Software-defined network support for transport resilience

João Taveira Araújo, Raúl Landa, Richard G. Clegg, George Pavlou
University College London
Email: {j.araujo, r.landa, r.clegg, g.pavlou}@ucl.ac.uk

***Abstract*—Existing methods for traffic resilience at the network and transport layers typically work in isolation, often resorting to inference in fault detection and recovery respectively. This both duplicates functionality across layers, eroding efficiency, and leads to protracted recovery cycles, affecting responsiveness. Such misalignment is particularly at odds with the unprecedented concentration of traffic in data-centers, in which network and hosts are managed in unison.**

This paper advocates instead a cross-layer approach to traffic resilience. The proposed architecture, INFLEX, builds on the abstractions provided by software-defined networking (SDN) to maintain multiple virtual forwarding planes which the network assigns to flows. In case of path failure, transport protocols proactively request to switch plane in a manner which is unilaterally deployable by an edge domain, providing scalable end-to-end forwarding path resilience.

I. Introduction

Despite being broadly designed for robustness, the current Internet architecture remains remarkably vulnerable to failures. Managing faults still poses a significant operational challenge, in part because faults can occur at every layer of the networking stack, and can affect any element along a network path. This inherent diversity has traditionally favoured placing resilience mechanisms at the transport layer. Since endpoints possess both intrinsic knowledge of application needs and fine-grained measurements on end-to-end path characteristics, transport protocols offer a natural fit for providing scalable resilience. Modern transport protocols such as Stream Control Transmission Protocol (SCTP) [1] and Multipath TCP (MPTCP) [2] address resilience by providing transparent failover through multihoming. Unfortunately, neither is likely to be widely deployed in the near future, with the former lacking critical middlebox support and the latter still undergoing standardization. This is compounded by the fact that end-host multihoming in itself poses a barrier to deployment. Finally, both SCTP and MPTCP, being confined to the transport layer, rely only on knowledge of faults within an individual flow. Hence, each flow must detect failures independently, even when many flows are affected by the same fault.

An alternative approach to resilience is to target the network layer, hence shifting the onus of resilience from the endpoints to the network operators. Traditionally, this has been achieved by designing improved routing algorithms. The deployment of real time applications with harder constraints on reliability coupled with better failure detection methods embedded in linecards have provided both the motivation and the means for achieving sub-second recovery within Interior Gateway Protocol (IGP) networks [3]. Even with reduced recovery times however, the transient effects of routing changes can still disrupt the forwarding path. Other frameworks [4], [5], [6] have been proposed to provide repair paths for use between the detection of a failure and the convergence of the routing process. Unfortunately, such methods are rarely sufficient. Firstly, their application is circumscribed to individual domains, and as such cannot provide end-to-end coverage in a federated, best-effort Internet. In order to detect failures in remote domains, an operator may employ active monitoring techniques [7], but such approaches can neither scale to cover most destinations nor operate at a small enough timescale to ensure timely recovery. Secondly, there are many faults which do not directly pertain to routing [8], such as middlebox misconfiguration or hardware malfunctions. For these kinds of faults, routing-based approaches are completely ineffective. Finally, fault reparation purely at the network layer often disregards (and can even potentially disrupt) the transport layer by causing out-of-order delivery of packets.

Software-Defined networking (SDN) [9] is a promising tool for improving network resilience. By decoupling the control and data planes, SDN provides vastly improved flexibility for scalable, policy-based network control. However, network resilience problems cannot be solved by this flexibility alone. Instead, what is needed is a cross-layer approach that bridges the gap between the transport and the network layers and provides end-to-end insight into network routing decisions.

This paper presents INFLEX, an SDN-based architecture for cross-layer network resilience which provides on-demand path fail-over for IP traffic. INFLEX operates by allowing an SDN-enabled routing layer to expose multiple *routing planes* to the transport layer. Hence, traffic can be shifted by one routing plane to another as a response to end-to-end failure detection. INFLEX then operates as an extension to the network abstraction provided by IP, and can be used by any transport protocols. While this architecture requires changes across both network and host, it is deployable because it can be adopted *unilaterally*, providing benefits even when used by individual domains, and is inherently end-to-end, potentially covering third party failures.

At the host, the proposed architecture allows transport protocols to switch network paths at a timescale which avoids flow disruption and which can be transparently integrated into existing congestion control mechanisms. Within the network, INFLEX provides both greater insight into end-to-end path quality, assisting fault detection, and more control over flow path assignment, enabling more effective fault recovery. In addition to describing our architecture design and justifying our

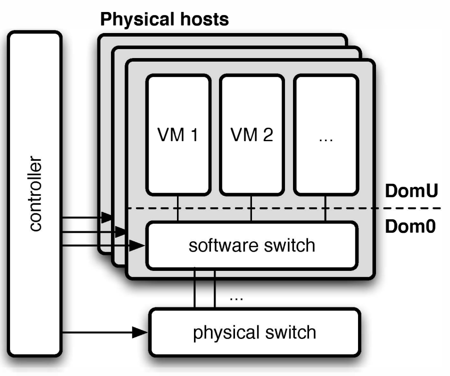


Fig. 1: SDN architecture

design choices with extensive network measurements, we also implement INFLEX and verify its operation experimentally. We make our modifications to both the TCP/IP network stack and a popular OpenFlow controller [10] publicly available.

The remainder of the paper is structured as follows. Section II provides a brief background on software-defined networks. Section III reviews some of the design decisions in the light of longitudinal measurements performed on the MAWI dataset [11]. An overview of the proposed architecture is the presented in Section IV, followed by an evaluation of INFLEX in Section V. Finally, conclusions are drawn in Section VII.

II. OpenFlow Background

Software defined networks decouple the data plane and control plane, allowing both to evolve independently. A traditional instantiation of a software defined network for datacenters is shown in Figure 1. Each physical host runs a number of virtual machines, each connected locally through a software-based edge switch (such as Open vSwitch [12]) running on the underlying physical host operating system. This switch is in turn connected to further forwarding devices, ensuring access to a wider network. The forwarding logic of each device can be accessed and configured by a controller through the establishment of a control channel through a common protocol, of which OpenFlow is the most widely used [9]. Both software and physical switches are indistinguishable from a controller's perspective: *how* a device implements OpenFlow is immaterial, so long as a forwarding device conforms to the given API.

An OpenFlow flow table is composed of multiple flow entries. Each flow entry is comprised of a pattern to be matched, and the corresponding *instructions* to be executed. The *match fields* over which an entry can be compared span from data link to transport layers, covering not only source and destination addresses at each protocol header, but also traffic classes and labels for VLAN, MPLS and IPv4 headers. Additionally, a *counter* keeps track of the number of times the entry is matched. If more than one matching entry is found, only the entry with the highest *priority* is processed. Finally, each entry has a pair of *timeout* values: a soft timeout, within which an entry is expired if no matching packet arrives, and a hard timeout, by which an entry is irrevocably expired. If neither timeout is set, a flow entry persists indefinitely.

An OpenFlow switch in turn maintains multiple flow tables. Every packet received at an OpenFlow compliant switch is processed along a pipeline which starts by matching the packet against *table 0*. From this first, default table, corresponding *instructions* may redirect the packet for further matching against another table, thereby chaining processing. This pipeline processing ceases once a matching entry fails to include a redirection request, with the accumulated instruction set being executed. In addition to redirections, valid instructions include modifying packet fields, pushing and popping packet tags, and defining through which ports a packet should be forwarded. If at any point no matching entry is found, the packet is buffered at the switch, and the truncated packet header is sent to the controller. Based on the header contents, a controller may decide to install a new flow entry on the switch, or allow the packet to be dropped altogether. Compared to the underlying, abstracted network elements which compose the data path, the controller is often expected to be entirely software based, and as such is not constrained in how it should process packets. In practice, this freedom is curbed as increasing complexity at the controller both reduces the rate at which packets are processed, as well as increasing latency for packets buffered at the switch.

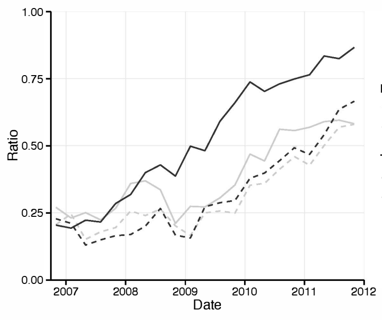
The overall performance of the described architecture is subject to two further critical tradeoffs. Firstly, the granularity at which flow entries are installed determines how often a controller is called to intervene. While installing an entry at a flow granularity may allow fine-grained control of resources, it increases both the load on the controller and the latency of the withheld packet. Conversely, as the granularity becomes coarser, the overhead incurred by the controller is reduced at the cost of flexibility in controlling traffic. Secondly, controller placement is critical [13]. At one extreme, a fully centralized controller is omniscient within a domain at the expense of reliability and scalability. At the other, a distributed system of controllers forsakes consistency and liveness in order to scale.

III. Design considerations

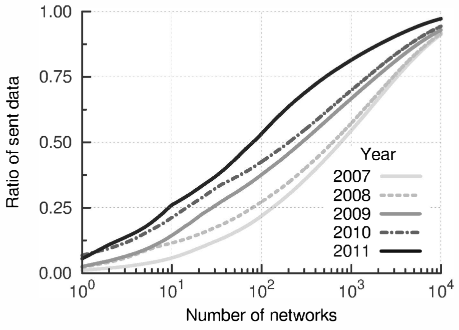
SDN provides an abstraction over which different architectural paradigms can be adapted and even coexist. It does not however prescribe or advocate a specific design – network practitioners must still consider system properties when grappling with fundamental tradeoffs affecting consistency, isolation, reliability and efficiency.

This section provides design considerations for scalable traffic management based on observations obtained across a longitudinal analysis of interdomain traffic. The dataset used is a five year subset of the unanonymized MAWI traces [11], spanning from late 2006 to early 2012. In addition to the raw packet traces, collected for 15 minutes every day, both source and destination IPs are mapped to the respective AS by reconstructing archived BGP routing dumps archived at routeviews. Both routing information and traffic traces are collected from the same point – within the WIDE AS – which provides transit for a Japanese research network.

Ideally resilience could be implemented at the transport layer alone, for the same motives rate control is best left to end-hosts: ultimately, the host is best positioned to detect end-to-end path faults and can often react over shorter timescales than the network, which must concern itself with reconvergence. This approach for path fail-over was a significant feature in



(a) Windowscale deployment



(b) CDF of outbound traffic by prefix

Fig. 2: Longitudinal traffic properties for the MAWI dataset.

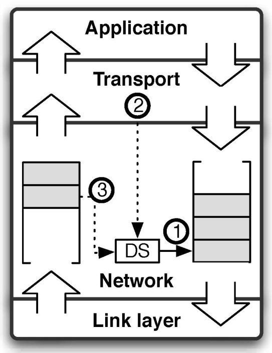


Fig. 3: INFLEX stack.

SCTP [1]. Unfortunately, deployment of SCTP has been negligible in over a decade since standardization, in part because the pervasiveness of middleboxes has significantly affected the ability for new transport protocols to be deployed. More recently Multipath TCP [2] has been proposed addressing many of the same concerns as SCTP whilst maintaining the same wire format as TCP, thereby ensuring middlebox compatibility. Despite this, widespread deployment is far from guaranteed, and is largely tied to the rate of OS adoption. As a reference point, Figure 2a tracks the use of the TCP windowscale option by overall volume in flows and bytes across both directions in the MAWI dataset. Without windowscale negotiation, a sender's congestion window cannot exceed 65KB. Despite offering a clear benefit to both endpoints, being simple to implement and incurring a low overhead, windowscale deployment has only recently picked up momentum, two decades since standardization. Expecting substantial deployment of a more complex and costly extension such as MPTCP over the near future is likely optimistic. Critically, transport extensions require receiver adoption and are therefore subject to the willingness and ability of users to upgrade their OS.

Receiver side deployment of even modest TCP extensions can be protracted, even when incentives are aligned. Rather than proposing a path for incremental deployment, this work focuses on how to obtain similar benefits immediately – modifying sender side hosts only. A host, however, cannot directly affect routing without changing destination address, which would break legacy TCP receiver side implementations. Additional extensions are required on the sender side network to enable multipath forwarding. Conventional wisdom suggests that maintaining parallel routing planes requires a proportional increase in table size [14], which itself can be subject to exponential growth. In practice however, this state can be significantly reduced by forsaking coverage for a small proportion of traffic. Rather than reflect the entirety of its potential path diversity for all traffic, an edge provider can instead provide additional routing planes for only a subset of popular prefixes.

The extent to which such a gain is possible for the MAWI dataset is quantified in Figure 2b, which displays the cumulative distribution function of outbound traffic across network prefixes announced by BGP neighbours. Over five years, traffic to approximately 340,000 unique prefixes was observed. Invariably however, an increasing amount is sent to a small group of prefixes – by 2011, over 50% of traffic went to the top 100 prefixes alone. This reflects ongoing structural changes in the Internet architecture as content providers interconnect directly edge, *eyeball* networks, and content becomes increasingly consolidated across a set of large content providers and national and regional ISPs.

Multipath routing state can be significantly reduced by covering fewer destinations while still benefiting most traffic. Within the MAWI dataset virtually all inbound and outbound traffic could be mapped to 10,000 unique network prefixes. Existing SDN tools such as RouteFlow [15] are already capable of overlaying routing on commodity switches, but the incurred overhead can still be a concern for production networks. Rather than address the scalability challenges inherent to multipath routing directly, these results suggest that a tangible deployment path lies instead in reducing the scope over which it is applied.

IV. Architecture

This section describes INFLEX, an architecture which provides edge domains with greater end-to-end resilience. Rather than probing paths through active or passive means, the network delegates the responsibility for fault detection to end-hosts. The system relies on packet marking at the host to select a path through the local domain. This provides far greater scalability in terms of the proportion of traffic and destinations which can be covered, at the cost of requiring small changes to the end-host TCP/IP stack. INFLEX is therefore particularly suited for managed environments, such as datacenters or enterprise networks, which not only have greater control over the end-host operating system, but also generate large volumes of traffic towards destinations which cannot be readily upgraded.

An overview of the proposed architecture as applied to a single domain is shown in Figure 4. Hosts are connected to the local network through an OpenFlow-enabled *edge switch*. While edge switches typically reside within each physical machine, alternative aggregation levels such as the top of rack or end of row may also be used. Each such switch is configured by a specialized controller which resides locally, referred to as an *inflector*. The local network is configured by a centralized routing controller to provide multiple *virtual routing planes*. While these planes are necessarily intradomain in scope, some degree of interdomain diversity can also be achieved by changing egress node.

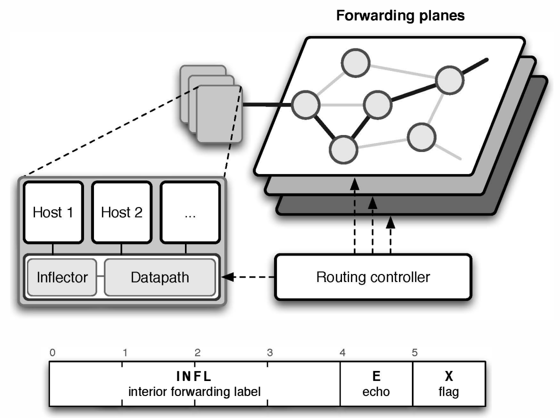


Fig. 4: INFLEX architecture (above) and INFLEX header (below). The edge switch forwards traffic across virtual planes set up by a centralized routing service.

The core of the architecture relies on repurposing the Differentiated services (DS) field in each IP packet to provide an in-band signalling channel between the end-host and the inflector. The header on inbound traffic is set by the edge switch and read by the host, and is used by the inflector to signal which plane a flow has been assigned to. The header on outbound traffic is set by the host and read by the edge switch, and is used by the transport protocol to ensure that all traffic for the flow is forwarded along the given plane. Hosts can request a new plane to be assigned by the inflector in case of an end-to-end path fault; this provides efficient cross-layer failure recovery. The DS standard [16] reserves a pool of code points for local use identified by setting the right-most bit, henceforth referred to as the INFLEX $flag$. When set, the rest of the DS field should be interpreted as containing two fields, shown in Figure 4. An Interior Forwarding $label$, which determines the plane over which a packet is forwarded, and an $echo$ bit, which explicitly signals a request from the host or a reply from the network. The remainder of the description of INFLEX is split across its main components: the end-hosts, the edge switch and the inflector.

A. INFLEX end-hosts

INFLEX hosts set the INF label of outbound packets according to the value assigned by the inflector, in a similar fashion to the path re-feedback mechanism introduced in [17]. The changes required to support this at the sender side network stack are minimal, and are illustrated in Figure 3. Every transport connection occurs over a socket, a local structure containing the variables associated to the ongoing flow. At the network layer, the local socket has a DS value which is copied to every outbound packet (point 1). Within INFLEX, the transport protocol can trigger a request (point 2), which leads to a network response contained in incoming packets (point 3).

Assume a transport protocol wishes to switch the plane it is currently assigned. With INFLEX, it can send an *inflection request* by setting the *echo* bit of the local DS field (point 2, Figure 3). All subsequent outbound packets will be marked with the resulting value. The network layer then proceeds to inspect inbound packets, waiting for a network response, as delineated in Figure 5. After demuxing an incoming packet, pkt, to the corresponding socket, $sock$, a receiver first verifies whether the INFLEX flag is set on the incoming packet (line 2), establishing whether the underlying network supports INFLEX for the given connection. The receiver must then decide whether it should change the virtual plane the socket is currently assigned. This can only happen under two conditions. Firstly, if the DS value for the current socket does not have the INFLEX flag set (line 3). This typically occurs on flow start, where a connection is spawned with a default DS value. Secondly, if the local DS value has the echo bit set, there is a *pending* inflection request. If the incoming packet has the same bit set, it corresponds to the network *reply* (line 4). Under both previous cases, the connection switches forwarding plane by copy the interior forwarding label from the incoming packet to the local socket, and setting the INFLEX flag (lines 5-6). These changes are all applied at the IP layer – transport protocols need only to decide when to send inflection requests – while applications can remain unchanged.

```
 1 ...
 2 if (is_inflex(pkt)) {
 3   if (!is_inflex(sock) ||
 4       (is_pending(sock) && is_reply(pkt))) {
 5     copy_label(sock, pkt);
 6     clear_echo(sock);
 7   }
 8
 9 } else if (is_inflex(sock))
10   clear_inflex(sock)
11 ...
```

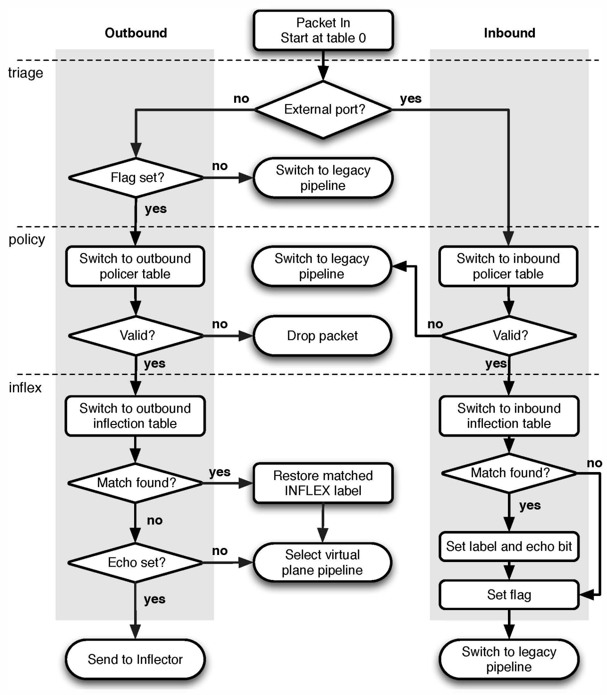
Fig. 5: Pseudo-code for packet reception using INFLEX.

B. The edge switch

The edge switch is primarily responsible for mapping INFLEX marked packets to the appropriate forwarding plane. On start up its datapath is configured by the local *inflector*, which installs the appropriate flow entries on it in order to construct the processing pipeline in Figure 6. This pipeline can be partitioned into three distinct blocks, responsible for *triaging*, *policing* and *inflecting* packets. For clarity, the processing pipeline is conceptually described as a sequence of flow matches across distinct tables. In practice, an implementer is free to collapse flow tables and entries to improve performance. An important safeguard is that a legacy pipeline must be present, establishing a default forwarding plane expected to be used by traffic to which INFLEX is not applicable.

The *triage* phase is responsible for distinguishing whether a packet is *capable* of using INFLEX. Firstly, INFLEX is only applicable to IP packets. Traffic is then differentiated according to the port on which the packet arrived: if connected to a host, the interface is said to be *internal*, otherwise it is *external*. Any inbound IP traffic may potentially be INFLEX capable and as such can proceed to the next stage. For outbound IP traffic, only packets with the INFLEX flag set require further processing. Packets for which this flag is not set are assumed to be legacy traffic.

The *policy* phase decides whether a packet is *permitted* to use INFLEX. For either direction, a packet is compared against a policer table, which contains a set of rules describing local policy concerning INFLEX usage. The rules applied to



Packet In
Start at table 0
Outbound
Inbound
triage
External port?
no
yes
Flag set?
no
Switch to legacy pipeline
yes
policy
Switch to outbound policer table
Switch to legacy pipeline
Switch to inbound policer table
Valid?
no
Drop packet
no
Valid?
yes
yes
inflex
Switch to outbound inflection table
Switch to inbound inflection table
Match found?
yes
Restore matched INFLEX label
Match found?
no
yes
no
Set label and echo bit
Echo set?
no
Select virtual plane pipeline
Set flag
yes
Send to Inflector
Switch to legacy pipeline

Fig. 6: Pipeline installed to the edge switch datapath.

each direction however may differ, particularly since outbound packets can be further scrutinized according to the INF *label*. For example, this allows the outbound policer to enforce which virtual planes are available to specific tenants or applications. For this reason, the action applied if a packet is matched within the policer table also differs according to direction. For inbound traffic, a matching rule indicates that the packet does not satisfy local requirements for INFLEX use, and is consequently treated as legacy traffic. For outbound traffic, a packet is already marked as being INFLEX capable. Any matching entry therefore indicates that it is in violation of local policy and should consequently be dropped.

Finally, the *inflex* phase processes the respective header and forwards the packet. A packet is first matched against an *inflection* table in either direction. This table is detailed in the next section, and can be assumed to contain no matching entry initially. For outbound traffic, the packet is typically redirected to the plane mapped by the interior forwarding label. The one exception are inflection requests, which are forwarded to the local inflector for further processing. For inbound traffic, the INFLEX flag is marked in order to notify hosts that the flow is INFLEX capable, and the packet is then processed according to the legacy pipeline.

C. The inflector

Each edge switch is controlled by an inflector, an SDN controller expected to reside locally. An inflector is firstly responsible for configuring the underlying datapath according to the previously described pipeline. Secondly, an inflector must process inflection requests.

Inflection requests require network intervention in assigning a packet to a forwarding plane. The dynamic nature of this decision process cannot readily be instantiated as a set of static rules at the edge switch, since a same flow must be able to be reassigned to a different plane in case of path faults. Therefore, inflection requests intercepted at the edge switch must be sent to a controller for further processing. Rather than overloading a centralized controller however, this decision can be taken locally – since the inflector manages the local rules associated to each virtual network, it already has full knowledge of the routing table associated to each plane. Upon receiving such a request, the inflector proceeds in three steps. It first verifies which virtual networks maintain a valid route for the given destination address. Given this list of potential planes, it then inspects local policy to verify which planes the packet is allowed to use. The intercepted packet contains the plane which the flow is currently using – this plane should be excluded from the candidate list unless there is no other option available. Finally, a single plane, identified by an interior forwarding label, is selected from the resulting list of candidates. The selection algorithm is not prescribed by the INFLEX specification, but a reasonable baseline is to select a routing entry proportionally to the assigned route weight.

Having selected an appropriate plane, the inflector installs forwarding rules into either *inflection table*. In the inbound direction, all packets matching the reverse flow are set to be marked with the corresponding INF *label*. This conveys the selected forwarding plane back to the host. In the outbound direction, all packets matching the flow are to be processed according to the *label*. This guarantees that any packet sent between the inflection request and its response are forwarded in a consistent manner. Rules installed to the inflection tables are ephemeral by nature, with a hard timeout of 1 second (the minimum permitted in the OpenFlow standard). This enables per-flow granularity with minimum flow state while also rate limiting inflection requests. Furthermore, flow entries can be configured to be sent to the controller upon expiry. This potentially allows the inflector to collect realtime information on the availability of each forwarding plane, allowing for further refinement of the plane selection algorithm.

V. Analysis

This section details the evaluation of INFLEX as well as details pertaining to its implementation. A reference implementation of the inflector was developed as a component of the POX network controller [10]. Additionally, INFLEX support for TCP was added to the Linux kernel, and is available as a small patch for version 3.8. All source code, as well as a virtual machine to replicate subsequent tests, is being made publicly available. The use of POX in particular invalidates any rigorous performance evaluation, as the implementation is not geared towards efficiency. Instead, the contributed code acts as a proof-of-concept for INFLEX, allowing the mechanics of the specification to be inspected and fine-tuned.

Open vSwitch 1.9 and OpenFlow 1.1 are used, enabling multiple table support. Unfortunately, current versions of OpenFlow do not support bitmasking the TOS field, and as such ECN bits are cleared when assigning INFLEX tags. This is a current limitation of OpenFlow which will likely be addressed in future versions in order to support manipulating the DS field while keeping the ECN field unchanged.

A simple evaluation scenario is used, illustrated in Figure 7. On one end is an INFLEX capable domain: a set of

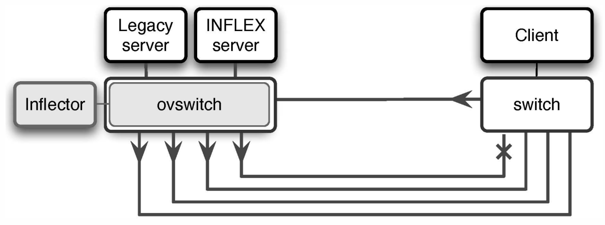


Fig. 7: Simulation setup.

virtual hosts acting as *servers* connected to an Open vSwitch edge switch controlled by an inflector. On the other end is a remote *client*. Typically this is an end-user device outside network operator control. We assume that the client is running a legacy network stack and connected to a switch with no SDN functionality. A single physical connection between the client and this switch acts as the bottleneck for all flows, with the bandwidth set to 10Mb/s. The edge switch has four potential planes over which it can forward traffic between the *servers* and the *client*. We simulate failures within the INFLEX domain by artificially dropping all forwarded packets belonging to a given plane; we denote this plane as being *down*. At any given moment one of the four available planes is down; each such simulated failure lasts for 15 seconds at a time, affecting planes cyclically. The reverse path, connecting from client to server, is always assumed to be functional. Propagation delay between both switches is set to 50ms.

A. Sender-side resilience

The first case study under review is one of the most common use cases for datacenters: a remote client *downloading* data from hosted servers. Under the conditions described previously, the forwarding path will be periodically affected by recurring failures. Since the nature and the origin of the fault are not always apparent to network devices, it is assumed that network elements within the INFLEX domain have no indication of the ongoing failure. Instead, it is up to the servers to detect and recover from perceived problems by issuing inflection requests. Clearly, requesting a path incurs some cost to network and host alike. For the network, an inflection request requires additional processing. For the host, this processing manifests itself as increased delay. This begs the question: when should a host request an inflection? The obvious candidate is to piggyback inflection requests on retransmissions spawned by *retransmission timeouts* (RTO). This leverages an existing transport mechanism which is well understood and only triggered under anomalous path conditions (as opposed to congestive losses). From the perspective of the host, any delay incurred by the inflection request is amortized by the retransmission timeout itself, which has a minimum value of 1 second. From the perspective of the network, such inflection requests should be both rare, reducing the amount of processing required, and critical to improve availability, justifying the expense in addressing them.

Figure 8 displays the congestion window over time for two concurrent flows towards a remote client. The first connection traced is a download from a server without INFLEX support, in which all packets are forwarded over the default path. The vertical lines signal the points at which the default forwarding path, plane 0, fails. Despite only failing for 15sec,

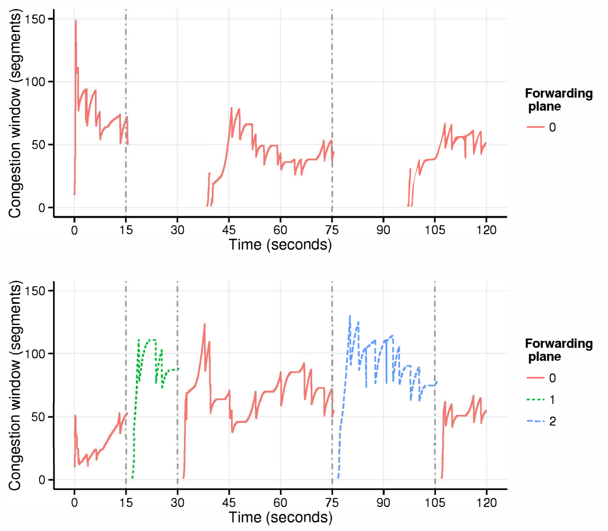


Fig. 8: Congestion window for concurrent downloads towards client from legacy (above) and INFLEX (below) servers.

the disruption to the transport flow lasts twice as long due to the exponential nature of the retransmission timeout, which doubles in duration at each occurrence. The second connection traced is a download occurring in parallel from an INFLEX capable server. In this case, each path failure is recovered by sending an inflection request on each retransmission timeout. The returned path is randomly assigned, as our basic proof-of-concept inflector does not currently keep track of network conditions. The time between path failure and flow recovery is directly tied to the RTO, in this case taking approximately one second. This value cannot be improved upon within the INFLEX framework, as the duration of flow entries introduced by inflection requests has a minimum timeout of 1 second. Conveniently however, this matches the lower bound of the RTO as defined by TCP, and it is therefore unlikely that a transport protocol would desire faster fail-over. In practice, the recovery time may be extended in order to account for spurious timeouts. For connections over wireless media in particular, timeouts may occur due to transient effects such as interference. While this is functionally equivalent to path failure, the transient nature of such events does not always require changes to the forwarding path.

An interesting implication of Figure 8 is that **TCP senders using INFLEX can accommodate path fail-over seamlessly**. Retransmissions conveniently distinguish between congestion events, triggering fast retransmission and similar heuristics, and pathological network conditions, which spawn repeated retransmission timeouts. In the case of the latter, the adopted response is to reset the congestion window and resume slow start – effectively restarting the flow. This behaviour is extremely conservative, but is a natural corollary of assuming as little as possible about the underlying network. As a result, no substantial change is required to the congestion control mechanisms employed by TCP in retrofitting cross-layer path fail-over at the sender using INFLEX.

B. Receiver-side resilience

Path failures can also affect the reverse path with equally nefarious consequences: the sender will repeatedly timeout in

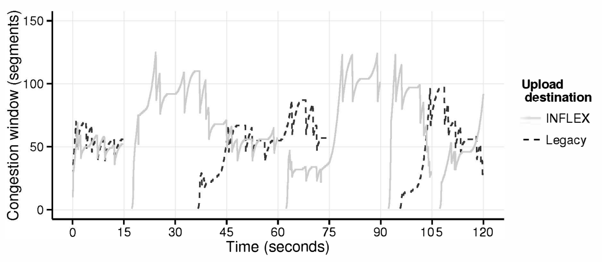


Fig. 9: Concurrent uploads from client towards servers.

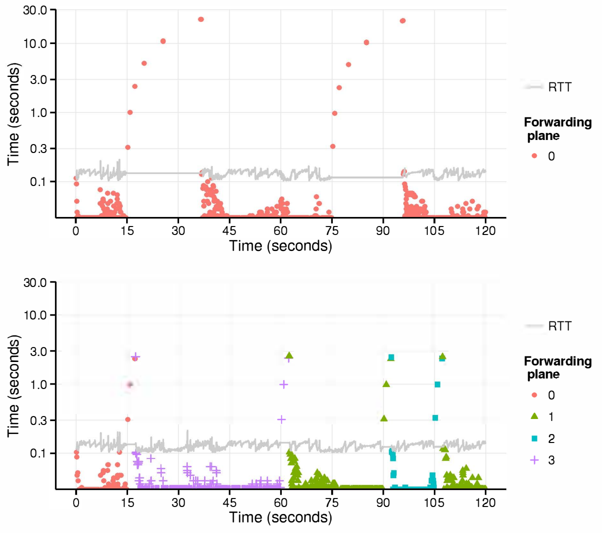


Fig. 10: Data packet inter-arrival time for legacy (above) and INFLEX (below) receivers.

the absence of acknowledgements from the receiver. Unlike failures affecting the forward path however, the INFLEX host does not actively track the reliability of the ongoing connection. TCP is sender driven, with receivers primarily generating acknowledgements as a response to inbound data. Hence, the reverse path lacks the reliable delivery mechanisms available in the forward path; if the *TCP Timestamp* option is not used, the receiver often lacks even an accurate RTT estimate. Furthermore, in the absence of data packets to be sent, there is no RTO on which to trigger inflection requests.

A receiver must instead rely on inferring path failure from the packet inter-arrival time when generating duplicate acknowledgements. With the exception of cases where immediate receiver feedback is required, such as a TCP timestamp request, duplicate acknowledgements are typically sent on the arrival of out-of-order data. Under path failure, the arrival time between such out-of-order events will rise exponentially as the sender TCP stack becomes tied to its own retransmission timeout. This behaviour is illustrated in figures 9 and 10, which show the result of using INFLEX with the same experimental setup but a reversed flow of data. Figure 9 displays the evolution of the congestion window size over time as the client uploads data concurrently to both a legacy and an INFLEX server. While the single forwarding path does not experience outages, the reverse path is periodically affected by failures. The corresponding data packet inter-arrival time is shown in Figure 10, with each sample point also displaying the routing plane used. For an ongoing TCP flow with sufficient data from

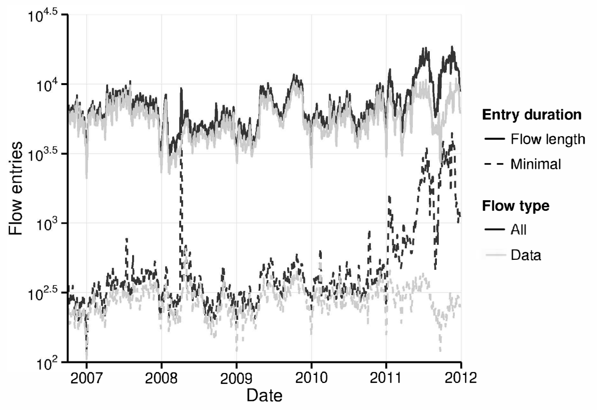


Fig. 11: Mean flow state for outbound traffic

the application layer the packet inter-arrival time at the receiver should be consistently low. RTT level dynamics are apparent on slow start, in which the sender is clocked by incoming ACKs, and during congestion events, in which out-of-order delivery temporarily affects the throughput. On path failure however, the inter-arrival time increases exponentially, with each inbound packet triggering a duplicate acknowledgement. For the upload to the legacy server, successive RTOs result in a recovery time of nearly 30sec.

An INFLEX receiver can use this information to decide when to trigger an inflection request. It can achieve this by setting a threshold for the time elapsed between duplicate acknowledgements, henceforth referred to as $dupthresh$. Comparatively to the sender, the receiver should be more conservative, as by design it has less information on which to act upon and does not typically exert control on the congestive feedback loop. Furthermore, neither sender nor receiver can reliably detect whether the forward or reverse path are at fault. By acting conservatively, a receiver allows the sender, which may also be INFLEX capable, to initiate recovery before trying to correct the reverse path. For the experiment displayed in Figure 10, the $dupthresh$ is set to twice the RTO, resulting in an overall downtime of approximately 3 seconds. Since each data point is generated on inbound data packets, recovery is signalled by a packet pair. A first inbound packet exceeding $dupthresh$ triggers an inflection request, which piggybacks on the acknowledgement sent out in response. A second inbound packet returns approximately 1 RTT later with the forwarding plane assigned by the network attached. Clearly some failures may not be recoverable, particularly if the remote host is not INFLEX capable and the fault lies on the reverse path. Nonetheless, the overhead incurred at the host is negligible, merely complementing congestion avoidance mechanisms with additional signalling. Remarkably, INFLEX incurs no additional memory costs at the host, operating as an extended API over the existing $inet\_connection$ socket, rendering it equally applicable to all transport protocols which use this socket structure, such as SCTP and DCCP.

C. Network overhead

The granularity at which an SDN deployment should manage traffic is often subject to debate. On one hand, hardware advances such as TCAMs offer fast lookup times over large tables, affording flow precision for many potential SDN deployments. On the other, deployments will often include cheaper, more flexible software switches which are less capable of scaling performance with the number of flow entries. Importantly, operating on a per-flow granularity is more likely to overload the controller, which itself can be a considerable source of latency. As a result, managing flow aggregates is often the preferred means of reducing this overhead, at the cost of flexibility in affecting flows individually.

INFLEX does neither strictly, exerting network control at a sub-flow granularity while pushing flow state to the end-host. Figure 11 investigates the relative expected overhead incurred by the network on adopting such an architecture. The graph tracks the mean flow state from applying different flow entry policies for outbound traffic in the MAWI dataset. The solid lines track the resulting flow table size if traditional per-flow state were maintained, with every unique five tuple inserting a table entry for the entirety of the flow's lifetime. This is equivalent to the mean number of flows at the observed link and is further refined according to whether data was traced for the unique five tuple. For domains which exchange traffic with the wider Internet, per-flow state can be particularly crippling as malicious SYN floods and port scans regularly inflate the required state in the network. Such attacks had visible impact in 2011 in particular, nearly doubling the number of flows.

INFLEX however inserts ephemeral rules in response to inflection requests. For the worst possible case, all existing flows would trigger an inflection request simultaneously – matching the overhead incurred by a per-flow approach. In practice even this is overly pessimistic, as an inflector could resort to a per-aggregate granularity in the case of widespread outages. Actual network state would strongly depend on the exact inflection strategy adopted by the transport protocol. One practical reference point is to investigate the resulting overhead if paths were requested on flow start, as this number will exceed retransmission timeouts under normal operating conditions. This is further illustrated in Figure 11, which also tracks flow table size if each unique five tuple were to only generate a flow entry for 1 second, the minimum expiry time for OpenFlow. This is functionally equivalent to the flow arrival rate, and determines the expected number of requests per second sent to the controller. The resulting flow table size is reduced dramatically in comparison to the traditional case where state is allocated for the duration of the flow, and the order of magnitude difference is crucial for software switches in particular. However, under such conditions state becomes more strained by the large fluctuations imposed by DOS attacks, suggesting that inflection requests should only be used after connection establishment; this corresponds to the grey dotted line in Figure 11. Importantly, such an approach also opens the possibility of using inflection requests for assisting traffic management in addition to enabling improved resilience.

VI. Acknowledgements

This research has received funding from the Seventh Framework Programme of the European Commission, through the FLAMINGO Network of Excellence (318488) and the ALIEN (317880) projects.

VII. Conclusions

This paper presented INFLEX, a scalable and easily deployable end-to-end resilience framework based on the cross-layer control of an SDN-enabled network layer. The proposed architecture is shown to perform end-to-end path fail-over on much shorter time scales than existing solutions and is inherently modular, providing failure recovery through cooperation between end-hosts and the IP network. In comparison to reliability mechanisms operating purely at the transport layer, INFLEX enables resilience when communicating with legacy endpoints and does not require host multi-homing. Conversely, when compared to mechanisms operating purely at the network layer, INFLEX provides end-to-end visibility into path failures, allowing both fast detection and fine-grained network control over recovery. The architecture design presented is implemented as a set of extensions to the Linux kernel and a popular OpenFlow controller and evaluated experimentally, demonstrating that high availability over multiple routing planes can be achieved without compromising scalability.

References

[1] R. Stewart, "RFC4960: Stream Control Transmission Protocol," IETF, Sep. 2007, updated by RFCs 6096, 6335.

[2] D. Wischik, M. Handley, and M. Braun, "The resource pooling principle," *ACM SIGCOMM CCR*, vol. 38, no. 5, 2008.

[3] P. Francois, C. Filsfils, J. Evans, and O. Bonaventure, "Achieving sub-second IGP convergence in large IP networks," *ACM SIGCOMM CCR*, vol. 35, no. 3, 2005.

[4] S. Bryant, C. Filsfils, S. Previdi, and M. Shand, "IP Fast Reroute using tunnels," *IETF Internet draft*, 2007.

[5] R. Torvi, A. Atlas, G. Choudhury, A. Zinin, and B. Imhoff, "RFC5286: Basic Specification for IP Fast Reroute: Loop-Free Alternates," IETF, Jan 2008.

[6] S. S. Lor, R. Landa, and M. Rio, "Packet re-cycling: eliminating packet losses due to network failures," in *Proc. of ACM SIGCOMM HotNets*, 2010.

[7] W. W. T. Fok, X. Luo, R. K. P. Mok, W. Li, Y. Liu, E. W. W. Chan, and R. K. C. Chang, "MonoScope: Automating network faults diagnosis based on active measurements," in *Proc. of IFIP/IEEE IM*, 2013.

[8] D. Turner, K. Levchenko, A. C. Snoeren, and S. Savage, "California fault lines: understanding the causes and impact of network failures," *ACM SIGCOMM CCR*, vol. 41, no. 4, Aug. 2010.

[9] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks," *ACM SIGCOMM CCR*, vol. 38, no. 2, Mar. 2008.

[10] "POX OpenFlow Controller," *http://www.noxrepo.org/pox*.

[11] K. Cho, K. Mitsuya, and A. Kato, "Traffic data repository at the WIDE project," in *Proc. of USENIX ATC*, 2000.

[12] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker, "Open vSwitch: Extending networking into the virtualization layer," in *Proc. of ACM SIGCOMM HotNets*, 2009.

[13] B. Heller, R. Sherwood, and N. McKeown, "The controller placement problem," in *Proc. of ACM SIGCOMM HotSDN*, 2012.

[14] N. Wang, K. Ho, G. Pavlou, and M. Howarth, "An overview of routing optimization for internet traffic engineering," *IEEE Communications Surveys & Tutorials*, vol. 10, no. 1, pp. 36–56, 2008.

[15] C. E. Rothenberg, M. R. Nascimento, M. R. Salvador, C. N. A. Corrêa, S. Cunha de Lucena, and R. Raszuk, "Revisiting routing control platforms with the eyes and muscles of software-defined networking," in *Proc. of ACM SIGCOMM HotSDN*, 2012.

[16] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss, "An architecture for differentiated services," 1998.

[17] J. Taveira Araújo, R. Clegg, I. Grandi, M. Rio, and G. Pavlou, "Balancing by PREFLEX: congestion aware traffic engineering," in *Proc. of IFIP NETWORKING*, 2011, pp. 135–149.