FASTR: Fast Resilience for Stateful Programmable Data Planes

Chenxing Ji and Fernando Kuipers
Delft University of Technology
{c.ji, f.a.kuipers}@tudelft.nl

***Abstract*—Programmable data plane devices have enabled various in-network applications that rely on locally stored state for delivering low-latency and high-throughput services. However, these applications are susceptible to network failures, which can disrupt state access and network functionality. Timely and reliable failure detection is therefore a critical component of a stateful data plane. In this paper, we propose a data plane framework, FASTR, that enables microsecond-scale fast failure detection between directly connected switches. FASTR can achieve sub-10μs detection latency by implementing a heartbeat mechanism in the data plane. In addition, FASTR also incorporates traffic-awareness to reduce overhead and priority queuing to avoid false alarms. We validate FASTR with hardware experiments, demonstrating that it can consistently detect failures within 10 μs using a 4 μs interval while remaining robust to network congestion.**

***Index Terms*—Programmable Data Plane, P4, Resilience, Failure Detection.**

I. Introduction

Over the past decade, the evolution towards network programmability has transformed the field of networking. Moreover, programmable network devices [1] were developed, offering programmability to both the control and data planes. These devices have the ability to maintain local states, introducing the capabilities of stateful network functions directly within the data plane. Such integration of stateful capabilities directly into the data plane has led to a breadth of novel in-network applications, including load balancers, cache systems, and monitoring systems [2]–[5]. These applications rely on timely and accurate locally stored states to provide correct and advanced services.

However, such reliance on local states also makes stateful in-network applications vulnerable to failures. A sudden link or device failure can cause inaccuracy or even a loss of local states, degrading application correctness and availability. While some Network Functions (NFs) (e.g., sketch-based [6]) tolerate approximate states, many others, such as NATs and firewalls, require highly accurate states to function correctly. Moreover, for high-performance stateful data planes operating at Tbps, existing works on millisecond-level failure detection [7], [8] could impact Gigabytes of traffic next to the inconsistent states.

As a result, microsecond-level failure detection becomes not only a performance optimization for the network but also a critical requirement for ensuring the availability and correctness of the stateful data plane. This is particularly necessary for modern data center networks where link and device failures are frequent [9].

In the context of network service management, fast failure detection in the data plane can (1) minimize recovery latency to reduce service disruption, (2) reduce control-plane dependency to enhance robustness, and (3) improve service quality by preventing false positive alarms.

In this paper, we propose FAST Resilience (FASTR), a data-plane framework for ultra-low-latency detection and mitigation of neighboring switch failures. FASTR enables sub-10 μs failure detection directly within the data plane, eliminating the involvement of the control plane in the detection logic. While FASTR supports in-data-plane backup rerouting for resilience, its core novelty lies in its ability to detect failures at a microsecond timescale within the data plane. FASTR achieves this by combining recirculated probe clones with data-plane logic to implement a microsecond-scale heartbeat mechanism. It also incorporates a traffic-aware mechanism to reduce overhead and employs priority scheduling to eliminate false positives caused by congestion. These components, when combined, allow FASTR to minimize detection latency, mitigate overhead, and maintain robustness. We have open-sourced our implementation of FASTR[1].

II. Background and Motivations

Programmable data planes have enabled NFs such as load-balancers [3], [11], security applications [12], and telemetry systems [13], [14], to provide more sophisticated and efficient network functionalities. While such an advancement enables rich data-plane processing logic, it also introduces new challenges for maintaining the service in the event of failures.

A. The Need for Fast Failure Detection

Failures are frequent in data centers [15], making rapid failure detection essential to minimize service interruptions. Traditional protocols, such as BFD [10], are control-plane-driven and typically detect failures on the order of hundreds of milliseconds. Such a detection interval is too slow for latency-sensitive and stateful NFs such as NATs and firewalls, which rely on timely and consistent states.

To address this gap, research has explored in-data-plane detection using programmable data planes. SPIDER [7] implements a finite-state machine inside the data plane to detect link

[1]Available at: https://gitlab.tudelft.nl/lois/fastr_camera_ready.

978-3-903176-75-1

TABLE I: Comparison of failure detection frameworks.

Framework	Failure Detection Latency	False Alarm Resilience	Control Plane Free	Detection Mechanism
BFD [10]	∼100ms	✓	✗	Timer-based
SPIDER [7]	∼10-50ms	✗	∼	Probe-based
vFFR [8]	∼500μs	✓	✗	Event-based
FASTR (this work)	$\mathbf{< 10\,\mu s}$	✓	∼	Probe-based

failures, achieving detection latencies of tens of milliseconds. However, due to the lack of native timers within the data plane, SPIDER evaluates timeouts only when packets arrive. Such a mechanism may be delayed during periods of low traffic. Additionally, it does not provide resilience against false alarms caused by transient congestion.

vFFR [8], on the other hand, achieves sub-millisecond detection latency by reacting to hardware link status changes. However, it relies on the local control plane to initiate failover, which introduces additional latency in the react path. Moreover, vFFR does not distinguish between link and device failures. Table I provides a comparison between FASTR and other failure detection frameworks.

B. Link and Device Failures

For a stateful data plane, link failures and device failures differ significantly in their impact and mitigation strategies. Link failures involve a single connection but leave the local state intact, allowing traffic to be rerouted with minimal disruption. Device failures, however, involve a total loss of the locally maintained states and pipeline functionality. MRC [16] distinguished the two and applied a different mechanism for the mitigation through pre-planning.

C. Programmable Switches

Today's programmable switches, e.g., Intel Tofino [17] and SmartNICs, offer line-rate packet processing with stateful abstractions, such as registers, counters, and meters. These components support advanced in-network applications, yet come with limitations on the amount of hardware resources. Moreover, the lack of native timers within the data plane makes time-sensitive operations such as fast failure detection difficult to achieve.

III. FASTR Framework

The increase of reliance on stateful NFs in programmable data planes highlights the need for faster failure-detection methods. In this section, we introduce FASTR, a framework designed to achieve microsecond-level fast failure detection between directly neighboring switches, using data-plane mechanisms. We begin by elaborating on the design goals, followed by an overview and a detailed explanation of each phase of the framework.

A. Goals

The primary design goal of FASTR is to enable fast and reliable detection of failures. To improve the overall service provided by the network, the failure detection framework must minimize detection latency while incurring minimal overhead and false positives.

Fast Response FASTR detects failures at the microsecond level to minimize disruptions, which is critical for latency-sensitive applications of both current and future networks.

Low Overhead FASTR must operate with minimal bandwidth overhead and minimal hardware resource consumption. This also involves reduced control-plane involvement after probe initialization to minimize the detection overhead.

Robustness FASTR targets the detection of failures between neighboring switches and must be robust against false alarms caused by congestion-induced packet loss.

B. Overall Framework Design

FASTR comprises three phases: (1) setup, (2) detection phase, and (3) failure mitigation. The core of FASTR lies in the detection phase, which employs a heartbeat mechanism entirely within the data plane for rapid failure detection. Additionally, it comes with two mechanisms to enable low overhead and robustness.

In the setup phase, FASTR uses the network-operator-supplied topology to pre-compute backup paths and configure probe routes. It also leverages switch timestamps to accurately measure the round-trip time between each switch pair and computes the timeout for the heartbeat mechanism. During the detection phase, periodic probe packets are initiated by local controllers but fully processed in the data plane. Detection logic, including probe tracking, timeout, and detection trigger, is implemented entirely within the data plane, without the involvement of a runtime control plane. The detection phase of FASTR suppresses probe processing when data traffic is arriving from the target switch. FASTR also enhances reliability by eliminating false alarms caused by transient congestion using prioritized probe packets. The failure mitigation phase uses pre-installed coordination logic to distinguish link and device failures and reroutes traffic accordingly.

C. Setup Phase

In the setup phase, FASTR prepares by computing per-path timeout thresholds. The local controller of a source switch initiates a sequence of custom probes, each carrying fields for

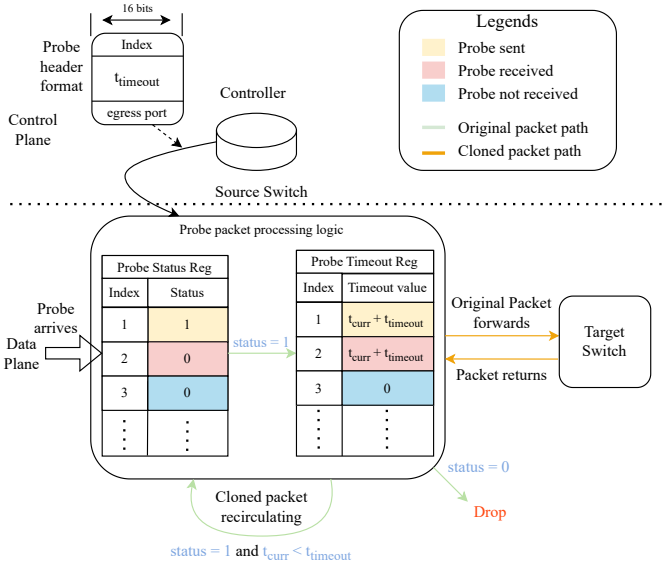


Fig. 1: Detailed probe-mechanism data-plane logic in the detection phase.

two timestamps ts_1 and ts_2. The backup route pre-planning part is discussed in Section III-E.

To ensure robustness under various traffic conditions, FASTR collects Round-Trip Time (RTT) measurements from 50 such probes. Then it computes the timeout value using Equation 1, a formula inspired by the retransmission timeout of TCP.

$$t_{\text{timeout}} = \text{SRTT} + 4 * RTT_{\text{variance}} \tag{1}$$

Here, SRTT represents the smoothed RTT value, and RTT_{variance} is the variance in the observed RTT.

D. Detection Phase

In the detection phase, FASTR performs a heartbeat-based low-latency failure detection using periodically transmitted probe packets. It is worth noting that the detection logic is implemented within the data plane, allowing programmable switches to monitor path health. However, probe packets still need to be periodically sent via a local controller to initiate each detection phase.

To enable the heartbeat mechanism, FASTR leverages replication and recirculation of the programmable switches. This approach simulates a timer-based heartbeat mechanism entirely within the data plane. Although replication and recirculation introduce overhead in the pipeline, FASTR mitigates this with a traffic-aware mechanism with which probes are only processed when no traffic comes from the associated target switch. Figure 1 illustrates the core detection logic and Algorithm 1 describes the detailed packet process inside the pipeline.

Probe Header Design We have designed a special header format, called probe header, for FASTR. The probe packet header contains three different parts: (1) index identifier, (2) timeout value, and (3) `egress_port` number. The index identifier is used inside the data plane to access the correct register index for retrieving stored data. The timeout value indicates the amount of time the data plane should wait for the probe packet to be received. The egress port number is used to correctly forward the probe packet to the outgoing link so that the next switch can receive and return the probe packet.

Detection Time Analysis The detection-time threshold is based on the following three main factors. Let us define (1) t_{interval} to be the period for the local controller to send probe packets, (2) t_{timeout} to be the timeout value defined for the probe packet, and (3) t_{RTT} the round-trip time of probe packets between the source and destination switch. Let t_{recirc} be the time it takes for the probe packet to be recirculated. We also denote t_{source} and t_{dest} as the processing time of the packet in the source and destination switches, respectively. t_{link} is the time the probe packet spends on the physical link.

Based on the different values chosen, we have the following three scenarios:

1) $t_{\text{RTT}} < t_{\text{timeout}} < t_{\text{interval}}$
2) $t_{\text{RTT}} < t_{\text{interval}} < t_{\text{timeout}}$
3) $t_{\text{interval}} < t_{\text{RTT}} < t_{\text{timeout}}$

Case 1 & 2): The worst-case detection time occurs when a failure occurs immediately after the last probe arrives and just after its recirculated copy is dropped, as shown in Equation 2. The best-case detection time occurs when a failure immediately follows the transmission of a probe packet by the source switch, as shown in Equation 3.

$$t_{\text{detection}} = (t_{\text{interval}} - t_{\text{RTT}}) + t_{\text{timeout}} + (1-\epsilon)\, t_{\text{recirc}} \tag{2}$$

$$t_{\text{detection}} = t_{\text{timeout}} - (1-\epsilon)\, t_{\text{RTT}} + \epsilon\, t_{\text{recirc}} \tag{3}$$

Case 3): It can be guaranteed that whenever a failure happens, the following probe packet has already arrived inside the data plane. Thus, the detection time can be expressed as shown in Equation 4:

$$t_{\text{detection}} = t_{\text{timeout}} - (t_{\text{RTT}} - t_{\text{interval}}) + (1 - [0,1]) \cdot t_{\text{recirc}} \tag{4}$$

1) Control Plane: The control plane is not directly involved in the detection procedure and hence does not add to detection latency. The primary role of the control plane in the detection phase is to prepare and send the probe packets at a designated rate.

Avoiding Data-Plane Entry Overwriting To ensure the correct timeout behavior in the data plane, each probe packet carries a 16-bit index to access and update the corresponding status register entry in the data plane. The control plane handles the allocation of registers inside the data plane and computing the correct index for each probe.

FASTR pre-allocates the exact number of required register entries based on four variables: (1) t_{RTT}, (2) the number of adjacent switches, (3) $t_{interval}$, and (4) $t_{timeout}$.

During the failure detection phase, FASTR employs a round-robin scheduling strategy to assign the correct index of the probe packets. This ensures that each entry will not be overwritten until a timeout decision can be safely made. We show below that FASTR can guarantee that no entry will be overwritten, ensuring no failure signals are missed. As

Algorithm 1 Data-Plane Logic of Probe Packets.

Input: $p \leftarrow$ **hdr.probe, ig_intr_md, CPUPort**
Metadata: $C_{\text{pkt\_cnt}} \leftarrow 0, t_{timeout} \leftarrow 0$

1: **if** ig_intr_md.ingress_port == CPUPort **then**
2: $C_{\text{pkt\_cnt}} \leftarrow$ PktCount(`p.eg_port`)
3: $D_{\text{pkt\_cnt}} \leftarrow$ StoreAndDiff(`p.eg_port`)
4: **if** $D_{\text{pkt\_cnt}} == 0$ **then**
5: $t_{\text{now}} \leftarrow$ gTimestamp;
6: $t_{timeout} \leftarrow t_{now} +$ `p.timeout`
7: StatusRegInc(`p.index`); ▷ Set status
8: TimeoutRegSet(`p.index`) ; ▷ Set timeout
9: set_mirror_type(), set_qid();
10: `egress_port` $\leftarrow$ `p.eg_port`;
11: **else**
12: drop(); ▷ Packets received between interval
13: **end if**
14: **else if** ig_intr_md.ingress_port == RecircPort **then**
15: $t_{\text{now}} \leftarrow$ gTimestamp;
16: $f_{\text{stat}} \in \mathbb{B} \leftarrow$ StatusRegRead(`p.index`);
17: $f_{\text{timeout}} \in \mathbb{B} \leftarrow$ TimeoutRegRead(`p.index`)
18: **if** f_{stat} & f_{timeout} **then**
19: table_error(); ▷ Send error
20: **else if** $\neg f_{\text{stat}}$ **then**
21: drop(); ▷ Return packet received, no timeout
22: **else**
23: recirculate(); ▷ Return packet not received
24: **end if**
25: **else**
26: StatusRegDec(`p.index`); ▷ Reset status
27: **end if**

a result, FASTR is capable of detecting and responding to failures immediately upon occurrence.

FASTR uses Equation 5 for calculating the number of entries per monitored path, where $t^{(i)}_{\text{interval}}$ represents a possible probe sending interval:

$$n_{\text{entries}} = \left\lceil \frac{\max(t_{\text{timeout}}, t_{\text{RTT}})}{\min_i \{t^{(i)}_{\text{interval}}\}} \right\rceil \tag{5}$$

To guarantee the correctness across all possible probe intervals, FASTR uses the minimum interval value when computing the number of register entries per switch.

2) Data Plane: Figure 1 illustrates the processing logic implemented in the data plane. We first describe the end-to-end probe packet flow inside the data plane. Finally, we highlight two essential aspects of FASTR, namely (1) minimizing bandwidth consumption, and (2) eliminating the risk of false failure alarms.

Packet Processing We outline the data-plane logic in Algorithm 1. Probe packets have three special cases. First, when a probe arrives from the CPU port, we initialize the status and timeout registers and tag it for Traffic Manager (TM) replication, which also steers the replica to the recirculation port. The original packet is forwarded, while the replica loops. On each replica recirculated arrival, it reads both the status and timeout via the header index. If $status = 0$ (the backward clone has returned), drop the recirculating packet, and if $status = 1$, compare the current time with the stored timeout. If the timeout has not expired, keep recirculating and recheck in the next iteration. However, if the timeout has expired, an error will be raised indicating failure.

Reducing Overhead Probe packets use a custom header type with a size of 22 bytes. At a high sending rate of one probe packet every 100 ns, this results in a bandwidth consumption of 5.12 Gbps. Such overhead highlights the importance of minimizing its impact on the network.

FASTR reduces bandwidth overhead by employing a traffic-aware mechanism based on per-port traffic activity. Specifically, it maintains a per-port counter to monitor the ingress traffic rate of a port. Each probe packet computes the difference between the current ingress port counter value and the previously recorded value (in another register) to infer whether traffic has been received from the corresponding target switch since the last probe packet. The logic is illustrated in lines 2-4 in Algorithm 1.

Eliminating False Alarms The data-plane heartbeat-based failure mechanism introduced previously is susceptible to false positives caused by congestion-induced packet loss. Research has demonstrated that such congestion is a common problem, even in modern data centers, which impacts their flow completion time [18], [19]. When the queues at the egress are full due to congestion, probe packets may be dropped, leading to a falsely raised alarm by FASTR. The false alarm is less costly for link failure events, where a fast failover mechanism is triggered to reroute traffic. Yet this false alarm is worse in the context of a stateful data plane, where stateful components need to be synchronized across multiple devices.

To mitigate this risk, FASTR leverages strict priority scheduling. In FASTR, we dedicate one queue exclusively for probe packets and assign the highest priority to it. This ensures that during periods of congestion, probe packets are still guaranteed to be transmitted first, thereby eliminating the risk of false alarms.

More importantly, the traffic-aware mechanism described above reduces the bandwidth overhead of the probe packets. Therefore, applying higher priority to the probe packets eliminates the false alarms without severely impacting the goodput of the normal traffic.

E. Failure Mitigation

The failure mitigation phase is designed to operate entirely within the data plane, following a one-time setup phase. During the setup phase, backup routes and associated state components are pre-computed and installed based on the provided network topology. After this initialization, failure mitigation proceeds automatically without the involvement of the control plane. Upon detection of a failure, switches coordinate using a pre-configured multicast group and an error propagation table to signal the event. A dedicated register

and an associated routing table then steer traffic through the corresponding pre-planned backup path.

Link Failure When a source switch detects a neighboring link failure, it uses the pre-configured multicast group to notify all switches directly connected to the target switch. Similar to [16], one bit of the traffic routing register is flipped and used by the routing table to reroute traffic through the corresponding pre-planned backup route.

Device Failure Device failures render in-switch states hosted on the failed switch inaccessible. To account for this, FASTR raises a signal when all reachable paths to a target switch are unavailable. A bitmap tracks failures, and a device failure is inferred when it indicates no viable paths. While FASTR handles detection and mitigation, this bitmap may also trigger higher-level mechanisms (e.g., state recovery), which are outside the scope of this work.

IV. Evaluation

We have implemented and conducted experiments on the performance of FASTR using a triangle topology with one 10 Gbps link to simulate congestion and two 25 Gbps links. All programmable switches used are BF2556x-1T [20] switches equipped with Tofino1 ASIC.

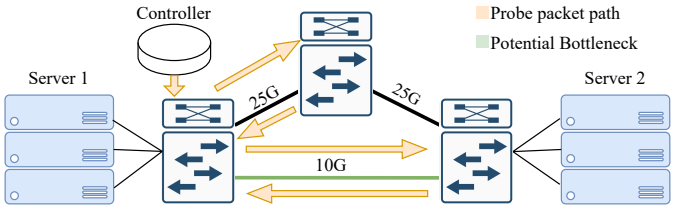


Fig. 2: Evaluation setup of the triangle topology.

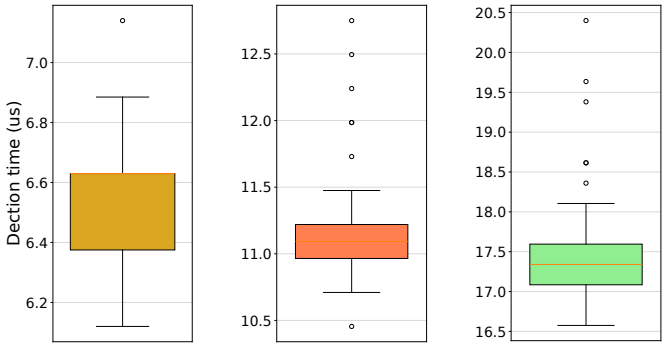
A. Fast Failure Detection Time

We have conducted an experiment on the link failure detection time of the FASTR implementation on Tofino hardware. In the experiment, probe packets were sent with a fixed timeout of 10 μs and different intervals.

To accurately measure the detection time, we added a register to maintain the timestamp of the latest ingress packet arriving at the "failed" link ingress port, t_{failure}. When a link fails, no more ingress packets are received from the corresponding port. As traffic and probe packets continuously traverse the link, such a value is the best candidate for estimating when a link failure occurs. We also collected the timestamp when a timeout expiration was triggered t_{detected}. Therefore, based on the two previously collected data-plane timestamp values, we can compute the failure detection time to be:

$$t_{\text{detection}} = t_{\text{detected}} - t_{\text{failure}} \tag{6}$$

Note that such a calculation is only for the collection of actual detection time inside the data plane, and is merely for experimental purposes to demonstrate the capability of fast detection. In a deployment setting, the detection trigger is directly generated inside the data plane via multicast and sent to the rest of the connected switches through the pre-configured multicast group.



(a) 4μs. (b) 8.5μs. (c) 15μs.

Fig. 3: Failure detection time over different probe intervals, with timeout being 10 μs and RTT around 7 μs.

Figure 3 illustrates the failure detection time of FASTR under various probe intervals across 50 distinct experiments, with probe intervals 4, 8.5, and 15 μs representing cases 3), 2), and 1) in Section III-D, respectively. Results show that FASTR is capable of providing a bounded detection time across different probing rates, with a detection time of less than 10 μs achieved.

B. Recirculation Latency

We have conducted another experiment on recirculation latency. Our results show that a recirculated packet spends ~516 ns inside the pipeline per iteration. This implies that, based on the detection time analysis in Section III-D, the recirculation latency introduces a variation to the detection time of at most 1 μs. More importantly, probe packets will only be cloned and recirculated when there is no incoming traffic between intervals, thereby minimizing the number of clones and recirculating packets needed.

C. Bandwidth overhead and Detection correctness

FASTR needs to minimize the impact on the traffic and detection failures correctly. To demonstrate the bandwidth overhead caused by inserting probe packets into the data plane, we conducted an experiment in which the link was fully saturated with normal traffic sharing the bandwidth with probe packets. In addition to demonstrating the bandwidth, this experiment can also verify the correctness of FASTR detection. In the experiment, we sent normal traffic along with probe packets with different sending intervals $t_{interval}$ of the probe packets over a 10 Gbps link. Figure 4 depicts the goodput of traffic under different scenarios, where goodput is the average throughput of normal traffic for 30 s.

Figure 4a illustrates the goodput of traffic sent from Server 1 towards Server 2, and congestion occurs at the source. With low inbound traffic, probes are required and use the strict-priority queuing to guarantee their delivery. Thus, the probe

	Stage 0	Stage 1	Stage 2	Stage 3	Stage 4
SRAM %	5.0%	7.5%	2.5%	2.5%	6.3%

TABLE II: SRAM usage per stage from 0 to 4. Note that only 5 stages were used.

has an impact on the goodput of the normal traffic. However, with a probe rate at a low interval of 5 μs, FASTR still only impacts a small portion of the goodput of less than 1.5%. Such an impact is a static amount to ensure the timely delivery of probe packets.

In Figure 4b, no goodput degradation was observed due to the traffic-aware mechanism. Figure 4c shows the bidirectional traffic goodput with various probe intervals, and results indicate that probe packets introduce overhead to neither side. This demonstrates the effectiveness of FASTR's traffic-aware mechanism. The mechanism suppresses probe packets when sufficient bidirectional traffic is observed, thereby reducing unnecessary overhead while still maintaining timely failure detection. More importantly, during the entire experiment, which involved different probe sending rates and various congestion scenarios, no failures were detected by FASTR, demonstrating the robustness of FASTR.

D. Resource consumption

As hardware-programmable data-plane devices are often resource-constrained, it is critical to evaluate the hardware resource consumption of the proposed FASTR. The ability to have enough resources to run other NFs on the same switch is critical for achieving high utilization of the networking devices.

As such, we analyzed the resource consumption using the P4Insight tool provided by the Tofino SDE. The results revealed that the implementation of FASTR spans a total of 5 different stages inside the Tofino pipeline, which leaves enough stages for other P4 applications to coexist with the existence of FASTR. In addition, Table II depicts the usage of SRAM per stage from Stage 0 to Stage 4. This shows that although the implementation of FASTR spans five different stages, it leaves sufficient spare capacity in each stage to incorporate other logic, if designed carefully.

V. Discussions

Control-plane Intervention FASTR avoids runtime control-plane involvement by executing failure detection and reaction entirely in the data plane. While programmable switches lack native timers, FASTR emulates timeouts using registers and recirculated probes to achieve μs-level detection.

Scalability FASTR scales to multi-switch networks with bounded per-peer state where each switch allocates a small, fixed number of entries per monitored neighbor. With no inbound traffic in the worst case, sustaining sub-10 μs detection requires probe overhead of 100 Mbps per neighbor. Aggregate link overhead grows linearly, $O(n) \approx 0.1nGbps$, e.g., 100 neighbors consume 10 Gbps. Under typical or high load, the traffic-awareness of FASTR suppresses probes when data traffic proves liveness, reducing the induced overhead.

Beyond Neighbors FASTR targets one-hop links with stable delay and no reordering. Extending to multi-hop paths introduces jitter and dynamics that can break microsecond timers, and FASTR needs to adopt a more resilient mechanism to withstand such variability challenges.

Robustness To ensure robustness, FASTR targets failure detection between neighboring switches, operating in a highly stable environment with minimal latency variance. Unlike wide-area or multi-hop networks, such direct links do not experience path changes, packet reordering, or significant jitter. Therefore, FASTR does not need to account for these factors in the failure detection logic.

VI. Related Work

A. Failure detection

FANcY [21] provides gray-failure detections and localization for ISP networks with a second-level detection duration. In contrast, [22] employs a probabilistic graphical model to detect failures in data center networks. Franco et al. [23] evaluated native failure detection capabilities of Tofino switches and reported port failure detection latency of hundreds of μs. Their Blink [24] detects failures by leveraging TCP retransmissions at the end hosts.

B. Fast Failover

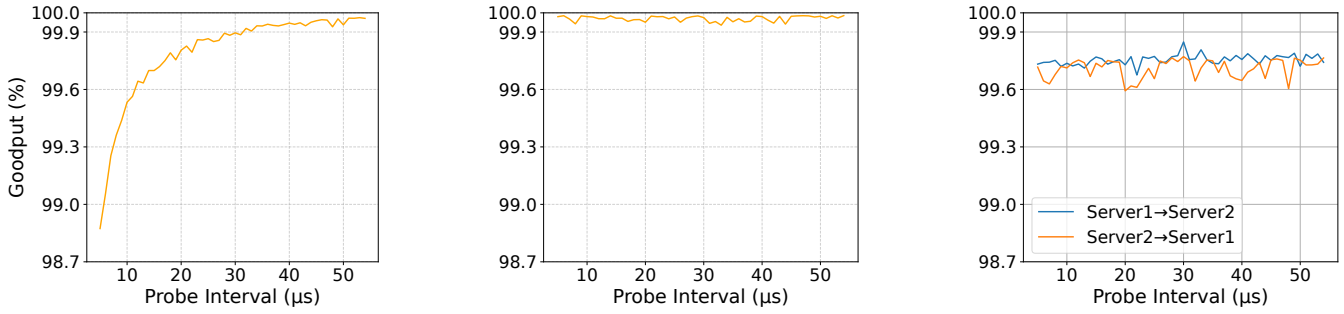
Early work, such as [25], utilizes pre-engineered topologies and multi-staged failover mechanisms to facilitate rapid failover at the routing layer. More recent work leverages programmable data planes to achieve fast failover. Miura et al. [26] apply Multiple Routing Configuration (MRC) in P4 to enable fast in-switch path switching, yet rely on an external detection.

SQR [27] and P4Neighbor [28] achieve stateless data plane failover by pre-encoding alternate path information inside the packet header space. However, neither work provides fast failure detection mechanisms.

Felix [29] approached the problem by using heuristic-driven rerouting in the data plane, yet offering hundreds of milliseconds of latency for detection and recovery. [30] leverages data-plane state replication to an external storage for recovery, yet incurs high restoration delay due to the communication overhead to the external device.

C. Data-plane state management

Several systems explore managing persistent state components inside the data plane. [31] proposes the usage of in-data-plane replication for stateful component migration. Swish [32] provides an abstraction to distribute shared memory across different programmable switches, allowing collaborative in-network stateful network applications. [33] explored the consistency of stateful components during switch reconfiguration. LOADER [34] enables programmable switch programs to make decisions through both local and replicated global states via abstractions for defining and managing synchronized state components.



(a) Server 2 → Server 1 (b) Server 1 → Server 2 (c) Bidirectional Traffic

Fig. 4: Goodput of traffic of different directions with probe interval between 5 to 55 μs over a 10 G link.

VII. Conclusion

This paper proposes FASTR, a solution for fast failure detection and mitigation for stateful programmable data planes. Using lightweight probes triggered by a local controller, failure detection and reaction are fully handled in the data plane, with mechanisms to reduce overhead and eliminate false alarms. Implemented on Intel Tofino, FASTR achieves sub-10 µs detection with $< 1.5\%$ bandwidth overhead in worst cases, minimal traffic impact, and no false alarms under congestion.

References

[1] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN," *SIGCOMM Comput. Commun. Rev.*, aug 2013.

[2] B. Turkovic, J. Oostenbrink, F. Kuipers, I. Keslassy, and A. Orda, "Sequential Zeroing: Online Heavy-Hitter Detection on Programmable Hardware," in *IFIP Networking*, 2020.

[3] T. Barbette, E. Wu, D. Kostić, G. Q. Maguire, P. Papadimitratos, and M. Chiesa, "Cheetah: A high-speed programmable load-balancer framework with guaranteed per-connection-consistency," *IEEE/ACM Transactions on Networking*, 2021.

[4] J. Xing, W. Wu, and A. Chen, "Ripple: A programmable, decentralized Link-Flooding defense against adaptive adversaries," in *USENIX Security Symposium*, USENIX Association, Aug. 2021.

[5] S. Sengupta, H. Kim, and J. Rexford, "Continuous in-network round-trip time monitoring," in *SIGCOMM*, 2022.

[6] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with univmon," in *SIGCOMM*, 2016.

[7] C. Cascone, L. Pollini, D. Sanvito, A. Capone, and B. Sansó, "Spider: Fault resilient sdn pipeline with recovery delay guarantees," in *2016 IEEE NetSoft*, 2016.

[8] D. Franco, M. Higuero, A. Sanz, J. Unzilla, and M. Huarte, "vffr: A very fast failure recovery strategy implemented in devices with programmable data plane," *IEEE Open Journal of the Communications Society*, 2024.

[9] R. Potharaju and N. Jain, "When the network crumbles: an empirical study of cloud network failures and their impact on services," in *SOCC*, 2013.

[10] D. Katz and D. Ward, "Bidirectional Forwarding Detection (BFD)." RFC 5880, June 2010.

[11] C. Rizzi, Z. Yao, Y. Desmouceaux, M. Townsley, and T. Clausen, "Charon: Load-aware load-balancing in p4," in *CNSM*, 2021.

[12] M. Zhang, G. Li, S. Wang, C. Liu, A. Chen, H. Hu, G. Gu, Q. Li, M. Xu, and J. Wu, "Poseidon: Mitigating volumetric ddos attacks with programmable switches," in *NDSS*, 2020.

[13] C. Jia, T. Pan, Z. Bian, X. Lin, E. Song, C. Xu, T. Huang, and Y. Liu, "Rapid detection and localization of gray failures in data centers via in-band network telemetry," in *NOMS*, 2020.

[14] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger, "Sonata: query-driven streaming network telemetry," in *SIGCOMM*, 2018.

[15] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: measurement, analysis, and implications," in *SIGCOMM*, 2011.

[16] H. Miura, K. Hirata, and T. Tachibana, "P4-based design of fast failure recovery for software-defined networks," *Computer Networks*, 2022.

[17] Intel, "Intel® Tofino™ Intelligent Fabric Processors."

[18] A. Zapletal and F. Kuipers, "Slowdown as a metric for congestion control fairness," in *HotNets*, 2023.

[19] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," *SIGCOMM Comput. Commun. Rev.*, Aug. 2010.

[20] "Advanced Programmable Switches (APS), BF2556X-1T Advanced Programmable Switch, APS Networks."

[21] E. C. Molero, S. Vissicchio, and L. Vanbever, "Fast in-network gray failure detection for isps," in *SIGCOMM*, 2022.

[22] V. Harsh, T. Meng, K. Agrawal, and P. B. Godfrey, "Flock: Accurate network fault localization at scale," *Proc. ACM Netw.*, July 2023.

[23] D. Franco, M. Higuero, E. O. Zaballa, J. Unzilla, and E. Jacob, "Quantitative measurement of link failure reaction time for devices with P4-programmable data planes," *Telecommunication Systems*, 2023.

[24] T. Holterbach, "Blink: Fast connectivity recovery entirely in the data plane," in *NSDI*, p. 84, 2019.

[25] V. Liu, D. Halperin, A. Krishnamurthy, and T. Anderson, "F10: A Fault-Tolerant engineered network," in *NSDI*, 2013.

[26] K. Hirata and T. Tachibana, "Implementation of multiple routing configurations on software-defined networks with p4," in *2019 APSIPA ASC*, 2019.

[27] T. Qu, R. Joshi, M. C. Chan, B. Leong, D. Guo, and Z. Liu, "Sqr: In-network packet loss recovery from link failures for highly reliable datacenter networks," in *ICNP*, 2019.

[28] J. Xu, S. Xie, and J. Zhao, "P4Neighbor: Efficient Link Failure Recovery With Programmable Switches," *IEEE Transactions on Network and Service Management*, 2021.

[29] J. A. Marques, K. Levchenko, and L. P. Gaspary, "Responding to network failures at data-plane speeds with network programmability," in *NOMS*, 2023.

[30] D. Kim, J. Nelson, D. R. K. Ports, V. Sekar, and S. Seshan, "Redplane: Enabling fault-tolerant stateful in-switch applications," in *SIGCOMM*, 2021.

[31] S. Luo, H. Yu, and L. Vanbever, "Swing State: Consistent Updates for Stateful and Programmable Data Planes," in *SOSR*, 2017.

[32] L. Zeno, D. R. K. Ports, J. Nelson, D. Kim, S. Landau-Feibish, I. Keidar, A. Rinberg, A. Rashelbach, I. De-Paula, and M. Silberstein, "SwiSh: Distributed shared state abstractions for programmable switches," in *NSDI*, Apr. 2022.

[33] C. Ji and F. Kuipers, "State4: State-preserving reconfiguration of p4-programmable switches," in *NetSoft*, 2023.

[34] G. Sviridov, M. Bonola, A. Tulumello, P. Giaccone, A. Bianco, and G. Bianchi, "LOcAl DEcisions on Replicated States (LOADER) in programmable dataplanes: Programming abstraction and experimental evaluation," *Computer Networks*, 2021.