UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

RICARDO PARIZOTTO

In-network computing: overcoming constraints, failures, and configuration challenges

Thesis presented in partial fulfillment of the requirements for the degree of Doctor of Computer Science

Advisor: Prof. Dr. Alberto Egon Schaeffer Filho
Co-advisor: Prof. Dra. Israat Haque

Porto Alegre
August 2024

CIP — CATALOGING-IN-PUBLICATION

Parizotto, Ricardo

In-network computing:
overcoming constraints, failures, and configuration challenges / Ricardo Parizotto. – Porto Alegre: PPGC da UFRGS, 2024.

132 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2024. Advisor: Alberto Egon Schaeffer Filho; Co-advisor: Israat Haque.

1. Programmable data planes. 2. In-network computing. 3. Hardware constraints. 4. Fault-tolerance. 5. Intent-based networks. I. Schaeffer Filho, Alberto Egon. II. Haque, Israat. III. Título.

"The times
they are a-changin'."
— Bob Dylan

AGRADECIMENTOS

I would like to thank my advisor, Professor Alberto Egon Schaeffer Filho, for his invaluable guidance and teachings over nearly seven years. Alberto was always open to discussing research directions and sharing technical insights, which has been a key part of my academic journey. I also want to thank Professor Israat Haque for co-advising me, providing support, and, along with Alberto and Miguel, making my visit to Dalhousie University possible, where she kindly welcomed me into her lab. I am also grateful to Israat's research group for their receptiveness, help, and friendship, especially Hesam and Miguel for their collaboration.

The remainder of these acknowledgments are in Portuguese.

Gostaria de agradecer à UFRGS por se posicionar em favor da ciência quando foi necessário. A UFRGS, junto com o professor Alberto, fez esforços essenciais para que eu pudesse continuar minha pesquisa mesmo durante a pandemia. Agradeço aos amigos e colegas que fiz nessa caminhada, especialmente aos colegas dos laboratórios de redes da UFRGS, onde fiz amigos, colaborações e momentos inesquecíveis. Agradeço também aos amigos distantes que trouxeram motivação e inspiração nos momentos difíceis, e aos colaboradores que, além de terem sido essenciais para o desenvolvimento desta tese, se tornaram bons amigos. Em particular, agradeço ao Professor Braulio pelos ensinamentos e inspiração e aos demais colegas da UFFS que tive oportunidade de colaborar.

Agradeço à minha família pelo amor, carinho, suporte e compreensão nos momentos difíceis. Meus pais, irmãs e cunhados foram inspiração e força para chegar até aqui. Agradeço à Jessica, agora minha esposa, pelo companheirismo, confiança e amor. Estudar e trabalhar ao seu lado não só foi prazeroso, mas também trouxe alegria e significado à minha vida. Estendo esse agradecimento à família da Jessica, que considero minha, e que foi um pilar essencial durante os últimos meses de desenvolvimento desta tese, enquanto esperávamos pelo nosso pequeno Valentin. A chegada do Valentin me fez identificar a importância das pequenas coisas na vida.

ABSTRACT

Programmable switches are networking forwarding devices that allow customized, stateful functionalities to run at line rate. Unlike fixed-functionality switches, programmable switches offer higher versatility and innovation potential. The advantages of programmable switches have led researchers to move functionality previously performed on servers to the network itself, resulting in the concept of *In-Network Computing* (INC). However, offloading functionalities to the data plane is subject to several distinctions compared to how computation is traditionally performed on servers. This thesis investigates the in-network computing paradigm under three different aspects that set it apart from traditional computation. Firstly, we study the *constraints* imposed by the data plane, which can impact the offloading of application functionality to the switch hardware. We propose a terminology and taxonomy of design considerations to be used when offloading a functionality. We then present a system called NetGVT that employs the considerations to build a customized design that offloads virtual time synchronization to switches. Furthermore, we show that NetGVT can accelerate distributed simulations and outperform a traditional server-only solution. However, once we move the computation to the data plane, INC failures can disrupt the system and make it unavailable. Therefore, the second aspect we investigate is the impact of failures and the necessary consistency requirements for existing INCs for correctness after failure. In response, we propose RESIST, a system that applies lightweight techniques and building blocks to provide fault tolerance for in-network computing. Although we observe that fault tolerance can be achieved without compromising the performance gains achieved with INC, managing the functionality at forwarding devices reveals a complex and time-consuming process compared to running them on traditional servers. To understand this configuration challenge, we investigate methods for simplifying INC fault tolerance management using high-level intents. We propose a system called Araucaria, which facilitates the specification of fault tolerance intents in a structured natural language and a process for intent refinement that instruments INCs with fault tolerance building blocks. Finally, we demonstrate a running example using NetGVT as a concrete use case, showing the feasibility and scalability of the proposed approaches in this thesis both in a testbed with real programmable switch hardware and in a behavior model emulator.

Keywords: Programmable data planes. In-network computing. Hardware constraints. Fault-tolerance. Intent-based networks.

Computação em rede: superando restrições, falhas, e desafios de configuração

RESUMO

Switches programáveis são dispositivos de encaminhamento de rede que permitem a execução de funcionalidades personalizadas que funcionam na taxa de linha. Ao contrário de *switches* com funcionalidade fixa, *switches* programáveis oferecem maior versatilidade e potencial de inovação. As vantagens dos *switches* programáveis levaram os pesquisadores a transferir funcionalidades anteriormente realizadas em servidores para a própria rede, resultando no conceito de *In-Network Computing* (INC). No entanto, a transferência de funcionalidades para o plano de dados está sujeita a várias distinções em comparação com a forma como a computação é tradicionalmente realizada em servidores. Esta tese investiga o paradigma de computação em rede sob três aspectos diferentes que o diferenciam da computação tradicional. Em primeiro lugar, estudamos as *restrições* impostas pelo plano de dados, que podem impactar a transferência de uma funcionalidade de aplicação para o *hardware* do *switch*. Propomos uma terminologia e uma taxonomia de considerações de design a serem utilizados ao transferir uma funcionalidade para o plano de dados dos dispositivos de rede. Apresentamos então um sistema chamado NetGVT, que utiliza as considerações para construir um *design* personalizado para transferir a sincronização de tempo virtual para *switches*. Além disso, mostramos que o NetGVT pode acelerar simulações distribuídas e superar uma solução tradicional que utiliza apenas servidores. No entanto, uma vez que movemos a computação para o plano de dados, *falhas* na INC podem interromper o sistema e torná-lo indisponível. Portanto, o segundo aspecto que investigamos é o impacto das falhas e os requisitos de consistência necessários para INCs existentes permanecerem corretas após uma falha. Em resposta, propomos o RESIST, um sistema que aplica técnicas eficientes e blocos de construção para fornecer tolerância a falhas para a computação em rede. Embora observemos que a tolerância a falhas pode ser alcançada sem comprometer os ganhos de desempenho obtidos com a INC, gerenciar a funcionalidade em dispositivos de encaminhamento revela um processo complexo e demorado em comparação com a execução em servidores tradicionais. Para entender esse *desafio de configuração*, investigamos métodos para simplificar o gerenciamento de tolerância

a falhas na INC usando intenções de alto nível. Propomos um sistema chamado Araucaria, que facilita a especificação de intenções em uma linguagem semelhante à natural, e um processo para refinamento de intenções para instrumentar INCs com blocos de construção de tolerância a falhas. Em seguida, demonstramos um exemplo prático usando o NetGVT, mostrando a viabilidade e escalabilidade das abordagens propostas nesta tese, tanto em um *testbed* com *hardware* de switches programáveis reais quanto em um emulador de modelo de comportamento.

Palavras-chave: Planos de dados programáveis. Computação na rede. Restrições de hardware. Tolerância a falhas. Redes baseadas em intenções.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

ACL Access Control List

ALU Arithmetic Logic Unit

API Application Programming Interface

ASIC Application-Specific Integrated Circuit

AST Abstract Syntax Tree

BMV2 Behavioral Model Version 2

BNN Binary Neural Network

CC Concurrency Control

CRDT Conflict-free Replicated Data Types

CNN Convolutional neural network

EC Eventual Consistency

FPGA Field Programmable Gate Arrays

GUI Graphical User Interface

GVT Global Virtual Time

IBN Intent Based Networking

IETF Internet Engineering Task Force

INA In-Network Aggregation

INC In-Network Computing

JCT Job Completion Time

JSON JavaScript Object Notation

LVT Local Virtual Time

ML Machine Learning

MMT Multiple Match Tables

NIC Network Interface Card

NF Network Function

NN Neural Network

PCIe Peripheral Component Interconnect Express

PDP Programmable Data Planes

PHV Packet Header Vector

PISA Protocol Independent Switch Architecture

QoS Quality of Service

RMT Reconfigurable Match Tables

RTT Round Trip Time

SC Strong Consistency

SDN Software-Defined Networking

SEC Strong Eventual Consistency

SMT Single Match Table

SRAM Static Random-Access Memory

TCAM Ternary Content Addressable Memory

TCP Transmission Control Protocol

TNA Tofino Native Architecture

CONTENTS

1 INTRODUCTION

This thesis investigates the notion of offloading computing tasks to programmable data plane devices, using the paradigm of *in-network computing*. This chapter provides an introduction to the thesis. We present the contextualization in Section §1.1. Next, in Section §1.2, we present the research problems we investigate in this thesis. Section §1.3 describes our motivation and goals, also presenting our contributions. Finally, Section §1.4 details the outline of this document.

1.1 Contextualization

Programmable networks allow the behavior of the network to be modified. The idea of programmable networks dates back to the 90s when researchers investigated a paradigm called *Active Networks*. Active networks were motivated by the claim that it was necessary to move functionality to network devices (TENNENHOUSE; WETHERALL, 1996), requiring a more flexible switch or router architecture. Furthermore, some researchers considered the standardization of new protocols by the IETF to be a slow process. As a response to that concern, research efforts were made to develop a novel architecture enabling operators to incorporate novel features into the network without waiting for the standardization process. The community proposed several functionalities that could be enhanced by in-network processing, including data fusion and caching (FEAMSTER; REXFORD; ZEGURA, 2014a). Although the active network could provide performance benefits, the architecture was too complex to adopt, and the industry had no real problems that needed solutions with these performance benefits (WETHERALL; TENNENHOUSE, 2019). Instead, because of the increasing traffic demands, network operators were looking for solutions for management tasks, such as efficient traffic engineering.

Managing networks was challenging, requiring network operators to understand complex distributed protocols from equipaments of different vendors. Separating the control and data plane was an alternative for addressing this management problem. By making the network decision-making centralized in the control plane and only forwarding packets in the data plane, this separation enabled more programmability and flexibility for networking (CASADO et al., 2007). The creation of the OpenFlow protocol solidified the separation between data and control plane

and led to the Software-Defined Networking (SDN) concept (MCKEOWN et al., 2008). The SDN logically centralized control plane provides a programmable API and OpenFlow as a standardized protocol, facilitating the network management and the creation of new applications. In addition, the *match+action* abstraction of OpenFlow enabled its implementation without requiring arduous modifications to the networking hardware, motivating a widespread adoption of the protocol. These characteristics from OpenFlow provide the opportunity to create new applications in a centralized manner using software, accelerating the innovation cycle.

Although SDN provides opportunity and innovation, the *match+action* abstraction only allows the configuration of a fixed set of switch operations, allowing switches to forward packets from an input port to an output port based on a subset of header fields. Precisely, a packet matches forwarding tables with a fixed set of matching keys in fixed functionality switches. For example, OpenFlow 1.0 supports matching IP addresses and submasks. After matching an entry, the switch may perform actions on the packet, including dropping it, forwarding it to an output port, and sending it to the controller. The protocol also allows populating match entries with parameters for actions, such as the IP source and destination ports. However, due to the fixed set of matching keys and actions supported by OpenFlow, a firmware update from the equipment vendor is necessary whenever a different matching or action set is required, which delays the innovation cycle and hinders the deployment of new functionalities.

Programmable switches are network forwarding devices that allow customized, stateful functionalities to run at the line rate. Programmable switches offer higher versatility and innovation potential by enabling the network operator to develop new functionalities for his network without waiting for industry protocol updates. This flexibility is possible because state-of-the-art programmable switches are based on the *reconfigurable match tables* (RMT) (BOSSHART et al., 2013) technology that enables modification of the logic of matching tables and packet parsers without changing the hardware.

The advent of programming languages such as P4 (BOSSHART et al., 2014a) enables the specification of the functionality of switches that support reconfigurable hardware. The reconfigurable hardware allows programmers to change the packet processing pipeline at runtime. P4 provides the abstractions that hide specific details of the packet processing pipeline from developers. The abstractions include ways

to define how a packet is parsed and deparsed or how packet headers reflect on the switch stateful elements. After specified, it is the job of a compiler to map the packet processing logic set in P4 to the stages of the switch pipeline.

The advantages of programmable switches and the P4 language have led researchers to move functionality previously performed on servers to the network itself, resulting in the paradigm called *In-Network Computing* (INC) (SAPIO et al., 2017). By performing these functions directly on the switch data plane, it is possible to reduce the time it takes to receive a response (propagation delay) by completing requests faster than a typical round-trip time while reducing the workload of the server. There are several successful examples of INC systems, including NetCache (JIN et al., 2017a), which caches frequently accessed key-value items on the switch; NetLock (YU et al., 2020a), which offloads lock management to the switch; and DAIET (SAPIO et al., 2017), which performs aggregation for distributed machine learning training.

In this thesis, we aim to study how to offload functionalities of distributed systems to the network and also understand the advantages and disadvantages of this paradigm. In addition, we aim to investigate and address problems that emerge when we offload functionality to the network, such as fault tolerance and configuration management. To achieve the goals mentioned above, we aim to provide a comprehensive literature overview of the techniques employed to offload the functionality to a programmable data plane and analyze practical case studies in the field. By analyzing a practical use case, we aim to investigate the challenges and limitations of dealing with constraints imposed by the existing data plane hardware, also showing a contrast with other techniques from the literature. We aim to investigate the implementation details of the analyzed methods and the impact on performance according to different factors such as latency, throughput, and resource utilization in different offloading scenarios.

1.2 Research problems

Moving functionalities to the data plane has already presented promising results and benefits but poses several challenges, including finding abstractions for offloading, fault tolerance, and expressiveness. There has been tremendous progress in solving these challenges in the last few years, yet several are still open. Therefore,

in this work, we are going to address the following specific research questions:

Research Question #1: *What techniques, including customized designs, optimized algorithms, and data structures, can be used to overcome the computational constraints from the data plane programming model?*

Building an application with functionality offloaded to the data plane is challenging because of the constraints of the existing hardware (HOGAN et al., 2022). The state-of-the-art switches have a limited amount of memory available (around 10MB) and often limit the amount of *arithmetic logic unit* (ALU) operations we can perform in the pipeline. Consequently, to offload functionality to a switch, a developer must build tailored designs to deal with those constraints. Examples of tailored designs include approximations of floating points (COELHO; SCHAEFFER-FILHO, 2022), using probabilistic data structures instead of deterministic ones (NAMKUNG et al., 2023), and using primitives such as *recirculation* or *resubmit* that can be used for performing additional processing passes for the same packet, but that impact the switch throughput (SENGUPTA et al., 2022). Therefore, developers must evaluate the trade-off between accuracy and performance when building those customized designs.

Research Question #2: *How can we mitigate the impact of switch failures on the data plane of distributed systems and prevent their negative effects on the correctness and performance of the corresponding applications?*

Once we move an application computation from a traditional server to a switch, a failure can make the overall system inconsistent (KIM et al., 2021). Failures in switches can happen and have 2% chances of occurring every 3 months, according to a recent study (SINGH et al., 2021). Because applications may rely on the data plane to keep stateful elements like key-value stores in the switch memory (JIN et al., 2017a; YU et al., 2020a), a switch failure (e.g., crash in fail-stop mode) can affect the corresponding application's correctness and performance. For instance, if the key-value functionality is used to manage the locks of a distributed system, a naive restoration of the system state may lead to deadlocks or race conditions.

Research Question #3: *What approaches can simplify data plane programming and configuration, and how can we systematize this process to reduce complexity*?

Manually dealing with the programmable data plane constraints and deploying fault-tolerance mechanisms for every new functionality is difficult and may take precious time. Creating an INC customized design requires the developer to deal with the constraints using low-level constructs from the P4 language, such as registers entries or *match+action* tables. Consequently, programming and configuring an INC system is complex and often involves manual intervention for each device in the network, which can be time-consuming and error-prone. Manual configuration was acceptable in the past when network devices had fixed functionality. However, with the increasing adoption of programmable data plane technologies and the emergence of complicated INC designs, the complexity of configuring the network is exacerbated. The network configurations need to be more dynamic and flexible, and as such, INC requires new approaches to manage network programmability.

1.3 Contributions

At a high level, this thesis is divided into three main parts, summarized in Figure 1.1s: (1) studying the constraints of the data plane programming model, investigating techniques to circumvent those constraints, and showing a use case for in-network computing, including a tailored design for dealing with the data plane constraints; (2) investigating the impact of switch failures for INC and identifying which are the essential building blocks for making INC systems fault-tolerant; and (3) understanding the need for having more abstract techniques for the data plane programming and configuration, enabling the deployment of different applications without having to configure them or establish protocols to tolerate failures manually.

In the first part of this thesis, our goal is to study the constraints of the data plane programming model and identify common design techniques employed in the literature. Using these design techniques, we aim to understand how to deploy customized data structures and algorithms to overcome those constraints. We focus specifically on the computational limits of the existing PISA hardware, which

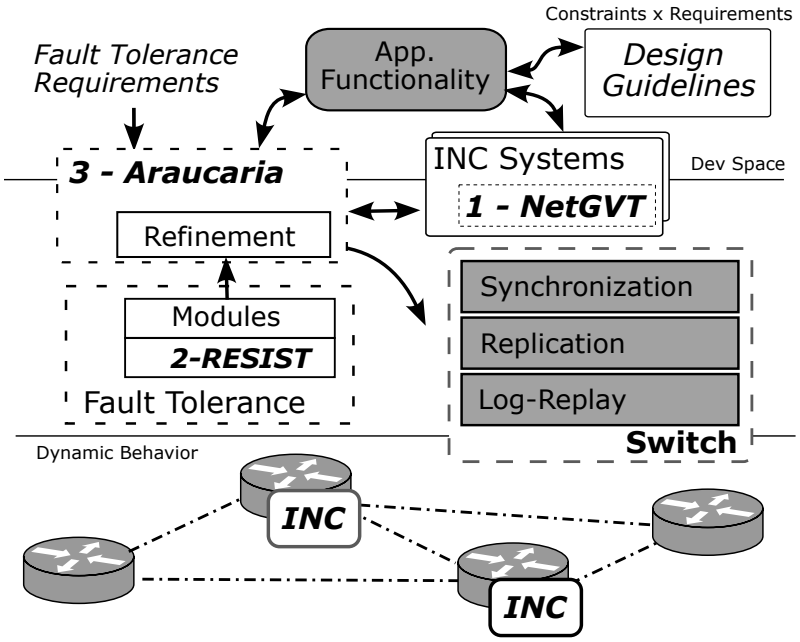

Figure 1.1 – Summary of contributions

is limited to a specific amount of ALU operations and memory at each pipeline. By offloading into the programmable ASIC computation that requires more ALU operations than available in a single pipeline, developers are usually left with implementation choices that lead to throughput degradation. This leads us to our first contribution.

Contribution (1): We first study the data plane constraints, mainly regarding available memory and supported operations of existing programmable switches. After that, we proposed a terminology and categorization for different considerations developers can take when offloading functionalities. We also classify existing INCs from the literature according to our categorization. Next, we benefit from the investigated considerations for designing and implementing NetGVT, which moves virtual time synchronization to programmable switches. Computing the virtual time synchronization in a programmable switch would consume more stages than what is available in a single pipeline of state-of-the-art switches. Thus, we designed an efficient data structure to make the virtual time synchronization fit in the data plane with minimal throughput degradation. NetGVT experiments in a Tofino testbed show that our in-network solution outperforms a server-based solution in terms of simulation completion time.

After offloading the functionality to the data plane, we identified a critical limitation for INC: *switches can fail*! Then, we decided to study the impact of failures on the application's correctness in the second part of this thesis. To investigate the effects of failures, we analyze a widely studied set of INC systems (e.g., aggregation, key-value storage), contrasting whether different applications require different consistency notions to be correct. Then, we investigate techniques to recover from INC failures without adding significant overhead for non-failure scenarios. This leads us to our second contribution:

Contribution (2): We show that different INCs may be affected differently by switch failures and that different consistency semantics could be employed to circumvent them. We also present techniques to tolerate INC failures, aiming to cut the overhead for non-failure scenarios. To demonstrate the feasibility of our methods, we implement them in RESIST, a system to make INC fault-tolerant. RESIST is a system that enables us to augment INCs with fault tolerance building blocks, allowing us to define the necessary consistency semantics to operate. The central insight in RESIST is that we can decouple consistency from the replication mechanism, achieving strong consistency notions without overhead for non-failure scenarios. Our experiments in a Tofino testbed show that an INC system augmented with RESIST can tolerate failures with minimal overhead in non-failure scenarios. We also present BMv2 emulations that reveal the behavior of the system during recovery time, enabling us to better understand the results in scenarios with an increasing number of hosts.

We observe that offloading functionality to the data plane and making it resilient to failures is a complex and specialized process. INC operators must manually set the code that implements fault tolerance requirements in P4 using domain knowledge and source code analysis. This includes a logic constrained by the asynchronous behavior of programmable switches, requiring the implementation of customized failover mechanisms. These manually implemented designs may take time to configure and impact latency, throughput, and correctness if the parameters are misconfigured. For example, deploying a new RESIST experiment requires specialized knowledge and may take time. In the third part of this thesis, we investigate techniques to automate the INC configuration using more abstract specification techniques. We work to understand this subject in a specific INC scenario, considering

the need to configure multiple devices for fault tolerance.

Contribution (3): This study investigates the concept of intent-based networks (IBN). We study a particular use case on specifying intents for INCs in which we want to define fault tolerance for INC at an abstract level. Our study provides a precise process that enables us to refine high-level fault tolerance requirements into concrete INC configurations. We propose the architecture of Araucaria, a system that provides a high-level language and a refinement process for INC fault tolerance. To that end, we investigate the structure of intents used in the literature and define the necessary constructs for an intent-based language that enables the specification of fault tolerance requirements. Aiming to translate the high-level language intents to low-level INC code, we propose a refinement process that translates and instruments the INC code and generates configuration for the instrumented P4 switch. To conclude the intent-refinement life-cycle, we conduct an investigation to ensure that an intent is consistently guaranteed, even in the event of network modifications. This involves monitoring availability metrics and reconfiguring the data plane in the event of crashes. We show a working use case, illustrating how we generate configurations and source code for making NetGVT fault-tolerant with RESIST.

1.4 Outline

The rest of this thesis is organized as follows. We begin discussing some background on programmable data plane technologies and in-network computing (Chapter 2). The programmable data plane background includes the historical perspective and a detailed description of the packet processing architectures and the P4 programming language. The in-network computation background introduces the concept, highlighting its advantages and exemplifying a use case. The following chapters describe the research we have been doing for each part of this thesis. In particular, the needed additional background and related work for each part of this thesis will be provided in each chapter. In the first part of the thesis, we present our study of how we can offload functionality to the data plane subject to the constraints of the programmable devices (Chapter 3). We present the NetGVT system design, including a tailored design for the data plane and its evaluation. In the second part of this thesis, we describe our investigation of the need for fault tolerance (Chapter 4). We propose the system design of RESIST, including the building blocks for

decoupled fault tolerance and our evaluation of the system. Chapter 5 provides an overview and architecture of Araucaria, a system that enables the refinement of fault tolerance requirements for INC. We also present a detailed use case, showing how to make the INCs studied in this thesis fault tolerant. Finally, Chapter 6 summarizes our contributions, discusses future research on in-network computing, and concludes the thesis.

2 BACKGROUND AND MOTIVATION

In this chapter, we present the essential background for reading this thesis. First, we discuss programmable networks in Section §2.1, briefly discussing its history and OpenFlow. Secondly, we present details about data plane programmability in Section §2.2, including packet processing architectures and programming languages. Finally, we present background on in-network computing in Section §2.3.

2.1 Programmable networks

Programmable networks are a type of network architecture that enables network functionality to be defined and modified dynamically through software (FEAMSTER; REXFORD; ZEGURA, 2014b). Programmable networks can include various components, such as programmable switches and routers, FPGAs, SmartNICs, and virtualized servers. These components can be programmed to perform specific tasks or implement custom protocols, making the network more flexible and adaptable to changing requirements.

2.1.1 Historical perspective

Although programmable networks are becoming more common, the idea traces back to the 90s, when *active networks* were introduced. The concept of active networks aims to allow developers to run customized programs on switches and routers. This was motivated by the need for researchers to implement their protocols and functionalities without waiting for the IETF standardization process to deploy new functionality, as mentioned in (WETHERALL; TENNENHOUSE, 2019).

In active networks, researchers idealized two ways of processing code in routers:

- ***Capsules:*** the first one is replacing packets by *capsules*. Capsules refer to a forwarding routine that an active node processes. Some of these routines are common to all nodes; others are application-specific and rely on a code distribution engine to find an active node (WETHERALL; GUTTAG; TEN-

NENHOUSE, 1998).

- ***Programmable Switch/Router:*** the other approach was the *programmable switch*, enabling the loading of programs in the switch that would control the infrastructure (TENNENHOUSE; WETHERALL, 1996).

However, the idea of active networks had no "transference of results" to industry and ended up not having a widespread deployment (WETHERALL; TENNENHOUSE, 2019). Despite not being widely deployed, researchers envisioned a programmable network for the first time, a radically different network design. Active networks envisioned network devices exposing their state to a programmable interface, a concept later revisited by OpenFlow and data plane programmability with P4 (FEAMSTER; REXFORD; ZEGURA, 2014b).

Meanwhile, network operators needed help managing and configuring their networks. Networks were composed of devices from different vendors, including distributed protocols, often requiring too much manual effort to configure them. Typically, the data and control plane were tightly integrated with the devices, making performing traffic engineering or debugging tasks challenging. To simplify management, researchers started arguing for separating the control from data planes, building a more centralized controller instead of the distributed one required by the tight integration of the planes (FEAMSTER; REXFORD; ZEGURA, 2014b). Several researchers made significant contributions showing that it was possible to separate the planes, but this separation was only concretely achieved with the emergence of OpenFlow (MCKEOWN et al., 2008).

2.1.2 SDN and OpenFlow

OpenFlow was conceived as an approach for researchers to run experiments in their networks efficiently (MCKEOWN et al., 2008). OpenFlow leveraged a necessary common abstraction between multiple switches and routers: the *match+action* abstraction. Using this abstraction, an OpenFlow API allowed external programs to change entries of flow tables from the switch chip, typically implemented as TCAM. The match+action abstraction was simple but powerful and quickly became widespread because the flow table hardware was common between existing switch chips. This was advantageous for equipment vendors because they did not

need to change their switch chips completely.

Meanwhile, researchers and operators could use SDN to study and deploy their network solutions. For example, a network operator could write a control plane application to balance the network load effectively. This application could require the switch to forward packets from a new flow to the controller, run a custom routing algorithm, and define new forwarding entries for subsequent packets. The strategy of changing table entries is opposed to how load balancing was implemented in traditional networks. A conventional network would be limited to changing the link weights of the control plane protocol (e.g., OSPF) implemented by the vendor, not allowing operators to build their solution.

OpenFlow played a significant role in developing the Software-Defined Networking (SDN) technology. SDN provides a logically centralized controller abstraction, being able to interact with the data plane using APIs such as OpenFlow (KREUTZ et al., 2014). The rise of SDN motivated the notion of network operating systems, such as ONOS (BERDE et al., 2014) and NOX (GUDE et al., 2008). In addition, because of concerns about the risks of having a centralized controller, the SDN community has pushed into the idea of having a distributed control plane, maintaining consistency across the distributed state, and providing support for general networking applications.

By motivating an open interface between the data and control planes, OpenFlow networks allowed researchers to experiment with new ideas and simplified management. Although OpenFlow has brought those benefits to network operators, the degree of programmability in the network using SDN is still small. An OpenFlow switch has a Flow Table that maps flows to a specific action. The only programmability in the switch is made by changing the flow table entries through the OpenFlow API. This allows for executing particular actions, but the OpenFlow match+action hardware can match only a fixed set of packet header fields and protocols. In addition, the set of available actions is fixed, including only dropping, forwarding, and sending to the controller actions. However, this has changed with programmable packet processing architectures.

2.2 Programmable packet processing architectures

A packet processing pipeline defines a network device's sequence of operations to process a packet (GUNTURI; JOHNSON; SEOW, 2005). The components of a packet processing pipeline may vary according to different pipeline architectures. The programable packet processing architecture allows the network owner to define data plane functionality using software artifacts. The switch functionality is often expressed using domain-specific languages (BOSSHART et al., 2014a; SONG, 2013) and then is tailored into an abstract data plane model (HAUSER et al., 2021). The resulting code is then compiled into a packet processing architecture that supports the data plane model. A concrete example of packet processing architecture is the Protocol Independent Switch Architecture (PISA), which generalizes the Reconfigurable Match-Table (RMT) (BOSSHART et al., 2013) model.

Reconfigurable Match-Table. The RMT model is a technology that makes match+action tables dynamically programmable without modifying the hardware. This distinguishes switches that use RMT from OpenFlow switches in several ways. In OpenFlow, the packet processing architecture transitioned from the Single Match Table (SMT) architecture to the Multiple Match Tables (MMT) (PFAFF et al., 2012). While in the SMT, the switch is composed of a single match+action table, the MMT has multiple match+action tables. However, the number of tables, their pipeline, their match types, and sizes are determined during the manufacture. Consequently, these parameters always remain the same, limiting flexibility.

In contrast to OpenFlow switches, RMT allows network administrators to define the fields within the match+table for their specific needs. Furthermore, the RMT model enables changing the width and depth of a match+table, supporting different sizes of entries and tables with other types. The action triggered and the number of available actions by a match are also programmable, being able to run customized functionality in case a packet matches an entry in a table. Finally, the programmer can define the topology between match+action tables in the pipeline instead of determining it during manufacture.

PISA Architecture. The PISA architecture leverages the RMT technology to provide line-rate packet processing. Figure 2.1 illustrates the PISA architecture.

In the PISA architecture, packets go through a packet parser, which instantiates user-defined protocols to the *packet header vector* (PHV). The PHV stores the

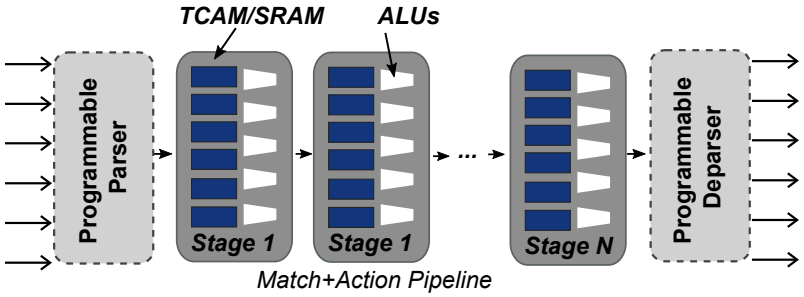

Figure 2.1 – PISA Architecture

necessary header fields for the program, such as TCP, IP destinations, and essential *metadata*. *Metadata* are per-packet variables that store temporary values to assist packet processing, such as input and output ports. After the parser processes a packet, it follows to a packet processing pipeline. A packet processing pipeline can comprise several constructs, such as *match+action* tables and registers. Matches can be in several forms, such as the longest prefix match, ternary match, or exact match. Each match corresponds to executing an action that runs computation and performs storage operations. *Match+action* tables can run on TCAM or SRAM, while actions are implemented using ALUs and must be able to run at line rate (SIVARAMAN et al., 2015). Each pipeline has a fixed set of stages, and each stage has available registers that store persistent states. Each stage also has a constrained set of ALU operations, divided between stateless and stateful. After the control flows process a packet, packet headers are emitted by a deparser. The deparser reconstructs the packet headers using the PHV variables and sends the resulting packet to an output port.

2.2.1 P4 language

The P4 language (BOSSHART et al., 2014b) is a high-level specification language for packet processing processors. The language is designed to make it easier for developers to describe the packet processing pipeline. P4 provides an abstraction layer above the PISA architecture, making the P4 compiler's job to map the specified functionality to the hardware stages (HOGAN et al., 2022).

P4 is commonly referenced as a declarative language (SHAHBAZ; FEAMSTER, 2015; KHERADMAND; ROSU, 2018; EICHHOLZ et al., 2022), aiming to provide a high-level abstraction, and releasing developers from the need to worry

about how the high-level specification is implemented using the low-level data plane hardware. Packet headers can be declared similarly to *structs* in C. The parser can be determined using a state machine approach, where states often correspond to a portion of the packet header, and transitions between states are transitions between headers. During runtime, the parser processes the packet by extracting the values of bits from the packet to internal variables of the program, which can then be accessed and modified in the other processing elements.

```
1  header ethernet_t { macAddr_t dstAddr;
2                      macAddr_t srcAddr;
3                      bit<16>   etherType;}
4  parser MyParser(packet_in packet, out headers hdr,
5                  inout metadata meta,
6                  inout standard_metadata_t standard_metadata){
7      state start {
8         packet.extract(hdr.ethernet);
9          transition select(hdr.ethernet.etherType) {
10             0x800: reject;
11             default: accept;
12 }}}
```

Figure 2.2 – P4 header and parser examples

Figure 2.2 presents an example of a P4 header and parser. The header type `ethernet_t` expresses the standard format of ethernet frames. The parser consists of only one state that extracts the bits from the ethernet header and instantiates them to an internal program variable. A transition selects the following state by checking the ethernet `etherType` and rejecting it if it is `0x800` (the IPv4 type), accepting it otherwise.

The P4 language also provides the abstraction of a control block. A control block can be used to specify how to process packets. Each control block may include primitives to modify the contents of a header and the actions the program may contain. Tables can be used in control blocks. A packet can match a table entry the way the programmer defines it. Once a packet matches a table entry, the table may process the packet using an action. Actions may be composed of primitive actions that change the header or modify switch variables. An action can have input parameters; an external program configures those parameters and the table entries that initiate the action. A control block may also maintain stateful or stateless registers that can be modified at runtime without external input. Finally, control

blocks include an apply block, which enables specifying in an imperative way how packets flow from one table to another. The control flow may consist of specific functions, conditions, tables, and action references.

```
1  control MyIngress ( inout headers hdr, inout metadata meta,
       inout standard_metadata_t standard_metadata ) {
2      action drop() {
3          mark_to_drop( standard_metadata);}
4
5      action forward(egressSpec_t port) {
6          standard_metadata.egress_spec = port;}
7
8      table eth_table {
9          key = { hdr.ethernet.dstAddr: exact;}
10         actions = { forward; drop;}
11     }
12     apply { eth_table.apply(); }
13 }
```

Figure 2.3 – P4 Control and forwarding examples

Figure 2.3 exemplifies a control block written in P4 language. The control block definition begins with the name `MyIngress` and two input variables: `hdr` for packet headers, `meta` for user-specified metadata, and `standard_metadata` for primitive metadata. The control block manipulates those variables during its execution. Two actions are defined inside the control: the `drop` and the `forward` actions. The `drop` action, which has no parameters, calls a primitive function `mark_to_drop`, that marks standard metadata accordingly. The `forward` action has a parameter `port`, which sets the standard metadata corresponding to the packet output port to the value passed as an argument.

Subsequently, a table called `eth_table` is defined. This table uses the packet header named `hdr.ethernet.dstAddr` as a key and matches packets using the exact primitive. Two actions can be called when a packet matches an entry in this table, namely `drop` or `forward`. In the apply block, this table is referenced using a primitive function called `apply`. It is important to note that the entries in the table are typically installed by the control plane.

2.2.2 Data plane programming challenges

Although P4 and the programmable packet processing pipelines provide more flexibility for building and managing networks, they come with several challenges. The state-of-the-art hardware and the languages used to specify functionalities present characteristics that make data plane programming challenging.

2.2.2.1 Hardware constraints

Despite their flexible processing capabilities, programmable switches have several design constraints to ensure that in-network functionalities run at line rate.

Limited memory. The state-of-the-art switch hardware has limited memory. For example, the amount of SRAM in a Tofino switch is around 10MB (LI et al., 2022). In addition, the amount of concurrent memory access is also constrained. When a packet arrives, the switch program can read only one memory address, retrieving a limited set of elements stored in a memory block. Consequently, we can not create a program that reads the entire memory block allocated for a register vector at once (BEN-BASAT et al., 2018).

Static Memory Allocation. State-of-the-art switches also do not enable dynamic memory allocation, limiting developers to allocate space for variables during development statically. Changing the memory allocation would require modifying the data plane source code, recompiling and reloading it. In the meantime, the switch needs to be stopped and restarted (ZHU et al., 2022). Also, match+action tables can not be modified at the data plane, requiring the intervention of the control plane to change table entries.

Read/Write only once. The switch programming model also does not enable multiple accesses to the same memory region in the same pipeline. This implies that if a stateful variable was accessed once when a packet traverses the pipeline, the same stateful variable could not be reaccessed in the same traversal[1]. Consequently, operations that depend on the same memory element must be performed on different pipelines or unrolled using primitives such as recirculation or resubmission, i.e., send the packet back to a *loopback* port (BEN-BASAT et al., 2018). However, recirculation primitives are costly in terms of throughput.

[1] Only once semantics differs from the limitation on concurrent access, which is about the number of memory addresses a single read can access from a memory block.

Limited computation on stages. A switch pipeline is composed of a few stages. Each stage can run simple ALU operations, restricted to simple arithmetic, logical, and bit manipulation (BOSSHART et al., 2013). Because float-point operations are often expensive, they are often not implemented in hardware and thus unavailable for use on stages. The number of ALU operations performed in each stage is also limited to preserve the line rate. In addition, branches are limited inside a stage, not allowing complex comparisons or more than two branches inside the same stage.

Fixed amount of stages. The switch pipeline is short to ensure the line rate is achieved (BEN-BASAT et al., 2018). Because the pipeline is short, the number of stages is limited; consequently, the amount of ALU operations a program can compute is limited accordingly. An alternative design is necessary if a program requires more ALU operations than can be performed in a single pipeline. Recirculation can be a possible solution, as pointed out in (BEN-BASAT et al., 2018).

Lack of shared memory. Modern switches are composed by multiple pipelines (CHIESA; VERDI, 2023). These multiple pipelines improve parallel processing in switches and help diminish race conditions, but they come with their costs. Typically, each pipeline has its own variables, including registers and match+action tables. Switches do not share memory between pipelines, thus creating synchronization problems for offloaded functionalities.

2.2.2.2 P4 programming challenges

Naturally, the P4 programming language inherits the constraints from the hardware. However, beyond the hardware limitations, the language poses its own set of challenges.

Lack of loops. The current language specification does not allow developers to use loops to iterate over state variables. The absence of loop structures requires unrolling strategies to iterate over memory elements using if-else statements and decomposing memory elements. This occurs because programmable data planes do not enable recursion and require sending the packet back to the beginning of the pipeline to process the same packet more than once (SHAH et al., 2018). However, due to the lack of loops at the language level, implementing simple operations may lead to more code voluminosity because of code repetition (ALCOZ et al., 2022).

Code Repetition. Code repetition is easily visible in P4 programs (HOGAN

et al., 2022). Repeated data-structure declarations are often found in data plane programs, as well as repeated actions to manipulate those data structures (HOGAN et al., 2020). Finally, the apply blocks often reference the repeated data structures multiple times. This is a consequence of both the lack of parametrization in the language and the lack of abstractions in the language, which requires manually dealing with the hardware constraints (ALCOZ et al., 2022) and repeating the action code for different registers or tables, even running the same logic.

Ineffective Abstractions. Although P4 is meant to be a high-level language, engineers and developers often say it is *not abstract enough* (LI et al., 2022; GYÖRGYI; LAKI; SCHMID, 2023; HOGAN et al., 2020). P4 often subjects the developer to hardware constraints, thus requiring to reason about low-level hardware while coding, often leading to trial and error and suboptimal designs (LI et al., 2022; GYÖRGYI; LAKI; SCHMID, 2023). Nonetheless, the P4 programming language mixes two programming paradigms: a state-machine definition style for parsers and an imperative definition for control flows (SONI et al., 2020). This can hinder development, making it a time-consuming task compared to writing the code for the same functionality for a server.

Limited feedback. The lack of feedback from the P4 compiler is also limited or even may present bugs (GYÖRGYI; LAKI; SCHMID, 2023). If the compilation fails, it is not uncommon for the compiler not to show the reason for the problem in the specific code. This may lead developers to debug code using *adhoc* ways, such as changing parameters to "magical values" or simplifying comparisons in if/else statements. This makes development and debugging a trial-and-error process (HOGAN et al., 2020).

2.3 In-network computing

In-network computing (INC) is an emerging concept in computing networks. Also called In-network Computation (SAPIO et al., 2017), NetCompute or In-Network Compute (BENSON, 2019), should not be confused with Computing in the Network (COIN) (KUNZE et al., 2023b), that still has the scope open (KUNZE et al., 2023a). INC has been used to characterize systems with functionality offloaded to the programmable data plane of networking devices (MICHEL et al., 2021). Instead of introducing new equipment to the infrastructure, in-network com-

puting leverages existing devices within the infrastructure, particularly those that are P4-configurable, to not only handle traffic forwarding but also to execute customized functionality (ZILBERMAN, 2019). This approach can potentially reduce the need for specialized equipment, such as accelerators or middleboxes.

2.3.1 The advantages of INC

Adopting an offloading strategy to the network is mainly motivated by the performance benefits it can provide.

Latency. With INC, it is possible to reduce latency by intercepting and processing packets in the data plane of networking devices instead of sending packets to be processed by the end servers. As such, instead of requiring a request to complete an entire RTT by sending the packet to a server, some application packets can be quickly processed in the network hardware and forwarded to their end destination. This can save the latency required to send the packet to the end host and all the other nodes in the path (ZILBERMAN, 2019).

Bandwidth. By processing packets at the network devices, INC also saves bandwidth in links from the switch running the INC to the servers running the end host functionality. By saving bandwidth, it is possible to avoid congestion and obstructions that often cause buffer occupancy, packet drops, and performance degradation. Thus, INC can enhance application performance, freeing a portion of network links to process packets from other applications at higher rates (KIANPISHEH; TALEB, 2022).

Throughput. Another possible advantage is throughput increase. Once existing switch hardware can process packets at 12Tb/s, applications can be continuously served. This occurs because switches are designed as pipelines and do not delay packets because of queuing or stalling delays (ZILBERMAN, 2019). In addition, programmable switches ensure the program runs at line rate, ensuring that any offloaded program can operate at this scale. Therefore, beyond reducing the time to process packets (e.g., latency), they can keep processing operations per second in the scale of terabits/s (e.g., throughput), which is higher than the rate usually observed on servers.

Energy Consumption. Power consumption is a surprising advantage of INC. Programmable switches hosting INCs consume less or the same amount

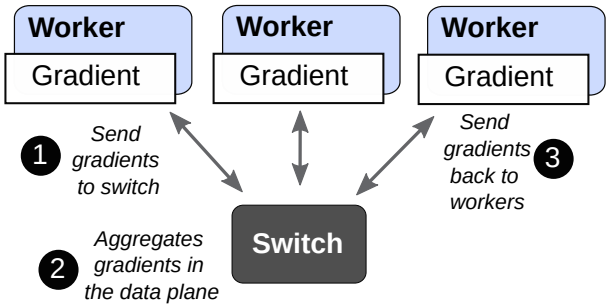

Figure 2.4 – In-network computing example

of resources compared to traditional switches. By offloading INC functionalities to a programmable device, it is possible to observe the same amount of resource consumption observed in an idle switch (TOKUSASHI et al., 2019). Furthermore, increasing the traffic rate increases energy consumption according to the rate in a constant way. Therefore, when compared to servers, which can double the consumption as the rate increases, switches can scale to larger traffic rates with small increases in energy consumption. Nonetheless, although a server consumption is generally lower than a switch, if we assume switches are already available in the infrastructure, it is possible to save server consumption by moving functionality using INC (TOKUSASHI et al., 2019).

2.3.2 Use-cases

Recent efforts in INC have shown many applications that can be used in the network. The most straightforward use cases are network functions, such as DDoS detection, load balancing, and NAT. More unexpected use cases include systems like ML aggregation (SAPIO et al., 2021) for data-parallel training and inference (SANVITO et al., 2018), which has been studied in several applications such as traffic classification, controlling robot arms, and even computer vision (GLEBKE et al., 2019a). Caching key-value storage (JIN et al., 2017a; JIN et al., 2018a) and lock management (YU et al., 2020a) is also possible, enabling fast database transactions in data centers.

To better illustrate the usage of INC, Figure 2.4 presents an example of an INC system for aggregation. The figure shows a system that performs machine learning aggregation inside the programmable data plane of the network device,

such as DAIET (SAPIO et al., 2017). Instead of requiring a centralized server to perform the aggregation, workers send their gradients to a switch responsible for aggregating them. The resulting gradient is encapsulated into a packet and sent back to workers to update their local model. In particular, the machine learning aggregation has some characteristics in common that make their offloading suitable for programmable data planes (MICHEL et al., 2021). By running the aggregation on switches, it is possible to reduce the amount of data exchanged and thus reduce network utilization. The resulting gradients can be computed faster because programmable data plane devices are closer to the workers and run at a line rate. Finally, since most of the operations are commutative and associative, they can be applied in an arbitrary order, being suitable for processing packets at a high concurrency rate on the programmable hardware (MICHEL et al., 2021).

3 OFFLOADING COMPUTATION SUBJECT TO LIMITATIONS OF THE PDP

In this chapter, we discuss the first part of this thesis, which addresses the research problem related to dealing with the hardware limitations of the data plane. Section §3.1 first introduces the research problem and proposes a terminology and categorization for design guidelines that circumvent these constraints. Section §3.2 motivates a use case to study how to handle the constraints when offloading computation to the data plane. Section §3.3 presents the design of NetGVT, used to study in practice the customized solutions by offloadind virtual time synchronization to switches. Section §3.4 presents the results we obtained with a prototype of the system. Section §3.5 presents the related work for this chapter. Finally, Section §3.6 sheds light on the complexities and lessons learned in the chapter, categorizing design guidelines to other existing INCs from the literature.

3.1 Investigating INC design guidelines

Distributed systems are often highly versatile and can support rich requirements due to the abundant hardware and general-purpose architectures employed. However, replicating the same functionalities of a traditional distributed system in an in-network solution is a significant challenge due to the limitations of programmable switches. The capabilities of programmable switches are considerably more constrained than their server-based counterpart, often resulting in a reduced feature set. In-network solutions must operate within the constraints of programmable switches and often face difficulties in supporting complex functionalities with rich requirements.

Devising a tailored design for in-network solutions can be a challenging task. The system designer must often consider different optimization goals, such as maximizing throughput while minimizing switch resources. In addition, identifying whether alternative solutions can fulfill the essential application requirements is necessary but is only possible with knowledge of application behavior. For example, a solution that considers only the network functionality may create a solution that violates correctness conditions of an application. Although some research pa-

pers propose solutions to automatically deploy a functionality described at a high level (XU et al., 2023; HOGAN et al., 2022), the existing INCs are still manually developed using P4, the *de-facto* programming language for the data plane.

Instead of attempting to automate the resolution of data plane constraints, we adopt a developer-centric approach. We investigate common design considerations that programmers use while developing INCs to work around these restrictions. Next, we categorize these considerations and provide terminology that developers can use to discuss different design alternatives. Specifically, this categorization describes design considerations developers can use when a functionality does not preserve specific programmable switches' constraints.

We divide the design considerations based on the constraints they aim to address, including the lack of memory and limited computation resources.

3.1.1 Memory management

When offloading a functionality, the constrained memory of existing programmable data planes requires intelligent memory management mechanisms. As the layout of conventional data structures often cannot be directly mapped to programmable switches memory, it is necessary to tailor them and employ specific memory management techniques. We categorize these techniques according to different aspects: (1) those employed to handle memory-intensive functionalities and (2) the functionalities that require multiple read and write operations.

Memory Intensive Functionalities. Different techniques can be employed to offload functionalities that consume a large amount of memory:

- **Distributed Designs.** To support functionalities that consume a large amount of memory, it may be necessary to consider building designs that incorporate methods for storing only the most crucial features in the data plane memory and use servers as a fallback option (JIN et al., 2017b). One approach to achieve this is by selectively keeping in the data plane elements relevant to a specific optimization objective (YU et al., 2020a). For example, instead of storing the entire key-value storage in the data plane, keeping only the most frequently accessed key-value pairs in the data plane is an alternative for optimizing throughput. In addition, distributed designs in the data plane

can also offer benefits (LAO et al., 2021). This points to using the memory of multiple data plane devices, such as smartNICs and switches, to collaborate. When a key miss occurs in one device, it can be routed to the subsequent data plane device with the key-value item. However, distributed designs may introduce additional overhead due to the delay in accessing the distributed memory.

- **Approximations.** Another approach to tailor memory-intensive functionalities is to utilize approximations instead of storing only raw data. For specific applications, instead of storing complete data streams in the data plane, it is possible to consider keeping a combination of statistical or aggregated values (KIM et al., 2015), such as quantiles or moving averages (BASAT et al., 2020). Approximations can be achieved using data structures like bloom filters, hash maps, or sketches, which organize elements based on a hash of their key and are easily implemented in the existing hardware since they use fewer memory elements. While storing approximations in these data structures allows for practical implementation, it might lead to some loss of accuracy due to collisions caused by overlapping keys. Therefore, this approach is only suitable for certain applications as an alternative solution and is not enough for cases like storing locks for concurrency control (YU et al., 2020a), which can lead to race conditions in cases shared across multiple processes. In such cases, more precise data storage techniques are necessary to ensure proper synchronization and avoid conflicts.

- **Compression.** Compressing the information to fit in the data plane is also an alternative. Data can be compressed before transmission and decompressed upon reception or compressed by the switch using tables and ALUs (SAPIO et al., 2021). A concrete example of compression is found in in-network aggregation, which uses quantization to transform floating point values used as parameters during aggregation into smaller integers. By using quantization, existing works on *in-network aggregation* (INA) reduce the size of models and make their aggregation possible in the network.

Read/Write. Functionalities that require intensive reading or writing into switch stateful memory are also subject to tailored designs. Because the state-of-the-art hardware does not allow reading or writing the same register more than once in

the same pipeline, tailored designs often can either reiterate a packet or decompose the data structures into different registers.

- **Reiterate.** To circumvent the constrained memory access in the same pipeline, we can leverage recirculation or resubmitting primitives, allowing a packet to traverse the pipeline more than once. These primitives can be employed to iterate over the same memory element repeatedly (SONCHACK et al., 2021). After resubmitting or recirculating, the switch code can identify a reiterated packet and read or write the memory element once more (SENGUPTA et al., 2022). However, this benefit comes with a cost. Once the packet is recirculated, the packet will occupy the buffers again, degrading the switch throughput (WU et al., 2019).

- **Decomposition.** Another alternative to circumvent constrained memory access is to divide the memory element that needs to be accessed more than once into multiple elements. Decomposing a data structure (e.g., an array) into different data structures allows a packet to read/modify them at the same pipeline, creating the same effect of reading/modifying the original data structure multiple times. While reiterating packets can impact the throughput, the decomposition can lead to the use of more pipeline stages.

3.1.2 Computational resource

Beyond dealing with the limited memory of the existing data plane hardware, the small amount of computational resources in the data plane also requires tailored designs. Computational resources in this discussion refer to both (1) the set of primitive operations available and (2) constraints regarding the limited number of pipeline stages (around 12 stages) and the number of operations in each stage. These constrained characteristics make running functionalities that require more complex operations or a large number of operations hard.

Complex Operations. Complex operations, such as tensor aggregations, matrix multiplication, or division, can be easily implemented on servers. However, implementing those operations is not easy in the existing programmable hardware. Instead, alternative approaches need to be used.

- **Approximations.** Computing approximations of the desired complex op-

erations are often employed (SAPIO et al., 2019). This is especially relevant for calculations involving floating-point numbers, which can be approximated within the data plane hardware itself (COELHO; SCHAEFFER-FILHO, 2022).

- **Simplifications.** In some cases, a viable approach is not trying to approximate the computations but instead finding a condensed and simplified algorithm or operation by removing unnecessary elements. An illustrative example is the deployment of a Neural Network. Neural networks (NN) comprise aggregation operations on weights and activation functions, which are complex to run in a programmable switch. An alternative is translating the NN into a binary neural network (bNN) by simplifying aggregation and activation functions to binary operations. This makes executing an NN more feasible in the data plane (SIRACUSANO et al., 2022).

Compute Intensive. Functionalities that require the computation of a large amount of instructions are limited by several constraints of the data plane. This is not only because pipelines have a limited amount of stages but also because those stages may need to run essential functionality for the network, consuming some of the resources available.

- **Memoization.** One alternative for handling an intensive amount of operations for a packet is memoization. Memoization can take form by processing a packet in the pipeline and storing preliminary states. Subsequent packets that need to compute the same state can rely on the memorized result instead of calculating it again.

- **Division of Labor.** Another alternative for more intensive computation is partitioning the calculation between multiple smaller packets that perform the calculation as they traverse the pipeline (BASAT et al., 2020). One example of this approach is investigated in Qpipe (IVKIN et al., 2019), which computes quantiles from a sample of packets in the data plane. To compute the quantiles efficiently, it relies on packets not sampled, named worker packets, to traverse the pipeline and perform part of the tasks before being forwarded to their output port. Since the data plane functionalities are enforced to run at the line rate, this technique does not affect the performance of the worker

packets. Furthermore, it does not rely on primitives such as resubmission or recirculation to perform the remaining tasks, which could reduce throughput.

3.1.3 Summary and takeaway

In summary, we observe that offloading functionality to the existing data plane models requires customized designs to overcome problems such as the lack of memory and limited computation capabilities. Figure 3.1 categorizes the methods we can rely on.

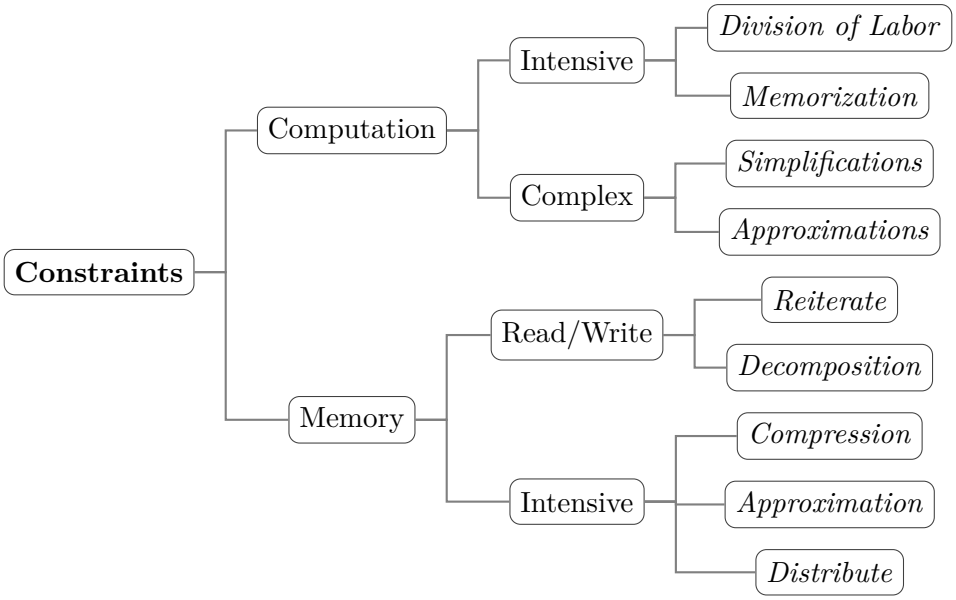

Figure 3.1 – Summary of techniques for each specific functionality category

In the first part of the thesis, we aim to investigate more concretely the existing techniques used to deal with the limitations in the data plane. To achieve this, we present a case study in which we propose and design a new functionality to be offloaded from a distributed system to the switch. Specifically, we focused on offloading the global virtual time synchronization functionality from distributed simulations to the switches. The computation of virtual times requires calculating a minimum value between several values, which is not easily performed in the data plane, especially considering that virtual times change discretely. Thus, a tailored design suitable to the switch constraints becomes necessary. By devising this distributed design, we illustrate a practical use case of how we can use the design considerations discussed in this section to handle the data plane constraints.

3.2 Motivating in-network GVT

Large-scale distributed simulations play an essential role in scientific research and many other domains, including weather forecast (MAZZURANA, 2021), military training (HANNAY; BERG, 2017), large-scale system design simulation acceleration (e.g., VLSI layout (GONSIOROWSKI; CAROTHERS; TROPPER, 2012; BRITO et al., 2015)), manufacturing and supply chains design (MORINAGA; ARAI; WAKAMATSU, 2012; SARLI; LEONE; GUTIÉRREZ, 2016), etc. Traditionally, different parts of a simulation model run on distributed servers or in a cluster and need to be periodically synchronized to exchange timing information and establish a *global virtual time* for consistent event processing (ABEYDEERA; SANCHEZ, 2020). Global virtual time (GVT) (JEFFERSON, 1985) significantly impacts the performance of distributed simulations. Efficient computation and propagation of GVT ensure synchronization across distributed processes, reducing waiting periods and improving overall system performance and simulation speed.

Developers usually have two choices for synchronizing simulations using GVT: a centralized server-based mechanism to coordinate the computation or a decentralized synchronization algorithm for application processes to perform the calculation on servers. Examples are the Chandy and Misra Null-message protocol (CHANDY; MISRA, 1979), the Granted Time Window algorithm based on the Distributed Snapshot protocol proposed by Mattern (MATTERN, 1989), and the Time Warp Mechanism (JEFFERSON, 1985), which uses rollback to recover from causal consistency violations. However, these schemes impact the simulation completion time due to the RTT message propagation delay and the software stack latency while computing a new GVT value in servers. As such, server-only implementations may become a bottleneck for synchronization because of the processing, buffering, and transmission operations (KUZNIAR et al., 2022; NORONHA; ABU-GHAZALEH, 2002). Even if GVT computation is pushed to a server placed in an optimal location in the network to minimize RTT, the delay will still be imposed because of the server software stack.

In this context, the recent advances in programmable data planes create a different alternative for GVT computation. Instead of using server-based protocols, we can write a P4 program to intercept synchronization packets and perform the computation necessary in the networking devices. By offloading the computation to

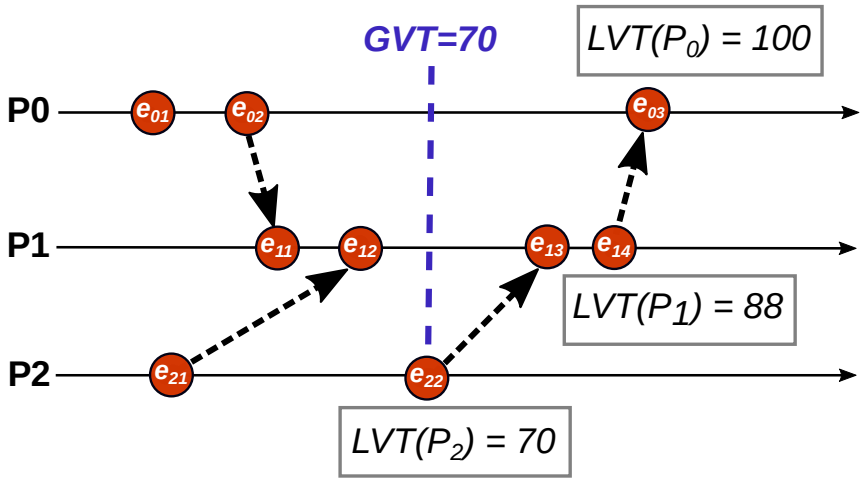

Figure 3.2 – An example of GVT value in an event diagram.

the network, there is an opportunity to revisit GVT synchronization protocols and process their information, saving bandwidth and reducing the propagation delay.

Different deployment scenarios can benefit from offloading GVT synchronization to the network. For example, a cloud provider can run an in-network GVT service for its tenants that need to run simulations. Alternatively, an enterprise can incrementally deploy switches near a server cluster to accelerate existing simulations. Devising an in-network GVT synchronization mechanism that relies on programmable switches will reduce the propagation delay compared to a server-based deployment. As such, it can speed up GVT synchronization and accelerate large-scale distributed simulations.

3.2.1 Distributed Simulations and GVT

A distributed simulation can be partitioned into a set of processes (P). Each process in P can simulate an entire simulation component (LI; LI; KAUFMANN, 2022). For example, a component can be the physics of a hurricane or a gate from a circuit. During the simulation, the GVT is used to synchronize these components' events (JEFFERSON, 1985). It is used to measure progress and define barriers to synchronization. The GVT is defined as the minimum value among all local virtual times (LVTs) and timestamps of all events in transit (JEFFERSON, 1985).

Figure 3.2 presents, in an event diagram, a snapshot of three different processes, P_0, P_1, and P_2. An arrow represents a message exchange between two processes, and its endpoints consist of send and receive events. The Global Virtual

Time (GVT) is defined by checking the timestamps of the last events processed by each individual process. GVT is calculated as the minimum of these timestamps, representing the earliest point in time among all processes where we are secure that all events have been processed. In this example, $GVT = LVT(P_2) = 70$, the timestamp of event e_{22}. Ideally, we want to have fast GVT computations so that the simulation will be subject to less wait time to order events.

The frequency at which GVT is computed depends on the synchronization algorithm implemented by the distributed simulator: it can be synchronous or asynchronous (JUNIOR et al., 2020). In a synchronous algorithm, events are performed in a lock-step manner (BREITNER; SMITH, 2017). Each event is processed only when its local virtual time meets the GVT, and all the distributed components simulate the same virtual time. In an asynchronous algorithm, the processes can simulate the component events without a barrier in a lock-free manner (EKER et al., 2019), leading to increased performance. However, by allowing this freedom, events can be processed out-of-order, violating causal consistency and possibly leading to inaccurate simulation results. Thus, to address this problem, simulators rely on checkpoint-rollback mechanisms to save consistent states of a process and restore the state when a causal violation occurs, ensuring the simulation remains accurate. In this case, GVT can be used as a barrier to define the moment a process should take a checkpoint of its state. By ensuring checkpoints are created using GVT as a barrier, there is also a guarantee that a single rollback will lead to a consistent state. Thus, in asynchronous scenarios, computing a GVT faster can lead to a faster reaction of mechanisms that create a checkpoint, avoiding unnecessary rollbacks to earlier points in time.

3.2.2 Why in-network computing?

We aim to reduce server and network bottlenecks and increase the performance of GVT synchronization by leveraging programmable data planes and in-network computing to offload the virtual time synchronization to programmable switches. Our solution is an alternative to server-based synchronization protocols. Offloading the GVT computation to switches is beneficial for several reasons (TOKUSASHI et al., 2019). Firstly, performing the computation as early as possible decreases network traffic, reduces congestion, and cuts the number of network hops

necessary to complete a GVT computation. Secondly, because the GVT computation computes the minimum value of a set of events, simple arithmetic operations that run at a line rate with modern switch hardware can be easily implemented. Finally, since the computation on switches occurs at line rate, the switch can implement the GVT comparison for an increasingly high number of processes with no significant impact on processing delay. Thus, we propose a system that can speed up distributed simulations running in data centers or clouds, which is common in the distributed simulation field (FUJIMOTO et al., 2010).

3.3 NetGVT system design

This section presents NetGVT[1], a system for performing GVT computation using programmable data planes. The system can intercept event messages and encapsulate logical clocks in a custom protocol header, which switches use to compute the logical clock barriers.

3.3.1 Challenges

As we already discussed in Chapter 2, performing computation in programmable switches has many advantages, including line rate processing and the reduction in propagation delay. However, the switch programming constraints impose challenges for calculating virtual time synchronization.

Limited set of ALU operations. There are restrictions on which operations are allowed and how these operations are performed. For example, the number of ALUs at each pipeline stage is fixed. This creates a challenge if the program's layout requires more ALU operations than those available in the pipeline stage (HOGAN et al., 2022). Additionally, P4 does not support loops, making it challenging to implement the GVT computation considering the constraints of the switch programming model. To overcome this challenge, NetGVT unrolls the GVT computation using resubmission. However, since using too many resubmissions can negatively impact throughput, we judiciously use it by *decomposing* multiple logi-

[1]Contents of this chapter have been published in Parizotto, Ricardo, et al. "NetGVT: offloading global virtual time computation to programmable switches." Proceedings of the Symposium on SDN Research. 2022.

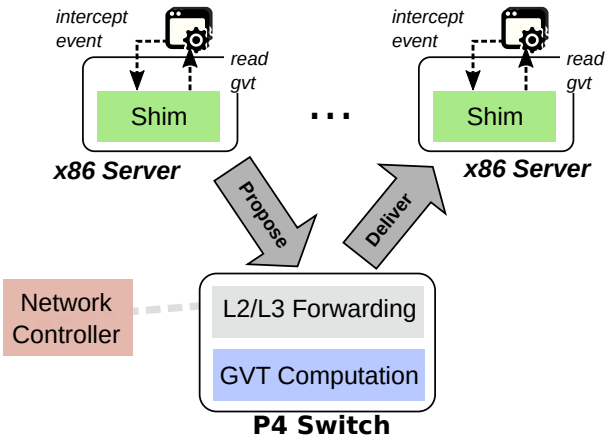

Figure 3.3 – NetGVT architecture overview.

cal clocks into hierarchical chunks and only traversing the essential chunks for the required computation.

Limited amount of memory. Within a processing pipeline, we use *stateful registers* to store logical clocks inside the switch. However, stateful registers consume SRAM space, which is limited. Such constraints limit the virtual clocks we can keep in the switch. To overcome this challenge, we propose a *compression* mechanism for virtual clocks that only needs to store an absolute difference between the virtual clock and an integer scalar. We periodically update the scalar to keep the memory demand low, ensuring that the fundamental difference between process clocks and the scalar can fit in short bit vectors.

3.3.2 Overview

The architecture of NetGVT, presented in Figure 3.3, consists of multiple servers connected to a programmable switch and an SDN controller.

Programmable switch. The switch is the main component of NetGVT. Simulation processes exchange event messages, and the switch is responsible for intercepting packets and storing a *compressed* version of their local virtual time to compute a new GVT value. First, the switch intercepts and processes a new protocol header that encapsulates virtual clocks. Next, the comparisons required to determine the new GVT are unrolled across multiple pipeline stages, subject to ALUs constraints. Finally, the switch sends the resulting computed value to all servers participating in the simulation using multicast primitives.

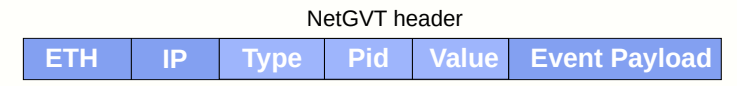

Figure 3.4 – NetGVT protocol packet format.

Shim layer. Existing distributed simulations often hardcode the synchronization protocol and how simulation nodes update their GVT. These traditional protocols do not consider any computation in the network and often operate over TCP, which is not suitable for INC. Instead of relying on TCP, NetGVT work over UDP and provides a shim layer that resides in simulation nodes. The shim layers intercept event messages and encapsulate the local clocks within a custom protocol header to offload the GVT computation to switches.

SDN controller. Besides deploying simple forwarding rules, the NetGVT SDN controller is responsible for updating the switch when the set of servers in a cluster is modified (addition/removal) or when there are changes in the set of processes participating in a distributed simulation. When a simulation starts, the logical clock of each process is reset to zero. Finally, the controller is also responsible for configuring multicast rules to deliver GVT values to all the servers.

3.3.3 Handling event messages

Distributed simulation processes execute events according to a global virtual time. Consequently, processes must have ways to read the current GVT before running an event. Additionally, after processing events, a process will have a different local virtual time, possibly leading to a new GVT value. Consequently, it is necessary to provide ways for the simulation processes to encapsulate these values so that the NetGVT switch can intercept and process them.

Intercepting packets. The shim layer intercepts event messages from distributed simulation processes. After intercepting a message, the shim extracts the virtual time of the process and encapsulates this information using a custom packet header as shown in Figure 3.4. The custom header added by the NetGVT shim layer is composed of the following attributes:

- `Type`: denotes an operation, whether the packet is *proposing*, *delivering* a new value, or *starting* a new execution.

- `Process ID (Pid)`: is the unique identifier of a process and specifies the *process* that created a message.
- `Value`: stores a value to be exchanged between servers and the network device. It can be a GVT or an LVT, depending on the context of the operation.

These header fields are modified by the switch and by shim layer instances during event message transmission. Finally, the shim layer intercepts packets arriving from switches to maintain a common timing reference among the distributed processes and stores the new GVT.

Packet losses. Although packet losses are uncommon in a cluster environment, NetGVT employs a mechanism for dealing with this loss. Because switches can not create new packets, we delegate most tasks for dealing with packet losses to the shim layers running on servers. The shim uses timeouts and re-transmissions to detect and recover from packet losses. The shim associates a timer to each proposal. If the timer triggers a timeout, the system assumes a packet was lost and retransmits it to the switch. Duplicate messages are identified by their LVT, i.e., if the received LVT from a process p is smaller than the one stored in the switch, we consider it a duplicate and acknowledge the receivement of the most recent packet.

3.3.4 Data plane layout

Figure 3.5 presents the layout of the NetGVT switch data plane. Upon receiving a packet, the switch checks whether it is an event or a standard forwarding packet. Standard forwarding packets are forwarded normally to an output port, thus enabling our system to be incrementally deployable. This also allows us to deploy NetGVT into existing distributed simulators without requiring us to change how event message exchange occurs.

3.3.4.1 Updating virtual times

In the case of event packets, the switch updates the respective LVT in a `LVTList` according to the process identification. The `LVTList` could be implemented either using *match+action* tables or as an array of on-chip registers. The former would require interaction with the control plane to update the list's contents, imposing the overhead of a control loop as part of the GVT computation. Because

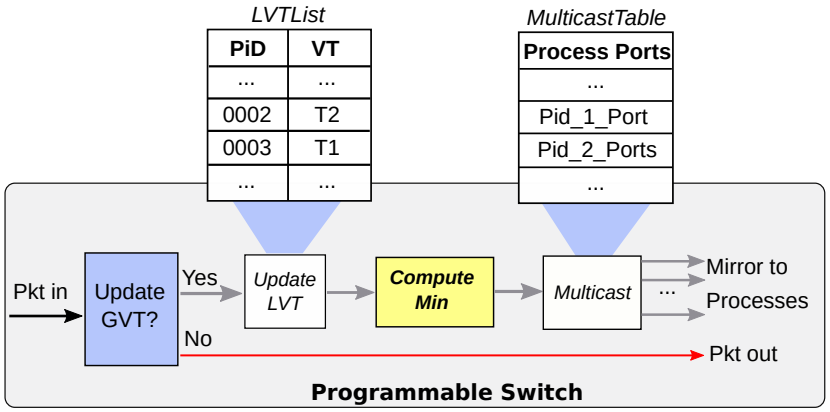

Figure 3.5 – The layout of NetGVT switch data plane.

of these drawbacks, which could impact the overall performance of the GVT computation, we opt for the usage of on-chip registers. By using on-chip registers, we can easily update the virtual time of each process without any interaction with the control plane, avoiding the need for a control loop. This also enables these updates to be processed at line rate, providing further improvements in terms of speed.

The `LVTList` holds a local virtual time for each simulation process and is indexed by a unique identifier. To read the identifier of each specific packet, NetGVT employs a *match+action*, that maps the PID as obtained from the header to an internal index of the data plane program. Once we obtain the index using this *match+action* mapping, the index is written in *metadata*, which is further used to index the proper LVT value from the list. Because *metadata* is a per-packet variable, this ensures the operation is atomic. Atomicity guarantees that the correct index is consistently managed for each packet without the possibility of concurrency problems.

Saving memory. However, as mentioned in Section §2.2.2.1, the switch has a small amount of memory, and registers consume precious SRAM memory. Consequently, naively allocating registers for the LVTList can consume a critical memory percentage. To address this limitation, we propose a compression mechanism that stores only the absolute difference between the LVT and a *scalar* to keep SRAM memory usage low. For example, in a scenario where the simulation process time is $t = 100000$, and the scalar is $s = 99070$, NetGVT only needs to store the difference $d = 930$, which can be represented with a bit vector of size 12, instead of regular 32-bit integers. By periodically updating the scalar to a value close to the processes LVT (e.g., the last GVT value), we ensure that the absolute difference is a small

integer that can fit in a tiny bit vector.

Although using a scalar can save precious memory resources, naively updating the scalar can face computational limitations. A naive approach would be to update all the LVTs in the switch once a process exceeds the range of a scalar. Although this can ensure all the LVTs are consistent according to the scalar, updating all the LVTs would require reading and writing all the LVTs in the list. However, this can easily exceed the number of operations available in a single pipeline traversal, as discussed in Section §2.2.2.1. We take an alternative approach to avoid the overhead of traversing all the elements.

Instead of updating all LVTs during a scalar update, the NetGVT switch detects and changes *on demand* only the LVTs that exceed the scalar range. Specifically, the system defines a *shadow scalar*, based on the current GVT, corresponding to the scalar that will succeed the current one. The switch then identifies any LVT that needs a new scalar and uses the shadow one[2]. The remaining LVTs still keep using the current scalar. Over time, all LVTs will exceed the current scalar range and converge to the shadow one, becoming the current scalar. Supporting the notion of a shadow in addition to the existing scalar avoids updating all LVTs at once when the scalar needs to change, at the cost of only two extra registers.

3.3.4.2 Computing the global virtual time

After updating the LVT of the process, the switch will start computing the GVT. Computing the GVT requires iterating through other virtual times and calculating the minimum value among all the LVTs. However, as discussed in Section 2.2.2.1, iterating through many values and computing a minimum value is subject to several hardware constraints.

Design Alternatives. Given that the P4 language does not support loops, we unroll GVT calculation as a set of *if-else* statements that iterate through the list of local virtual times. However, having a large number of processes leads to a large number of local virtual times. Consequently, the chain of *if-else* statements may not fit in the pipeline because it would require more ALU operations than what is available in a pipeline. To address this challenge, our design partitions the *if-else*

[2]Since these LVTs are on a different scale, we invalidate them by setting up a bit in a bitmask. This does not affect the GVT computation accuracy because it only occurs in processes that are not the new GVT.

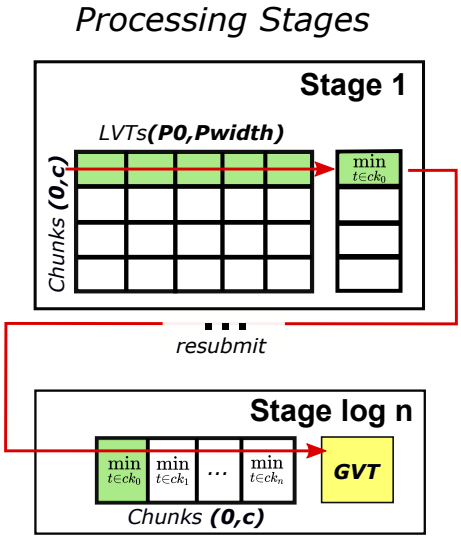

Figure 3.6 – Performing GVT computation on a programmable ASIC.

chain into different *chunks*, where each chunk compares a portion of virtual times. Assuming that n virtual times are partitioned into chunks that can fit in the pipeline *width* size, a naive approach to computing the GVT is to use resubmissions to iterate through all the chunks. This would enable a correct computation by iterating through the list of virtual times, even if the list is extensive. However, this approach still requires $r = n/width = |chunks| = \mathcal{O}(chunks)$ resubmissions, imposing additional overhead for packet processing. Minimizing the number of resubmissions is essential to optimize throughput from the switch and reduce the overhead of packet processing.

Our Approach. In NetGVT, we thus aim to provide an approach that judiciously uses resubmissions. Our insight into reducing the number of necessary resubmissions relies on the observation that a packet event only modifies the virtual time of a single chunk in each pipeline pass. However, the packet must observe the minimum value from each chunk to compute the global minimum value. Therefore, we propose to employ memorization techniques, where the minimum value of each chunk computed by processing previous events is stored in a cache. This cache acts as a repository of pre-compute local minimum values. When calculating the GVT, only the iteration of the cache values is utilized instead of iterating through all the virtual clocks. This approach minimizes redundant computations and reduces the overhead associated with resubmissions. Figure 3.6 illustrates how this design performs the GVT computation. In the approach presented in the figure, the system identifies the chunk that should be updated when a packet arrives. After updating

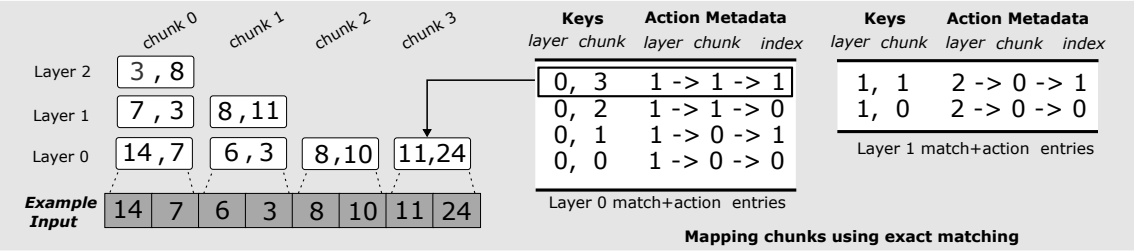

Figure 3.7 – An example of mapping of GVT computing

that specific chunk, the switch reiterates the packet. Next, the system compares only the minimum between the chunks to compute the GVT.

Hierarchical approach. However, reading and comparing the memorized values can also exceed the ALU operations available in a single pipeline. We devise another strategy that generalizes the memorization approach to also occur between the cached values. To achieve this, we employ a hierarchical approach for processing chunks. First, we create chunks of minimum values of lower-level chunks. This process continues recursively from the bottom up, creating multiple levels of chunks that fit in the pipeline. The creation of chunks occurs until the number of operations required to compute the local minimum is, at most, the amount of ALU operations available in the pipeline.

A mapping of a set of virtual clocks in our hierarchical approach is presented in Figure 3.7. The example shows a recursion tree with a depth of three in a switch capable of performing two comparisons at each pipeline. Each layer comprises a set of chunks, and only a single chunk is modified at each pipeline pass. Once a packet traverses the pipeline and is resubmitted to the beginning, the *match+action* tables use the layer and chunk as keys. The table will match an action that updates these variables according to the current layer and specific chunk. For example, consider an update performed at layer 0 chunk 3. The next iteration will match the first entries in the table, mapping it to layer 1, chunk 1, and index 1 in the chunk. This process repeats until a new minimum is found.

Complexity analysis. We can represent the computation of the minimum using the following recurrence relation:

$$T(N) = \begin{cases} T(N/width), & \text{if } n > width. \\ min(N), & \text{otherwise.} \end{cases} \tag{3.1}$$

As we can observe in Figure 3.8, performing these operations to the pipeline

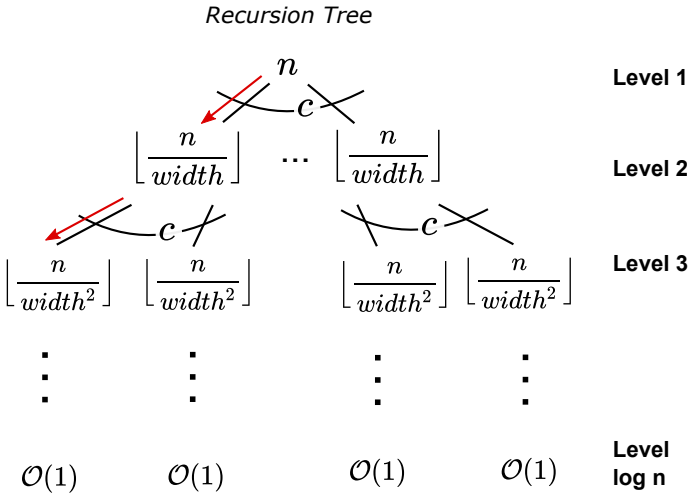

Figure 3.8 – The analysis of our algorithm

can be seen as a recursion tree, with the branching factor of *width*. Each node has at most *width* operations, which can run at a line rate. In contrast, the following levels have more *width* operations associated with each of the *width* chunks, and so on. The tree has height $\log_{width}(n)$, and at the leaves is our base case, which has as the default $\frac{n}{width}$ leaf chunks. Since sometimes $\frac{n}{width}$ can not fit in a single pipeline, we perform the hierarchical process by dividing the entry size by the pipeline *width* at each recursion level. In level i, we remove $\lfloor n/width^{i-1} \rfloor - 1$ chunks from the list of chunks the switch needs to visit using resubmission. Computing a new GVT value will require starting from a leaf node and resubmitting until reaching the root chunk. In the worst-case scenario, the amount of resubmissions is $\mathcal{O}(\log chunks)$ instead of $\mathcal{O}(n)$ from the naive approach. At that point, NetGVT can compute the current GVT value.

Delivering the GVT. After computing and saving the new GVT value, the switch inserts this value in the packet header and sends the packet to match the `MulticastTable`. `MulticastTable` will create a copy of the packet with the updated GVT for each shim instance and change the output ports. Finally, all packets are forwarded to their destination.

3.4 Experimental results

In this section, we present the experimental results. Our experiments focus on answering two main questions: (i) How does the performance of NetGVT compare

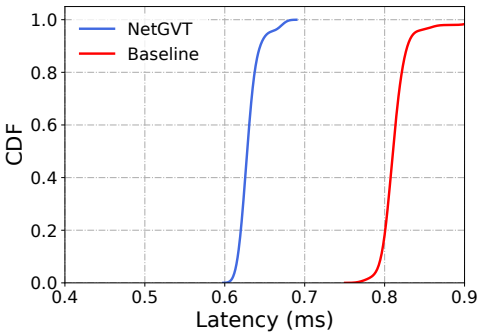

Figure 3.9 – Latency

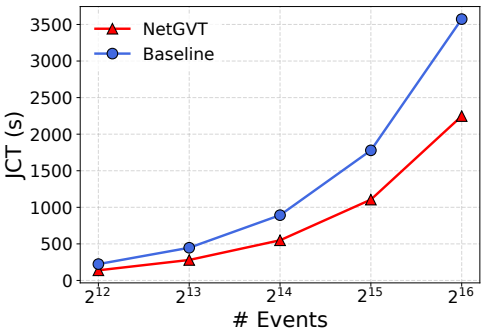

Figure 3.10 – JCT

to a server-only solution, and (ii) How does NetGVT scale with different workloads?

Experimental setup. We implemented the switch logic as a P4-16 (BOSSHART et al., 2014b) program based on the `TNA` model in approximately 600 lines of code. The controller is a Python program (∼40 lines of code). The P4 compiler generates an API that the controller uses to modify the multicast groups and addresses according to the mechanism described earlier. We developed the shim layer in Python using Scapy (∼120 lines of code). The source code is available in (NETGVT, 2022). The experiments were conducted in a testbed with two servers connected by a Wedge 100BF-32X 32-port programmable switch with a 3.2 Tbps Tofino ASIC. Each server is an Intel(R) Xeon(R) Silver 4210R CPU @ 2.4 GHz, with ten cores and 32 GB memory.

Workload. We run a microbenchmark that instantiates one process at each server of our testbed. We configured the processes to repeatedly send proposals that update the GVT value for the microbenchmark and used `tcpdump` to measure the latency. We compared our system to a server-only solution implemented with the same functionality as NetGVT. We configured the server-only solution in one of the servers, but instead of using the NetGVT switch, we used a simple L2 switch to forward event messages to the central server.

Computation time. We measured the latency to propose, compute, and deliver a new GVT value using both NetGVT and the server-only solution. Figure 3.9 presents the CDF for the latency with a sample of 1,000 GVT updates. This experiment demonstrates that our solution consistently outperforms the server solution. We observed that the latency using NetGVT is not higher than 0.65ms for 50% of the proposals. Conversely, the server solution takes about 0.85ms. This happens because NetGVT cuts a network hop, avoiding a round trip to a server solution.

Job completion time. To understand the impact of the GVT computation time in a distributed simulation, we implemented a lock-step (synchronous) simulation that executes different amounts of events. In this simulation, the processes keep a local virtual time (LVT) and only run the next event when the LVT is less or equal to the current GVT; otherwise, the process is locked. We measured the time to complete the simulation and presented it in Figure 3.10. We can see that for 4096 events, the server solution performs similarly to NetGVT. However, as the number of events increases, we observe that NetGVT can complete simulations considerably faster than the server-only solution. This happens because events that the GVT delays will start earlier when using NetGVT. For example, instead of a process waiting 0.8 ms for a new GVT value to begin processing a new event, it will wait for about 0.6 ms. This early start makes a considerable difference as the number of events processed in a simulation increases. As part of ongoing work, we are studying the integration of NetGVT with existing simulators, such as NS-3[3] and NEST[4].

Resource utilization. We evaluated resource consumption of NetGVT on top of a simple L3/L4 forwarding switch and configured NetGVT for an increasing number of processes. Because we observed that the majority of simulations NetGVT aims to deal with often require around 2-128 processes (EKER et al., 2019; WILLIAMS et al., 2021; MOHAMMAD et al., 2017; PELKEY; RILEY, 2011), we assumed this setup in our analysis. To handle 128 processes, we compare LVTs of up to 8 processes (8 is the pipeline width) at each pipeline stage and need up to 3 levels of chunks, incurring two resubmissions for updating the GVT. Unlike the previous experiments on the testbed, we used Intel P4 Insight to measure resource consumption as the number of processes increases (Figure 3.11). For 128 processes, the virtual time storage increases the consumption of SRAM and MAP RAM up to less than 10% of the memory available in the switch. ALU consumption is close to 40%, and the logical table ID to 20%.

Resubmission analysis. As mentioned earlier, NetGVT may use resubmissions to enable the GVT computation. We note that resubmission adds around $0.65 - 0.75$ ns to latency (WU et al., 2019). Figure 3.12 shows the number of resubmissions required by the naive approach (which does not use the hierarchical approach) compared to NetGVT. For 128 processes, NetGVT requires only two resubmissions; resubmitting adds only $2 \times \pm 0.7$ ns to the latency of the scenario

[3]https://www.nsnam.org/docs/models/html/distributed.html
[4]https://www.nest-simulator.org/

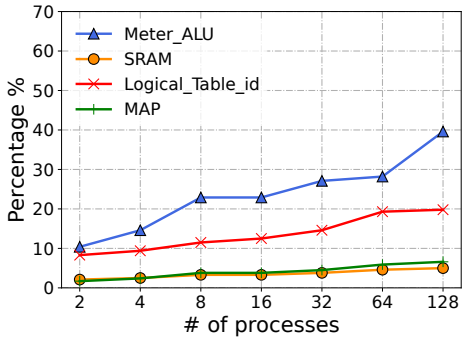

Figure 3.11 – % Resources

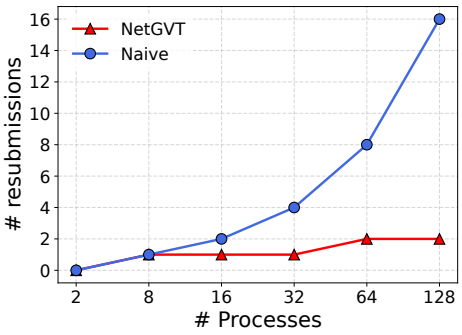

Figure 3.12 – # Resubmissions

using only two processes. Instead, the naive approach would need to traverse all 128 LVTs without optimizations, requiring up to 16 resubmissions (considering a pipeline width of 8). Thus, minimizing the number of resubmissions is necessary for the overall computation performance.

3.5 Related work

Synchronization protocols. Fujimoto et al. (FUJIMOTO; HYBINETTE, 1997) proposed a GVT protocol for shared-memory processors. Their protocol requires a round of communication for computing GVT since there is no need for message exchange between processors. Mattern (MATTERN, 1993) presented an algorithm for GVT computation for distributed simulations. This algorithm uses a variant of a distributed snapshot to compute the GVT value. However, none of these works considers network programmability. The most similar to our work is (NORONHA; ABU-GHAZALEH, 2002), which migrates the GVT computation to a programmable NIC. Although using the NIC avoids the overhead of the server software stack, this approach still requires an entire RTT to estimate the GVT. Differently, our work supports the GVT computation using programmable switches, thus avoiding the overhead and latency of an end-to-end communication.

In-network computing. Research efforts have proposed several solutions related to in-network computing, such as in-network concurrency control (JEPSEN et al., 2018; YU et al., 2020a; LI; MICHAEL; PORTS, 2017). Coordination services (DANG et al., 2020; ISTVÁN et al., 2016) offload the Paxos consensus protocol to the network hardware in order to minimize exchanges with servers. HovercRaft

(KOGIAS; BUGNION, 2020) uses programmable switches to collect quorum and accelerate communication. Others run vertical Paxos between switches to build a reliable storage (JIN et al., 2018a) or between servers to tolerate failures of network applications running on switches (KIM et al., 2021). However, the purpose of consensus is to make sure the same value is delivered to all participants without any calculation in the switch to decide which value should be agreed upon. Instead, NetGVT performs the GVT calculation before returning it to servers. Further, differently from these works, NetGVT captures the notion of causality which is required for GVT computation.

Our work is also aligned with recent efforts that leverage the benefits of in-network computing to accelerate the processing of scientific workloads for high-performance computing. Kim et al. (KIM et al., 2020) presented NSinC, an architecture that provides a closed control-loop for in-network acceleration of scientific workloads and simulations. However, NSinC does not focus on synchronization but instead enables telemetry over scientific data using programmable data planes.

A related area of research is moving physical clock synchronization to data plane devices. HUYGENS (GENG et al., 2018) improves the synchronization of datacenter servers by moving it to the NIC. Also, DTP (LEE et al., 2016) improves the synchronization by moving it to the physical network layer. Further, DPTP (KANNAN; JOSHI; CHAN, 2019) leverages high-resolution clocks available in programmable switching ASICs to respond to physical synchronization queries entirely in the data plane. Although keeping a reference to a real clock in the data plane improves accuracy, the clock skew of physical clocks could make it impossible to define precisely if an event happens before another. Instead, virtual clock synchronization ensures causal consistency, preserving the Lamport *happens-before* relation (LAMPORT, 1978). We propose ways to offload virtual time synchronization using logical clocks to programmable switches to speed up distributed simulations.

3.6 Discussions

In this chapter, we investigated how to deal with the constraints of the programmable data planes when offloading computation from a distributed system to the switch. We discussed techniques that we can use to customize functionalities to fit them in the data plane of forwarding devices. After studying these techniques,

we analyzed a practical use case of those techniques. Our use case is the offloading of the global virtual time synchronization for a distributed simulator to the network switches. More specifically, we proposed NetGVT, a system that offloads GVT synchronization into programmable switches. Since computing the GVT requires calculating the minimum value between a set of LVTs, which is not a primitive readily available in the P4 language or PDP hardware, we designed an efficient approach to compute the minimum. Our approach's design considers the memory constraints and the limited amount of ALU operations at each pipeline. We presented our design and an evaluation of our prototype and showed the scalability of our solution. Our results demonstrated that offloading the GVT computation to programmable switches is possible, promoting reduced time to complete a simulation compared to techniques that do not rely on network programmability. We also show that by employing our approach, we may be able to compute the global virtual time over a more significant number of clocks compared to a naive solution.

Although NetGVT has been primarily designed to compute the minimum value between a subset of logical clocks, being able to run experiments on real hardware allowed us to understand the different customizations and designs from the literature. This section briefly discusses several lessons we learned about the design considerations applied in the literature. Table 3.1 provides a comprehensive summary of considerations being employed in the literature according to the categorization presented in this Chapter. The table categorizes the functionalities (i.e., operations, services) offloaded to the network according to the respective design considerations. By employing these design considerations, the in-network implementation of the functionality employs several customizations compared to the server counterpart. It is clear that various functionalities, such as real-time computer vision, computing quantiles, and aggregation, are addressed with different algorithms and data structures, each tailored to the specific needs of the functionality, but can rely on the same design consideration.

Bellow we discuss the lessons learned in this chapter.

Lesson 1 - Commonality and Variation in Customizations: Although several techniques are available, different INC functionalities rely on a common subset of techniques from other data plane problems. For example, caching hot key-value items (JIN et al., 2017c) and detecting heavy hitters (SIVARAMAN et

Table 3.1 – Summary of techniques from the literature classified according to the customizations studied in this chapter

Technique	Funcionality	Customization	References
Division of Labor	Quantiles Sketch	Unsampled packets to compute *argmin*	(IVKIN et al., 2019)
Memoization	Sketch Optimization	Reuse hash values	(NAMKUNG et al., 2022)
Simplification & Approximations	Computer Vision	Limited size of filters & assumes high contrast	(GLEBKE et al., 2019b)
	HH Detection	Compute min in constant accesses	(SIVARAMAN et al., 2017)
	NN Inference	Converts NN into Binary NN	(SANVITO et al., 2018)
	Lock management	Identify hot-keys not present in the cache.	(YU et al., 2020a)
	Key-Value Cache	Identify hot-keys not present in the cache.	(JIN et al., 2017a)
	ML training	Quantize large floating point gradients	(SAPIO et al., 2021)
Reiterate	RTT monitoring	Recirculate entry for another chance	(SENGUPTA et al., 2022)
	IoT fingerprinting	Update state machine for signature checking	(KUZNIAR et al., 2022)
	Lock management	The same request can release/lock	(YU et al., 2020a)
	Computer vision	Loop to complete convolution	(GLEBKE et al., 2019b)
	Quantiles Sketch	Carry value to the first stage	(IVKIN et al., 2019)
	Graph Mining	Obtain next sub-graph	(HUSSEIN et al., 2023)
Decompositions	Aggregation	Decomposes gradients smaller ones	(SAPIO et al., 2021)
	Sketch Optimization	Decompose sketches into multiple stages	(SIVARAMAN et al., 2017)
	Load Balancing	Decompose resources from different devices	(TAJBAKHSH et al., 2022a)
	Graph Mining	Creates a copy of edges in multiple stages	(HUSSEIN et al., 2023)
Compression	Quantiles Sketch	Compacts a set of integers into half of its size	(IVKIN et al., 2019)
	Load balancing	Reduces bits to store resources	(TAJBAKHSH et al., 2022a)
	Aggregation	Sparsification & quantization	(SAPIO et al., 2021)
Distribute	Key-Value Cache	Keeps only hot-keys	(JIN et al., 2017a)
	Aggregation	Trains ML models on different devices	(LAO et al., 2021)
	Lock management	Keeps only hot locks	(YU et al., 2020a)

al., 2017) for traffic analysis requires the computation of packet statistics. In the case of caching, switches need to keep track of statistics about key-value pairs stored in the server. Based on these statistics, the system can identify the hot key-value items that need to be in the cache. Conversely, keeping a counter for flows is necessary for heavy hitter detection. Both of those applications accept a few errors, and thus, a common solution to these problems includes using probabilistic data structures (e.g., bloom filters and count-min sketches). *They trade the accuracy for*

the enhanced performance of programmable data planes.

Although we can reuse these data structures in multiple systems, deciding if the functionality accepts the inaccuracies is difficult. For example, load balancers can also use probabilistic data structures to keep per-connection status to maintain consistency (TAJBAKHSH et al., 2022a). The solution to these problems includes keeping a bloom filter or sketch that maps all packets from an existing connection to the same (consistent) destination. However, using these data structures may lead to conflicts because of hash collisions, resulting in *approximated* value. To avoid conflicts, sketches can be divided into multiple-level sketches held in the data plane (one sketch at each stage) and a conflicting key at each sketch, or rely on the control plane to solve conflicting entries. This enables updating only the sketch that stores the value corresponding to the key, making the system more robust. However, how could a framework guess the level of robustness a functionality will tolerate? This question is hard to answer because a desired functionality may require a tailored design that only makes sense at the application level. An example functionality is performing NN inference, in which the tailored design requires redesigning the entire NN into a simplified model. How do we infer such specific simplification without knowledge of the application? (PAN et al., 2023).

We can also apply different customizations to solve similar problems. For example, obtaining the *argmin()* of a set of integers for computing quantiles in the data plane is fundamental. Instead of doing several recirculations or approximating the minimum value, as done in NetGVT and for heavy hitter detection, the quantile approach from (IVKIN et al., 2019) uses unsampled packets to work the computation of the minimum. In NetGVT, we unroll the computing of the *min()* between multiple pipelines and employ memorization to avoid unnecessary resubmissions. *Given this scenario, it is difficult to choose the best alternative implementation for each application.*

Lesson 2 - The Role of Reiteration. The applicability of reiterations has a broad scope and occurs in at least two scenarios: reading data structures multiple times and unrolling large functionalities.

Firstly, reiterating is essential in scenarios where it is necessary to read or update the same register value or table more than once. A concrete example of this is seen in in-network lock management (YU et al., 2020b). When a lock is

released, the corresponding object becomes available for acquisition by another request waiting in the queue. The reiteration enables the packet to read the queue again by sending the packet to the beginning of the pipeline, enabling enqueued requests to acquire the lock. The second reason reiteration is applied is to allow functionalities to use more ALU operations than in a single pipeline. A concrete example is computing a convolution in a switch for computer vision (GLEBKE et al., 2019b). The convolution process includes creating a matrix composed of several filters. Even after simplifying the filters, computing the entire convolution is not feasible within a single pipeline pass because of the small amount of available operators. A concrete solution to this problem is iterating (or looping) over the pipeline using recirculation. This iterative approach ensures that the convolution computation can continue processing after a packet reaches the end of the pipeline, effectively addressing the limitation imposed by a single pass through the pipeline.

Lesson 3: Importance of Decompositions: By combining reiteration, memoization, and decomposition, NetGVT reduces the complexity of resubmissions from linear to logarithmic. Without decomposing the set of virtual times into multiple arrays (but still employing the hierarchical approach), the complexity of recirculations would fall to $\mathcal{O}(width \log chunks)$ because it would be necessary for each branch of the tree to reiterate for $width$ times. Thus, decomposition is a powerful design technique for NetGVT. Decomposition is also essential for other systems. For load-aware load balancing, for example, it is necessary to distribute load into different servers according to the available resources (TAJBAKHSH et al., 2022a). Thus, keeping this information on switches is essential for computing a load-balancing policy. Once resources from multiple servers have to be checked, an array is a natural way for programmers to store this information. However, because of the constraints, their resources must be decomposed and saved in different variables to be accessed in various stages of the same pipeline. Another alternative would be to reiterate, but this would cost in throughput, thus making decomposition an attractive solution to this problem.

Lesson 4: The Benefits of Compression. For similar reasons to NetGVT, resource-aware load balancers employ compression techniques. The reasoning is to release resources for other functionalities running in the same switch. In the case

of load balancing, compressing the resource releases switch resources to store state about a higher number of connections. In NetGVT, it is possible to keep a higher number of LVTs. Other functionalities also need to employ compression techniques. In-network aggregation would not be possible without compressing gradients (that are large floating points) (SAPIO et al., 2021). Compressing the gradients makes it possible to fit them in the switch memory and reduce the bandwidth between servers and the aggregator. The same occurs with quantile sketches, which compress a set of integers into a smaller set.

Functionalities that can not fit into the pipeline memory even after compressing may rely on a distributed design. Aggregating gradients for multiple jobs simultaneously requires a switch to keep gradients for the training throughout the entire aggregation process. Supporting multiple jobs needs a distributed design where multiple gradients are aggregated in collaboration between various switches, for example, by using a switch for each job and servers as a fallback, as adopted in ATP (LAO et al., 2021). Deciding how to distribute those tasks is a challenging task. Simply distributing the aggregation functionality by optimizing switch resources without inferring the frequency of traffic or priority of jobs could lead to deployments that do not maximize throughput and latency.

4 INVESTIGATING THE IMPACT OF INC FAILURES TO CONSISTENCY

In this chapter, we answer the second research question investigated in this thesis, regarding studying the impact of failures on INC. Section §4.1 presents a study of the effects of failures on existing INCs and the different consistency notions required by each system. Section §4.2 presents the overview of RESIST, the system we propose to mitigate the effects of failures. Section §4.3 presents how RESIST synchronizes state between multiple devices, which is a fundamental building block for providing fault tolerance. After discussing how to synchronize INC state, Section §4.4 presents how the system can provide different consistency notions for INC even in case of failures. Next, Section §4.5 presents the data plane building blocks and how we can use them to configure state-of-the-art INC systems. Section §4.6 presents implementation details of a RESIST proof-of-concept and the results obtained from experiments running both in a testbed and an emulator. Section §4.7 presents the related work. Finally, Section §4.8 presents our discussions.

4.1 Understanding the INC fault tolerance requirements

The idea behind INC is to perform computing tasks close to the data source, reducing the data transmission overhead and thus improving the overall system efficiency. However, this offloading also brings new challenges, and one of the significant issues is failures. INC failures can negatively affect the operation of a distributed system, disrupting and potentially creating an end-to-end system inconsistency. When computing tasks are executed in the data plane, switches act as the critical infrastructure that supports these tasks. Hence, failures can result in significant downtime, data loss, and a negative impact on system performance and consistency. In order to emphasize the potential impact of failures, we examine an in-network concurrency control system, an in-network event synchronization system, and an in-network aggregation system, which work as exemplary examples of the issue.

4.1.1 Concurrency control

In-network concurrency control offloads the concurrency control (CC) into switches, thus enabling the processing of transactions with reduced RTT compared to server-based approaches. For example, in Netlock (YU et al., 2020a), when a client wants to acquire a lock for an object, the system first tries to obtain the lock from the switch. The switch checks in a forwarding table (e.g., key-value data structure) whether the switch is responsible for locking that object. If the switch manages the lock for the object, the packet is processed by another table responsible for writing the lock into a persistent register array. However, if the switch is not responsible, the system will acquire the lock from the server. This process is illustrated in Figure 4.1.

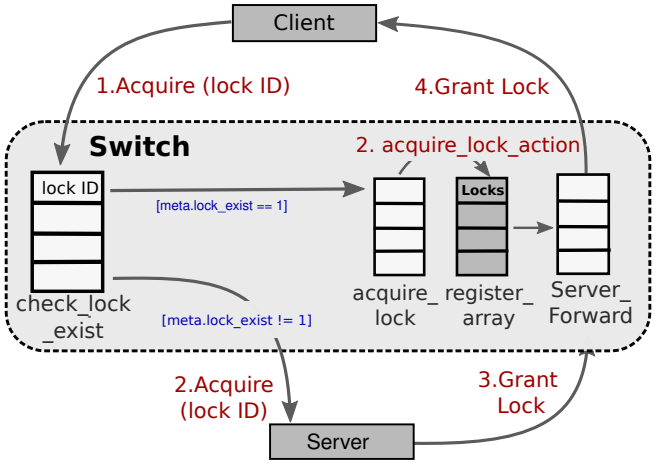

Figure 4.1 – Netlock Overview

Impact of failures: However, if the switch fails (fail-stop mode) and the switch state is erased, the locks managed by the switch will be lost, making the system inconsistent. For example, suppose an *exclusive lock* is acquired by application A and managed by the switch. In case of failure, another application, C, requires a lock for the same key in the server since the switch is unavailable. Naively restoring the switch state allows two applications to have an exclusive lock for the same key, enabling different clients to simultaneously change the same critical element.

4.1.2 Event synchronization

As we saw in the last chapter, NetGVT (PARIZOTTO et al., 2022) is a system that employs in-network computing to achieve efficient event synchronization

in distributed systems. It is designed to offload the computation of a global virtual time (GVT) into network switches, enabling them to synchronize events with a reduced round-trip time (RTT) compared to a traditional server-based solution. The system operates by intercepting event packets sent between processes running on servers and storing the local virtual time of the sender process in a register within the switch. The system then compares the virtual time stored in the registers and the minimum virtual time of all existing processes in the system. If the virtual time from the process is the new minimum, the switch performs a register action that writes the new minimum into a persistent register and multicasts the new value to all processes. On the other hand, if the virtual time from the process is not the new minimum, the event packet is forwarded normally to its destination. This process is illustrated in Figure 4.2. The system's design leverages the network devices' processing capabilities and reduces the overhead of transmitting data to a centralized server for processing. This results in improved performance for distributed systems that require event synchronization.

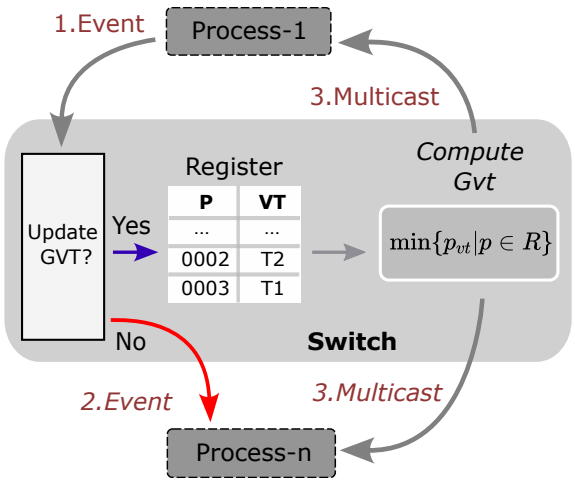

Figure 4.2 – NetGVT Overview

Impact of failures: When a failure occurs in the NetGVT system (fail-stop mode), it erases the switch's state. This has significant implications, as the failure results in the loss of the barriers managed by the switch, leading to inconsistencies. These inconsistencies can cause serious problems, such as starvation of processes and incorrect computation of the GVT. For instance, consider a scenario where one process waits for another, and the switch manages the GVT barrier between them. In the event of a switch failure, a naive recovery approach can cause one of the processes to starve, as the system can not coordinate the communication between

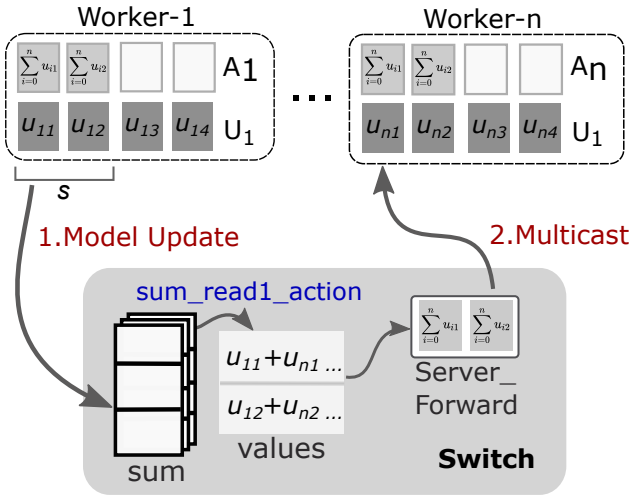

Figure 4.3 – DAIET Overview

the processes. Additionally, a naive recovery approach can lead to incorrect computation of the GVT, potentially resulting in causality violations in the distributed system running on the servers. Therefore, it is crucial to have robust and effective recovery mechanisms in place to handle switch failures in the NetGVT system. Such mechanisms should ensure that the system remains consistent and that the computation of the GVT is accurate, even in the presence of failures.

4.1.3 In-network aggregation

DAIET (SAPIO et al., 2017) is a system for map reduce that leverages the programmable switch for efficient aggregation. Map reduce models are mapped across multiple servers, called workers, who can perform operations in parallel, which are aggregated later in the reduce phase. By offloading the aggregation process to the switch hardware, DAIET can avoid the overhead of server-only solutions and bring the aggregation closer to the workers. The model is divided into s parts, and each part is sent to a switch for aggregation. The switch maps each packet to an action, aggregating and storing the values into a register pool (*values*). Once all the values have been aggregated, they are returned to the workers. This process is repeated for each of the s parts of the model in a synchronous manner. The overall process is illustrated in Figure 4.3. DAIET offers several advantages over traditional aggregation systems, including improved scalability, reduced latency, and higher efficiency due to the close proximity of the aggregation to the workers and

Required Consistency Model	INC Functionality	State
Strong Consistency	Concurrency Control (YU et al., 2020a) (JEPSEN et al., 2018; JEPSEN et al., 2021)	Lock/Queues
	L4 Load Balancing (BARBETTE et al., 2020; MIAO et al., 2017) (TAJBAKHSH et al., 2022b)	Connection Status
	Virtual Time Synchronization (PARIZOTTO et al., 2022)	Logical Clocks
Eventual Consistency	Key-Value Cache (JIN et al., 2017a; LIU et al., 2017)	Key-Value
	Aggregation (SAPIO et al., 2021) (LAO et al., 2021; HE et al., 2023)	Gradients/ Iteration

Table 4.1 – INCs by their consistency requirements

the line rate capabilities of switches.

Impact of failures. In the event of a failure during the training process in DAIET, the workers must recompute the model and send it back for aggregation in a different switch. This is because the failed switch would have lost all of its pre-computed aggregation data. As a result, the workers must train, at minimum, the last iteration again to obtain their gradients. However, unlike the event synchronization example described above, there is no need to recompute the lost gradients in the same order to ensure correctness in DAIET. This is because the aggregation operation is associative, meaning it can be executed in any order and yield the correct result. This associativity property allows for more relaxed notions of consistency for fault tolerance of in-network aggregation. By leveraging this property, DAIET can provide a robust and efficient solution for aggregation, even in the presence of switch failures.

4.1.4 Takeaway

In summary, these examples have two different requirements regarding consistency. On the one hand, global virtual time computation and concurrency control require a strong notion of consistency to ensure correctness. On the other hand, the in-network aggregation would require a weaker notion of consistency. Still, the failure recovery process can enhance the aggregation performance by avoiding restarting the aggregation after a failure.

Table 4.1 generalizes the discussion presented in this section to a list of INC systems. The table also identifies, for each system, the switch functionality (i.e., the

state) that must preserve the respective notion of consistency.

4.2 RESIST overview and workflow

RESIST provides fault tolerance for In-Network Computing (INC). The goal is to provide fault tolerance reducing the performance impact in non-failure scenarios. To achieve this goal, our insight is to decouple the replication strategy from the consistency assurance, ensuring consistency using a log-replay mechanism.

4.2.1 System model

INC assumptions. Servers are directly connected to a network of programmable switches. We assume that servers can temporarily store packets before transmitting them to the NIC output port. For systems with clients outside the network, we expect a proxy attached to our network to intercept and store packets. Besides forwarding network packets, a switch may also run an INC functionality. We do not make any assumptions about using cryptography to packet payloads or INC headers. We assume the same in-network functionality runs in different switches and that the functionality is deterministic.

Failure model. We assume several disruptions can occur in message transmission, such as lost, duplicated, or unordered messages. RESIST focuses on *switch* failures, and thus we assume *servers* can rely on traditional mechanisms for fault tolerance, for example, using well-established state machine replication mechanisms, such as Paxos (LAMPORT, 2019) or Raft (ONGARO; OUSTERHOUT, 2014). The system comprises $f + 1$ replica switches, where f replicas can fail. We assume failures can occur by crashing, but switches do not experience an arbitrary behavior (i.e., no byzantine cases). Finally, failures may occur simultaneously, both in the main switch and in its replicas, without blocking the processing, but we assume that at least one INC replica remains non-faulty.

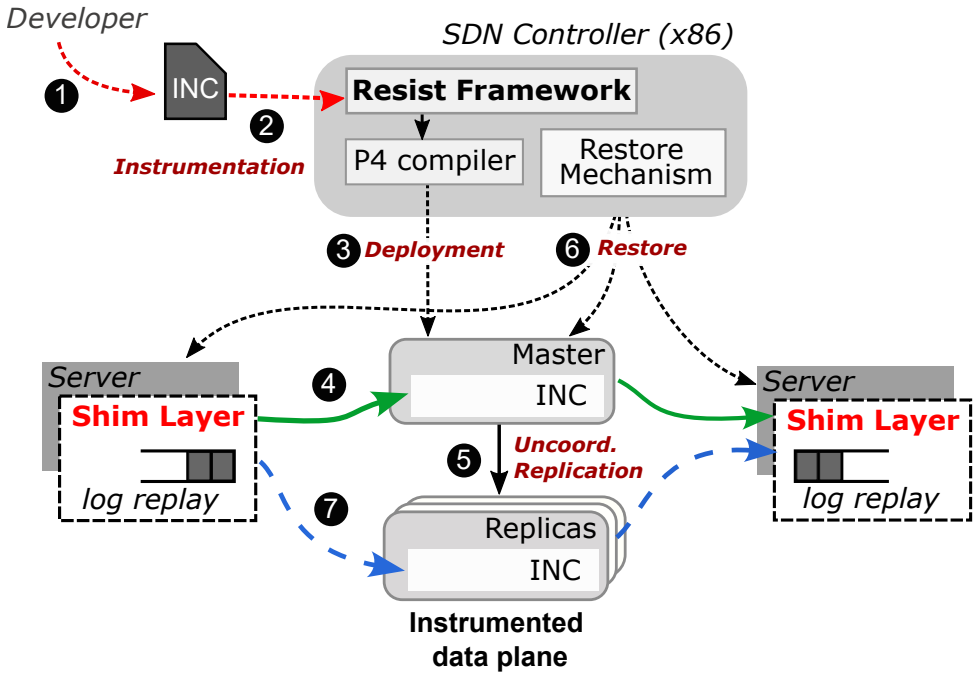

Figure 4.4 – RESIST workflow with the RESIST Framework (top), Shim layers (sides), and the INC replicas (middle)

4.2.2 Architecture and workflow

RESIST provides fault tolerance primitives to instrument INCs, deploying the instrumented source code on master and replica switches for asynchronous replication (§4.3). To ensure the fault tolerance mechanism is transparent to applications, RESIST implements a shim layer on servers (§4.4.3). This shim layer can replay packets in case of a failure, ensuring the application remains available and consistent (§4.4.4). To enhance the system's expressiveness further, developers can choose between different consistency requirements using a high-level control plane configuration interface (§4.5).

Figure 4.4 describes the workflow of RESIST. RESIST enables developers to instantiate fault tolerance building blocks to restore the INC to a consistent state after a failure (❶). The framework instruments the data plane with the essential replication building blocks (❷). After instrumenting the code, RESIST deploys the INC into master and replica switches and configures them for replication (❸). The master switch runs the main INC functionality and will process packets in non-failure scenarios (❹, green arrows). Every packet processed by the master will be processed in the same order by the replica INC, ensuring linearization (❺). In addition to replication, RESIST employs log-replay mechanisms to restore the INC

to a consistent state. We co-design our log replay mechanism between servers and switches: a shim layer on servers intercepts and logs the essential information for failure recovery. RESIST controller is accountable for identifying the failures and collecting the necessary information (❻). After the failure is detected, the controller gathers information from the shim layers, and triggers recover using a *model-based* replay mechanism that meets the desired consistency requirement. Finally, after the recovery, the INC will start processing packets again using the replica (❼, blue arrows).

4.3 INC state synchronization

We now discuss how to replicate the INC state between master and replica switches and how RESIST can ensure linearization during the replication.

Alternative solutions. Periodic snapshot collection is a widely used method for synchronizing replicas in distributed systems. However, this approach comes with its own set of challenges. For example, when considering INCs, there are two alternatives for collecting snapshots: exporting the switch state to the switch CPU or using in-band telemetry to export data. Exporting the state elements to the switch CPU can be costly, as the PCI-Express interface is slower than the data plane, leading to performance degradation (SONCHACK et al., 2018). On the other hand, the in-band telemetry approach would require additional resources in the switch to dump register information into packets, which can increase the system's complexity. To overcome these challenges, RESIST employs packet replication to send operations to a switch replica instead of collecting snapshots periodically.

Our Approach. Coordinating the packet replication synchronously could ensure linearization by guaranteeing that the main switch only processes a packet and forwards it to servers once all replicas acknowledge the receipt. However, this type of coordination would impose additional overhead for processing the INC. Instead, to avoid the performance overhead of a synchronous approach, RESIST uses an *asynchronous* design for fault tolerance. In this design, the primary INC process packets *speculatively*, similarly to (PARK; OUSTERHOUT, 2019). Packets are replicated from the master to the replicas, but the master does not need to wait for an acknowledgment from the replicas to process a packet. This approach eliminates the need to coordinate the replication, as the master does not need to be completely

synchronized with the replicas to continue processing. This design enables the system to operate more efficiently, at the cost of requiring a more complex recovery process in case of failures.

However, naively replicating could lead to packets being processed out of order or cause inconsistencies because of packet losses. Instead, to preserve linearization (HERLIHY; WING, 1990) and reliability, RESIST takes inspiration from Redplane (KIM et al., 2021). In particular, the master switch attaches a *round number* (a monotonically increasing sequence number) to each packet before replicating. Replicas identify out-of-order packets by comparing the packet round number with the round number from the previous packet. Whenever an out-of-order packet arrives at the replica, the system reorders it. Since buffering packets is prohibitive in the switch because of memory limitations, replicas send out-of-order packets back to the primary. The primary then forwards them to the replica again. This process repeats until all previous packets have been processed in order.

Besides ensuring linearization, to recover from packet losses, the master switch mirrors packets in RESIST header in the switch egress. The mirrored packet recirculates in the master until a replica sends an acknowledgment. If a packet exceeds a timeout in the master switch because it was lost, the mirrored headers are resent to the replica.

4.4 Maintaining consistency after INC failures

After identifying a failure, RESIST orchestrates a recovery procedure to restore the replica to a consistent state.

4.4.1 Consistency models

RESIST supports different consistency models by relying on packet replaying to ensure the replica INC is consistent with the state that the master INC had before failure:

- ***Strong Consistency (SC):*** The goal of strong consistency is to ensure multiple replicas act as a single sequential process. All replicas will have the same state at a specific time. Any updates applied to replicas will consistently

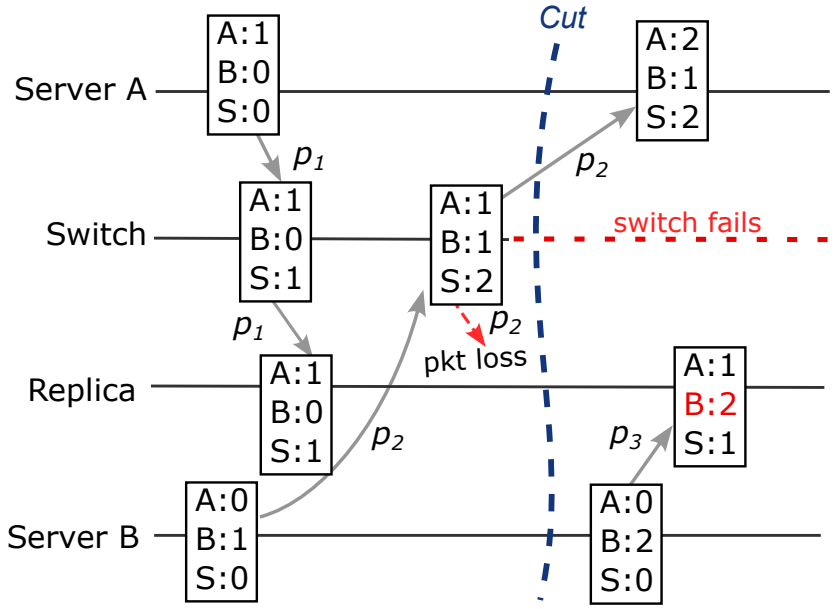

Figure 4.5 – Inconsistent cut created after failure because packet p_2 became an orphan packet

result in an identical state as if the updates were executed in a total order sequence on all replicas.

- ***Eventual Consistency (EC):*** In EC, there is no need to ensure ordering. All packets lost during failure must be replayed, but a replayed packet is not guaranteed to be delivered to the replica switch in any specific order.

4.4.2 The need to preserve dependency

Recovering from a failure requires preserving the consistency models RESIST provides and the dependency relation between events processed before the switch crashes. The dependency can, however, be violated due to asynchronous replication. Our key observation is that asynchronous replication enables *a packet to show up in the server state without being seen by the replica, indicating a dependency violation*; we call this type of packet an *orphan* packet. If a failure occurs in the INC switch, and there are orphan packets, the state of replicas becomes inconsistent with servers.

Example. Figure 4.5 presents an example of an inconsistent state created after a switch failure. In this example, nodes A and B exchange messages intercepted and asynchronously replicated by the INC running in the switch. Node A sends

packet p_1 to the INC that replicates the packet. Concurrently, Node B sends packet p_2 to the INC, which is sent to Node A and to the replica. However, the main switch crashes, and packet p_2 never reaches the replica switch. After noticing the failure, the RESIST framework will resume using the replica switch. However, a naive recovery procedure would make the system inconsistent because there is an orphan packet – in the example, packet p_2 is an orphan because it is reflected in the state of Node A but was never reflected as a send event by the replica state. This can be observed because Node A has information that the switch is in state $S = 2$, but the replica has state $S = 1$, which is inconsistent.

To detect dependency violations, it is necessary to provide ways to define and correlate the current replica state with the information already reflected on servers.

4.4.3 How to keep essential information

For detecting dependency violations and later having the necessary information to correct that, our insight is to augment application requests with an additional header that includes a monotonically increasing *logical clock*. By logging packets and correlating the corresponding logical clocks that the packets observe while being processed by the INC, RESIST identifies dependency violations. However, logging this information at the switches would be prohibitive because of their limited storage space (around ∼10MB). We take a different approach in which we rely on servers that interact with the INC to maintain logging information in a distributed and collaborative manner. The logs will help identify and fix dependency violations using a (consistency) model-based *packet replaying* in case the primary switch fails. This process is transparent to the applications because we assume the existence of a *shim layer* at the clients and servers that can help interface this communication.

Logging at shim layers. Figure 4.6 provides an overview of shim layers. Each shim layer maintains its monotonically increasing logical clock and two logs, the `input` and `output` logs. A logical clock is supported independently by each shim layer, helping to identify and repair any violation of dependencies in the event of a switch failure. The `output log` stores information about packets sent from the application to the network; the `input log` stores information about packets received from the network.

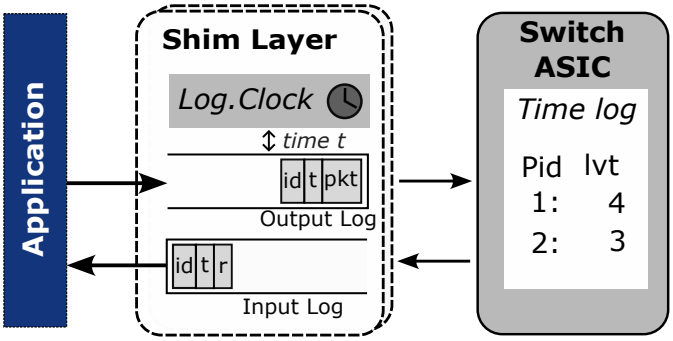

Figure 4.6 – Shim Layers in RESIST

- ***`Output Log:`*** When intercepting a packet from the application, the shim layer encapsulates the packet with a custom protocol header and keeps a copy in the output log (along with additional metadata) before forwarding it to the network. The custom header encapsulation adds the logical clock value to the packet and a unique identifier for the shim layer (e.g., a shim layer ID created from its IP and port) before forwarding the packet to the output interface.

- ***`Input Log:`*** When receiving a packet, the shim layer stores the RESIST custom header information, which includes the shim layer ID, the logical clock, and a round number (added by the switch) in its `Input log`. After we log this information in the shim layer, we take out the custom header and deliver the packet to the application.

The shim layer logs will be crucial in identifying which packets should be replayed after a failure; however, it could also harm the system's memory consumption. To address this, the RESIST framework *periodically* garbage collects the shim layer logs by removing information from packets already reflected at the replica switches.

One key observation is that once a replica switch has processed a packet, the corresponding packet logs are no longer needed for system restoration and can thus be removed from the shim layers. The RESIST framework periodically garbage collects the shim layer logs by collecting determinants[1] from the replica switch. Due to hardware constraints, the switch only logs the last round number seen by the replica. We utilize this information to determine which packets from the server can

[1]Determinants (SHERRY et al., 2015) is the kind of information required to restore a system to a consistent state. In RESIST, determinants include the round number from the replica switch and logs kept in the shim layer of the communicating nodes.

be removed. Specifically, the system clears all packets in the shim layer with a timestamp less than the current round number collected from the switch. This mechanism ensures the fault tolerance process can be performed with minimal memory usage on the shim layers.

4.4.4 Replay-based recovery process

RESIST can recover from failures by reconstructing the switch state using a replay mechanism. This process has three different steps. The first step of the recovery process involves collecting the *determinants* stored in the shim layers and in the replica switch. The next step involves identifying the subset of packets that must be replayed based on the collected determinants. Finally, the third step sends the necessary operations to the new master switch, ensuring that the necessary coordination is employed to preserve the consistency model. These steps are explained next.

Algorithm 1: Determinants collection and pre-processing

```
   Data: N: the number of RESIST servers
   pool: the set of servers in the RESIST pool;
1: inlog[N], sw.rnd ← Collects determinants from servers and replicas
   /* aggregates the determinants                                    */
2: for each server ∈ pool do
3:     for each p ∈ inlog[server] do
4:         Log(p) ← Log(p) ∪ {server}
5:         Round(p) ← pkt.round
   /* send determinants to servers                                   */
6: Send⟨Log, Round, sw.rnd⟩
```

Step #1: Determinants collection and pre-processing. The determinants collection aims to gather the essential information to identify orphan packets. Initially, the RESIST framework collects determinants from the set of servers involved in the recovery procedure and stores these determinants in `inlog` and `sw.rnd` (Algorithm 1, line 1). Next, the system checks the `input logs` to identify packets received by the shim layers. For each packet `p`, we map in a data structure `Log(p)` the set of servers that have logged that packet (according to their `input log`). In addition, we keep in `Round(p)` the information about the round number of each packet `p` and the replicas (Algorithm 1, lines 2-5). Once all this information has

been collected and aggregated, the RESIST framework encodes the values from `Log`, `Round` and `sw.rnd` and uses them to trigger the replay process (Algorithm 1, line 6).

Step #2: Detecting orphan-packets. Next, RESIST uses the information collected in Step #1 to detect orphan packets. This process occurs for all the shim layers in the system. First, packets previously received are sorted in non-descending order, according to their round number (Algorithm 2, line 3). The system identifies orphans as packets that are logged by the `input log` of a server ($Log(pkt) \neq \emptyset$) but whose round number was not seen by the replica switch ($pkt.round > sw.round$) (Algorithm 2, lines 4). RESIST maintains those packets to replay their execution so a new master switch can reach a consistent state (Algorithm 2, line 5).

Algorithm 2: Identifying orphan packets for replaying

```
   Data: N: the number of RESIST servers
   outlog[N]: the output log for each server;
   pool: the set of servers in the RESIST pool;
   Log: the mapping of servers that saw each packet;
   sw.rnd: the last round number seen by the switch;
   Round: the information about packets round number;
   /* repeating for all the servers in the pool                    */
1: for each server ∈ pool do
2:     Sort the outlog[server] in nondescending order
       /* replay messages in order                                 */
3:     for pkt ∈ outlog[server] do
4:         if Log(pkt) ≠ ∅  and  pkt.rnd > sw.round then
5:             orphan ← orphan ∪ {pkt} ;          ▷ pkt is orphan
```

Step #3: Model-based replay. The replay process is the core mechanism for achieving the consistency of the replicas in RESIST. This process aims to eliminate the inconsistencies that may have arisen from the existence of orphan packets by identifying and replaying these packets according to a pre-defined configuration. RESIST replay is customizable, allowing operators to choose between three replay models: *Strong Replay*, *Eventual Replay*, and *Strong Eventual Replay*. Figure 4.7 presents an example of valid replica states after a failure according to each supported model. The primary INC state is presented at the top after processing events from three servers: x, y, and z. Each event updates the state of the switch and is processed in a specific order. Events not received by the replicas are

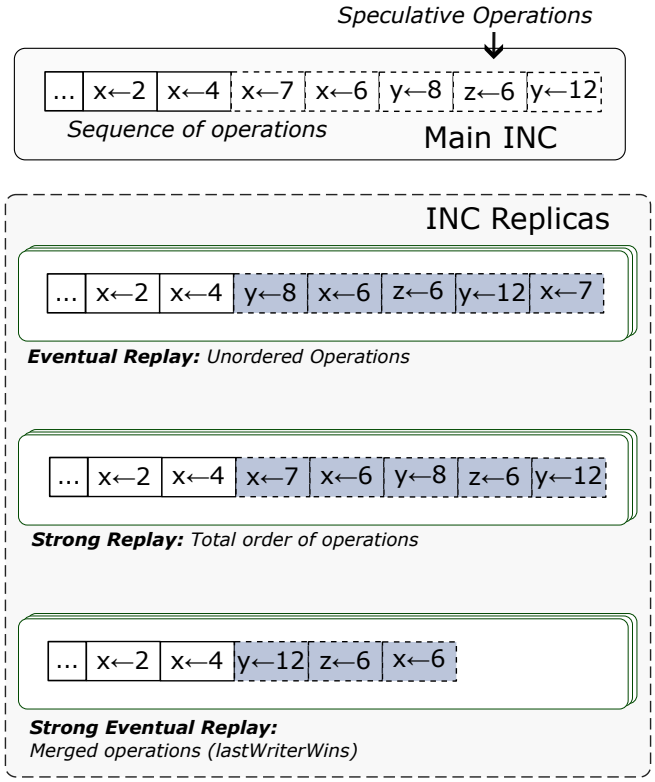

Figure 4.7 – Valid states for RESIST replay mechanisms

marked as operations that run speculatively. The figure presents operations in a replica after a failure according to each of these replay mechanisms.

- **Eventual Replay (ER):** The replay provides EC by allowing multiple shim-layers to send all the replay packets without worrying about the ordering. Switches will process packets replayed as soon as they arrive without re-ordering operations. The replay ends when all the packets are acknowledged. In the example in the figure, the replica processes the packets in an order that is different from the order that the main switch processed before the failure.

- **Strong Replay (SR):** Packets are replayed, ensuring a strict total order. Before starting the detection, the system blocks the processing of the application running on servers, buffering new requests and preventing the application from sending new packets to the switches. After blocking the application, the shim layers send a message notifying the switch at which round number the replay finishes. RESIST must guarantee that the order of packets processed by the replica corresponds to the order of the *round number* included in the packets. The replica does this by employing a reordering mechanism similar to the one used in the replication. For any packet p_i processed before p_j in the

main switch before the failure ($p_i \prec p_j$), we must ensure that the delivery of p_i happens before p_j in the replica switch. If a replica receives an unordered replay packet, the packet is forwarded back, and the process continues until it has been processed in the correct order. The figure shows that all replayed packets are processed in the replica in the same order as the master before the failure.

- **Strong Eventual Replay (SER):** Similarly to SR, it can also block the application processing. But before starting the replay, the shim layers can go through a customizable *merge* function that pre-processes packets locally within each shim layer, solving conflicts between them. The merge function outputs a subset of packets replayed with no ordering constraints compared to packets replayed by other nodes. One example of a merge function is the `Last-Writer-Wins` that selects only the most recent packet from each shim layer to be replayed. The figure presents the state of a replica that received merged packets using the `Last-Writer-Wins` but processed packets from different shim layers in any order.

Takeaway. While ER and SR can provide strong and eventual consistency, SER depends on the system model of applications. In particular, SER can be helpful in applications where shim layers have to replay multiple packets, and the operation performed by those packets in the INC is *commutative*. Commutative operations are those whose order of operations can change without affecting the result (SHAPIRO et al., 2011). Examples of commutative operations are updating monotonically increasing logical clocks from different processes (PARIZOTTO et al., 2022), summing gradients for in-network aggregation (SAPIO et al., 2021), or even updating different keys of a key-value storage (as pointed out in (PARK; OUSTERHOUT, 2019)).

Consider the example of computing barriers for logical clocks. Shim layers can employ a merge function locally to solve conflicts between the packets from the `output_logs`. The resulting packet can be processed by the replica in any order while still achieving strong convergence at the end of the replay.

4.5 Data plane and replay configuration

We now discuss how to instrument existing INCs to include RESIST fault tolerance capabilities and how to configure an application's consistency semantics. In summary, RESIST allows data plane developers to instantiate pre-existing building blocks that implement asynchronous replication in their INC code. Further, the network operator can configure the replay strategy according to the consistency requirement and define a merge function when necessary.

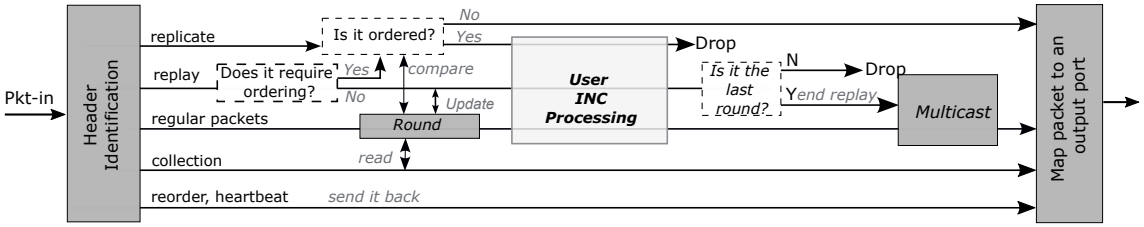

Figure 4.8 – The control flow of a switch with an instrumented fault tolerant INC

Switch building blocks. Figure 4.8 illustrates the control flow of an INC that has been instrumented with RESIST. Heartbeat packets are transmitted periodically between the RESIST shim layers running in the servers and the main switch, to which the switch responds with *pong* packets. Regular packets are received in the switch and will be replicated and forwarded to replicas using a multicast primitive. To ensure a switch replica process requests in a specific order, our building blocks ensure that once a replica receives an out-of-order replicated packet, it automatically forwards this packet back to the main switch as a reordered packet. These requests are retransmitted to the replica, which will process the requests in the desired order in the user INC. This mechanism assembles the approach from RedPlane (KIM et al., 2021), and it is necessary because buffering the packet in the data plane is unsuitable.

During the recovery phase, *collection* packets gather the replica round number and forward it to the shim layers. Further, *replay* packets ensure that the INC will be restored consistently. Before the INC processes a replayed packet, the switch maps the specific consistency requirement configured by the operator: packets can be strictly ordered by the round number, ensuring linearization or processed in the order they arrive. Finally, after the INC processes the last replayed packet, the switch is fully recovered and can process *regular* packets again.

Replay configuration. A control plane API enables developers to modify

Replay Mode	Blocks Application	Ordering	Merge Function
SR	✓	✓	✗
ER	✗	✗	✗
SER	✓	✗	✓

Table 4.2 – Operators used by each replay mode

the replay model of the replicated INC without needing to modify the underlying data plane code, thus reducing the complexity of expressing fault tolerance requirements. The configuration choice is entirely up to the developer, who can make trade-offs between performance and consistency based on the application's requirements. RESIST offers a `Resist_API(model, app)` control plane API call. The `model` parameter provides ways to change the replay model required. The `app` parameter is the name of the application. When a developer selects a replay model for an application, a configuration is updated at the shim layers and switches corresponding to the selected model. The configuration reflects the selected model and customizes the replay process to match the behavior of the chosen model.

Table 4.2 presents the configuration used by each replay model. The configuration can set shim layers to replay packets asynchronously, i.e., without blocking the application, or to operate synchronously. The switch is configured to guarantee the sequential order of the retransmitted messages, thereby achieving total order, or alternatively, it can permit replayed packets to be executed without any order. Finally, the merge function is optional and only used for SER. RESIST can support the three replay modes by combining the configurations mentioned above.

4.6 Evaluation

This section presents the experimental evaluation of RESIST. Our experiments focus on answering the following questions: (1) How does RESIST impact the overall INC performance in non-failure scenarios? (2) How does RESIST behave in failure scenarios, and what is the performance of the replay mechanisms? (3) How does the replay behave for different amounts of servers and workloads? (4) What is the hardware resource consumption of RESIST?

4.6.1 Experiments setup

Proof-of-Concept implementation. In our PoC prototype, we developed the shim layer as a multi-threaded Python program using Scapy[2]. The shim layer sniffs the network interfaces and filters the captured packets using eBPF filters. The RESIST switch was built using P4-16 code and is based on the TNA model utilizing the Tofino SDK. The switch operates a 32-bit register to store the switch's round number, which is updated at the beginning of the ingress pipeline for every INC operation. The framework uses multicast groups to facilitate asynchronous replication, implemented using the `mcast_grp_a` metadata.

Evaluation Setup. We evaluate our PoC of RESIST in a Tofino testbed. The experiments were conducted with two servers connected to two Wedge 100BF-32X 32-port programmable switches with a 3.2 Tbps Tofino ASIC. Each server is an Intel(R) Xeon(R) Silver 4210R CPU @ 2.4 GHz, with ten cores and 32 GB memory. Each server has a network interface card with two interfaces (one per switch). In addition, the server has a programmable NIC. The SmartNIC is a dual-port SFP28, PCIe Gen3.0/4.0 x8, BlueField(R) G-Series, with 16 cores and 16GB of onboard DDR4 RAM.

Applications. We evaluate RESIST by instrumenting the Tofino code of two INC systems, NetGVT (PARIZOTTO et al., 2022) and a modified version of Serene (NUNES et al., 2023). NetGVT synchronizes distributed simulation events using an in-network solution to compute the global virtual time (GVT). The modified version of Serene can perform in-network aggregation for distributed training in real hardware, as opposed to the original version that runs in BMv2. The system trains a CNN for image classification with one hidden layer using the MNIST dataset.

4.6.2 Performance benchmarks

Job completion time. To assess how RESIST impacts in non-failure scenarios, we performed a lock-step (synchronous) simulation that executes different amounts of events using NetGVT. In this experiment, the processes keep a local

[2]Optimizing the implementation of the shim layer using packet processing frameworks such as DPDK is planned as future work.

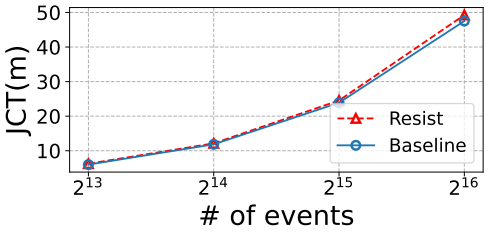

Figure 4.9 – JCT of simulations

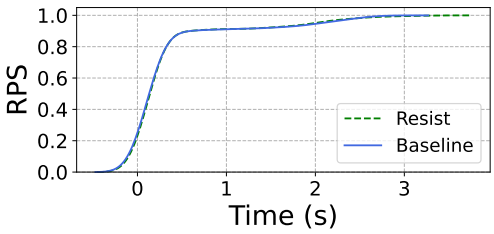

Figure 4.10 – CDF of iterations

virtual time (LVT) and only run the next event when their LVT is less or equal to the GVT. We measured the time to complete the simulation and presented it in Figure 4.9. For 8,192 events, the time to complete a simulation is not affected by RESIST replication mechanism, meaning our asynchronous approach does not add any delays for non-failure scenarios. As we increase the number of events in a simulation to 65,536, we see only a negligible impact on the JCT.

Iteration time. We train the CNN model in the in-network aggregation with and without RESIST and measure the time to complete each training iteration. Because the aggregation is an associative and commutative operation, we configured the asynchronous replication using the SER replay. Figure 4.10 presents the CDF of the training iteration time with and without RESIST. We observe that RESIST imposes around 0.02s overhead for each iteration. That occurs because the shim layer instrumentation intercepts packets for logging and appending new headers. However, this overhead does not significantly affect the job completion time (i.e., the system added only around 20 seconds to the 3h:21m of the entire training process).

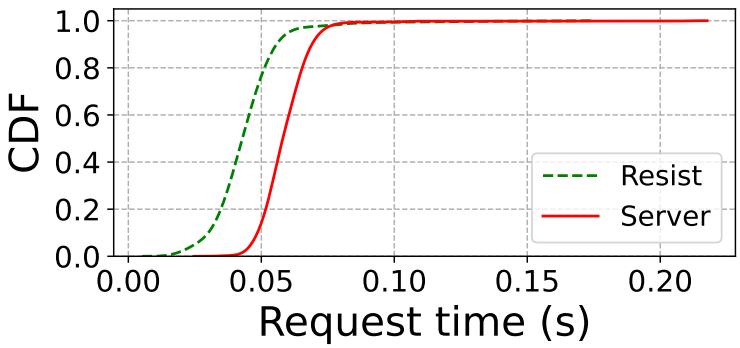

Figure 4.11 – SmartNIC vs Switch

Comparison with server solution. We implemented an alternative version of a synchronous replication mechanism that runs on a server (an approach

similar to the one in RedPlane (KIM et al., 2021)) to compare to our in-switch solution. This approach, as opposed to the asynchronous one used in RESIST, relies on coordination. In our setup, this alternative replication mechanism runs on a SmartNIC. We then measure the impact in the replication in cases where we use a server instead of a switch as a replica. We ran the experiment and recorded the time required to process each event in the SmartNIC-based fault tolerance. Figure 4.11 presents the CDF for the request processing time with RESIST switches and with the alternative server solution. We observe that RESIST completes requests earlier than using the replicas in a server, which may affect the JCT of large jobs.

4.6.3 Handling failures

To assess the availability of an INC system in the event of switch failures, we compare the performance of NetGVT with and without RESIST instrumented code for fault tolerance. We configured an experiment where clients send requests to the switch to perform virtual time computation, and we monitored the number of requests per second (RPS) being processed. We simulated a failure at the 10-second mark by dropping packets in the switch and manually restarting the switch after around 25 seconds.

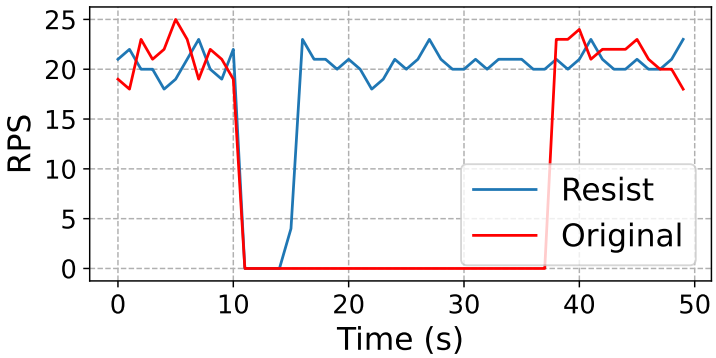

Figure 4.12 – RPS during failures

Figure 4.12 illustrates the RPS during the experiment, highlighting the differences between the two scenarios. We observed that in the scenario without fault tolerance mechanisms, requests triggered during the switch failure were delayed in the server queue for the entire period that the switch was down. However, after the restart, the switch resumed processing packets. In contrast, when using RE-

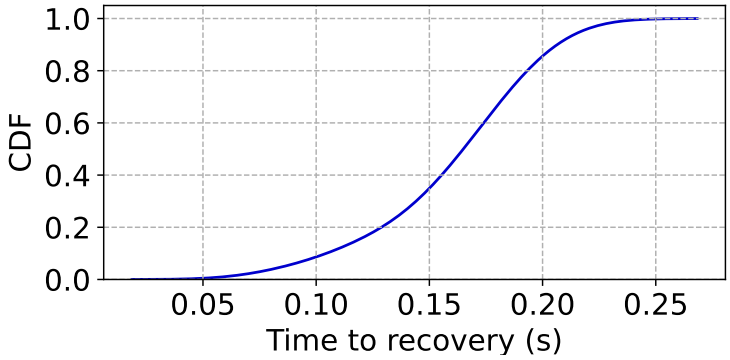

Figure 4.13 – Failover Time.

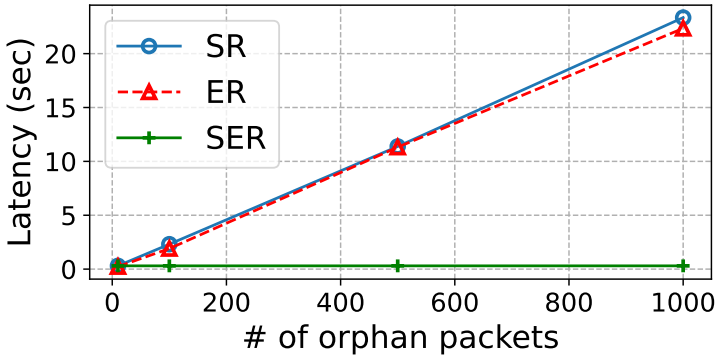

Figure 4.14 – Latency due to replays

SIST fault tolerance, we observed that the system could resume request processing earlier. Consequently, a lower amount of requests are impacted by the switch failure.

The results in Figure 4.13 show that the system requires, on average, 0.16 seconds to recover from a failure. The standard deviation is 0.03 sec. These results indicate that RESIST can rapidly recover from failures while maintaining a strong notion of consistency.

Replay overhead. To understand the replay overhead, we ran a microbenchmark configuring shim layers to trigger replays with a variable number of packets operating with different consistency models. We vary the number of replayed packets from 10 to 1K, where each server sends half of the packets (ranging from 5 to 500 each). To ensure that we assess the overhead of reordering with SR, we randomly distribute the rounds between the packets in each shim layer. We configure a merge function for the SER that selects only the last packet intercepted by the shim layer using the *LastWriteWin* policy.

Figure 4.14 presents the time to complete the replay according to different

replay modes. We observe that replaying 1k packets in SR and ER substantially increases the latency for replaying. SR imposes significant delay because the number of out-of-order packets in the new master increases. At the same time, the ER is also required to send and acknowledge all the packets, which leads to overhead. Out-of-order packets lead to reordering and increased latency as the number of packets needed to replay increases. Instead, the delay is around one second when using SER because only a single merged packet per process has to be replayed without requiring any specific ordering between different servers.

4.6.4 Functional evaluation

To better understand the scalability of RESIST, we run experiments in our emulated setup, varying the number of servers and using different configuration scenarios to define the recovery strategy:

- ***Scenario 1***: For the replication, the main INC periodically *exchanges snapshots* with its replicas in a 4-second interval, and uses *server replaying of lost packets* to achieve total order (strong consistency).
- ***Scenario 2***: For the replication, the main INC *sends all packets* to replicas, and uses *server replaying of lost packets* to achieve total order (strong consistency).
- ***Scenario 3***: For the replication, the main INC *sends all packets to replicas*, but relies on a *merge function and CRDTs* during recovery. The merge function solves conflicts locally at each server before the server retransmits, outputting only the last packet retransmitted before failure (strong eventual consistency).

During the experiment, we intentionally dropped packets from the main switch to the replica but delivered the original packet to the host destination. After a fixed interval of 4 seconds, we injected a crash in the main switch. This situation creates dependency violations that need to be corrected by the recovery procedure.

Recovery latency. Figure 4.15 presents how long the system takes to recover for each scenario. We observe that the recovery is slower as the number of servers increases. Achieving total order with eight servers requires about 7 seconds in Scenario 1 and approximately 4 seconds in Scenario 2. This latency increase is

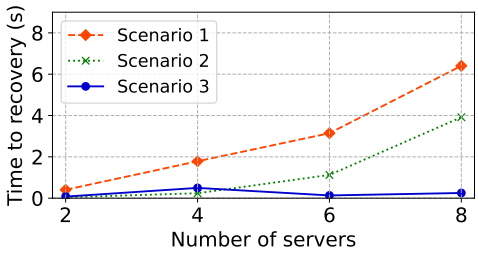

Figure 4.15 – Recovery on different amounts of servers.

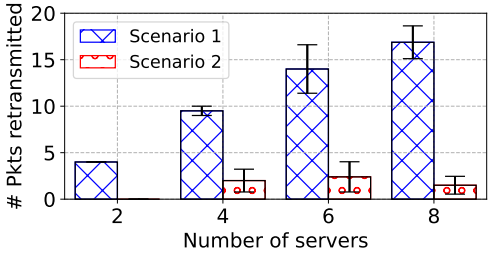

Figure 4.16 – Number of packets to replay on different amounts of servers.

attributed to the higher number of dependencies that need correction. However, the latency does not exhibit the same growth in Scenario 3, which employs a merge function to resolve conflicts and uses Conflict-Free Replicated Data Types (CRDTs). Conflicts are resolved by consistently selecting each server's highest NetGVT clock value, eliminating the need to retransmit all packets. Operations performed on resulting packets are commutative, allowing them to be processed in any order in the replicas. This significantly reduces the number of packets requiring retransmission and avoids the need for reordering, resulting in recovery times of less than 2 seconds for any number of servers in our evaluation.

Retransmissions and dependencies. Figure 4.16 presents the number of packet retransmissions due to dependency violations we observed per server in experiments with scenarios 1 and 2. We omit Scenario 3 since the merge function solves dependency violations in this recovery strategy. We observe that as the number of servers increases, there is a corresponding increase in the number of retransmissions for Scenario 1. This explains the higher overhead to recover. In contrast, Scenario 2 displays a lower number of retransmissions (3 packets on average) because the failure in the INC (and the subsequent loss of packets) has a lower impact than losing entire snapshots (Scenario 1). Although reducing the number of retransmissions can improve the time to recovery compared to Scenario 1, it still requires reordering packets from multiple servers.

4.6.5 Resource Consumption

Switch. The amount of switch resources that are consumed by RESIST is presented in Table 4.3.

RESIST is not consuming more than 10% of each computational resource

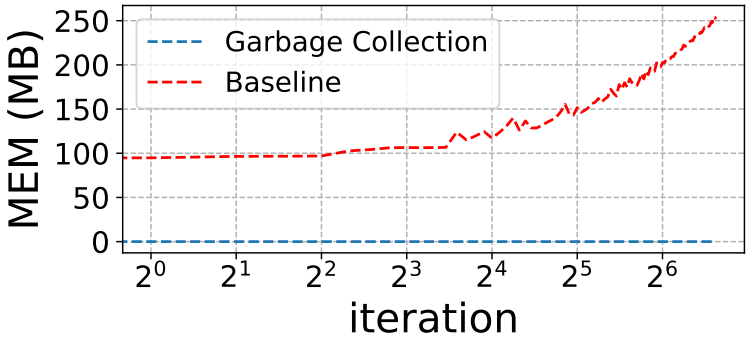

Figure 4.17 – Shim Layer memory

available. Specifically, a relatively small percentage of available arithmetic logic unit (ALU) operations, approximately 5.2%, are used for comparing round numbers and detecting out-of-order packets. Additionally, identifying the types of packets being sent for the different building blocks (see Section 4.5), including *replay*, *collection*, *linearization*, requires 9.4% of table IDs and 5.2% of Gateway resources. Furthermore, the memory consumption of the switch is negligibly low, with only 1.1% of static random-access memory (SRAM) being utilized for storing round numbers and flags that indicate the end of the recovery process. Finally, switch resources are spread across only four pipeline stages.

Table 4.3 – RESIST switch resource consumption

Computational resources			**Memory**	
Meter ALU	*Table IDs*	*Gateway*	*SRAM*	*TCAM*
5.2%	9.4%	5.2%	1.1%	0.0%

Shim layer memory. We conducted multiple iterations within the testbed setup while running in-network aggregation and recorded the memory consumption using RESIST shim layers. We configured the frequency for garbage collection to 5s. Figure 4.17 illustrates the amount of memory utilized with active garbage collection in RESIST compared to the baseline without it. Without garbage collection, the mechanism would be impractical due to the high memory consumption of storing packets with gradients. Nonetheless, by implementing packet garbage collection in the shim layers, the memory utilization in the output logs is negligible for almost the entire training.

4.7 Related work

As related work, we consider several research efforts with the goal of tolerating failures in programmable switches.

Fault tolerance for INC. FRANCIS (HAN et al., 2022) designs a system to recover from switch and link failures for source-routed multicast and clock synchronization. In contrast, LIBRA (PAN et al., 2022) provides switch fault tolerance for synchronous machine learning systems. NetChain (JIN et al., 2018b) stores and manipulates key-value items in switch memory and implements Vertical Paxos (LAMPORT; MALKHI; ZHOU, 2009) using Chain-Replication to tolerate switch failures, achieving strong consistency. Finally, HyperSFP (HUANG; WU, 2022) implements fault-tolerant service-function chaining in the programmable data plane. RESIST is different because it provides fault tolerance by decoupling consistency from replication and employing a log-replay approach.

RedPlane (KIM et al., 2021) and Swish (ZENO et al., 2022) are the most aligned works to RESIST. RedPlane provides fault tolerance for applications running on switches using replicated data storage on servers. The system provides both strong and eventual consistency. However, because the system uses coordination to achieve strong consistency, the system delays every packet. This overhead increases when a server-based storage is used to run the replicas. RESIST reduces these overheads by placing a replica in the data plane and performing replication at line rate. Furthermore, instead of coordinating the replication, RESIST decouples the consistency guarantees to be ensured using log-replay techniques. This decoupling allows RESIST to reduce the overheads of non-failure scenarios while providing efficient recovery and strong convergence guarantees.

Swish (ZENO et al., 2022) provides abstractions that can enable distributed network functions on programmable switches by replicating the state and operations between multiple devices. While Swish could offer consistency guarantees to the INCs we are working with, they rely on a mechanism to coordinate replicas and recover from packet losses using the control plane to log and send packets again in case of failures. This is different from RESIST. Once a packet arrives at the Swish switch, it is forwarded to the control plane, where it is logged and waits until the replicas acknowledge the previous packet. These techniques add overhead for non-failure scenarios because Swish coordination ensures a packet is only retrieved from

the log and forwarded to replicas once the replicas processed the previous packet.

INC for fault tolerance. Previously, we discussed the works that aim to provide fault tolerance for INC systems. Now we briefly present research that uses the data plane of switches to enhance traditional fault tolerance mechanisms. Recent work leverages programmable switches for optimizing fault tolerance for applications running on servers (SUN et al., 2023; CHOI et al., 2023; LI et al., 2016; PORTS et al., 2015; LIU et al., 2023; KIM; LEE, 2022). These solutions use switches to intercept and attach a sequence number on requests, ensuring different servers will process the same request in the same order. By ordering requests in the network, it is possible to provide total order without the overhead of coordination between servers. RESIST is orthogonal to these works as it protects the switch functionality from failures. Our system also orders requests in the network, but by doing so, it protects the switch functionality from failures. Thus, besides ordering, we need to handle the replication and recovery of switches.

Although these papers are orthogonal to ours, we highlight NeoBFT (SUN et al., 2023), which, beyond using the network to order requests, employs asynchronous replication using speculation in a way similar to ours. Once a server receives a request, it processes it in the correct order without waiting for a majority to complete. If a packet is lost, a server sends a request to another server that will send the lost packet. RESIST is different because it keeps a copy of the packet header in the switch, avoiding asking for another server.

Asynchronous replication and commutativity. Asynchronous replication is widely used to avoid coordination overhead while employing state-machine replication for fault tolerance (BIRMAN; JOSEPH, 1987). Recently, commutativity operations started gaining attention because of their strong convergence, especially in conflict-free replicated data types (CRDT) (SHAPIRO et al., 2011). CURP (PARK; OUSTERHOUT, 2019) is a protocol that explores the commutativity of operations for fault tolerance. Clients actively replicate requests asynchronously, executing requests in 1 RTT and relying on replay in case of failures. Replayed packets do not need to be ordered as long as their operations are commutative. In RESIST, SER explores commutativity to enhance the recovery of INC failures. In addition, RESIST also enables the customization of a merge function, reducing the amount of replayed packets and, consequently, the recovery overhead.

Table 4.4 summarizes the most representative related work, the consistency

Table 4.4 – Related work comparison

System	Consistency Model	Mechanism	Scope
Swish	Strong Consistency	Chain Replication	Read-intensive workloads (NAT, load balancer)
	Eventual Consistency	Periodic Synchronization	Arbitrary inconsistency (e.g., rate limiter)
	Strong consistency	Window Synchronization	Sketches (e.g., DDoS detection)
RedPlane	Strong Consistency	Synchronous replication	NFs (e.g, NAT)
	Bounded inconsistency	Periodic snapshot	Bloom Filter, Sketches
Resist	Strong consistency	Asynchronous replication Log (model-based) replay	Strict Ordering (e.g., event synchronization)
	Eventual Consistency	Asynchronous replication	Associative Operations (e.g, aggregation)

model employed, the kind of mechanisms, and the scope. We have chosen the most representative work by selecting ones that focus on INC systems (as opposed to NFs, for example) and propose different notions of consistency instead of a single notion specific to one application.

4.8 Discussions

This chapter investigated how to mitigate crashes in programmable data plane devices when offloading computation from a distributed system to switches. We study systems from the literature and show that these systems may apply different consistency models and mechanisms to preserve correctness.

- **Concurrency control:** Various approaches exist for concurrency control (e.g., timestamp ordering, optimistic CC, transaction commit), all being investigated in the context of data plane programmability. Regardless of the mechanism, it is necessary to operate in a strict notion of consistency.

- **Distributed training:** This process can perform aggregation synchronously or asynchronously, allowing workers to keep a distance from each other. The synchronous approach ensures all the workers are in the same iteration. However, because aggregation is commutative, more relaxed approaches can also converge. This means that a mechanism that provides strong consistency or

offers a weaker notion of consistency can satisfy the accuracy needs because gradients from different workers do not conflict, thus also fitting the definitions of SER. Asynchronous aggregation permits workers to be in different iterations while achieving similar convergence guarantees, therefore remaining correct in most cases. This is observed in Serene (NUNES et al., 2023), one of the systems investigated in this chapter.

- **Distributed simulation:** Similar to distributed training, distributed simulation relies on comparable principles for synchronization, employing either synchronous or asynchronous methods. However, like concurrency control, correctness is maintained only with a strong consistency model.

To answer our second research question, we proposed RESIST, a system that enables INC systems to be fault tolerant. The system employs asynchronous packet replication between data plane devices and allows replicas to operate at different consistency notions when a failure occurs. These multiple consistency notions can be employed in RESIST to preserve consistency from each application. In particular, we proposed mechanisms that can preserve the consistency of both in-network aggregation and also virtual time synchronization with a small overhead.

5 EXPRESSING FAULT TOLERANCE REQUIREMENTS FOR INC USING INTENTS

Until now, we have focused on understanding how to map INC functionality subject to the data plane constraints and studying the impact of failures and techniques that can keep INC fault tolerant with a small overhead. This chapter addresses the third and final research problem presented in Chapter 1, Section §1.2. In Section §5.1, we discuss the research problem, intent-based network concepts, and the challenges we will face in investigating the research problem. Section §5.2 presents the Araucaria system design that addresses the INC configurations challenges, including our specification and refinement methodology. Section §5.3 presents the results from experimental evaluations. Section §5.4 presents the related work, including literature that studies intent-based networks and research that tries to raise the abstraction level of data plane programming. Section §5.5 presents our discussions about the content from this chapter.

5.1 Understanding the complexity of INC fault tolerance

Offloading functionality to the network has several advantages, e.g., reducing latency and improving bandwidth by intercepting and processing network packets at the switch, thus avoiding the need to forward them to servers. However, these advantages come at the cost of data plane configuration complexity due to the necessity of writing low-level P4-based operations (e.g., table entries, action parameters, and register values). Configuring such functions in the data plane is tedious and requires substantial training. Moreover, data plane programmability must deal with failures in forwarding devices and their applications, adding an additional layer of complexity. For example, existing fault tolerant systems (KIM et al., 2021; ZENO et al., 2022) employing replication techniques on data plane devices introduce other complexities. In particular, they require configuring an INC and its replicas while offering consistency besides fault tolerance.

One way to mitigate the above complexities is to develop a DevOps-friendly automated policy- or intent-based data plane configuration. This approach could enable INC operators to express fault tolerance requirements at a higher abstraction

Table 5.1 – Intent frameworks in the literature.

Examples	Purpose	Target
Nile (JACOBS et al., 2018), InsPire (SCHEID et al., 2017), PGA (PRAKASH et al., 2015), Arkham (MACHADO et al., 2017)	QoS, Control Access	SDN, NFV
Janus (ABHASHKUMAR et al., 2017)	Bandwidth QoS, Temporal Policies	SDN, NFV
JingJing (TIAN et al., 2019a)	ACL rules	WAN
Gherkin (ESPOSITO et al., 2018)	Firewall, SFC	SDN
P4-iO (RIFTADI; KUIPERS, 2019), (LEWIS et al., 2018)	Routing, HH Detection	P4

level (PANG et al., 2020). Subsequently, the underlying software would automatically translate these specifications into detailed low-level data plane code and configurations. Although early research efforts in policy management facilitated some configuration capabilities, particularly in the domain of security and quality of service (QoS) (DAMIANOU et al., 2001), they can not enable the specification of policies for programmable data planes. Recent developments in *intent-based networking* (IBN) allow the deployment of high-level intents directly into more fine-grain network configurations, such as OpenFlow *match+action* rules, middleboxes (JACOBS et al., 2018; ESPOSITO et al., 2018; HEORHIADI et al., 2018; ALALMAEI et al., 2020), or P4 (ANGI et al., 2022; LEWIS et al., 2018; RIFTADI; KUIPERS, 2019) - an in-depth list is presented in Table 5.1. However, existing works in this domain do not support fault tolerance requirements and abstractions. Consequently, adding fault tolerance to INC still requires handling low-level switch code without a clear and organized methodology.

5.1.1 Intent-based networking

An intention (or intent) is an abstract declaration of an application or user desires from the network (TSUZAKI; OKABE, 2017). Intent-based systems are made to be easy to manage and require little or no intervention from external entities (CLEMM et al., 2022). An intent-based system comprises several components: profiling, translation, resolution, activation, and assurance.

The first component, named *profiling*, is where the users specify the intent. The intent specification is declarative, using a high-level representation, such as

natural language or GUIs. After establishing the intent, a *translation* component refines the high-level specification. During the translation, an intent goes through different abstraction levels: Policies and Configuration.

- **Policy.** First, the intent is translated into policies (CRAVEN et al., 2010) (ELKHATIB; COULSON; TYSON, 2017). A policy composes a set of rules used to control, change, or maintain a set of state objects. The policy is an abstract representation that coops with the heterogeneity of the networking devices. Thus, a policy does not represent a device-specific rule, control loop, or action but an abstraction that can be reused across different domains (CRAVEN et al., 2010).

- **Configuration.** After obtaining the policy, the intent is translated to device-specific configurations. The configuration is the lowest level of abstraction achieved in the translation and represents the concrete, device-specific commands.

Given that multiple configurations may contradict, an intent *resolution* component solves these conflicts and disambiguates them if necessary. Once there are no conflicts with other intents, the system *activates* the intent configuration, orchestrating the configuration deployment into network devices. Although the translation and conflict resolution mechanisms ensure the deployed intent aligns with the specified intent, deviations may occur over time. The dynamic nature of the network can lead to behavior changes that contradict the intents orchestrated earlier. To address this problem, an intent *assurance* component monitors the network to gather the network behavior. Subsequently, the assurance employs mechanisms that identify the discrepancies and generate new configurations to preserve the user intents (LEIVADEAS; FALKNER, 2022).

5.2 Araucaria design

In this section, we present the design of Araucaria[1], a system that relies on intents to enhance the fault tolerance of INCs. The design of Araucaria is based on the following *insights*.

[1] Araucaria is named after the *Araucaria angustifolia*, which are large and resilient trees that can be found in the south of Brazil. The Araucaria is a symbol of *resistance* in the fight for biodiversity conservation.

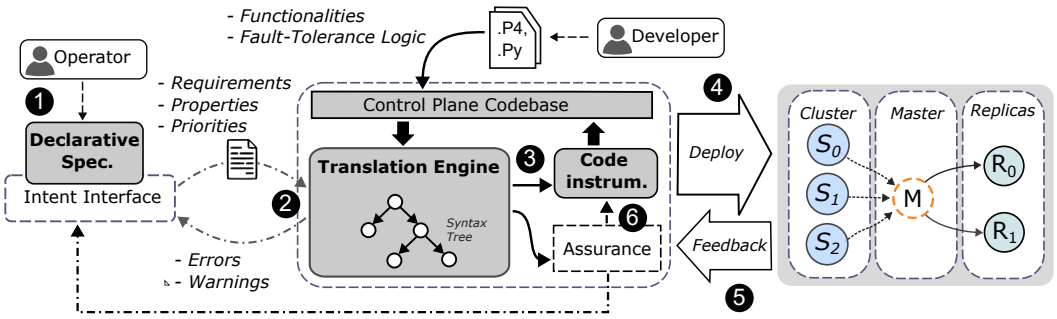

Figure 5.1 – The high-level architecture of Araucaria.

Fault tolerance specification. To simplify INC fault-tolerance, the specification of intents should abstract the implementation details. Intents must also be expressive enough to fulfill different fault tolerance requirements. To solve these issues, Araucaria defines a constrained natural language with primitive fault tolerance constructs, simplifying the specification of fault tolerance requirements.

Systematic instrumentation. Instrumenting fault tolerance into INC is difficult because of the limited composability of the P4 language – i.e., simply importing the fault tolerance functionality is impossible. To solve this challenge, Araucaria provides a refinement methodology that deduces the rules to provide fault tolerance from the input intent. The system also defines an instrumentation strategy capable of systematically instantiating fault tolerance building blocks into the INC source code.

5.2.1 Overview and workflow

Figure 5.1 illustrates the overview and workflow of Araucaria. Initially, the operator defines INC fault tolerance requirements (e.g., in terms of *consistency notion* and *number of replicas*) in a declarative manner (❶). The specification, made using a high-level language, goes through a translation process that analyzes the intent structure and semantics (❷). If the translation occurs without errors, Araucaria generates an intermediary representation, identifying pre-defined building blocks that implement the fault tolerance logic. For example, these building blocks include code fragments for enforcing *failure detection* or enabling *packet replaying* mechanisms. Araucaria then instruments the INC code by merging parsers and control flows from the fault tolerance building blocks into a single data plane program (❸).

This data plane program is instantiated into multiple switch replicas based on the required availability (❹), and an assurance module is instantiated in the control plane to coordinate fault recovery. Dynamically, Araucaria replicates the INC state across devices and provides periodic feedback to the assurance module about the status of the replicas (❺). In the event of an INC failure, the assurance module identifies the failure and adjusts the data plane configurations (❻). The new configuration ensures the system forwards subsequent application packets to a different replica INC.

5.2.2 Declarative intent specification

An *intent* is an abstract declaration of what an application or user desires from the network (TSUZAKI; OKABE, 2017). In Araucaria, each intent is associated with a predicate, including functionality and requirements. These predicates state a *property* of an intent. Inspired by (ELKHATIB; COULSON; TYSON, 2017; JACOBS et al., 2018), we formulate a constrained natural language to specify fault tolerance intents, where intents are structured as a tuple of primitive elements ⟨*operations, functionalities, requirements*⟩. Grammar 5.1 presents the language specification.

⟨*intent*⟩ ::= ⟨*op*⟩ `intent_name` '`{`' ⟨*pred*⟩ '`}`'
⟨*op*⟩ ::= '`Create`' | '`Delete`' | '`Update`' | '`Read`'
⟨*pred*⟩ ::= ⟨*req*⟩ '`,`' ⟨*func*⟩
⟨*func*⟩ ::= `functionality` '`fname`' '`[`' ⟨*input*⟩ '`]`' '`,`'
⟨*reqs*⟩ ::= ⟨*reqs*⟩ ',' ⟨*req*⟩ | ⟨*req*⟩
⟨*req*⟩ ::= ⟨*avail*⟩ | ⟨*cons*⟩ | ⟨*cons*⟩ '`[`' ⟨*merge*⟩ '`]`'
⟨*inputs*⟩ ::= ⟨*inputs*⟩ '`,`' ⟨*input*⟩ | ⟨*input*⟩ | ⟨*empty*⟩
⟨*input*⟩ ::= `name` '`:`' `value`
⟨*avail*⟩ ::= `tolerates` ⟨*int*⟩ '`failures`'
⟨*cons*⟩ :: = '`strong`' | '`eventual`'
⟨*merge*⟩ ::= max[hdr.value)] | '`add`'

Grammar 5.1 – The Araucaria grammar in BNF.

The language constructs are:

- ***Operations*** define actions (Create, Read, Update, and Delete) being applied

to instances of *functionalities*.

- ***Functionalities*** identify the specific INC that the intent aims to configure. Functionalities may be instantiated with customized *inputs*, used during the refinement to identify the necessary INC building blocks for deployment.

- ***Requirements*** is the core element in the Araucaria intent structure. A requirement aims to provide additional information about the intent:

  - *Availability* lets programmers ensure that specific INCs are available even if f failures occur (CHEUNG et al., 2021). Equivalent to the last chapter, we assume failures can occur by crashing, but switches do not experience an arbitrary behavior (i.e., no Byzantine cases).

  - *Consistency* allows programmers to specify replica correctness properties. The properties can vary between a strong or weaker notion that does not preserve ordering constraints. In addition, consistency may be followed by an optional merge function that provides ways to reduce conflicts between requests.

Listing 5.1 presents an example of an intent that can be built using the Araucaria intent language. In this example, an intent called '`syncIntent`' is created to manage an instance of NetGVT. The NetGVT functionality expects an optional parameter representing the number of processes interacting with the switches. The intent is for this instantiation of NetGVT to tolerate a failure while preserving the strong consistency semantics the system requires.

```
1 Create intent syncIntent{
2     functionality: NetGVT [
3                         size: 3
4                 ]
5     availability: tolerates 1 failure,
6     consistency: strong,
7 }
```

Listing 5.1 – Intent for synchronization functionality.

We implemented a compiler to translate Araucaria intents, including a *lexer* (to identify the tokens from the intent) and a *parser* (to analyze the syntactic structure of the intent and generate an abstract syntax tree (AST)). Also, the *semantic*

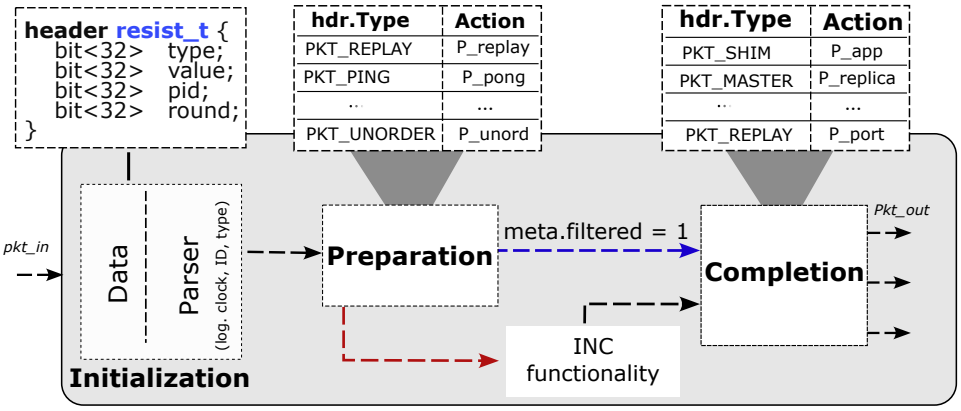

Figure 5.2 – Structure of an INC instrumented with the RESIST building blocks.

analysis ensures correct input formatting and examines potential conflicts, such as assessing whether the expressed merge function can achieve the desired consistency mode. The output of the intent compilation process is either an error or a valid intent represented at a lower level. This representation contains the fault tolerance functionality decomposed into smaller building blocks, which are discussed next.

5.2.3 Fault tolerance building blocks

Araucaria employs the fault tolerance protocol implemented by RESIST for recovering INCs from failures. As explained in Chapter 4, client traffic is processed by the main INC and replicated to a set of switch replicas in this protocol. If the main INC crashes, a control plane program (*SDN Controller*) identifies the failure using timeouts and collects the necessary state information from the replicas and the clients. The SDN controller can aggregate their information and identify the subset of packets that need to be retransmitted to a replica. The aggregated information may trigger a client replay, which recovers the replicas to a consistent state (PARK; OUSTERHOUT, 2019). The refinement process implemented by Araucaria merges the source code from RESIST within the INC source code. To allow the instrumentation, we first *decomposed* the RESIST into building blocks. Dynamically, the network operator can select one of the recovery strategies and Araucaria instruments the templates into the INC according to an application's consistency requirement, i.e., strong or weak.

Figure 5.2 provides a comprehensive overview of the underlying structure of a P4 program that has been instrumented by Araucaria to support fault tolerance

Table 5.2 – Summary of preparation actions according to the protocol packet types

Building Block	**Packet Type**	**Action**	**Process on INC**
Failure Detector	PKT_PING	Answers with pong	No
State Collection	PKT_COLLECT_ROUND	Collects current round and sends to the origin.	No
	PKT_NEW_SWITCH_ROUND	Updates the round to the new provided value and answers with ack.	No
Replication	PKT_BUFFERED	Mark to specific output port.	No
	PKT_UNORDERED	Mark to send it back to origin.	No
	PKT_ACK_MAIN_SWITCH	Updates packet state and drop packet.	No
	PKT_FROM_MASTER_TO_REPLICA	Mark as unordered in case it is.	**Yes**
Recovery	PKT_UNORDERED_REPLAY	Send it back.	No
	PKT_LAST_REPLAY_ROUND	Controller updates the last replay round register.	No
	PKT_REPLAY_STRONG_EVENTUAL	Checks if the amount of packets reaches the cluster size.	**Yes**
	PKT_REPLAY_FROM_SHIM	Mark it as unordered if it is. Otherwise, check if it is the last packet from the replay.	**Yes**

using a set of four reusable building blocks that resulted from our decomposition: *Failure Detector*, to identify if the main INC has failed; *Replication*, to synchronize state with other switches; *State Collection*, to determine how up to date a replica state is; and *Recovery*, to handle the recovery ensuring replicas follow a specific consistency notion after a failure. These building blocks are implemented as a set of P4 *templates* separated into different files. There are three types of templates: *initialization*, *preparation*, and *completion*.

Initialization. The initialization template includes per-packet variables, such as RESIST custom metadata, a new header and struct, and a parser state to extract this header. The header has information to identify servers and message types (e.g., recovery, collection) and to ensure linearizability through monotonically increasing round numbers. The message types are strictly important for ensuring consistency under failure. The parser state initializes these variables upon the arrival of a packet.

Preparation. This template includes a set of variables and actions to implement the logic building blocks discussed earlier. These variables are instantiated in the ingress pipeline. Additionally, the preparation template contains a set of commands for the apply block. These commands are arranged so that the template code precedes the INC functionality in the pipeline and prepares the packet for INC

Table 5.3 – Example of completion functionalities

Building Block	Packet Type	Action
Replication	PKT_FROM_SHIM_LAYER	Packet will be replicated and forwarded to servers
	PKT_FROM_MASTER_TO_REPLICA	Packet will acknowledge the main switch
	PKT_BUFFERED	Retransmit packet if the threshold is crossed
Recovery	PKT_REPLAY_FROM_SHIM	Mark to specific output port
	LAST_PACKET_RECEIVED	Updates the round to the new provided value and answers with acknowledgment

processing or filtering. The packet type triggers the P4 commands executed in the preparation phase, where each separated command is part of a logic building block. Table 5.2 outlines the operations performed in the preparation template based on the packet type and the corresponding building block. Examples of these operations include handling unordered and replay packets, detecting the last replayed packet, responding to ping requests, replicating packets, and acknowledging packets. The table also indicates the types of packets processed by the INC. The only packet types the INC should process are packets from the shim layer, replicated packets, and correctly ordered replayed packets. Filtering is crucial in certain cases, such as when the packet being handled is an acknowledgment for replication or a message for failure detection. In these instances, the INC should not process the packets.

Completion. Table 5.3 presents the packet types and actions implemented in the completion template. This template includes packet management mechanisms capable of applying multicast tables, keeping storage for packet losses, and changing header arguments. The template is included after the INC functionality to preserve headers and packet metadata for correct INC processing. The switch defines the packet destination in this block based on the header type. In addition, the completion template adds code to the egress, cloning packets to replicate and acknowledge shim layers on servers.

5.2.4 INC source code instrumentation

To instrument the *templates* discussed in the previous section into an INC, Araucaria systematically traverses the INC code and writes `include` pre-processors strategically to instantiate the building blocks in distinct parts of the INC source code. We require INC variables to follow a specific naming convention to avoid conflicts with variable names used by Araucaria. This requires that INC variable

names do not start with Araucaria *reserved words*, thereby establishing a contract between INC developers and DevOps.

Step #1: Metadata and header definitions. The first phase of the instrumentation includes the definitions of `headers` and `structs` at the beginning of the INC source code. Araucaria variables include a new header definition for ensuring linearizability during replication and specific metadata used in the control flow for making per-packet decisions. After merging the INC headers, `structs`, and metadata definitions, we start instrumenting the parser.

Step #2: Parser instrumentation. Our parser instrumentation leverages a modular design that decouples the Araucaria parser state from traditional protocol states, such as Ethernet and IPv4. This decoupling allows us to incrementally include the Araucaria protocol into the INC parser, avoiding ambiguities. This is achieved in two steps: first, placing the Araucaria state between the INC header extraction and the transitions, with the INC state working as the 'parent' node of the Araucaria state. Additionally, the Araucaria state incorporates previous INC state transitions. By ensuring that the extraction of an INC state is consistently followed by the extraction of the Araucaria header, we effectively mitigate the risk of introducing loops and non-determinism in the parser structure.

Figure 5.3 presents the parser of Araucaria, ignoring the states for standard protocols. Figure 5.4 presents a general INC parser, including the INC state. During the instrumentation, Araucaria includes the transitions from the INC in a transition of the `Parse_Araucaria`. The INC state transitions are also removed, adding a single transition to the Araucaria state. The resulting parser is presented in Figure 5.5.

Step #3: Control flow composition. After instrumenting the INC parser, we start instrumenting the control blocks. Control blocks in P4 can contain several constructs, such as *tables*, *actions*, *registers*, and *apply blocks*. To compose the INC source code with the fault tolerance logic, Araucaria extends the definition of tables, actions, and registers in the INC code to offer consistency. These include variables for serializing requests between replicas, keeping consistency, and actions to handle packets from the replica and coordinator. Next, Araucaria proceeds to instrument the source code within the apply block. Our approach includes the entire INC apply block between the *preparation* and *completion* templates. This will allow, for example, to identify unordered packets in the preparation template to avoid

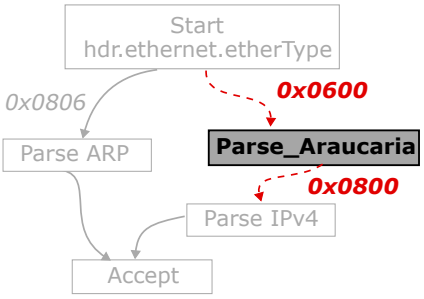

Figure 5.3 – Fault tolerance parser.

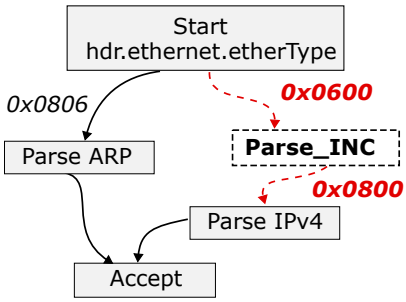

Figure 5.4 – INC parser.

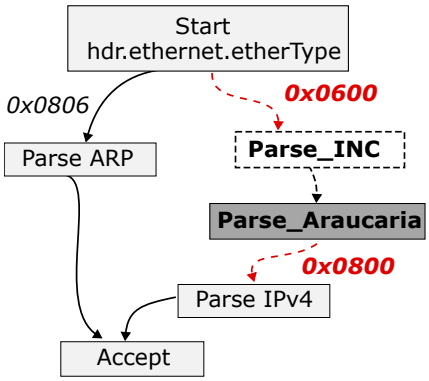

Figure 5.5 – Instrumented parser.

processing them in the INC code. Another example is to ensure that multicasts in the completion template do not process packets within the same switch.

5.2.5 Configuration

Beyond instrumenting the source code of the INC for a specific intent, Araucaria generates the configuration to the network devices. The configuration Araucaria creates is responsible for different tasks: (1) setting up the switch ports for replicating packets; (2) configuring the switches and servers to operate accordingly to a specific consistency model; and (3) setting up the communication with all the servers running applications using the INC.

- **Replication for availability.** The availability requirement is mapped to a set of replicas. Topology information containing the input/output ports of the devices is used to create multicast groups. These groups ensure the INC forwards packets to all replicas, thereby synchronizing the state of the replicas.

- **Defining consistency.** The consistency model to be used is configured in

the servers, establishing how they should replay packets (whenever necessary for recovery). In addition, the configuration of merge functions is created by mapping commands that solve conflicts during the recovery.

- **Recovery.** Finally, Araucaria creates rules to determine the need for retransmissions in case of a failure. These rules comprise a list of servers and their corresponding IPs. This list enables orchestrating the recovery by triggering route and interface changes after a failure.

5.3 Evaluation

In this section, we present experimental results to show that (i) the refinement components of Araucaria effectively provide fault tolerance; (ii) the system provides abstractions for reducing the overhead of recovery scenarios; (iii) the system scales for increasing amounts of intents.

5.3.1 Experimental settings

Implementation. Araucaria decomposes and reuses the source code from RESIST. The controller is implemented as a multithread application (∼250 LoC), capable of *sniffing* the network to collect devices' status information and computing the necessary information to maintain consistency using `Scapy`. We built refined building blocks using P4-16 for both V1Model (∼350 LoC) and a proof-of-concept for the TNA model (∼610 LoC). We employ a specific multicast for replication, combining cloning and recirculation to buffer packets and recovery from packet loss in the BMv2. Our hardware PoC still does not implement the mechanism to recover from packet losses, but implements the other building blocks. The intent compiler is a new contribution and is implemented using PLY (Python Lex-Yacc) to build the Araucaria language (∼120 LoC).

Testbed Setup. Our prototype for the V1Model is evaluated in a Linux virtual machine with an Intel® i5-10210U CPU @ 1.60GHz using two dedicated cores, 2 GB of memory, and Ubuntu 20.04 LTS. To assess the functionality of the system, we use BMv2, a behavior model for P4 programs. The network is emulated using mininet. The topology includes three hosts connected to 2 switches. All the

hosts run the application instantiated by the intents. One switch acts as a replica and the other as the main.

We evaluate our TNA PoC of Araucaria in a Tofino testbed. The experiments were conducted with two Wedge 100BF-32X 32-port programmable switches with a 3.2 Tbps Tofino ASIC using SDE 9.9.1.

Methodology and metrics. We run experiments to check the *feasibility* of achieving fault tolerance with Araucaria and measure the number of requests processed per second (RPS), a methodology similar to what we did for RESIST. To understand the tradeoffs and scalability of the system, we investigate the *latency* to deploy an instrumented code and the time to translate intents using multiple intent configurations. Finally, we measure the system overhead regarding rules and primitives used in the switches.

5.3.2 A running example

To understand the end-to-end intent specification and refinement process, we provide a use case in our emulated setup. In this use case, we examine the intent specified in Section §5.2.2, Listing 5.1, and check the refinement process and the effectiveness of fault tolerance. Our use case deploys NetGVT, an INC proposed in Chapter 3. Next, we demonstrate the step-by-step process of instrumenting this INC using Araucaria.

```
1
2 //multicast rules created for replication
3 "multicast_group_entries" : [{"multicast_group_id" : 1, "
    replicas" : [{"egress_port" : 1, "instance" : 1}]
4
5 //clone port for buffering
6 mirroring_add 500 3
7
8 //Writing specific consistency model
9 register_write consistency_model 0 1
```

Listing 5.2 – Fragment of commands and configurations created

Refinement. Listing 5.2 presents a fragment of configurations and commands created by the Araucaria refinement process. The '`syncnIntent`' (Listing 5.1) is refined into rules in JSON that instantiate two replicas and create a multi-

Figure 5.6 – Analysing the behavior after failure.

cast group. A mirroring port is used by Araucaria switches to clone packets and keep a copy internally in the switch. The refinement also translates the strong consistency notion to CLI commands for the switches by writing the `consistency_model` register value to 1 (corresponding to the strong consistency behavior).

Fault tolerance analysis. To analyze the ability of Araucaria to provide fault tolerance for existing INCs, we deployed the intent and injected a failure in the switch running the INC. We then analyzed the number of requests sent and acknowledged by each one of the servers.

Figure 5.6 presents the number of requests per second processed by each server. After injecting a failure at the switch, the controller identifies the crash after a timeout. The failure leads all servers to stop transmitting packets. After the controller collects the status of devices to achieve consistency, the servers are notified about the failure and start the recovery by switching their communication to a different replica ($\sim$ 16s). Next, each server replays packets (that were lost during the switch failure) to the new main replica. After the replica finishes processing all the packets, the application returns to regular operation ($\sim$ 18s). This indicates that the Araucaria translation and refinement process does not corrupt the fault tolerance mechanism.

5.3.3 Deployment micro-benchmarks

Compilation time. We also evaluate the compilation time of our hardware PoC to understand the impact of NetGVT on deployment. We compare the results using an instrumented program by Araucaria and the baseline (with no instrumen-

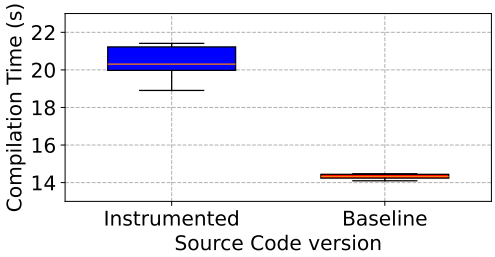

Figure 5.7 – Compilation Time.

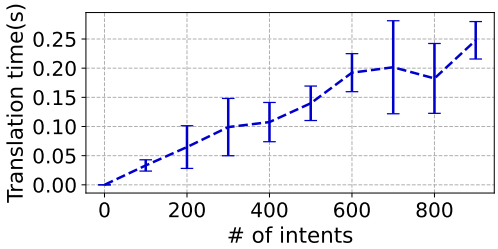

Figure 5.8 – Time to translate intents.

Table 5.4 – Rules and primitives used by Araucaria.

Primitives/Rules	**Usage**
Register Actions	8
Registers	3
Tables	4
Table Entries	15
Multicast Groups	1

tation). Figure 5.7 presents the results for the compilation time. We observe that Araucaria delays the compilation by nearly 8 seconds. Although this affects the deployment, we consider this an acceptable tradeoff.

Scalability. To show the scalability of the compiler, we generated a variable amount of intents in the Araucaria language. In Figure 5.8, we show the time it takes to complete translation for varying intents using batches of *multiple* sizes. These experiments were executed in a Linux virtual machine with an Intel® i5-10210U CPU @ 1.60GHz using two dedicated cores and 2 GB of memory. We observe that the time to translate intents increases linearly with the amount of intents. Translating a single intent takes less than 0.05 seconds while translating 800 intents takes only 0.20 seconds. Overall, this result indicates that the system can translate intents rapidly.

Program Primitives. To understand the impact on resource usage, Table 5.4 presents the number of P4 primitives generated by the refinement process. We focus on these primitives in our evaluation because they can be generalized to multiple targets. The Araucaria hardware PoC employs four match+action tables that match the protocol type field, defining the action to be taken. The actions of these tables can use eight register actions over three different registers. These results indicate that only a small amount of P4 primitives are used by Araucaria.

Code Volume. We also measured the number of lines of code (LoC) necessary to build INC fault tolerance. Table 5.5 summarizes the lines of code to

Table 5.5 – Line of codes of Araucaria modular templates.

Building Block	Lines of Code (LoC)			
	Initialization	Preparation	Completion	Total
Collection	2	23	0	25
Failure Detection	1	5	0	6
Replication	7	24	9	40
Replay	9	51	9	69
Common	37	57	12	106
Total	56	160	30	492

implement each building block. These results characterize the amount of P4 code Araucaria release developers to write by allowing simple requirements to be specified in our high-level language. Araucaria instruments the INC with reusable modules, allowing developers to focus only on INC-specific logic, thus reducing the code volume.

5.4 Related Work

Intents. Intents have been explored in the context of various platforms, including network function virtualization (SCHEID et al., 2017), software-defined networking (JACOBS et al., 2018; ESPOSITO et al., 2018; HEORHIADI et al., 2018; ALALMAEI et al., 2020), and industrial networks (SAHA et al., 2018). Researchers have been investigating techniques for managing Access Control Lists (ACL) (TIAN et al., 2019b; LI et al., 2021) and Quality of Service (QoS) using intents, along with exploring diverse abstractions to express intents. These abstractions include policy graphs (PRAKASH et al., 2015), natural language (JACOBS et al., 2018), graphical user interfaces (FEMMINELLA; PERGOLESI; REALI, 2020), and constrained natural language grammars (SCHEID et al., 2020). Recent works also used high-level intents to create *match+action* entries for P4 programs (ANGI et al., 2022; LEWIS et al., 2018). P4I/O (RIFTADI; KUIPERS, 2019) facilitates P4 adoption by using a language to express policies for switches and merges different source files using reusable templates to upgrade the switch configuration dynamically. However, the authors only demonstrated the concept of deploying heavy hitter detection. Araucaria focuses on other domains by extending INC functionality with specific abstractions for fault tolerance.

High-level data structures. An orthogonal research field focuses on

Table 5.6 – Tradeoffs between fault tolerance techniques

Technique	Synchronization	Recovery	Complexity
Synchronous Packet Replication	Slow Sync., Small Amount of Memory, Large bandwidth	Fast Recovery	Low
Asynchronous Packet Replication	Fast Sync., Large bandwidth.	Fast Recovery only with CRDTs, otherwise slow, Increased amount of memory. from servers	High
Asynchronous Snapshot Replication	Fast Sync., Small bandwidth, Double memory on switches	Fast Recovery only with CRDTs, Increased amount of memory	High

bringing higher-level abstractions to P4. Examples of high-level abstractions include data structure elasticity (HOGAN et al., 2022), loops (ALCOZ et al., 2022), modularity (FATTAHOLMANAN et al., 2021) composability (SONI et al., 2020), and heterogeneity (GAO et al., 2020). Although these efforts can simplify programming, they do not support functionalities to express intents.

5.5 Discussions

In this chapter, we presented Araucaria, a system to provide fault tolerance requirements for INC expressed as intents. The system enables the specification to be made in an intent language close to natural language. Subsequently, a refinement mechanism instruments the INC code to ensure fault tolerance. The instrumentation includes code for a fault tolerance protocol that generalized the techniques from RESIST.

This generalization enabled us to analyze tradeoffs for fault tolerance regarding different configuration setups, and it sheds light on possible priorities we can consider during the refinement. Table 5.6 summarizes these considerations. We can observe that approaches using asynchronous replication can lead to more negligible overhead for replication, but the complexity for recovery is higher and can only scale with CRDTs. In addition, recovery may require log-replay techniques to ensure strong consistency, which can consume memory from servers. More specifically, asynchronously replicating snapshots leads to smaller bandwidth consumption because of the lower rate for replication but consumes more switch resources. On the other hand, synchronous replication adds overhead for non-failure scenarios but ensures fault recovery is always fast.

By employing techniques from intent-based networks, we observed that making an INC fault tolerant was simpler than manually employing the process described in Chapter 4. In addition, devising a refinement approach required us to explore the design space from a top-down perspective. This exploration enabled us to identify missing pieces in the implementation of RESIST. More importantly, the instrumentation with NetGVT became easier to reproduce and more precise because the source code became modular to fit Araucaria's needs.

6 CONCLUSIONS

This chapter provides a summary of the contributions made through this thesis. We outline the future work that results from the contributions presented in the previous chapters. Finally, we present our concluding remarks.

6.1 Summary of contributions

In-network computing trades the rich set of functionalities from traditional distributed systems by the performance benefits from switches already in the network. The functionalities are often limited because of the hardware constraints, which require a customized design to circumvent the switch constraints. However, devising a customized design usually brings uncommon challenges for non-specialized programmers. These challenges include handling mechanisms that are unnecessary to worry about in the server-based implementation. In addition, reasoning about the functionality configuration often requires reasoning about low-level protocols and networking, which are also hidden by more traditional network management tools. This thesis is divided into three main challenges for INC: handling constraints of the data plane, supporting tolerance to failures, and providing simple ways to manage the offloaded functionality.

First, we presented a categorization of design considerations developers can take to customize their INCs when they face hardware constraints. We distinguish these design considerations according to computational aspects (e.g., ALU operations) and memory aspects (e.g., RAM, TCAM). These can be complemented by the type of tasks to map to the data plane, and for each of them, we propose design considerations that can be instantiated to transform the INC and overcome a constraint. We devised an in-network computing system called NetGVT that offloads virtual time computation to the data plane. NetGVT instantiates these design considerations to enable the scale of the system to multiple processes and keep a low memory footprint. Our evaluation demonstrates that our design can be used efficiently in a state-of-the-art platform.

Secondly, we studied the impact that switch crashes make on the entire distributed system. We analyzed the design and functionality of existing systems and identified inconsistencies that can occur in their behavior once a crash occurs. We

also identified what consistency models are needed to recover the state of the studied systems into a non-faulty consistent state. We then designed a system called RESIST that can make INC systems fault-tolerant. RESIST adapts fault-tolerance techniques, such as log replay and replication, to provide fault tolerance without introducing significant overhead for the INC. Our evaluation with benchmarks of RESIST main building blocks in the hardware platform demonstrates its viability. In addition, we evaluate the entire RESIST functionality with emulations, demonstrating the performance of our system at scale.

Thirdly, we investigated the complexity that INC brings to system management. We analyzed the particular case of INCs studied in the previous chapters and the configuration necessary to make them fault-tolerant. We then design Araucaria, a system to specify and refine fault tolerance requirements for INC using high-level intents. Our evaluation demonstrates that translating the high-level specification is a lightweight process. We also present a comprehensive functionality evaluation on a behavior model of P4, along with experiments on hardware.

6.2 Future work

In this section, we discuss the main limitations identified in the previous chapter, which we aim to address in future work.

The design considerations discussed in Chapter 3 still need to be formally described and cataloged, and the discussed techniques can be expanded by including other design insights. Although we focused on the most popular devices for data plane programmability, investigating and systematizing the methodology to employ the considerations and expanding to the different architectures of forwarding devices (e.g., dRMT, Linux TC, smart NICs) is necessary for generality. A promising research direction is investigating other use-cases for the proposed techniques, e.g., *algorithms with predictions*, also being able to reason about the consistency/robustness trade-off (MITZENMACHER; VASSILVITSKII, 2022) automatically. These will rely on considerations such as decomposition and memoization and adding real application semantics.

In the future, we aim to combine RESIST with reconfiguration mechanisms, e.g., employing the reconfiguration phases of the Viewstamp Replication Protocol (LISKOV; COWLING, 2012). In addition, merging the content from the replicas

to the updated switch may be necessary for live reconfiguration. This requirement is because the new switch may not support or require changing the rules from the previous switch configuration.

From the fault tolerance perspective, we plan to use these logs for reasoning about the cause of events in the network by replaying the logs in a controlled environment (BABAEI; BAGHERZADEH; DINGEL, 2020; WANG; HAO; CUI, 2022; BESCHASTNIKH et al., 2016). In future research, we plan to investigate more sophisticated instrumentation and study how to interoperate with varied applications written for other technologies, e.g., eBPF, DPDK, or even hosted in accelerators, e.g., GPUs, SmartNICs. This can allow the collection of these logs to be efficiently and subsequently used for different ends.

Placing replicas in optimal locations based on available resources is also in perspective. In the future, we plan to address other kinds of failures, including the system's resilience against threats (such as (LIU et al., 2023)) and the capacity of INCs to perform transitions into invalid states because of bugs or malfunctioning (STERBENZ et al., 2010).

We plan to generalize the high-level intent refinement to other distributed system functionalities running in different platforms, such as the Cloud or Edge. To that end, investigating techniques to extract intents from developer-centric functionalities and provide modular and automatic extensions for intent-based languages is an interesting future direction. Different scheduling techniques can be used to define updates. In addition, studying natural language usage to specify intents for the intent grammar is in perspective.

6.3 Closing remarks

In this thesis, we (I) presented a set of design considerations that we employed to build a new in-network system that can efficiently synchronize virtual time within switches; (II) we designed a fault tolerance mechanism for INC that can quickly react to failures without impacting on non-fault behavior; and (III) developed a system to configure fault tolerance for INCs using intents.

At a high level, this thesis focused on *designing robust in-network computing systems*, aiming to make INCs easier to build and manage. This problem is complex because the data plane imposes several constraints and demands handling faults with

customized designs, requiring the configuration of low-level commands from multiple devices. We leveraged the state-of-the-art programmable switches using P4 to create solutions to these problems, including design considerations for efficient algorithms, INC fault tolerance mechanisms, and intent refinement methodologies. In-network computing is a promising field, and circumventing constraints, failures, and configuration challenges is essential for making the INC paradigm viable. We are excited about designing the next generation of INCs that can rely on SmartNICs, eBPF, and DPUs, and, why not, new open packet processing architectures for switches.

REFERENCES

ABEYDEERA, M.; SANCHEZ, D. Chronos: Efficient speculative parallelism for accelerators. In: **Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems**. [S.l.: s.n.], 2020. p. 1247–1262.

ABHASHKUMAR, A. et al. Supporting diverse dynamic intent-based policies using janus. In: **Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies**. New York, NY, USA: Association for Computing Machinery, 2017. (CoNEXT '17), p. 296–309. ISBN 9781450354226. Disponível em: <https://doi.org/10.1145/3143361.3143380>.

ALALMAEI, S. et al. Sdn heading north: Towards a declarative intent-based northbound interface. In: IEEE. **2020 16th International Conference on Network and Service Management (CNSM)**. [S.l.], 2020. p. 1–5.

ALCOZ, A. G. et al. Reducing p4 language's voluminosity using higher-level constructs. In: **Proceedings of the 5th International Workshop on P4 in Europe**. [S.l.: s.n.], 2022. p. 19–25.

ANGI, A. et al. Nlp4: An architecture for intent-driven data plane programmability. In: IEEE. **2022 IEEE 8th International Conference on Network Softwarization (NetSoft)**. [S.l.], 2022. p. 25–30.

BABAEI, M.; BAGHERZADEH, M.; DINGEL, J. Efficient reordering and replay of execution traces of distributed reactive systems in the context of model-driven development. In: **Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems**. [S.l.: s.n.], 2020. p. 285–296.

BARBETTE, T. et al. A high-speed load-balancer design with guaranteed per-connection-consistency. In: **17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)**. [S.l.: s.n.], 2020. p. 667–683.

BASAT, R. B. et al. Pint: Probabilistic in-band network telemetry. In: **Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication**. [S.l.: s.n.], 2020. p. 662–680.

BEN-BASAT, R. et al. Efficient measurement on programmable switches using probabilistic recirculation. In: IEEE. **2018 IEEE 26th International Conference on Network Protocols (ICNP)**. [S.l.], 2018. p. 313–323.

BENSON, T. A. In-network compute: Considered armed and dangerous. In: **Proceedings of the Workshop on Hot Topics in Operating Systems**. New York, NY, USA: ACM, 2019. (HotOS '19), p. 216–224. ISBN 978-1-4503-6727-1. Disponível em: <http://doi.acm.org/10.1145/3317550.3321436>.

BERDE, P. et al. Onos: towards an open, distributed sdn os. In: **Proceedings of the third workshop on Hot topics in software defined networking**. [S.l.: s.n.], 2014. p. 1–6.

BESCHASTNIKH, I. et al. Debugging distributed systems. **Communications of the ACM**, ACM New York, NY, USA, v. 59, n. 8, p. 32–37, 2016.

BIRMAN, K.; JOSEPH, T. Exploiting virtual synchrony in distributed systems. In: **Proceedings of the eleventh ACM Symposium on Operating systems principles**. [S.l.: s.n.], 1987. p. 123–138.

BOSSHART, P. et al. P4: Programming protocol-independent packet processors. **ACM SIGCOMM Computer Communication Review**, ACM New York, NY, USA, v. 44, n. 3, p. 87–95, 2014.

BOSSHART, P. et al. P4: Programming protocol-independent packet processors. **SIGCOMM Comput. Commun. Rev.**, ACM, 2014.

BOSSHART, P. et al. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. **ACM SIGCOMM Computer Communication Review**, ACM New York, NY, USA, v. 43, n. 4, p. 99–110, 2013.

BREITNER, J.; SMITH, C. Lock-step simulation is child's play (experience report). **Proceedings of the ACM on Programming Languages**, ACM New York, NY, USA, v. 1, n. ICFP, p. 1–15, 2017.

BRITO, A. V. et al. A distributed simulation platform using hla for complex embedded systems design. In: IEEE. **2015 IEEE/ACM 19th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)**. [S.l.], 2015. p. 195–202.

CASADO, M. et al. Ethane: Taking control of the enterprise. **ACM SIGCOMM computer communication review**, ACM New York, NY, USA, v. 37, n. 4, p. 1–12, 2007.

CHANDY, K. M.; MISRA, J. Distributed simulation: A case study in design and verification of distributed programs. **IEEE Transactions on Software Engineering**, IEEE, n. 5, p. 440–452, 1979.

CHEUNG, A. et al. New directions in cloud programming. **In 11th Conference on Innovative Data Systems Research (CIDR' 21)**, 2021.

CHIESA, M.; VERDI, F. L. Network monitoring on multi-pipe switches. **Proceedings of the ACM on Measurement and Analysis of Computing Systems**, ACM New York, NY, USA, v. 7, n. 1, p. 1–31, 2023.

CHOI, I. et al. Hydra:{Serialization-Free} network ordering for strongly consistent distributed applications. In: **20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)**. [S.l.: s.n.], 2023. p. 293–320.

CLEMM, A. et al. **Intent-Based Networking - Concepts and Definitions**. RFC Editor, 2022. RFC 9315. (Request for Comments, 9315). Disponível em: <https://www.rfc-editor.org/info/rfc9315>.

COELHO, B.; SCHAEFFER-FILHO, A. Backorders: using random forests to detect ddos attacks in programmable data planes. In: **Proceedings of the 5th International Workshop on P4 in Europe**. [S.l.: s.n.], 2022. p. 1–7.

CRAVEN, R. et al. Decomposition techniques for policy refinement. In: **2010 International Conference on Network and Service Management**. [S.l.: s.n.], 2010. p. 72–79. ISSN 2165-963X.

DAMIANOU, N. et al. The ponder policy specification language. In: SPRINGER. **International Workshop on Policies for Distributed Systems and Networks**. [S.l.], 2001. p. 18–38.

DANG, H. T. et al. P4xos: Consensus as a network service. **IEEE/ACM Transactions on Networking**, IEEE, v. 28, n. 4, p. 1726–1738, 2020.

EICHHOLZ, M. et al. Dependently-typed data plane programming. **Proceedings of the ACM on Programming Languages**, ACM New York, NY, USA, v. 6, n. POPL, p. 1–28, 2022.

EKER, A. et al. Controlled asynchronous gvt: accelerating parallel discrete event simulation on many-core clusters. In: **Proceedings of the 48th International Conference on Parallel Processing**. [S.l.: s.n.], 2019. p. 1–10.

ELKHATIB, Y.; COULSON, G.; TYSON, G. Charting an intent driven network. In: IEEE. **2017 13th International Conference on Network and Service Management (CNSM)**. [S.l.], 2017. p. 1–5.

ESPOSITO, F. et al. A behavior-driven approach to intent specification for software-defined infrastructure management. In: **2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)**. [S.l.: s.n.], 2018. p. 1–6. ISSN null.

FATTAHOLMANAN, A. et al. P4 weaver: Supporting modular and incremental programming in p4. In: **Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)**. [S.l.: s.n.], 2021. p. 54–65.

FAVERO, N. et al. Quic-tr4ck: Mitigando ataques quic-flood usando planos de dados programáveis. In: **Anais do XXIII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais**. Porto Alegre, RS, Brasil: SBC, 2023. p. 307–320. ISSN 0000-0000. Disponível em: <https://sol.sbc.org.br/index.php/sbseg/article/view/27215>.

FEAMSTER, N.; REXFORD, J.; ZEGURA, E. The road to sdn: An intellectual history of programmable networks. **SIGCOMM Comput. Commun. Rev.**, ACM, New York, NY, USA, v. 44, n. 2, p. 87–98, abr. 2014. ISSN 0146-4833. Disponível em: <http://doi.acm.org/10.1145/2602204.2602219>.

FEAMSTER, N.; REXFORD, J.; ZEGURA, E. The road to sdn: an intellectual history of programmable networks. **ACM SIGCOMM Computer Communication Review**, ACM New York, NY, USA, v. 44, n. 2, p. 87–98, 2014.

FEMMINELLA, M.; PERGOLESI, M.; REALI, G. Simplification of the design, deployment, and testing of 5g vertical services. In: IEEE. **NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium**. [S.l.], 2020. p. 1–7.

FUJIMOTO, R. M.; HYBINETTE, M. Computing global virtual time in shared-memory multiprocessors. **ACM Trans. Model. Comput. Simul.**, Association for Computing Machinery, New York, NY, USA, 1997. ISSN 1049-3301. Disponível em: <https://doi.org/10.1145/268403.268404>.

FUJIMOTO, R. M. et al. Parallel and distributed simulation in the cloud. **SCS M&S Magazine**, Citeseer, v. 3, p. 1–10, 2010.

GAO, J. et al. Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics. In: **Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication**. [S.l.: s.n.], 2020. p. 435–450.

GENG, Y. et al. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In: **15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)**. [S.l.: s.n.], 2018. p. 81–94.

GLEBKE, R. et al. Towards executing computer vision functionality on programmable network devices. In: **Proceedings of the 1st ACM CoNEXT Workshop on Emerging in-Network Computing Paradigms**. [S.l.: s.n.], 2019. p. 15–20.

GLEBKE, R. et al. Towards executing computer vision functionality on programmable network devices. In: **Proceedings of the 1st ACM CoNEXT Workshop on Emerging In-Network Computing Paradigms**. New York, NY, USA: Association for Computing Machinery, 2019. (ENCP '19), p. 15–20. ISBN 9781450370004. Disponível em: <https://doi.org/10.1145/3359993.3366646>.

GONSIOROWSKI, E.; CAROTHERS, C.; TROPPER, C. Modeling large scale circuits using massively parallel discrete-event simulation. In: IEEE. **2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems**. [S.l.], 2012. p. 127–133.

GUDE, N. et al. Nox: towards an operating system for networks. **ACM SIGCOMM computer communication review**, ACM New York, NY, USA, v. 38, n. 3, p. 105–110, 2008.

GUNTURI, R.; JOHNSON, E.; SEOW, C. **Packet processing pipeline**. [S.l.]: Google Patents, 2005. US Patent App. 10/766,282.

GYÖRGYI, C.; LAKI, S.; SCHMID, S. P4rrot: Generating p4 code for the application layer. **ACM SIGCOMM Computer Communication Review**, ACM New York, NY, USA, v. 53, n. 1, p. 30–37, 2023.

HAN, W. et al. Francis: Fast reaction algorithms for network coordination in switches. **arXiv preprint arXiv:2204.14138**, 2022.

HANNAY, J. E.; BERG, T. van den. The nato msg-136 reference architecture for m&s as a service. In: **Proc. NATO Modelling and Simulation Group Symp. on M&S Technologies and Standards for Enabling Alliance Interoperability and Pervasive M&S Applications (STO-MP-MSG-149)**. [S.l.: s.n.], 2017.

HAUSER, F. et al. A survey on data plane programming with p4: Fundamentals, advances, and applied research. **arXiv preprint arXiv:2101.10632**, 2021.

HE, Y. et al. A generic service to provide in-network aggregation for key-value streams. In: **Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2**. [S.l.: s.n.], 2023. p. 33–47.

HEORHIADI, V. et al. Intent-driven composition of resource-management sdn applications. In: . New York, NY, USA: Association for Computing Machinery, 2018. (CoNEXT '18), p. 86–97. ISBN 9781450360807. Disponível em: <https://doi.org/10.1145/3281411.3281431>.

HERLIHY, M. P.; WING, J. M. Linearizability: A correctness condition for concurrent objects. **ACM Transactions on Programming Languages and Systems (TOPLAS)**, ACM New York, NY, USA, v. 12, n. 3, p. 463–492, 1990.

HOGAN, M. et al. Elastic switch programming with p4all. In: **Proceedings of the 19th ACM Workshop on Hot Topics in Networks**. [S.l.: s.n.], 2020. p. 168–174.

HOGAN, M. et al. Modular switch programming under resource constraints. In: **19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)**. [S.l.: s.n.], 2022. p. 193–207.

HUANG, H.; WU, W. Hypersfp: Fault-tolerant service function chain provision on programmable switches in data centers. In: IEEE. **NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium**. [S.l.], 2022. p. 1–9.

HUSSEIN, R. et al. Graphinc: Graph pattern mining at network speed. **Proceedings of the ACM on Management of Data**, ACM New York, NY, USA, v. 1, n. 2, p. 1–28, 2023.

ISTVÁN, Z. et al. Consensus in a box: Inexpensive coordination in hardware. In: **13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)**. [S.l.: s.n.], 2016. p. 425–438.

IVKIN, N. et al. Qpipe: Quantiles sketch fully in the data plane. In: **Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies**. [S.l.: s.n.], 2019. p. 285–291.

JACOBS, A. S. et al. Refining network intents for self-driving networks. In: **Proceedings of the Afternoon Workshop on Self-Driving Networks**. New York, NY, USA: Association for Computing Machinery, 2018. (SelfDN 2018), p. 15–21. ISBN 9781450359146. Disponível em: <https://doi.org/10.1145/3229584.3229590>.

JEFFERSON, D. R. Virtual time. **ACM Transactions on Programming Languages and Systems (TOPLAS)**, ACM, 1985.

JEPSEN, T. et al. In-network support for transaction triaging. **Proceedings of the VLDB Endowment**, VLDB Endowment, v. 14, n. 9, p. 1626–1639, 2021.

JEPSEN, T. et al. Infinite resources for optimistic concurrency control. In: **Proceedings of the 2018 Morning Workshop on In-Network Computing**. [S.l.: s.n.], 2018. (NetCompute '18).

JIN, X. et al. Netcache: Balancing key-value stores with fast in-network caching. In: **Proceedings of the 26th Symposium on Operating Systems Principles**. New York, NY, USA: Association for Computing Machinery, 2017. (SOSP '17), p. 121–136. ISBN 9781450350853. Disponível em: <https://doi.org/10.1145/3132747.3132764>.

JIN, X. et al. Netcache: Balancing key-value stores with fast in-network caching. In: **Proceedings of the 26th Symposium on Operating Systems Principles**. [S.l.: s.n.], 2017. p. 121–136.

JIN, X. et al. Netcache: Balancing key-value stores with fast in-network caching. In: **Proceedings of the 26th Symposium on Operating Systems Principles**. [S.l.]: ACM, 2017. (SOSP '17).

JIN, X. et al. Netchain: Scale-free sub-rtt coordination. In: **15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)**. [S.l.: s.n.], 2018. p. 35–49.

JIN, X. et al. Netchain: Scale-free sub-rtt coordination. In: **15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)**. [S.l.: s.n.], 2018. p. 35–49.

JUNIOR, E. R. M. et al. Closing the gap between lookahead and checkpointing to provide hybrid synchronization. In: SBC. **Anais do XLVII Seminário Integrado de Software e Hardware**. [S.l.], 2020. p. 104–115.

KANNAN, P. G.; JOSHI, R.; CHAN, M. C. Precise time-synchronization in the data-plane using programmable switching asics. In: **Proceedings of the 2019 ACM Symposium on SDN Research**. New York, NY, USA: Association for Computing Machinery, 2019. (SOSR '19), p. 8–20. ISBN 9781450367103. Disponível em: <https://doi.org/10.1145/3314148.3314353>.

KHERADMAND, A.; ROSU, G. P4k: A formal semantics of p4 and applications. **arXiv preprint arXiv:1804.01468**, 2018.

KIANPISHEH, S.; TALEB, T. A survey on in-network computing: Programmable data plane and technology specific applications. **IEEE Communications Surveys & Tutorials**, IEEE, 2022.

KIM, C. et al. In-band network telemetry via programmable dataplanes. In: **ACM SIGCOMM**. [S.l.: s.n.], 2015. v. 15.

KIM, D. et al. Unleashing in-network computing on scientific workloads. **arXiv preprint arXiv:2009.02457**, 2020.

KIM, D. et al. Redplane: enabling fault-tolerant stateful in-switch applications. In: **Proceedings of the 2021 ACM SIGCOMM 2021 Conference**. [S.l.: s.n.], 2021. p. 223–244.

KIM, G.; LEE, W. In-network leaderless replication for distributed data stores. **Proceedings of the VLDB Endowment**, VLDB Endowment, v. 15, n. 7, p. 1337–1349, 2022.

KOGIAS, M.; BUGNION, E. Hovercraft: Achieving scalability and fault-tolerance for microsecond-scale datacenter services. In: **Proceedings of the Fifteenth European Conference on Computer Systems**. New York, NY, USA: Association for Computing Machinery, 2020. (EuroSys '20). ISBN 9781450368827. Disponível em: <https://doi.org/10.1145/3342195.3387545>.

KREUTZ, D. et al. Software-defined networking: A comprehensive survey. **Proceedings of the IEEE**, Ieee, v. 103, n. 1, p. 14–76, 2014.

KUNZE, I. et al. **Terminology for Computing in the Network**. [S.l.], 2023. Work in Progress. Disponível em: <https://datatracker.ietf.org/doc/draft-irtf-coinrg-coin-terminology/00/>.

KUNZE, I. et al. **Use Cases for In-Network Computing**. [S.l.], 2023. Work in Progress. Disponível em: <https://datatracker.ietf.org/doc/draft-irtf-coinrg-use-cases/03/>.

KUZNIAR, C. et al. Iot device fingerprinting on commodity switches. In: **NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium**. [S.l.: s.n.], 2022. p. 1–9.

LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. **Commun. ACM**, New York, NY, USA, 1978. ISSN 0001-0782.

LAMPORT, L. The part-time parliament. In: _____. **Concurrency: The Works of Leslie Lamport**. New York, NY, USA: Association for Computing Machinery, 2019. p. 277–317. ISBN 9781450372701. Disponível em: <https://doi.org/10.1145/3335772.3335939>.

LAMPORT, L.; MALKHI, D.; ZHOU, L. Vertical paxos and primary-backup replication. In: **Proceedings of the 28th ACM Symposium on Principles of Distributed Computing**. New York, NY, USA: Association for Computing Machinery, 2009. (PODC '09), p. 312–313. ISBN 9781605583969. Disponível em: <https://doi.org/10.1145/1582716.1582783>.

LAO, C. et al. Atp: In-network aggregation for multi-tenant learning. In: **NSDI**. [S.l.: s.n.], 2021. v. 21, p. 741–761.

LEE, K. S. et al. Globally synchronized time via datacenter networks. In: **Proceedings of the 2016 ACM SIGCOMM Conference**. [S.l.: s.n.], 2016. p. 454–467.

LEIVADEAS, A.; FALKNER, M. A survey on intent based networking. **IEEE Communications Surveys & Tutorials**, IEEE, 2022.

LEWIS, B. et al. Using p4 to enable scalable intents in software defined networks. In: **2018 IEEE 26th International Conference on Network Protocols (ICNP)**. [S.l.: s.n.], 2018. p. 442–443.

LI, H.; LI, J.; KAUFMANN, A. Simbricks: end-to-end network system evaluation with modular simulation. In: **Proceedings of the ACM SIGCOMM 2022 Conference**. [S.l.: s.n.], 2022. p. 380–396.

LI, J.; MICHAEL, E.; PORTS, D. R. Eris: Coordination-free consistent transactions using in-network concurrency control. In: **Proceedings of the 26th Symposium on Operating Systems Principles**. [S.l.: s.n.], 2017. p. 104–120.

LI, J. et al. Just say {NO} to paxos overhead: Replacing consensus with network ordering. In: **12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)**. [S.l.: s.n.], 2016. p. 467–483.

LI, X. et al. Automatic policy generation for {Inter-Service} access control of microservices. In: **30th USENIX Security Symposium (USENIX Security 21)**. [S.l.: s.n.], 2021. p. 3971–3988.

LI, Y. et al. Cetus: Releasing p4 programmers from the chore of trial and error compiling. In: **19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)**. [S.l.: s.n.], 2022. p. 371–385.

LISKOV, B.; COWLING, J. **Viewstamped Replication Revisited**. [S.l.], 2012.

LIU, H. et al. Vulnerabilities and attacks of inter-device coordination in programmable networks. In: IEEE. **2023 IEEE/ACM 31st International Symposium on Quality of Service (IWQoS)**. [S.l.], 2023. p. 01–10.

LIU, M. et al. Incbricks: Toward in-network computation with an in-network cache. In: **Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems**. [S.l.: s.n.], 2017. p. 795–809.

LIU, Z. et al. Mrtom: Mostly reliable totally ordered multicast, a network primitive to offload distributed systems. In: IEEE. **2023 IEEE 43rd International Conference on Distributed Computing Systems (ICDCS)**. [S.l.], 2023. p. 638–648.

MACHADO, C. C. et al. Arkham: an advanced refinement toolkit for handling service level agreements in software-defined networking. **Journal of Network and Computer Applications**, Elsevier, v. 90, p. 1–16, 2017.

MATTERN, F. Virtual time and global states of distributed systems. In: AL., C. M. et (Ed.). **Proc. Workshop on Parallel and Distributed Algorithms**. [S.l.: s.n.], 1989.

MATTERN, F. Efficient algorithms for distributed snapshots and global virtual time approximation. **Journal of Parallel and Distributed Computing**, v. 18, n. 4, p. 423–434, 1993. ISSN 0743-7315. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0743731583710750>.

MAZZURANA, P. **Weather Research and Forecasting Model Workload Evaluation on IBM Cloud**. 2021. Disponível em: <https://www.ibm.com/cloud/blog/weather-research-and-forecasting-model-workload-evaluation-on-ibm-cloud>.

MCKEOWN, N. et al. Openflow: enabling innovation in campus networks. **ACM SIGCOMM computer communication review**, ACM New York, NY, USA, v. 38, n. 2, p. 69–74, 2008.

MIAO, R. et al. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In: **Proceedings of the Conference of the ACM Special Interest Group on Data Communication**. [S.l.: s.n.], 2017. p. 15–28.

MICHEL, O. et al. The programmable data plane: Abstractions, architectures, algorithms, and applications. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 54, n. 4, p. 1–36, 2021.

MITZENMACHER, M.; VASSILVITSKII, S. Algorithms with predictions. **Communications of the ACM**, ACM New York, NY, USA, v. 65, n. 7, p. 33–35, 2022.

MOHAMMAD, A. et al. dist-gem5: Distributed simulation of computer clusters. In: IEEE. **2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)**. [S.l.], 2017. p. 153–162.

MORINAGA, E.; ARAI, E.; WAKAMATSU, H. A basic study on highly distributed production scheduling. In: SPRINGER. **IFIP International Conference on Advances in Production Management Systems**. [S.l.], 2012. p. 638–645.

NAMKUNG, H. et al. Sketchlib: Enabling efficient sketch-based monitoring on programmable switches. In: **19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)**. [S.l.: s.n.], 2022. p. 743–759.

NAMKUNG, H. et al. Sketchovsky: Enabling ensembles of sketches on programmable switches. In: **20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)**. [S.l.: s.n.], 2023. p. 1273–1292.

NETGVT. [S.l.]: GitHub, 2022. <https://github.com/RicardoParizotto/p4app-NetGVT>.

NORONHA, R.; ABU-GHAZALEH, N. B. Using programmable nics for time-warp optimization. In: IEEE. **Proceedings 16th International Parallel and Distributed Processing Symposium**. [S.l.], 2002. p. 8–pp.

NUNES, D. C. et al. Serene: Handling the effects of stragglers in in-network machine learning aggregation. In: IEEE. **NOMS 2023-2023 IEEE/IFIP Network Operations and Management Symposium**. [S.l.], 2023. p. 1–10.

ONGARO, D.; OUSTERHOUT, J. In search of an understandable consensus algorithm. In: **2014 USENIX annual technical conference (USENIX ATC 14)**. [S.l.: s.n.], 2014. p. 305–319.

PAN, H. et al. Libra: In-network gradient aggregation for speeding up distributed sparse deep training. **arXiv preprint arXiv:2205.05243**, 2022.

PAN, M. et al. Nap: Programming data planes with approximate data structures. In: **Proceedings of the 6th on European P4 Workshop**. New York, NY, USA: Association for Computing Machinery, 2023. (EuroP4 '23), p. 33–39. ISBN 9798400704468. Disponível em: <https://doi.org/10.1145/3630047.3630196>.

PANG, L. et al. A survey on intent-driven networks. **IEEE Access**, IEEE, v. 8, p. 22862–22873, 2020.

PARIZOTTO, R. et al. Offloading machine learning to programmable data planes: A systematic survey. **ACM Computing Surveys**, ACM New York, NY, 2023.

PARIZOTTO, R. et al. Netgvt: Offloading global virtual time computation to programmable switches. In: **Proceedings of the Symposium on SDN Research**. New York, NY, USA: Association for Computing Machinery, 2022. (SOSR '22), p. 16–24. ISBN 9781450398923. Disponível em: <https://doi.org/10.1145/3563647.3563648>.

PARK, S. J.; OUSTERHOUT, J. Exploiting commutativity for practical fast replication. In: **16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)**. [S.l.: s.n.], 2019. p. 47–64.

PELKEY, J.; RILEY, G. Distributed simulation with mpi in ns-3. In: **Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques**. [S.l.: s.n.], 2011. p. 410–414.

PFAFF, B. et al. Openflow switch specification, version 1.3. 0. **Open Networking Foundation**, v. 42, 2012.

PORTS, D. R. et al. Designing distributed systems using approximate synchrony in data center networks. In: **12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)**. [S.l.: s.n.], 2015. p. 43–57.

PRAKASH, C. et al. Pga: Using graphs to express and automatically reconcile network policies. In: **Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication**. New York, NY, USA: ACM, 2015. (SIGCOMM '15), p. 29–42. ISBN 978-1-4503-3542-3. Disponível em: <http://doi.acm.org/10.1145/2785956.2787506>.

RIFTADI, M.; KUIPERS, F. P4i/o: Intent-based networking with p4. In: IEEE. **2019 IEEE Conference on Network Softwarization (NetSoft)**. [S.l.], 2019. p. 438–443.

SAHA, B. K. et al. Intent-based networks: An industrial perspective. In: **Proceedings of the 1st International Workshop on Future Industrial Communication Networks**. [S.l.: s.n.], 2018. p. 35–40.

SANVITO, D. et al. Can the network be the ai accelerator? In: **Proceedings of the 2018 Morning Workshop on In-Network Computing**. [S.l.: s.n.], 2018. p. 20–25.

SAPIO, A. et al. In-network computation is a dumb idea whose time has come. In: **Proceedings of the 16th ACM Workshop on Hot Topics in Networks**. New York, NY, USA: Association for Computing Machinery, 2017. (HotNets-XVI), p. 150–156. ISBN 9781450355698. Disponível em: <https://doi.org/10.1145/3152434.3152461>.

SAPIO, A. et al. Scaling distributed machine learning with in-network aggregation. **arXiv preprint arXiv:1903.06701**, 2019.

SAPIO, A. et al. Scaling distributed machine learning with in-network aggregation. In: **18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)**. USENIX Association, 2021. p. 785–808. ISBN 978-1-939133-21-2. Disponível em: <https://www.usenix.org/conference/nsdi21/presentation/sapio>.

SARLI, J. L.; LEONE, H. P.; GUTIÉRREZ, M. De los M. Ontology-based semantic model of supply chains for modeling and simulation in distributed environment. In: IEEE. **2016 Winter Simulation Conference (WSC)**. [S.l.], 2016. p. 1182–1193.

SCHEID, E. J. et al. Inspire: Integrated nfv-based intent refinement environment. In: IEEE. **2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)**. [S.l.], 2017. p. 186–194.

SCHEID, E. J. et al. A controlled natural language to support intent-based blockchain selection. In: IEEE. **2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)**. [S.l.], 2020. p. 1–9.

SENGUPTA, S. et al. Continuous in-network round-trip time monitoring. In: **Proceedings of the ACM SIGCOMM 2022 Conference**. [S.l.: s.n.], 2022. p. 473–485.

SHAH, R. et al. pcube: Primitives for network data plane programming. In: IEEE. **2018 IEEE 26th International Conference on Network Protocols (ICNP)**. [S.l.], 2018. p. 430–435.

SHAHBAZ, M.; FEAMSTER, N. The case for an intermediate representation for programmable data planes. In: **Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research**. [S.l.: s.n.], 2015. p. 1–6.

SHAPIRO, M. et al. Conflict-free replicated data types. In: SPRINGER. **Stabilization, Safety, and Security of Distributed Systems: 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings 13**. [S.l.], 2011. p. 386–400.

SHERRY, J. et al. Rollback-recovery for middleboxes. **SIGCOMM Comput. Commun. Rev.**, Association for Computing Machinery, New York, NY, USA, v. 45, n. 4, p. 227–240, ago. 2015. ISSN 0146-4833. Disponível em: <https://doi.org/10.1145/2829988.2787501>.

SINGH, R. et al. Surviving switch failures in cloud datacenters. **ACM SIGCOMM Computer Communication Review**, ACM New York, NY, USA, v. 51, n. 2, p. 2–9, 2021.

SIRACUSANO, G. et al. Re-architecting traffic analysis with neural network interface cards. In: **19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)**. [S.l.: s.n.], 2022. p. 513–533.

SIVARAMAN, A. et al. Packet transactions: A programming model for data-plane algorithms at hardware speed. **CoRR, vol. abs/1512.05023**, 2015.

SIVARAMAN, V. et al. Heavy-hitter detection entirely in the data plane. In: **Proceedings of the Symposium on SDN Research**. New York, NY, USA: ACM, 2017. (SOSR '17), p. 164–176. ISBN 978-1-4503-4947-5. Disponível em: <http://doi.acm.org/10.1145/3050220.3063772>.

SONCHACK, J. et al. Turboflow: Information rich flow record generation on commodity switches. In: **Proceedings of the Thirteenth EuroSys Conference**. New York, NY, USA: Association for Computing Machinery, 2018. (EuroSys '18). ISBN 9781450355841. Disponível em: <https://doi.org/10.1145/3190508.3190558>.

SONCHACK, J. et al. Lucid: A language for control in the data plane. In: **Proceedings of the 2021 ACM SIGCOMM 2021 Conference**. [S.l.: s.n.], 2021. p. 731–747.

SONG, H. Protocol-oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane. In: **Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking**. New York, NY, USA: Association for Computing Machinery, 2013. (HotSDN '13), p. 127–132. ISBN 9781450321785. Disponível em: <https://doi.org/10.1145/2491185.2491190>.

SONI, H. et al. Composing dataplane programs with μp4. In: **Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication**. [S.l.: s.n.], 2020. p. 329–343.

STERBENZ, J. P. et al. Resilience and survivability in communication networks: Strategies, principles, and survey of disciplines. **Computer networks**, Elsevier, v. 54, n. 8, p. 1245–1265, 2010.

SUN, G. et al. Neobft: Accelerating byzantine fault tolerance using authenticated in-network ordering. In: **Proceedings of the ACM SIGCOMM 2023 Conference**. [S.l.: s.n.], 2023. p. 239–254.

TAJBAKHSH, H. et al. Accelerator-aware in-network load balancing for improved application performance. In: **2022 IFIP Networking Conference (IFIP Networking)**. [S.l.: s.n.], 2022. p. 1–9.

TAJBAKHSH, H. et al. Accelerator-aware in-network load balancing for improved application performance. In: IEEE. **2022 IFIP Networking Conference (IFIP Networking)**. [S.l.], 2022. p. 1–9.

TENNENHOUSE, D. L.; WETHERALL, D. J. Towards an active network architecture. **ACM SIGCOMM Computer Communication Review**, ACM New York, NY, USA, v. 26, n. 2, p. 5–17, 1996.

TIAN, B. et al. Safely and automatically updating in-network acl configurations with intent language. In: **Proceedings of the ACM Special Interest Group on Data Communication**. New York, NY, USA: Association for Computing Machinery, 2019. (SIGCOMM '19), p. 214–226. ISBN 9781450359566. Disponível em: <https://doi.org/10.1145/3341302.3342088>.

TIAN, B. et al. Safely and automatically updating in-network acl configurations with intent language. In: **Proceedings of the ACM Special Interest Group on Data Communication**. [S.l.: s.n.], 2019. p. 214–226.

TOKUSASHI, Y. et al. The case for in-network computing on demand. In: **Proceedings of the Fourteenth EuroSys Conference 2019**. [S.l.: s.n.], 2019. p. 1–16.

TSUZAKI, Y.; OKABE, Y. Reactive configuration updating for intent-based networking. In: IEEE. **2017 International Conference on Information Networking (ICOIN)**. [S.l.], 2017. p. 97–102.

WANG, W.; HAO, Z.; CUI, L. Clusterrr: a record and replay framework for virtual machine cluster. In: **Proceedings of the 18th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments**. [S.l.: s.n.], 2022. p. 31–44.

WETHERALL, D.; TENNENHOUSE, D. Retrospective on" towards an active network architecture". **ACM SIGCOMM Computer Communication Review**, ACM New York, NY, USA, v. 49, n. 5, p. 86–89, 2019.

WETHERALL, D. J.; GUTTAG, J. V.; TENNENHOUSE, D. L. Ants: A toolkit for building and dynamically deploying network protocols. In: IEEE. **1998 IEEE Open Architectures and Network Programming**. [S.l.], 1998. p. 117–129.

WILLIAMS, B. et al. High-performance pdes on manycore clusters. In: **Proceedings of the 2021 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation**. [S.l.: s.n.], 2021. p. 153–164.

WU, D. et al. Accelerated service chaining on a single switch asic. In: **Proceedings of the 18th ACM Workshop on Hot Topics in Networks**. [S.l.: s.n.], 2019. p. 141–149.

XU, W. et al. Clickinc: In-network computing as a service in heterogeneous programmable data-center networks. In: **Proceedings of the ACM SIGCOMM 2023 Conference**. [S.l.: s.n.], 2023. p. 798–815.

YU, Z. et al. Netlock: Fast, centralized lock management using programmable switches. In: **SIGCOMM**. [S.l.: s.n.], 2020.

YU, Z. et al. Netlock: Fast, centralized lock management using programmable switches. In: **Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication**. [S.l.: s.n.], 2020. p. 126–138.

ZENO, L. et al. {SwiSh}: Distributed shared state abstractions for programmable switches. In: **19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)**. [S.l.: s.n.], 2022. p. 171–191.

ZHU, H. et al. {NetVRM}: Virtual register memory for programmable networks. In: **19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)**. [S.l.: s.n.], 2022. p. 155–170.

ZILBERMAN, N. **In-network computing**. 2019. Disponível em: <https://www.sigarch.org/in-network-computing-draft/>.

APPENDIX A — PUBLICATIONS

The results and collaborations made during this research appear in the following publications:

(PARIZOTTO et al., 2022) Parizotto, Ricardo, et al. "NetGVT: offloading global virtual time computation to programmable switches." Proceedings of the Symposium on SDN Research. 2022.

(PARIZOTTO et al., 2023) Parizotto, Ricardo, et al. "Offloading Machine Learning to Programmable Data Planes: A Systematic Survey." ACM Computing Surveys (2023).

(TAJBAKHSH et al., 2022b) Tajbakhsh, Hesam, et al. "Accelerator-aware in-network load balancing for improved application performance." 2022 IFIP Networking Conference (IFIP Networking). IEEE, 2022.

(NUNES et al., 2023) Nunes, Diego Cardoso, et al. "Serene: Handling the Effects of Stragglers in In-Network Machine Learning Aggregation." NOMS 2023 IEEE/IFIP Network Operations and Management Symposium. IEEE, 2023.

(FAVERO et al., 2023) Favero, Nicolle P. et al. "QUIC-Tr4ck: Mitigando Ataques QUIC-Flood usando Planos de Dados Programáveis." XXIII Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais, 2023.

APPENDIX B — RESUMO EXPANDIDO

Switches programáveis são dispositivos de encaminhamento de rede que permitem a execução de funcionalidades personalizadas que funcionam na taxa de linha. Ao contrário de *switches* com funcionalidade fixa, *switches* programáveis oferecem maior versatilidade e potencial de inovação. As vantagens dos *switches* programáveis levaram os pesquisadores a transferir funcionalidades anteriormente realizadas em servidores para a própria rede, resultando no conceito de *In-Network Computing* (INC). Esse conceito permite diversas melhorias no desempenho de sistemas em rede. Ao mover funcionalidade para os dispositivos da rede, é possível reduzir a latência ao processar requisições sem a necessidade de chegar até um servidor, e, ao processar requisições em taxa de linha, aumenta a vazão. No entanto, a transferência de funcionalidades para o plano de dados está sujeita a várias distinções em comparação com como a computação é tradicionalmente realizada em servidores. Os *switches* submetem os desenvolvedores a restrições do hardware, como uma quantidade pequena de operações lógicas e aritméticas por estágios do pipeline, pequena quantidade de memória e limitações de quantidade de leituras e escritas em uma única variável de estado.

Esta tese investiga o paradigma de computação em rede sob três aspectos diferentes que o diferenciam da computação tradicional. Em primeiro lugar, estudamos as *restrições* impostas pelo plano de dados, que podem impactar a transferência de uma funcionalidade de aplicação para o *hardware* do *switch*. Propomos uma terminologia e uma taxonomia de considerações de projeto a serem utilizadas ao transferir uma funcionalidade para o plano de dados dos dispositivos de rede. Apresentamos então um sistema chamado NetGVT, que utiliza as considerações para construir um *design* personalizado para transferir a sincronização de tempo virtual para *switches*. As personalizações realizadas no NetGVT incluem técnicas de compressão para armazenar relógios lógicos, e utilizam decomposições, divisão de trabalho e recirculações para computar o GVT reduzindo a utilização de primitivas custosas. Além disso, mostramos que o NetGVT pode acelerar simulações distribuídas e superar uma solução tradicional que utiliza apenas servidores. Os resultados obtidos usando uma carga de trabalho sintética de simulação demonstram melhorias de aproximadamente 40% em relação à versão em servidores em termos de tempo para completar a simulação. Além disso, as adaptações realizadas suportam

simulações com mais de 128 servidores, o que é suficiente para os simuladores da literatura.

No entanto, uma vez que movemos a computação para o plano de dados, *falhas* na INC podem interromper o sistema e torná-lo indisponível. Portanto, o segundo aspecto que investigamos é o impacto das falhas e os requisitos de consistência necessários para sistemas INC existentes permanecerem corretos após uma falha. Observamos que diferentes sistemas INC podem permanecer corretos depois de uma falha sob diferentes noções de consistência. Em resposta, propomos o RESIST, um sistema que aplica técnicas eficientes e blocos de construção para fornecer tolerância a falhas para a computação em rede que permite customizar os requisitos de consistência. De maneira adicional, advogamos por uma abordagem para tolerância a falhas que opera de maneira assíncrona, complementada por um mecanismo de *log-replay*. Este mecanismo de replay pode ser configurado de diferentes maneiras, atendendo tanto um nível de consistência mais forte, assim também como níveis de consistência mais fracos e menores onerosos. Nós demonstramos que RESIST pode superar falhas de maneira consistente e com um custo baixo para a aplicação. Além disso, mostramos que é possível reduzir os custos durante operações regulares do sistema em comparação com outras abordagens do estado da arte.

Embora observemos que a tolerância a falhas pode ser alcançada sem comprometer os ganhos de desempenho obtidos com a INC, gerenciar a funcionalidade em dispositivos de encaminhamento revela um processo complexo e demorado em comparação com a execução em servidores tradicionais. Configurar uma INC tolerante a falhas exige configurar não somente o dispositivo que hospeda a funcionalidade da INC, mas um conjunto de réplicas e servidores. Ao considerar este cenário distribuído, não somente configurar a INC é suficiente, mas também os elementos que tornam a INC tolerante a falhas. Para entender esse *desafio de configuração*, investigamos métodos para simplificar o gerenciamento de tolerância a falhas na INC usando intenções de alto nível. Propomos um sistema chamado Araucaria, que facilita a especificação de intenções em uma linguagem semelhante à natural, e um processo para refinamento de intenções para instrumentar INCs com a habilidade de tolerância a falhas. Para realizar a instrumentação da INC nós decompomos o RESIST em templates modulares reutilizados para instrumentar as INCs. Após a instrumentação, utilizamos as primitivas obtidas através da tradução da intenção para gerar a configuração de tabelas e registradores conforme o modelo de consistên-

cia exigido e a disponibilidade do sistema. Nós também mostramos que o custo para permitir o refinamento é muito baixo em tempos de tradução e configuração, além de diminuir a complexidade do código da INC, em que o desenvolvedor não precisa incluir manualmente.

Nós demonstramos um exemplo prático das técnicas propostas usando o NetGVT, mostrando a viabilidade e escalabilidade das abordagens propostas nesta tese, tanto em um *testbed* com *hardware* de switches programáveis reais quanto em um emulador de modelo de comportamento. Também mostramos analises de consumo de recursos, mostrando que as técnicas propostas podem ser implementadas implementadas com baixo custo em dispositivos do estado da arte.