Araucaria: Simplifying INC Fault Tolerance with High-Level Intents

Ricardo Parizotto[†], Israat Haque[*], Alberto Schaeffer-Filho[†]
[*]*Dalhousie University*, [†]*UFRGS*

***Abstract*—Network programmability allows modification of *fine-grain* data plane functionality. The performance benefits of data plane programmability have motivated many researchers to offload computation that previously operated only on servers to the network, creating the notion of *in-network computing* (INC). Because failures can occur in the data plane, fault tolerance mechanisms are essential for INC. However, INC operators and developers must manually set fault tolerance requirements using domain knowledge to change the source code. These manually set requirements may take time and lead to errors in case of misconfiguration. In this work, we present Araucaria, a system that aims to simplify the definition and implementation of fault tolerance requirements for INC. The system allows requirements specification using an intent language, which enables the expression of consistency and availability requirements in a constrained natural language. A refinement process translates the intent and incorporates the essential building blocks and configurations into the INC code. We present a prototype of Araucaria and analyze the end-to-end system behavior. Experiments demonstrate that the refinement scales to multiple intents and that the system provides fault tolerance with negligible overhead in failure scenarios.**

I. Introduction

Network programmability has changed how we manage and operate computer networks, providing agility for deploying new network functions in the network. The P4 language [6] has paved the way towards programmability, allowing fine-grain data plane functionality development. This programmability has motivated the notion of *in-network computing* (INC) [36]. INC advocates for offloading functionality traditionally running on servers to programmable network devices. This offloading has 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. Examples of INCs in the literature include functionalities such as aggregation [37], load balancing [5], [43], concurrency control [26], computer vision [15], [41] and IoT security [23].

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 needs to deal with failures in forwarding devices and in the applications running in them, which adds an additional layer of complexity. For example, existing fault tolerant systems [22], [47] employing replication techniques on data plane devices introduce additional complexities. In particular, they require configuring an INC and its replicas while offering consistency among these replicas to offer 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 level [30]. 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) [9], 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 [20], [11], [16], [2], or P4 [4], [25], [34] - an in-depth list is presented in Table I. 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.

TABLE I
Intent frameworks in the literature.

Examples	Purpose	Target
Nile [20], InsPire [38], PGA [33], Arkham [28]	QoS, Control Access	SDN, NFV
Janus [1]	Bandwidth QoS, Temporal Policies	SDN, NFV
JingJing [45]	ACL rules	WAN
Gherkin [11]	Firewall, SFC	SDN
P4-iO [34], [25]	Routing, HH Detection	P4

In this work, we propose Araucaria[1], a system that utilizes the expressiveness benefits of IBN for providing fault tolerance for INC. Araucaria enables operators to specify fault-tolerance requirements using a high-level and declarative language. We introduce a *refinement process* that translates the operator intents into data plane code and configurations to preserve the requirements for INC. This refinement process comprises three main steps: translation, instrumentation, and configuration. Initially, Araucaria *translates* the intent into an intermediary representation, identifying the essential building

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

blocks to achieve the desired level of robustness. Next, Araucaria *instruments* the INC source code with a set of constructs (e.g., the parser, definition of headers, and control flow) to enforce a replication protocol that satisfies the high-level fault tolerance intent. Finally, the system *configures* the data plane according to the translated intent and network topology information. The configuration includes several rules, such as multicast groups, ports, IP addresses, and register entries. These rules ensure replicas preserve the consistency model necessary and recover appropriately from failures. The instrumented code can then be automatically deployed and configured according to the intent priority.

To validate our approach, we have implemented a prototype of Araucaria both for the behavior model of P4 programs (BMv2) and for a Tofino switch ASIC (Edgecore Wedge 100BF-32X). Additionally, we conducted experiments to assess the scalability of our refinement process. As a practical case study, we evaluate Araucaria by analyzing the system behavior when injecting failures, and show that Araucaria can rapidly allow INCs to recover from failures.

Contributions. Overall this paper makes the following contributions:

- Identifies a set of requirements and abstractions to provide fault tolerance for in-network computing.
- Proposes a refinement technique to systematically instrument the INC source code with fault tolerance protocol building blocks, ensuring consistency guarantees.
- Evaluates a use case with an existing INC and analyses results in terms of feasibility and scalability. Results show that Araucaria can recover from failures in less than $0.2s$, while also keeping INC consistency.

II. Background

In this section we discuss the essential background for intent-based networking and programmable data planes.

A. Intent-based Networking (IBN)

Intent-based networking simplifies network management through five essential components: profiling, translation, resolution, activation, and assurance [8]. Users specify their abstract intentions in the profiling phase using high-level representations, such as natural language. Subsequently, the translation component refines these into policies and further into concrete configuration commands [10]. The resolution component solves conflicts, ensuring a conflict-free activation of the intent configuration onto network devices. Despite the reliance on translation and conflict resolution for deploying intents, the dynamic nature of the network may lead to configurations no longer satisfying the initial intent. To address this challenge, an intent assurance component employs mechanisms to identify the discrepancies and refine new configurations to deploy in the network [24].

B. In-Network Computing

In-network computing (INC) has been used to characterize systems with functionality offloaded to the data plane of networking devices [29]. INC relies on programmable switches or SmartNICs that are already deployed on the traffic path [48] to potentially reduce the need for additional specialized equipment, such as accelerators or middleboxes. INC is mainly motivated by the performance benefits it can provide. Those benefits include reducing latency and bandwidth usage, increasing throughput, and improving energy consumption [48].

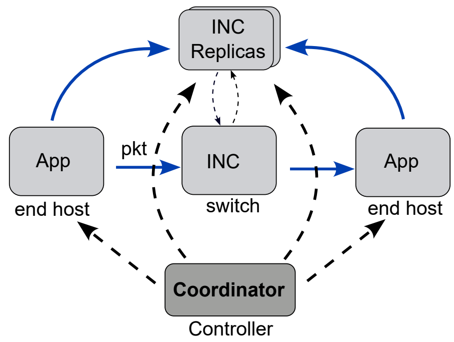

Fig. 1. Fault Tolerant INC system model.

However, these advantages come with a cost: once we offload functionality to the data plane of networking devices, failures in these devices may affect the correctness of the system. Fault tolerance is thus essential but adds additional complexity to configure the INC.

III. INC Fault Tolerance

To better understand fault tolerance requirements, we illustrate a hypothetical INC fault tolerance model in Figure 1. In particular, a switch running an INC intercepts packets from applications running on end hosts. The switch synchronizes the INC state with a pre-configured set of replicas and is able to recover from crashes. The fault tolerance of the INC is measured as reliability (e.g, by means of consistency) and availability of the INC when it is necessary.

Replication for availability. Redundancy is a common approach to ensure the system is available even in case of failures. Multiple INC replicas can keep replicated state, where replication can take different forms: it can be either *synchronous* in a way that resembles the execution of a sequential processor [17], or *asynchronous*, allowing temporary inconsistency between replicas [22]. In case of a switch crash, a coordinator can identify the failure and orchestrate the recovery by collaborating with replicas and end-hosts. The recovery process can also take different forms, from simply *rerouting* new application requests to the replicas, to a more complex process of *replaying* packets from the applications to restore the state of inconsistent replicas.

Consistency notions. Replicas can process packets in an *ordered* or *unordered* manner, impacting the application's consistency. Different types of applications may be correct under different consistency models, such as strong or eventual. On the one hand, *strong consistency* ensures replicas process all the requests in the same total order as the primary INC, while *eventual consistency* does not require strict ordering. In addition, specific applications can achieve a strong notion of consistency without ensuring strict ordering

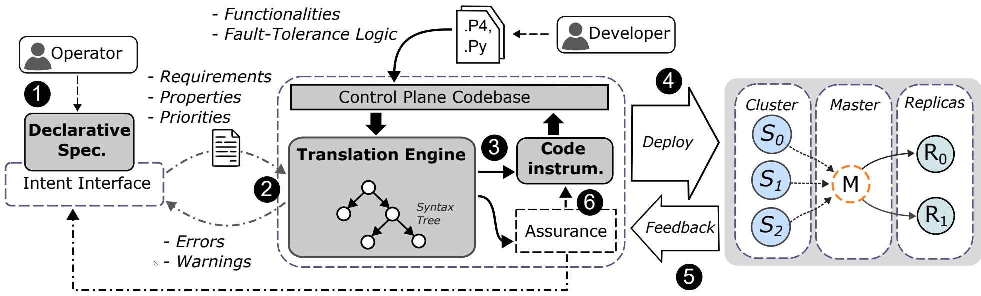

Fig. 2. The high-level architecture of Araucaria.

[47]. These applications follow a system model called *strong eventual consistency* (SEC), in which requests are processed as they arrive, and conflicts are solved automatically using a merge function. The data replicated from these applications are called *conflict-free replicated data types* (CRDT) and often apply to independent or monotonically increasing data [40]. Examples are operations in different keys of a key-value store or adding values to a distributed counter, e.g., counting likes in social media posts.

IV. Araucaria Design

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

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.

A. Overview and workflow

Figure 2 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 *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 refines the data plane configurations (❻), forwarding application packets to a different replica INC.

B. Declarative intent specification

An *intent* is an abstract declaration of what an application or user desires from the network [46]. In Araucaria, each intent is associated with a predicate, including functionality, requirements, and priorities. These predicates state a *property* of an intent. Inspired by [10], [20], 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 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 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* that are 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 [7]. 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 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. The synchronization functionality expects an optional parameter representing the number of processes interacting with the network functionality. The intent requires the INC to tolerate two simultaneous failures while preserving strong consistency.

```
Create intent syncIntent{
    functionality: synchronization [
                        size: 3
                  ]
    availability: tolerates two failures,
    consistency: strong,
}
```

Listing 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 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.

C. Fault tolerance building blocks

Araucaria defines a fault tolerance protocol for recovering INCs from failures. In this protocol (Figure 3), client traffic is processed by the main INC and replicated to a set of switch replicas. If the main INC crashes, a control plane program (*Coordinator*) identifies the failure using timeouts and collects the necessary state information from the replicas and the clients. The Coordinator 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 [32].

The refinement process implemented by Araucaria merges the source code of pre-existing building blocks that enforce the fault tolerance protocol with the INC source code. Dynamically, the network operator can select one of the recovery strategies instrumented into the INC according to an application's consistency requirement, i.e., strong or weak.

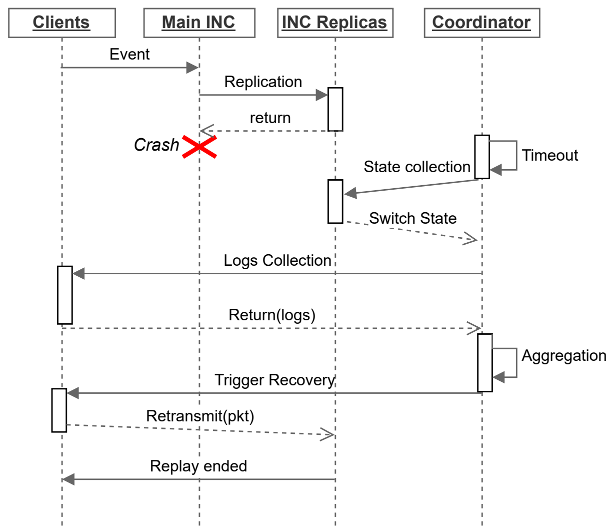

Fig. 3. Sequence diagram for Araucaria Recovery protocol in accordance with Figure 1.

Reusable building blocks. Figure 4 provides a comprehensive overview of the underlying structure of a P4 program that has been instrumented with Araucaria to support fault tolerance employing a set of four standard building blocks: *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.

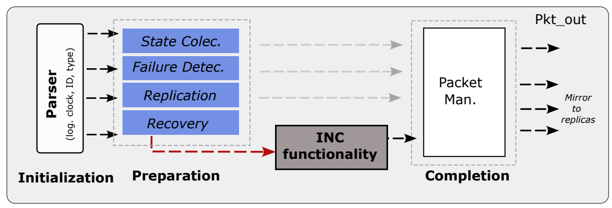

Fig. 4. Structure of an INC instrumented with fault tolerance building blocks.

These building blocks are implemented as a set of P4 *templates* that are merged with the INC source code. Templates include three blocks: *initialization*, *preparation*, and *completion*.

- **Initialization.** The initialization template includes per-packet variables, such as custom metadata, a new header and struct, and a parser state. The header has information to identify servers and message types (e.g., recovery, collection) and to ensure linearizability through monotonically increasing logical clocks. The parser state can initialize these variables upon the arrival of a packet.
- **Preparation.** This block includes a set of variables to implement the building blocks we mentioned earlier. The Preparation template precedes the INC functionality

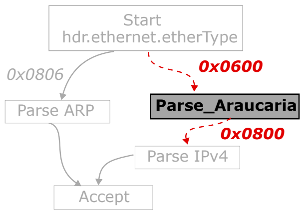

Fig. 5. Fault tolerance parser.

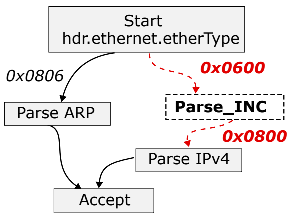

Fig. 6. INC parser.

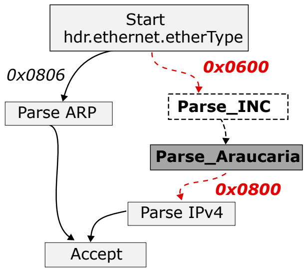

Fig. 7. Instrumented parser.

in the pipeline and consists of code that prepares the packet for INC processing or filtering. Filtering is essential in several cases, such as when the packet being handled is an acknowledgment for replication or a message for failure detection. In these cases, the INC itself should refrain from processing these packets.

- **Completion.** This block includes packet management mechanisms capable of applying multicast tables, keeping storage for packet losses, and changing header arguments. This block should be included after the INC functionality to preserve headers and packet metadata for correct INC processing.

D. 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. We start instrumenting the parser after merging the INC headers, `structs`, and metadata definitions.

Step #2: Parser instrumentation. Our parser instrumentation leverages a modular design that decouples the Araucaria parser state from traditional protocols 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 presents the parser of Araucaria, ignoring the states for standard protocols. Figure 6 shows 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 7.

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 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.

E. 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

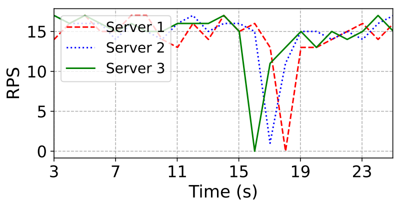

Fig. 8. Analysing the behavior after failure.

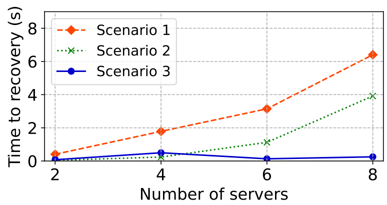

Fig. 9. Recovery on different amounts of servers.

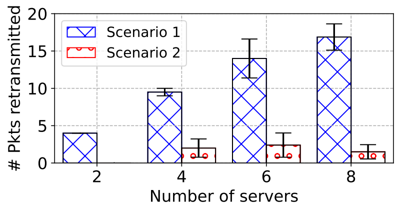

Fig. 10. Number of packets to replay.

IPs. This list enables orchestrating the recovery by triggering route and interface changes after a failure.

V. 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 increasingly amounts of intents.

A. Experimental settings

Implementation. The Araucaria coordinator 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`. The compiler is implemented using PLY (Python Lex-Yacc), to build the Araucaria language (∼120 LoC), while also using native Linux commands to manage the repository of functionalities and instantiate experiments automatically. We built the switch building blocks using P4-16 both for V1Model (∼350 LoC), and also 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.

Testbed Setup. Our prototype for the V1Model is evaluated in a Linux virtual machine with an Intel® i5-10210U CPU @ 1.60GHz using 2 dedicated cores, 2 GB of memory, and Ubuntu 20.04 LTS. To evaluate the functionality of the system we use BMv2, a behaviour model for P4 programs. The network is emulated using mininet. The topology includes 45 hosts connected to 2 switches, one acting as a replica and the other as the main.

We evaluate our TNA PoC of Araucaria in a Tofino testbed. The experiments were conducted in a setup 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).

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). To understand the scalability of multiple recovery strategies, we investigate the *latency* to recover from a failure and the number of packet retransmissions using different intent configurations. These measurements enable us to understand the trade-offs of different fault tolerance techniques for INC. Finally we measure the system overhead in terms of rules and resources used in the switches.

B. 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 IV-B, Listing 1, discuss the refinement process, and check the effectiveness of fault tolerance. Our use case deploys NetGVT [31], an INC capable of synchronizing logical clocks from distributed systems running on servers. The nodes in the system exchange event messages with timestamps, which are intercepted and synchronized by the INC. Next, we demonstrate the step-by-step process of instrumenting this INC using Araucaria.

Refinement. Listing 2 presents a fragment of configurations and commands created by the Araucaria refinement process. The '`syncnIntent`' (Listing 1) is refined into rules in JSON that instantiate two replicas and create a multicast 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).

```
//multicast rules created for replication
"multicast_group_entries" : [{"
    multicast_group_id" : 1, "replicas" :
    [{"egress_port" : 1, "instance" : 1}]

//clone port for buffering
mirroring_add 500 3

//Writing specific consistency model
register_write consistency_model 0 1
```

Listing 2. Fragment of commands and configurations created.

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

Figure 8 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 coordinator collects the devices' status to achieve consistency, the servers are notified about the failure and start the recovery by switching their communication to a different replica (∼ 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).

C. Analysing recovery configurations

To better understand the scalability of Araucaria, 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 9 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 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 the highest NetGVT's clock value from each server, eliminating the need to retransmit all packets. Operations to be 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 10 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 in this recovery strategy the dependency violations are solved by the merge function. 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.

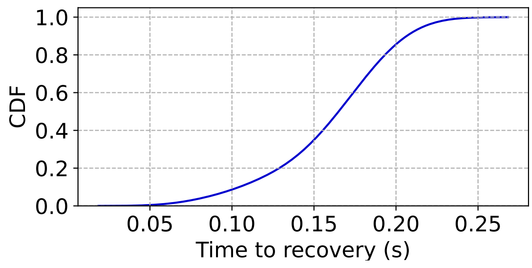

Fig. 11. Failover Time.

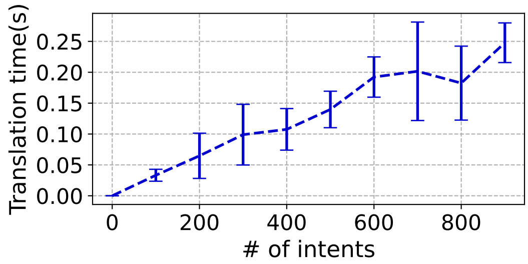

Fig. 12. Time to translate intents.

D. Hardware micro-benchmark

To understand the overhead of Araucaria in real hardware, we selected the optimal configuration scenario for NetGVT (Scenario 3) and conducted experiments to measure the recovery latency. Specifically, in this experiment, we used two Tofino ASICs running Araucaria and two servers exchanging events processed by the switches. We bring down the main switch and measure the time taken for servers to resume their operation after the recovery. The results in Figure 11 show that the system requires, on average, 0.16 seconds to recover from a failure. The standard deviation is 0.03 sec. These results show that Araucaria can rapidly recover from failures while maintaining a strong notion of consistency.

E. Scalability

To show the scalability of the compiler, we generated a variable amount of intents in the Araucaria language. In Figure 12, we show the time it takes to complete translation for a varying amount of intents using batches of *multiple* sizes. These experiments were executed in a Linux virtual machine with an Intel® i5-10210U CPU @ 1.60GHz using 2 dedicated cores, 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.

To understand the impact on resource usage, Table II presents the number of P4 primitives generated by the refinement process. We focus on these primitives in our evaluation

TABLE II
RULES AND PRIMITIVES USED BY ARAUCARIA.

Primitives/Rules	Usage
Match+Action Entries	49
CloneE2E	4
Multicast Groups	1
Recirculation	1

because they can be generalized to multiple targets. Considering our emulated setup, which includes 45 hosts connected to 2 switches, Araucaria has used 49 match+action entries to ensure proper communication between the coordinator and servers. The clone primitives were employed four times in the source code, enabling the creation of packet copies for acknowledgments, multicast, buffering, and retransmissions. A single recirculation primitive was used for buffering packets. These results indicate that only a small amount of P4 primitives are used by Araucaria.

VI. Related Work

Intents. Various platforms have been explored for intent management, including network function virtualization [38], software-defined networking [20], [11], [16], [2], and industrial networks [35]. Researchers have been investigating techniques for managing Access Control Lists (ACL) [44], [27] and Quality of Service (QoS) using intents, along with exploring diverse abstractions to express intents. These abstractions include policy graphs [33], natural language [20], graphical user interfaces [13], and constrained natural language grammars [39]. Recent works also used high-level intents to create *match+action* entries for P4 programs [4], [25]. P4I/O [34] facilitates P4 adoption by using a language to express policies for switches, and merges different source files using reusable templates to dynamically upgrade the switch configuration. However, the authors only demonstrated the concept for deploying heavy hitter detection. Araucaria focuses on other domain by extending INC functionality with specific abstractions for fault tolerance.

High-level data structures. An orthogonal research field focuses on bringing higher-level abstractions to P4. Examples of high-level abstractions include data structure elasticity [18], loops [3], modularity [12] composability [42], and heterogeneity [14]. Although these efforts can simplify programming, they do not support functionalities to express intents.

INC Fault tolerance. Fault tolerance is investigated in [21], [19], focusing on applications like aggregation, key-value store, and network functions. Notably, RedPlane [22] proposes techniques for making switch applications fault tolerant. RedPlane synchronizes the switch with state storage to provide strong and eventual consistency. Swish [47] provides abstractions that can enable distributed network functions on programmable switches by replicating the state and operations between multiple devices. However, none of the previous approaches support the specification of requirements at the level of intents.

VII. Conclusion

In this work, we presented Araucaria, a system to provide fault tolerance requirements expressed as intents for INCs. The system allows intents to be specified in a constrained natural language. Subsequently, a refinement mechanism instruments the INC code for ensuring fault tolerance. We have implemented a prototype of Araucaria both in an emulated setup based on BMv2 and on a Tofino testbed, and analyzed our translation and refinement processes. As future work, we plan to address other kinds of failures, including adversarial attacks, bugs and malfunctioning. Simplifying intent management using large language models is also an interesting perspective. Furthermore, we also plan to investigate how eBPF can be used to reduce the overhead of recovery.

References

[1] Anubhavnidhi Abhashkumar, Joon-Myung Kang, Sujata Banerjee, Aditya Akella, Ying Zhang, and Wenfei Wu. Supporting diverse dynamic intent-based policies using janus. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '17, page 296–309, New York, NY, USA, 2017. Association for Computing Machinery.

[2] Shiyam Alalmaei, Yehia Elkhatib, Mehdi Bezahaf, Matthew Broadbent, and Nicholas Race. Sdn heading north: Towards a declarative intent-based northbound interface. In *2020 16th International Conference on Network and Service Management (CNSM)*, pages 1–5. IEEE, 2020.

[3] Albert Gran Alcoz, Coralie Busse-Grawitz, Eric Marty, and Laurent Vanbever. Reducing p4 language's voluminosity using higher-level constructs. In *Proceedings of the 5th International Workshop on P4 in Europe*, pages 19–25, 2022.

[4] Antonino Angi, Alessio Sacco, Flavio Esposito, Guido Marchetto, and Alexander Clemm. Nlp4: An architecture for intent-driven data plane programmability. In *2022 IEEE 8th International Conference on Network Softwarization (NetSoft)*, pages 25–30. IEEE, 2022.

[5] Tom Barbette, Erfan Wu, Dejan Kostić, Gerald Q Maguire, Panagiotis Papadimitratos, and Marco Chiesa. Cheetah: A high-speed programmable load-balancer framework with guaranteed per-connection-consistency. *IEEE/ACM Transactions on Networking*, 30(1):354–367, 2021.

[6] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[7] Alvin Cheung, Natacha Crooks, Joseph M Hellerstein, and Matthew Milano. New directions in cloud programming. *In 11th Conference on Innovative Data Systems Research (CIDR' 21)*, 2021.

[8] Alexander Clemm, Laurent Ciavaglia, Lisandro Zambenedetti Granville, and Jeff Tantsura. Intent-Based Networking - Concepts and Definitions. RFC 9315, October 2022.

[9] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The ponder policy specification language. In *International Workshop on Policies for Distributed Systems and Networks*, pages 18–38. Springer, 2001.

[10] Yehia Elkhatib, Geoff Coulson, and Gareth Tyson. Charting an intent driven network. In *2017 13th International Conference on Network and Service Management (CNSM)*, pages 1–5. IEEE, 2017.

[11] F. Esposito, J. Wang, C. Contoli, G. Davoli, W. Cerroni, and F. Callegati. 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)*, pages 1–6, Nov 2018.

[12] Ali Fattaholmanan, Mario Baldi, Antonio Carzaniga, and Robert Soulé. P4 weaver: Supporting modular and incremental programming in p4. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*, pages 54–65, 2021.

[13] Mauro Femminella, Matteo Pergolesi, and Gianluca Reali. Simplification of the design, deployment, and testing of 5g vertical services. In *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*, pages 1–7. IEEE, 2020.

[14] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. 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*, pages 435–450, 2020.

[15] René Glebke, Johannes Krude, Ike Kunze, Jan Rüth, Felix Senger, and Klaus Wehrle. Towards executing computer vision functionality on programmable network devices. In *Proceedings of the 1st ACM CoNEXT Workshop on Emerging in-Network Computing Paradigms*, pages 15–20, 2019.

[16] Victor Heorhiadi, Sanjay Chandrasekaran, Michael K. Reiter, and Vyas Sekar. Intent-driven composition of resource-management sdn applications. CoNEXT '18, page 86–97, New York, NY, USA, 2018. Association for Computing Machinery.

[17] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[18] Mary Hogan, Shir Landau-Feibish, Mina Tahmasbi Arashloo, Jennifer Rexford, and David Walker. Modular switch programming under resource constraints. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 193–207, 2022.

[19] Hongyi Huang and Wenfei Wu. Hypersfp: Fault-tolerant service function chain provision on programmable switches in data centers. In *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*, pages 1–9. IEEE, 2022.

[20] Arthur Selle Jacobs, Ricardo José Pfitscher, Ronaldo Alves Ferreira, and Lisandro Zambenedetti Granville. Refining network intents for self-driving networks. In *Proceedings of the Afternoon Workshop on Self-Driving Networks*, SelfDN 2018, page 15–21, New York, NY, USA, 2018. Association for Computing Machinery.

[21] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 35–49, 2018.

[22] Daehyeok Kim, Jacob Nelson, Dan RK Ports, Vyas Sekar, and Srinivasan Seshan. Redplane: enabling fault-tolerant stateful in-switch applications. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 223–244, 2021.

[23] Carson Kuzniar, Miguel Neves, Vladimir Gurevich, and Israat Haque. Iot device fingerprinting on commodity switches. In *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*, pages 1–9. IEEE, 2022.

[24] Aris Leivadeas and Matthias Falkner. A survey on intent based networking. *IEEE Communications Surveys & Tutorials*, 2022.

[25] B. Lewis, L. Fawcett, M. Broadbent, and N. Race. Using p4 to enable scalable intents in software defined networks. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, pages 442–443, 2018.

[26] Jialin Li, Ellis Michael, and Dan RK Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 104–120, 2017.

[27] Xing Li, Yan Chen, Zhiqiang Lin, Xiao Wang, and Jim Hao Chen. Automatic policy generation for {Inter-Service} access control of microservices. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3971–3988, 2021.

[28] Cristian Cleder Machado, Juliano Araujo Wickboldt, Lisandro Zambenedetti Granville, and Alberto Schaeffer-Filho. Arkham: an advanced refinement toolkit for handling service level agreements in software-defined networking. *Journal of Network and Computer Applications*, 90:1–16, 2017.

[29] Oliver Michel, Roberto Bifulco, Gábor Rétvári, and Stefan Schmid. The programmable data plane: Abstractions, architectures, algorithms, and applications. *ACM Computing Surveys (CSUR)*, 54(4):1–36, 2021.

[30] Lei Pang, Chungang Yang, Danyang Chen, Yanbo Song, and Mohsen Guizani. A survey on intent-driven networks. *IEEE Access*, 8:22862–22873, 2020.

[31] Ricardo Parizotto, Braulio Mello, Israat Haque, and Alberto Schaeffer-Filho. Netgvt: offloading global virtual time computation to programmable switches. In *Proceedings of the Symposium on SDN Research*, pages 16–24, 2022.

[32] Seo Jin Park and John Ousterhout. Exploiting commutativity for practical fast replication. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 47–64, 2019.

[33] Chaithan Prakash, Jeongkeun Lee, Yoshio Turner, Joon-Myung Kang, Aditya Akella, Sujata Banerjee, Charles Clark, Yadi Ma, Puneet Sharma, and Ying Zhang. Pga: Using graphs to express and automatically reconcile network policies. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 29–42, New York, NY, USA, 2015. ACM.

[34] Mohammad Riftadi and Fernando Kuipers. P4i/o: Intent-based networking with p4. In *2019 IEEE Conference on Network Softwarization (NetSoft)*, pages 438–443. IEEE, 2019.

[35] Barun Kumar Saha, Deepaknath Tandur, Luca Haab, and Lukasz Podleski. Intent-based networks: An industrial perspective. In *Proceedings of the 1st International Workshop on Future Industrial Communication Networks*, pages 35–40, 2018.

[36] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. In-network computation is a dumb idea whose time has come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets-XVI, page 150–156, New York, NY, USA, 2017. Association for Computing Machinery.

[37] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling distributed machine learning with in-network aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 785–808. USENIX Association, April 2021.

[38] Eder J Scheid, Cristian C Machado, Muriel F Franco, Ricardo L dos Santos, Ricardo P Pfitscher, Alberto E Schaeffer-Filho, and Lisandro Z Granville. Inspire: Integrated nfv-based intent refinement environment. In *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 186–194. IEEE, 2017.

[39] Eder J Scheid, Patrick Widmer, Bruno B Rodrigues, Muriel F Franco, and Burkhard Stiller. A controlled natural language to support intent-based blockchain selection. In *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–9. IEEE, 2020.

[40] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems: 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings 13*, pages 386–400. Springer, 2011.

[41] Hisham Siddique, Miguel Neves, Carson Kuzniar, and Israat Haque. Towards network-accelerated ml-based distributed computer vision systems. In *2021 IEEE 27th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 122–129, 2021.

[42] Hardik Soni, Myriana Rifai, Praveen Kumar, Ryan Doenges, and Nate Foster. 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*, pages 329–343, 2020.

[43] Hesam Tajbakhsh, Ricardo Parizotto, Miguel Neves, Alberto Schaeffer-Filho, and Israat Haque. Accelerator-aware in-network load balancing for improved application performance. In *2022 IFIP Networking Conference (IFIP Networking)*, pages 1–9. IEEE, 2022.

[44] Bingchuan Tian, Xinyi Zhang, Ennan Zhai, Hongqiang Harry Liu, Qiaobo Ye, Chunsheng Wang, Xin Wu, Zhiming Ji, Yihong Sang, Ming Zhang, et al. Safely and automatically updating in-network acl configurations with intent language. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 214–226. 2019.

[45] Bingchuan Tian, Xinyi Zhang, Ennan Zhai, Hongqiang Harry Liu, Qiaobo Ye, Chunsheng Wang, Xin Wu, Zhiming Ji, Yihong Sang, Ming Zhang, Da Yu, Chen Tian, Haitao Zheng, and Ben Y. Zhao. Safely and automatically updating in-network acl configurations with intent language. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 214–226, New York, NY, USA, 2019. Association for Computing Machinery.

[46] Yoshiharu Tsuzaki and Yasuo Okabe. Reactive configuration updating for intent-based networking. In *2017 International Conference on Information Networking (ICOIN)*, pages 97–102. IEEE, 2017.

[47] Lior Zeno, Dan RK Ports, Jacob Nelson, Daehyeok Kim, Shir Landau-Feibish, Idit Keidar, Arik Rinberg, Alon Rashelbach, Igor De-Paula, and Mark Silberstein. {SwiSh}: Distributed shared state abstractions for programmable switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 171–191, 2022.

[48] Noa Zilberman. In-network computing, Apr 2019. https://www.sigarch.org/in-network-computing-draft [Accessed: Feb 22 2024].