm'
Blade	`rsh -l root {blade} "/opt/bin/i2c \ 2:0x60/2=0x02,0x00"`	{blade}: Component name of targeted blade. For example, {blade} = "c0-0c0s3"

Table A.2: Component Restoration Commands

Component(s)	**Command**	**Comments**
Link	`xtwarmswap -s {link}, {link_end} -p p0`	{link}: Failed link to restore {link_end}: Other end of failed link to restore
Connection	`xtwarmswap -s {links} -p p0`	{links}: Comma separated list of links to restore from impacted connection. Other ends of impacted links must also be specified.
Node	`xtbootsys --reboot -L CNL0 {node}`	{node}: Component name of node to restore. Note: Requires an interactive terminal session.
Blade	Remove: `xtwarmswap --force --remove {blade}` Swap: `xtwarmswap --add {blade}` Boot: `xtcli boot CNL0 {blade}`	{blade}: Component name of blade to restore. Remove, swap and boot commands must be executed in sequence.

APPENDIX B

JYC SYSTEM MAP

JYC is a 96-node Cray XE/XK testbed at the National Center for Supercomputing Applications (NCSA). The machine consists of one cabinet (three chassis) with 56 XE nodes, 28 XK nodes and 14 service nodes.

System components are referred to by their *component names* (cnames). Component names are based on their physical locations within the system and follow a hierarchical convention. Cabinets are referred to by their X and Y position in the physical machine layout. Since JYC is a single cabinet system, the cabinet name is simply `c0-0`. Chassis within each cabinet are referred to by their vertical position, with `c0` being the bottommost chassis and `c2` being to topmost chassis within a cabinet. Blades within each chassis are referred to by their position from left to right, with `s0` being the leftmost blade and `s7` being the rightmost blade. The two Gemini ASICs on each blade are referred to by `g0` and `g1`. Finally, the four nodes on each blade are referred to by `n0`, `n1`, `n2`, and `n3`. Nodes `n0` and `n1` are connected to Gemini `g0`, whereas nodes `n2` and `n3` are connected to Gemini `g1`. Figure B.1 demonstrates the layout and naming convention on JYC. Node identifiers (NIDs) are also provided within node boxes. Examples of component names mapped to NIDs are provided in Table B.1.

Table B.1: Example Component Names on JYC

Component/NID	**Component Name (cname)**
Cabinet 0	`c0-0`
Chassis 1 in Cabinet 0	`c0-0c1`
Blade 4 in Chassis 1, Cabinet 0	`c0-0c1s4`
Node 55	`c0-0c1s4n2`
Gemini 0 in Blade 4, Chassis 1, Cabinet 0	`c0-0c1s4g0`

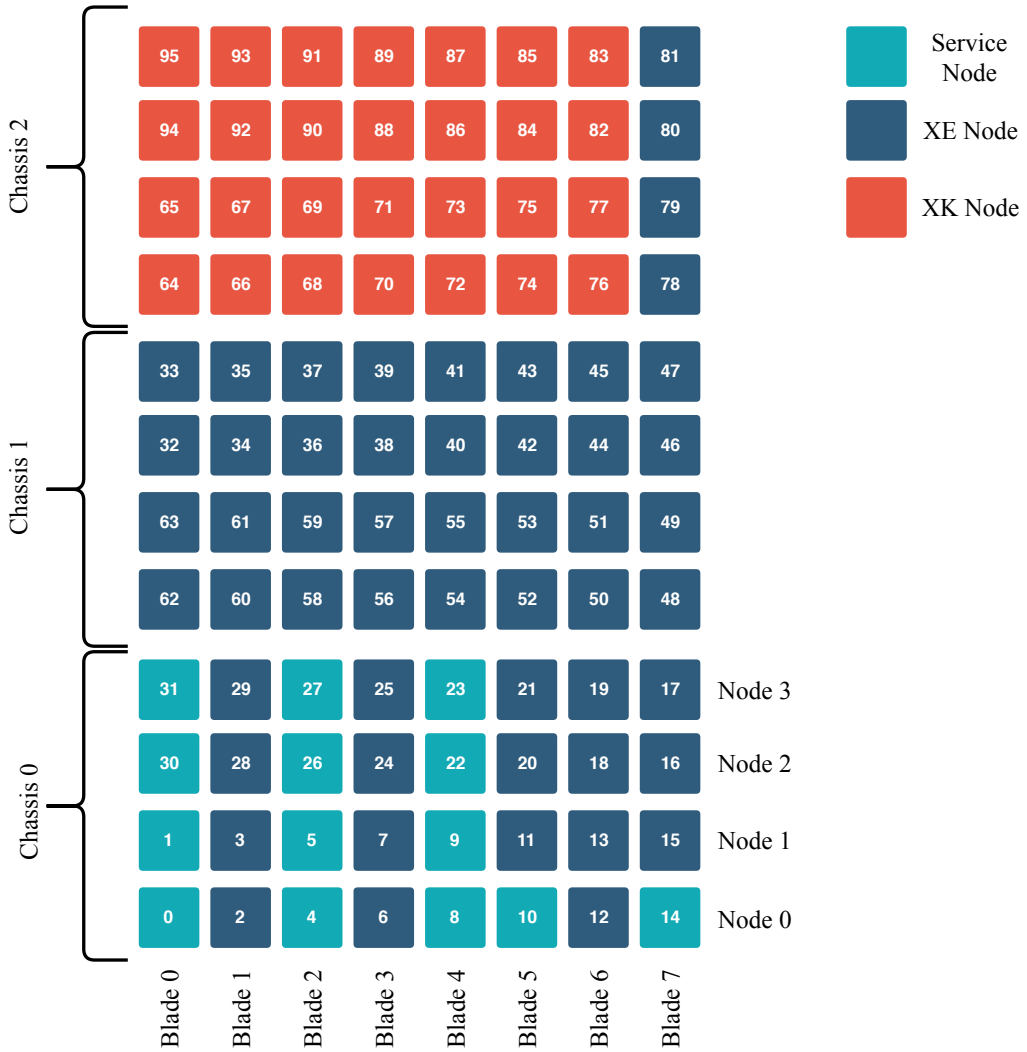

Figure B.1: JYC system overview with node types and identifiers

APPENDIX C

APPLICATION SETS AND PARAMETERS

In order to study the impact of injected faults and recoveries across the entire target system, we simultaneously launch a set of workloads for each experiment. Launching workloads across the entire system allows us to identify failures that propagate or cascade through the system and impact applications running further away from the injected component. Applications are chosen such that each set has a mix of runtime frameworks, application sizes, and overall system utilization. In our injection campaign, the *HPCArrow* workload generator is configured with eight unique application sets. Tables C.1 – C.8 define the following fields for their respective application sets:

- **Application Name**. For Charm++ applications, we distinguish between symmetric multiprocessing (SMP) and HugePages variants.
- **Size**, in terms of number of nodes and node types.
- **Processes per node (PPN)**. Cray XE nodes support 32 simultaneous processes/threads whereas Cray XK nodes support 16. When executing on a combination of node types, the number of processes per node is limited to the minimum number of processes supported by each node type. Additionally, Charm++ SMP applications abstract processing elements (PEs) from available processor cores. Effectively, each PE is composed of multiple threads. For such applications, we define their PPN as number of PEs $\times$ number of threads.
- **Node Identifiers**, describing the placement of applications within the system. Application locations are also visualized in Figs. C.1 – C.8.
- **Parameters**, describing input files and parameter values for each application.

Table C.1: Application Set 1 Configuration

Application Name	Size	PPN	Node IDs	Parameters

AWP-ODC	32 XE	16	$32-63$	Default benchmark parameters
				`NX = NY = NX = 712`
				`NPX = NPY = NPZ = 8`

Kripke (HugePages)	8 XK	16	$66-69, 90-93$	`NITER = 13`
				`ZSET = 5:5:5`
				`ZONES = 12,12,12`
				`DLIST = 1:12,2:6,3:4,4:3,6:2,12:1`
				`GLIST = 1:64,2:32,4:16,8:8,16:4,32:2,64:1`
				`LEGENDRE = 9`

LeanMD (HugePages)	8 XK	16	$72-75, 84-87$	`dimX = dimY = dimZ = 16`
				`Steps = 1900`
				`FirstLBStep = 20`
				`LBPeriod = 20`

UPC-FT	4 XE	32	$6, 7, 24, 25$	Class B with `MAX_ITER = 21500`
				`NX = 4`
				`NY = 32`

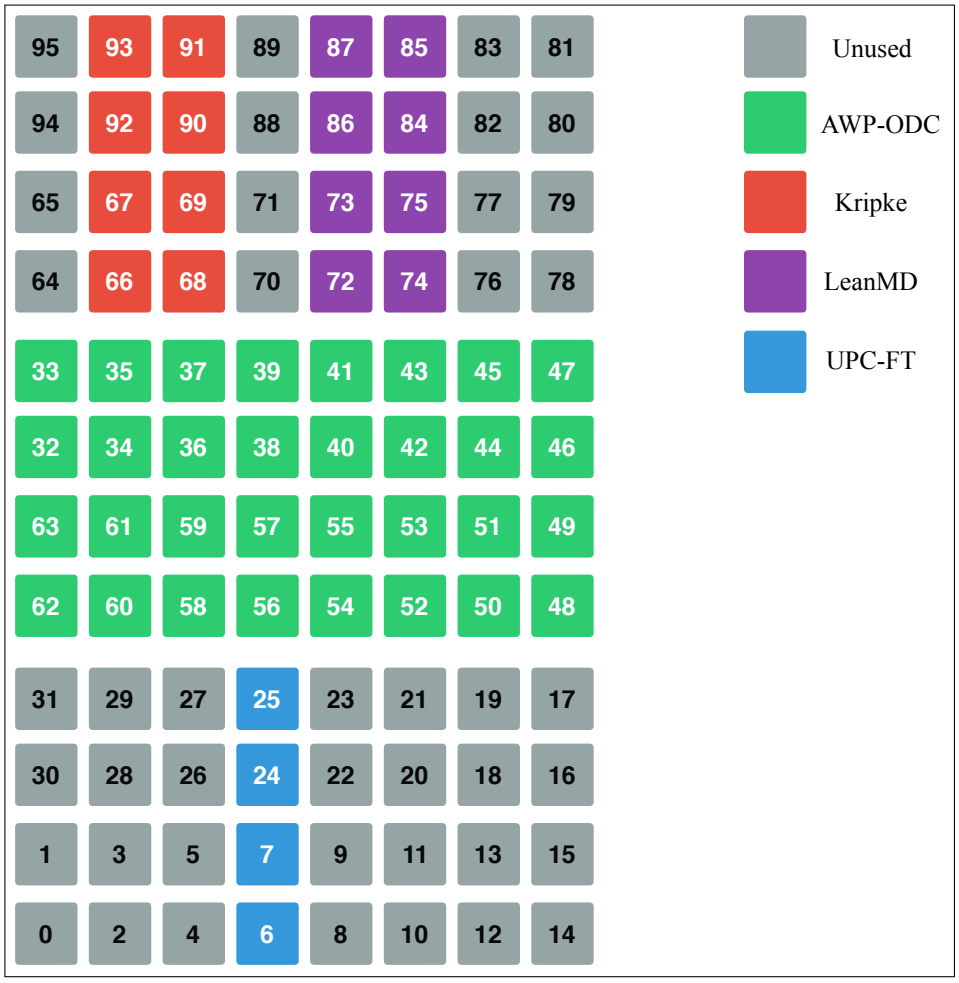

Figure C.1: Application set 1 placement

Table C.2: Application Set 2 Configuration

Application Name	**Size**	**PPN**	**Node IDs**	**Parameters**

AMR (HugePages)	4 XE	32	$2, 3, 28, 29$	`MAX_DEPTH = 10`
				`BLOCK_SIZE = 8`
				`ITERATIONS = 175`
				`LB_FREQ = 3`
				`ARRAY_DIM = 512`

AMR (SMP)	32 XE	2×16	$32 - 63$	`MAX_DEPTH = 10`
				`BLOCK_SIZE = 8`
				`ITERATIONS = 800`
				`LB_FREQ = 3`
				`ARRAY_DIM = 512`

AWP-ODC	2 XE	32	$15, 16$	Default benchmark parameters
				`NX = NY = NX = 368`
				`NPX = NPY = NPZ = 4`

Kripke (HugePages)	4 XE	32	$6, 7, 24, 25$	`NITER = 13`
				`ZSET = 5:5:5`
				`ZONES = 12,12,12`
				`DLIST = 1:12,2:6,3:4,4:3,6:2,12:1`
				`GLIST = 1:64,2:32,4:16,8:8,16:4,32:2,64:1`
				`LEGENDRE = 9`

LeanMD (HugePages)	4 XE	32	$12, 13, 18, 19$	`dimX = dimY = dimZ = 16`
				`Steps = 1775`
				`FirstLBStep = 20`
				`LBPeriod = 20`

LeanMD (HugePages)	2 XE	32	$11, 20$	`dimX = dimY = dimZ = 16`
				`Steps = 950`
				`FirstLBStep = 20`
				`LBPeriod = 20`

UPC-FT	4 XE, 28 XK	16	$64 - 95$	Class B with `MAX_ITER = 21500`
				`NX = 32`
				`NY = 16`

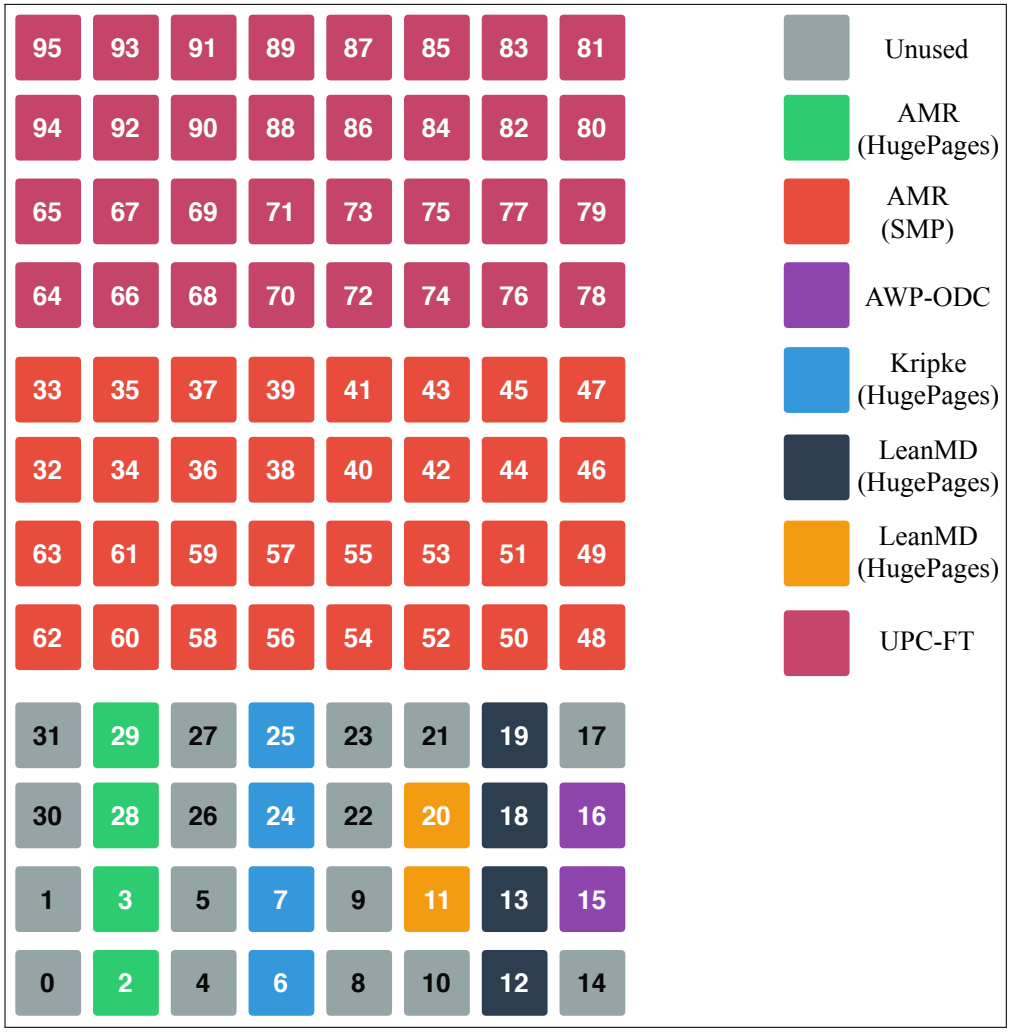

Figure C.2: Application set 2 placement

Table C.3: Application Set 3 Configuration

Application Name	Size	PPN	Node IDs	Parameters

AMR (HugePages)	2 XE	32	60, 61	`MAX_DEPTH = 10`
				`BLOCK_SIZE = 8`
				`ITERATIONS = 95`
				`LB_FREQ = 3`
				`ARRAY_DIM = 512`

AWP-ODC	4 XK	16	74, 75, 84, 85	Default benchmark parameters
				`NX = NY = NX = 364`
				`NPX = NPY = NPZ = 4`

LeanMD (HugePages)	4 XE	32	6, 7, 24, 25	`dimX = dimY = dimZ = 16`
				`Steps = 1775`
				`FirstLBStep = 20`
				`LBPeriod = 20`

UPC-FT	2 XE	32	16, 17	Class B with `MAX_ITER = 21500`
				`NX = 2`
				`NY = 32`

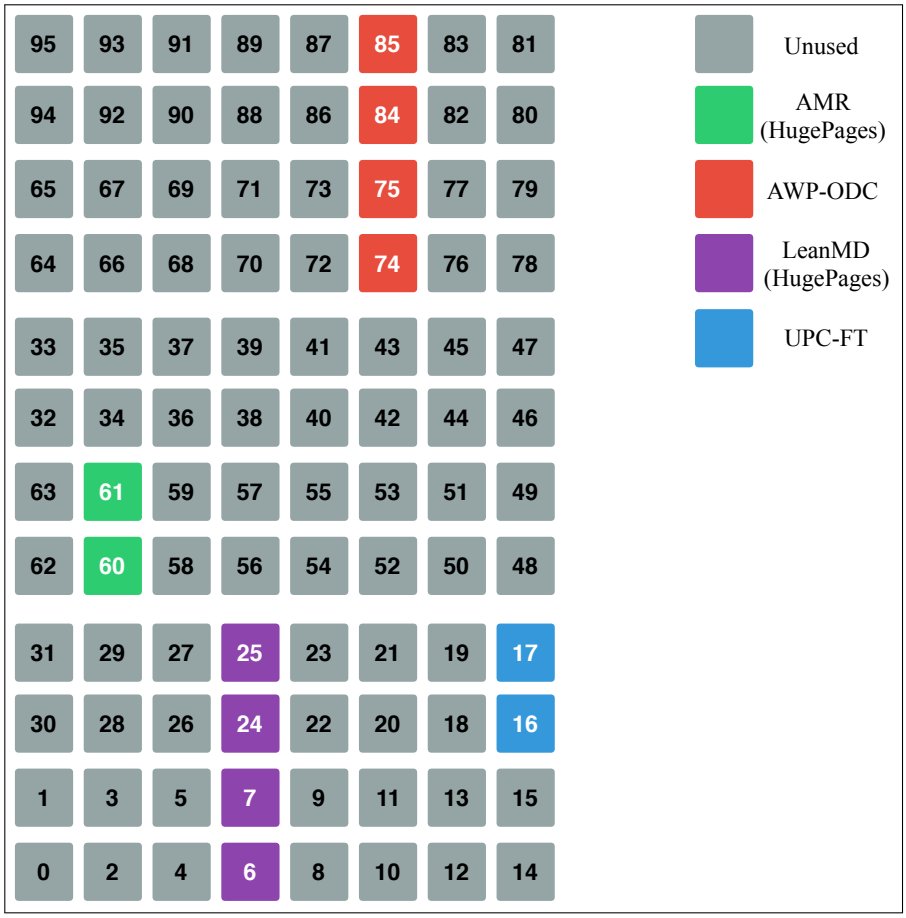

Figure C.3: Application set 3 placement

Table C.4: Application Set 4 Configuration

Application Name	Size	PPN	Node IDs	Parameters

AMR (HugePages)	2 XE	32	16, 17	`MAX_DEPTH = 10`
				`BLOCK_SIZE = 8`
				`ITERATIONS = 95`
				`LB_FREQ = 3`
				`ARRAY_DIM = 512`

AWP-ODC	4 XE, 28 XK	16	64 – 95	Default benchmark parameters
				`NX = NY = NX = 712`
				`NPX = NPY = NPZ = 8`

Kripke (HugePages)	2 XE	32	16, 17	`NITER = 13`
				`ZSET = 4:4:4`
				`ZONES = 12,12,12`
				`DLIST = 1:12,2:6,3:4,4:3,6:2,12:1`
				`GLIST = 1:64,2:32,4:16,8:8,16:4,32:2,64:1`
				`LEGENDRE = 9`

UPC-FT	8 XE	32	34 – 37, 58 – 61	Class B with `MAX_ITER = 21500`
				`NX = 8`
				`NY = 32`

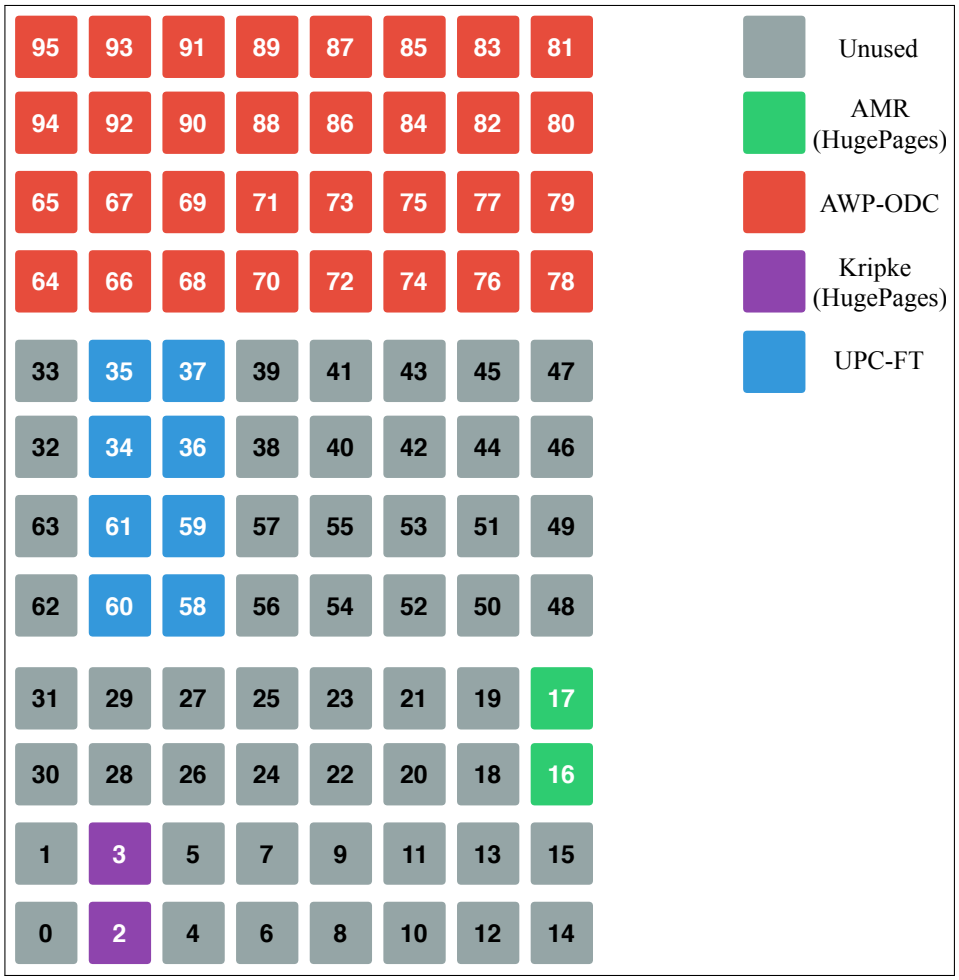

Figure C.4: Application set 4 placement

Table C.5: Application Set 5 Configuration

Application Name	Size	PPN	Node IDs	Parameters

AMR (SMP)	36 XE, 28 XK	1×16	$32 - 95$	`MAX_DEPTH = 10`
				`BLOCK_SIZE = 8`
				`ITERATIONS = 850`
				`LB_FREQ = 3`
				`ARRAY_DIM = 512`

AWP-ODC	2 XE	32	$18, 19$	Default benchmark parameters
				`NX = NY = NX = 368`
				`NPX = NPY = NPZ = 4`

UPC-FT	4 XE	32	$6, 7, 24, 25$	Class B with `MAX_ITER = 21500`
				`NX = 4`
				`NY = 32`

Figure C.5: Application set 5 placement

Table C.6: Application Set 6 Configuration

Application Name	Size	PPN	Node IDs	Parameters

AMR (HugePages)	2 XE	2×16	$20, 21$	`MAX_DEPTH = 10`
				`BLOCK_SIZE = 8`
				`ITERATIONS = 95`
				`LB_FREQ = 3`
				`ARRAY_DIM = 512`

Kripke (HugePages)	4 XE	32	$2, 3, 28, 29$	`NITER = 13`
				`ZSET = 5:5:5`
				`ZONES = 12,12,12`
				`DLIST = 1:12,2:6,3:4,4:3,6:2,12:1`
				`GLIST = 1:64,2:32,4:16,8:8,16:4,32:2,64:1`
				`LEGENDRE = 9`

LeanMD (HugePages)	4 XE	32	$40 - 55$	`dimX = dimY = dimZ = 16`
				`Steps = 1775`
				`FirstLBStep = 20`
				`LBPeriod = 20`

MILC	16 XE	32	$32 - 39, 56 - 63$	`NX = NY = NZ = NT = 32`
				`Warmups = 0`
				`Trajectories = 1`
				`Iterations = 10000`

MILC	2 XE	32	$16, 17$	`NX = NY = NZ = NT = 16`
				`Warmups = 0`
				`Trajectories = 2`
				`Iterations = 10000`

NAMD (SMP)	4 XE	2×16	$6, 7, 24, 25$	Input: `stmv.4.namd`
PSDNS	28 XK, 4 XE	16	$64 - 95$	Dimensions: 512
PSDNS	4 XE	32	$12, 13, 18, 19$	Dimensions: 128

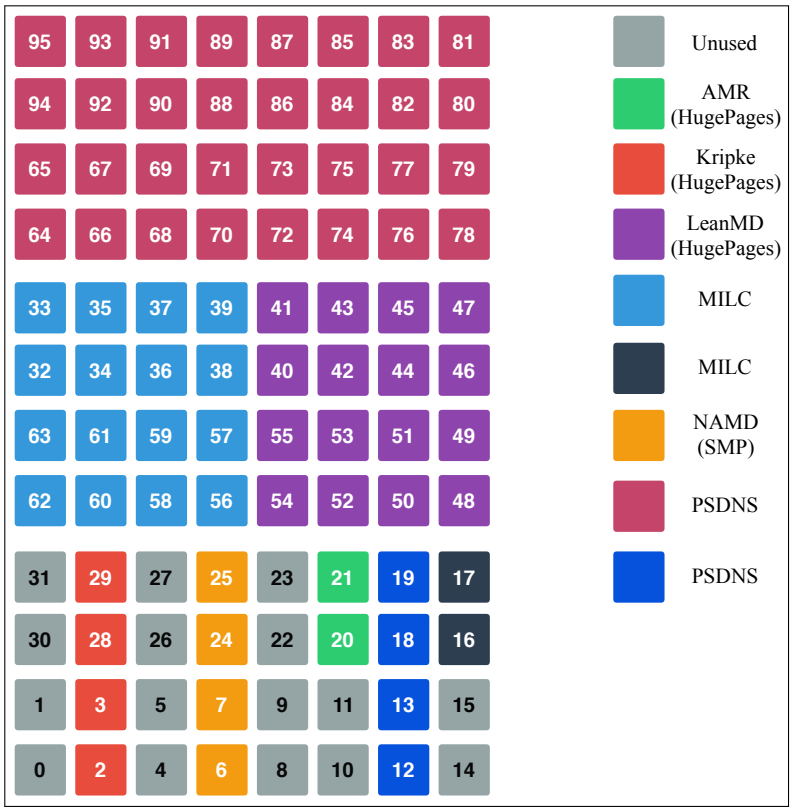

Figure C.6: Application set 6 placement

Table C.7: Application Set 7 Configuration

Application Name	Size	PPN	Node IDs	Parameters

AMR (HugePages)	4 XE	32	12, 13, 18, 19	`MAX_DEPTH = 10`
				`BLOCK_SIZE = 8`
				`ITERATIONS = 175`
				`LB_FREQ = 3`
				`ARRAY_DIM = 512`

Kripke (HugePages)	8 XK	16	72 – 75, 84 – 87	`NITER = 13`
				`ZSET = 5:5:5`
				`ZONES = 12,12,12`
				`DLIST = 1:12,2:6,3:4,4:3,6:2,12:1`
				`GLIST = 1:64,2:32,4:16,8:8,16:4,32:2,64:1`
				`LEGENDRE = 9`

LeanMD (HugePages)	8 XK	16	64 – 67, 92 – 95	`dimX = dimY = dimZ = 16`
				`Steps = 1900`
				`FirstLBStep = 20`
				`LBPeriod = 20`

MILC	4 XE	32	2, 3, 28, 29	`NX = NY = NZ = NT = 16`
				`Warmups = 0`
				`Trajectories = 4`
				`Iterations = 10000`

NAMD (SMP)	16 XE	2 × 16	32 – 39, 56 – 63	Input: `stmv.16.namd`
PSDNS	8 XE	32	44 – 51	Dimensions: 256

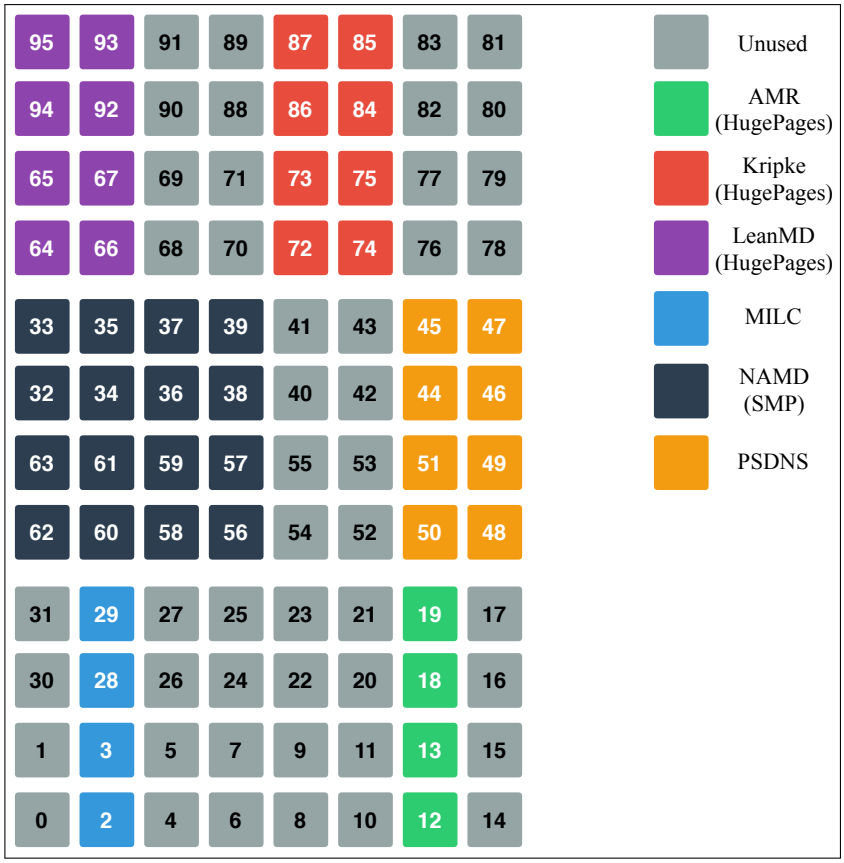

Figure C.7: Application set 7 placement

Table C.8: Application Set 8 Configuration

Application Name	Size	PPN	Node IDs	Parameters

AMR (HugePages)	4 XK	16	64, 65, 94, 95	`MAX_DEPTH = 10`
				`BLOCK_SIZE = 8`
				`ITERATIONS = 105`
				`LB_FREQ = 3`
				`ARRAY_DIM = 512`

Kripke (HugePages)	4 XK	16	70, 71, 88, 89	`NITER = 14`
				`ZSET = 4:4:4`
				`ZONES = 12,12,12`
				`DLIST = 1:12,2:6,3:4,4:3,6:2,12:1`
				`GLIST = 1:64,2:32,4:16,8:8,16:4,32:2,64:1`
				`LEGENDRE = 9`

LeanMD (HugePages)	2 XE	32	48, 49	`dimX = dimY = dimZ = 16`
				`Steps = 950`
				`FirstLBStep = 20`
				`LBPeriod = 20`

MILC	4 XE	32	78 – 81	`NX = NY = NZ = NT = 16`
				`Warmups = 0`
				`Trajectories = 4`
				`Iterations = 10000`

NAMD (SMP)	4 XE	2 × 16	36, 37, 58, 59	Input: `stmv.4.namd`
PSDNS	4 XE	32	42, 43, 52, 53	Dimensions: 128

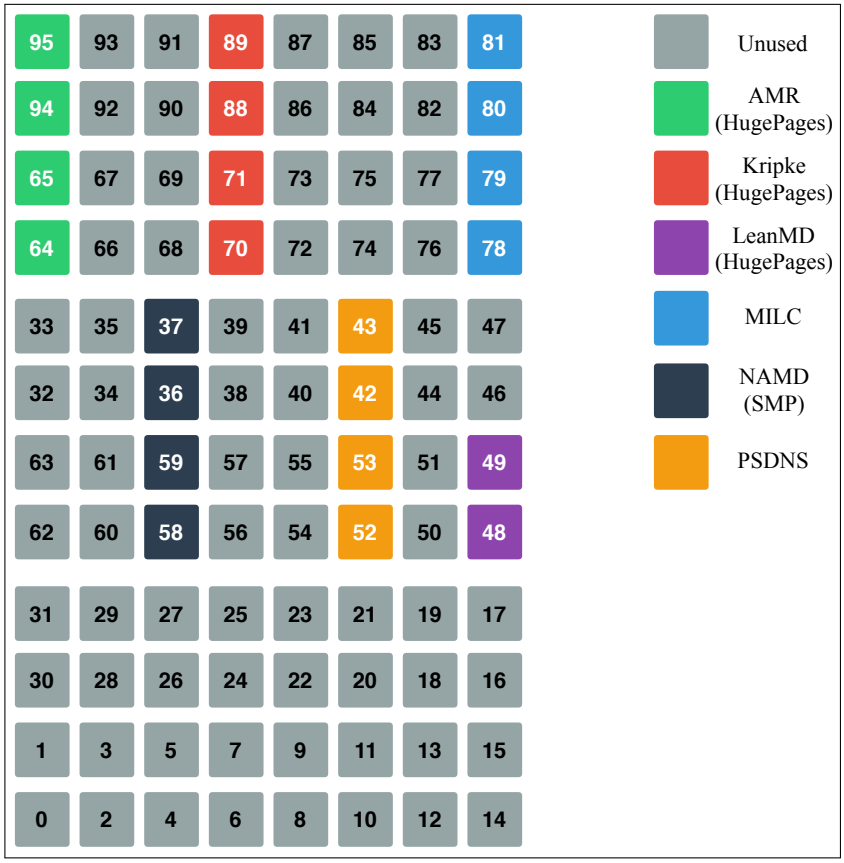

Figure C.8: Application set 8 placement