DEFT: Distributed, Elastic, and Fault-tolerant State Management of Network Functions

Md Mahir Shahriyar, Gourab Saha, Bishwajit Bhattacharjee, Rezwana Reaz
Department of Computer Science and Engineering, *Bangladesh University of Engineering and Technology*
Dhaka, Bangladesh
Email: {1605024, 1605053, 1605003}@ugrad.cse.buet.ac.bd, rezwana@teacher.cse.buet.ac.bd

***Abstract*—Network function virtualization is the key to developing elastically scalable and fault-tolerant network functions (e.g. load balancer, firewall etc.). By integrating NFV and SDN technologies, it is feasible to dynamically reroute traffic to new network function (NF) instances in the event of an NF failure or overload scenario. The fact that the majority of network functions are stateful makes the task more challenging. State migration and state replication are common approaches in achieving elasticity and fault tolerance. The majority of the studies in the literature either emphasize fault tolerance or elastic scalability while designing a state management system for network functions. In this paper, we have designed a complete state management system, called DEFT, that meets both elasticity and fault-tolerance goals. Our system also supports strong consistency on global state updates. While existing designs rely on a central controller or remote central storage to achieve strong consistency on state updates, DEFT utilizes distributed consensus mechanism to achieve the same. We have done a proof of concept implementation of DEFT and extensively tested DEFT under several model conditions to evaluate its scalability and performance. Our experimental results show that DEFT is scalable and maintains a considerably high throughput throughout. It incurs minimal performance overhead while achieving strong consistency on state updates.**

***Index Terms*—Network function virtualization, Software defined networking, Distributed computing, Fault tolerant systems**

I. Introduction

Network Function Virtualization (NFV) has replaced the dedicated hardware for implementing network functions with virtual machines (VMs) that run on physical servers. Significant benefits of NFV include scalability, elasticity, fault-tolerance, and cost-efficiency. In the rest of this article, a virtual processing unit implementing a network function is referred to as an NF.

A network function is called *stateful* when the processing of a current packet depends on the packets that have been processed earlier. This dependency is carried out by updating a predefined set of parameters while processing packets, called *network function states* or simply *states*. We can categorize states into the following two types.

- Per-flow states: States that are updated by a single flow are called *per-flow states*, also known as *local states*.
- Global states: States that are updated by multiple flows are called *global states*. For example, the number of flows originating from a particular source IP address at a particular time interval is an example of a global state.

NFs can be distributed across different physical servers. Traffic can be distributed on a per-flow or per-packet basis on NFs. *Elastic scaling* of NFs occurs when one or more NFs become overloaded, leading to the relocation of traffic to new NF instances. Failure of an active NF also requires redistribution of traffic to other (replica) NFs.

State management of network functions includes state sharing among the NFs, state migration under scaling operation if states are not shared across NFs, and state recovery due to NF failure. Some pioneering efforts to design NF state management systems are OpenNF [1], Split-Merge [2], StatelessNF [3], S6 [4] and so on.

OpenNF [1] supports state-sharing across NFs through a central controller and provides a loss-free and order-preserving state migration mechanism. Failures of NFs can be handled through replication. However, dependency on the central controller to facilitate state synchronization and state-sharing can lead to a single point of failure and limit scalability.

StatelessNF [3] and S6 [4] avoid state migration during scaling events. In StatelessNF [3], states reside in a low-latency, resilient remote server, eliminating the need for migration during regular or scaling events. However, this approach necessitates remote state access for per-packet processing and presents challenges in maintaining a consistently low-latency and resilient remote server. S6 [4] also avoids state migration as states are stored in a distributed object space and accessible to all NFs. However, read-heavy states may be exported to requesting NF. Thus, S6 does not eliminate state migration completely. The current design of S6 does not consider NF failure.

Fault-tolerance of NF failures can be achieved by replicating NF states. Pioneering efforts that deal with NF failure by NF replication are Pico replication [5], FTMB [6], REINFORCE [7]. In these works, for each NF there exists a replica NF and states are replicated from the active NF to the replica NF. These works do not consider elastic scaling. Since states are not shared across all replicas, explicit state migration is required in the event of elastic scaling and load balancing.

Unlike most of the previous works that primarily focus either on elasticity or NF failure, we aim to integrate both NF failures and elastic scaling while designing a state management system for stateful network functions.

A. Motivation of Our Work

- Distributed State Management under Elasticity: Most existing works focus on techniques to manage NF states (i.e., how the states are shared and migrated during scaling events) that rely on a central controller or remote storage. Therefore, there is a need for state management systems reliant on distributed mechanisms.
- State Management under NF Failure: Existing works ignore NF failures or simply refer to NF replication to recover from failure. Hence, there exists a need for devising detailed mechanisms showing how state replication should work under a chosen consistency model.
- Integration of Elasticity and NF Failure: Elastic scaling requires to deal with active flow migration from one NF to a non-replica NF during scaling events. On the other hand, fault-tolerance requires to deal with replication mechanism to meet the consistency requirement of the system. Thus, there exists a need for a complete state management system that meets both elasticity and fault-tolerance goals.

B. Design Challenges

1) Elasticity Challenges:

E1. Loss Free State Migration: When an active flow is migrated from a current instance to a new instance, the new instance must have the necessary states to process the incoming requests. Any in-flight traffic that reaches the source instance after the migration has started needs to be processed.

E2. Order Preserving State Migration: Per-flow state updates must be done in the order in which packets are received by the switch despite active flow reallocation. Global state updates must be done in the same order across all NFs despite flow migration.

2) Fault-tolerance Challenges:

F1. Loss Free Failure Recovery: State preservation must be guaranteed in the event of NF instance failure or node failure by performing state replication during normal operation. Any in-flight traffic that reaches failed NF instance needs to be processed.

F2. Order Preserving Failure Recovery: Per-flow state updates must be done in the order in which packets are received by the switch despite active flow relocation due to NF failure. Global state updates must be done in the same order across all NFs despite NF failure.

C. Our Contribution

In this paper, we have designed a complete NF state management system, called DEFT, that addresses both elastic scaling and NF failures and holds the following properties.

Before we explain the properties of DEFT, we define the consistency model for our system.

Strong Consistency: State updates are seen by all NFs in the same global order and the global update order of a state is consistent with the local update order of that of a state with respect to an NF.

DEFT Properties: DEFT holds the following properties.

P1. Replicated Global States: Global states are replicated across all NFs in the system.

P2. Distributed State Management: The tasks related to state management are performed without involving any central entity. The update order of global states is solely determined by NFs. Moreover, state migration and failure recovery are done peer-to-peer.

P3. Strongly Consistent Global States: Global states are strongly consistent across NFs.

P4. Order preserving Per-flow State Update: Per-flow states are updated in the order in which the corresponding packets were received by the system irrespective of the flow migration between NFs due to scaling events or NF failure.

P5. Loss-Free and order-preserving state migration and failure recovery.

P6. Loose Flow-Instance Affinity: An instance keeps record of per-flow states while the instance is actively processing the flow. When an instance fails, it only exhibits fail-stop behavior.

II. Literature Review

We divide the literature into two groups. One group of work deals primarily with elastic scaling, and the other group of work focuses on fault-tolerance of network functions.

A. Elastic State Management

State management during scaling events follows two strategies: state migration and migration avoidance.

1) State Migration Strategy: Among different studies that deal with state migration, Split/Merge [2] and OpenNF [1] align with DEFT the most. Split/Merge [2] introduced the concepts of partitioned and coherent NF states. Coherent state updates are periodically merged, and so remain eventually consistent across replicas. However, any in-flight traffic that arrives at the source NF during NF state migration is dropped. OpenNF [1] uses move, copy, and share operations to achieve different levels of consistency in NF states sharing. (specially share operation which manages to achieve strong [8] or strict [9] consistency). However, they all require central controller in order to operate.

2) Migration Avoidance Strategy: An alternative option to state migration is migration avoidance. Two notable studies that follow this strategy are StatelessNF [3] and S6 [4] to various degrees. StatelessNF [3] manages to achieve this by making states accessible to all NFs (global states) through data store. This adds to the latency of the system and makes the data store a single point of failure. In S6, the states reside in distributed shared object (DSO) space and are tied to NFs through a concept called state-instance affinity. While this does not completely eliminate state migration, it minimizes the issues that might arrive with extensive state migration and thus achieves eventual consistency.

B. Fault Tolerant State Management

Pico Replication [5] utilizes frequent checkpointing [10] for failure recovery while FTMB [6] does the opposite. Packets are processed in batches and released after the successful completion of a checkpointing. The model followed by Pico Replication [5] allows it to avoid packet replay during recovery from failure. On the other hand, FTMB [6] manages to achieve high throughput during normal operation following its strategy. Both systems utilize backups to handle the failure of primary NF. Backup restores to the last saved checkpoint upon failure of the primary NF. Pico [5] does not handle in-flight packets which may arrive at the primary during checkpointing.

REINFORCE [7] proposed a fault-tolerant system with frequent batch processing similar to Pico Replication [5] and infrequent checkpointing with packet-replay similar to FTMB [6]. Packets are buffered at a predecessor node and replayed to the backup node in case of failure. After a batch of packets are processed at the primary, only the values of Transmit Timestamp (TxTs) table are transmitted to the backup node and after that, the processed batch is released. Strict checkpointing [10] is imposed instead of lazy checkpointing when the processed batch contains non-deterministic [11] packets.

III. DEFT Design

In our system, a node comprises multiple NF instances, and each NF works either as a primary NF or as the backup of a primary NF. We call the later ones secondary NFs.

We design our system based on the following assumptions.

- An NF instance or a node hosting an NF instance may fail. However, we do not consider any communication link failure.
- Our system tolerates fail-stop but no byzantine failures.
- Packets may get reordered but no packet is lost in the communication channel.
- Traffic distribution, load balancing, and detection of NF failures are handled by an SDN controller. The handling mechanism and the failure recovery of the SDN controller are out of the scope of this paper.
- We assume per-flow distribution of packets among the primary NFs.

We will discuss more on our failure model in Section VI.

The SDN controller includes a Failure Detection Unit (FDU) responsible for identifying NF failures in the network. When a primary NF failure is detected, the FDU instructs the corresponding secondary NF to assume the primary role.

An NF instance is comprised of the following major components: input buffer, output buffer, network function processing unit, state manager, transaction coordinator, and a client to an NF cluster. All NFs in the network form this cluster. We present the architecture of an NF instance of our system in Figure 1.

An input buffer stores the incoming packets until they are processed and the output buffer holds the processed packets until they are released. The network function processing (or simply the processing unit) implements the underlying network function and processes input traffic. The state manager

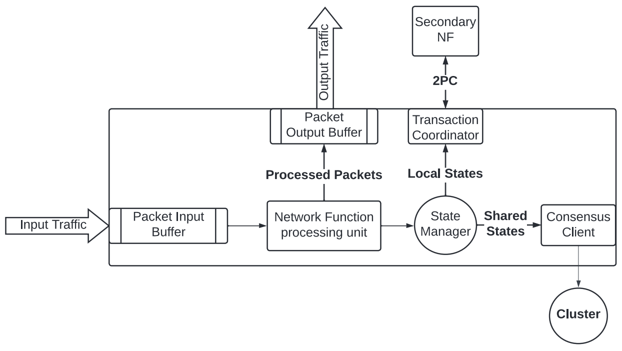


Fig. 1: Detailed Architecture of an NF

of a primary NF delegates the responsibility of sharing state updates with the secondary NF to the transaction coordinator. Whenever the primary NF fills the output buffer, the transaction coordinator initiates an atomic commit between the primary and the secondary NF to share the corresponding state updates. The responsibility of sharing global state updates is delegated to a client in the NF cluster. Whenever an NF wants to update any global state, it initiates a consensus protocol informing all other NFs within the cluster about the update.

IV. Workflow of DEFT

A. Packet Stamping and Duplication

All incoming packets pass through the *stamper module* before reaching NFs. The stamper module has multiple stamping units and a stamper manager. The stamper manager is responsible for distributing incoming packets to the appropriate stamping units based on a predefined hash calculated from the packet's header. This process ensures that packets belonging to the same flow are processed by the same unit. The task of a stamping unit is to assign a unique identifier consisting of two integers (*flow ID*, *per-flow packet counter*), to the packet's payload and forward the packet to the switch. Each stamping unit assigns *flow ID*-s from a disjoint subset of natural numbers. The stamped packet is then forwarded to a primary NF from the switch. The packet is also duplicated and forwarded to the corresponding backup NF.

B. Packet Processing

We want to achieve order-preserved per-flow state updates and strongly consistent global state updates. To achieve this, we want an NF to process the packets of a particular flow in the order they were received by the stamper module. But a packet might get reordered on the path to NF. The architecture in Figure 1 shows that the packets will reside in an input buffer before entering the processing unit. The input buffer can be considered as a FIFO queue.

The processing unit maintains a HashMap called `nextExpectedPktID` to select the packet to process from the input buffer. The `nextExpectedPktID` is a HashMap with *flow ID* as the key and one plus the count of the processed packets so far from the corresponding flow as the value.

A packet with an identifier (*flow ID*, *per-flow packet counter*) *matches* `nextExpectedPktID` if there is an entry in the corresponding HashMap with *flow ID* as its key and *per-flow packet counter* as its value.

The following steps describe the workflow of the packet processing unit.

Step 1: If there is no space in the output buffer, then no packet is retrieved from the input buffer to process.

Step 2: If there exists space in the output buffer, then a packet p is popped from the queue. If the packet identifier of p matches with `nextExpectedPktID`, then p is processed and placed into the output buffer. Also, the value of the corresponding entry in the `nextExpectedPktID` is incremented.

Otherwise, p is placed in a HashMap called `pendingList` with *flow ID* as the key and a priority queue as the value. The priority queue stores the packets that cannot be processed now. The priority queue is sorted by the value field (i.e. *per-flow packet counter*) of the packet identifier.

Step 1 and Step 2 are repeated. Each time a packet from a particular flow is processed, the `pendingList` is checked whether any subsequent packet from the same flow is pending to be processed. If the identifier of a packet p of that flow matches with `nextExpectedPktID`, then p is processed and removed from the `pendingList` and `nextExpectedPktID` is updated accordingly. The process is repeated until there is no pending packet of that flow that can be processed now. After processing and updating the states of a packet, primary NF will store that packet in the output buffer. The size of the output buffer is denoted as the *batch size*.

C. State Update

A packet might update both per-flow states and global states in the system. We can also term the per-flow state updates as local updates.

1) Global Update: Whenever a global update is encountered for any packet p, the primary initiates the consensus protocol with other NFs with state update along with the packet identifier of p. This identifier will only be remembered by the corresponding secondary NF. This is required because if the primary fails before completing the current batch, the secondary will reprocess the corresponding batch and update local states skipping the global updates up to the last received packet identifier. A successful return from a consensus protocol marks the end of processing of packet p and the `nextExpectedPktID` is updated accordingly.
2) Local Update: DEFT achieves order-preserving local state updates. The primary NF processes the packets of a flow in the same exact order in which the switch receives them. All the packets that leave the processing unit reside in the output buffer and are not immediately released.

D. State Replication

When a batch is full, the primary NF shares two pieces of information with the secondary NF: `packet clock` and `state clock`. `packet clock` consists of the most recent HashMap `nextExpectedPktID` after processing a batch and a batch ID t, where t is an integer value. `packet clock` is shared with secondary NF via two-phase commit protocol (2PC) [12]. Packet processing is halted during this time. Upon committing, the primary NF will release the batch from the output buffer. `state clock` consists of three elements: the most recent HashMap `nextExpectedPktID`, corresponding state updates, and a batch ID t, where t is an integer value.

Reason for sending `packet clock`: In case of primary NF failure, secondary NF will replay buffered packets to retrieve states. Since `packet clock` indicates the last batch of packets that the primary NF has released, the secondary NF will not release the same packets.

Reason for sending `state clock`: When the secondary receives `state clock` with batch ID t, it can be guaranteed that the last state updates with which it is consistent with the primary NF correspond to the updates by batch t of packets. So, upon failure, the secondary can process packets from the next batch to avoid duplicate updates of the same states.

V. Elastic Scaling

When any NF A gets overloaded, some flows need to be directed to a different NF B. Only relocation of flows is not enough as NF B does not possess all the states related to these flows. We discuss the scaling procedure below.

We consider that packets of a particular flow will be directed to a particular NF i.e. no packet of the same flow will be distributed among multiple primary NFs. So, if we want to scale, we require some set S of flows that were forwarded towards A to be now directed towards B. Let, s be a flow and $s \in S$. We now describe the methodology we follow while migrating flow s from NF A to NF B.

First, we start buffering the packets of flow s at NF A. Then we migrate the states pertinent to flow s to NF B via 2PC. After the migration is complete, both NF A and B have the necessary states to process the packets of flow s. The SDN controller then changes the flow rule at the switch so that the packets of flow s are now forwarded towards NF B. So, all new packets of flow s will now arrive at B instead of A. Finally, NF A will forward the buffered packets of s towards the switch and these forwarded packets too will be sent towards NF B. If there are any in-flight packets of flow s, they will be forwarded to B after reaching A.

We can show that no packet will be lost during or after state migration and per-flow state updates preserve the order in which the corresponding packets were received by the switch. The first proof is trivial because none of the packets that arrive at A are dropped and directed towards B later on. For the second proof, we have to understand that the new packets of flow s which are sent from the switch to B have higher identifiers than the packets of flow s that are forwarded from A to B. So, even though the new packets of flow s sent from the switch to B arrive before the packets forwarded from A to B, these packets would reside in the NF B's input buffer. NF B will process a packet of flow s if the identifier of that packet matches the `nextExpectedPktID` of NF B. Note that during the state migration, NF A will also share its `nextExpectedPktID` to B and B will update its own `nextExpectedPktID` accordingly. As a result, NF B knows the count of the last packet of flow s that has been processed by NF A.

VI. Fault Tolerance

Achieving fault tolerance is a prime objective of DEFT design. Studies have shown that middlebox-failure [13], [14] and software-failure [15], [16] happen quite often in a system. In our system, we consider both NF instance failure (software failure) and node failure.

Before getting into failure recovery we need to clarify some of the assumptions we make in our architecture.

- The failure detection unit (FDU) frequently checks for any form of NF failure. Handling the failure of the FDU unit is out of the scope of this work.
- Primary NF and the corresponding secondary NF do not reside on the same physical node and do not fail simultaneously.
- We only consider the crash failure of NFs in our system.

Now we discuss our failure recovery mechanism.

A. NF Failure Recovery

When a primary NF fails, the corresponding secondary NF takes over and continues packet processing. First, the FDU unit acknowledges primary NF A failure and informs the corresponding secondary NF B. Next, it assigns NF B as the new primary and NF C as the new secondary NF. Now, NF B has to migrate the state updates along with the last received `nextExpectedPktID` from NF A and the buffered packets to NF C so that the new secondary NF becomes consistent with the new primary NF. Note that, we would not incur any packet loss here as duplicate packets are always forwarded to the corresponding secondary NF and in this case, all the packets that already reached (or in-flight) to NF A but have not been processed due to the failure of A are also sent to NF B. Now, NF B needs to process these buffered packets. The SDN controller changes the flow rule at the switch such that all the packets related to that flow would now be forwarded to NF B and duplicate packets would be sent to the newly assigned secondary NF C.

Let the last received `packet clock` by NF B from NF A has a batch ID i. Also, let the last received `state clock` by NF B from NF A has a batch ID j. Now, one of the following two cases occurs.

1) Case 1: If $i = j$. In this case, NF B starts processing packets, updating states, and releasing packets according to the `nextExpectedPktID` value in the `packet clock`.
2) Case 2: $i = j + 1$. In this case, NF B starts processing packets and updating states according to the `nextExpectedPktID` value in the `state clock`. However, packets are released according to the `nextExpectedPktID` value in the `packet clock`.

All the newly processed packets' per-flow states will be shared with the newly assigned secondary NF C just like before. By following this mechanism, we can both ensure loss-free packet processing and achieve order-preserving per-flow state updates even if primary NF fails at any time.

If any secondary NF fails, the SDN controller would perform the following. 1) Assign a new secondary NF and change the forwarding rule at the switch such that duplicate packets will now be forwarded to the newly assigned secondary NF; 2) Instruct primary NF to share its states with the newly assigned secondary NF via 2PC. Packet processing will be halted during this transaction.

B. Node Failure Recovery

Our system distributes network function (NF) instances ensuring primary and secondary instances of the same NF always run on different nodes. This approach guarantees that if a node fails along with all the NFs hosted on it, the system will continue to operate normally.

Our system can tolerate the simultaneous failure of multiple primary NFs residing in the same node or across several nodes. It can also tolerate simultaneous failure of multiple nodes as long as the failed nodes do not host the primary and backup instances of the same NF.

C. Failure of Stamper Module

The stamper module consists of multiple stamping units and a stamper manager. In the event of a single stamping unit failure, only the flows assigned to that particular unit will be impacted (incoming packets will be dropped). Suppose, the failure of a stamping unit su impacts flow f. Upon recovery of su, the remaining packets of flow f will reach su due to predefined hashing. Flow f will now be recognized as a new flow by su.

To mitigate single-point failures, we can increase the number of stamping units, distribute them across multiple nodes, and replicate each unit as needed. If the stamper manager fails, incoming packets will be dropped momentarily. Upon recovery, the manager ensures that packets belonging to the same flow are always forwarded to the same stamping unit. A backup manager can reduce the impact of failure.

VII. Experiments and Results

In our experiments[1], we used latency, throughput, and packet drop as our evaluation metrics.

[1]All of our experiments were conducted on an Intel® Core™ i7-10700K CPU @ 3.80GHz CPU with 16 cores and 64GB RAM, Ubuntu 18.04.6 LTS. Available at https://github.com/MahirSez/DEFT

A. Local State Update

In the following experiments, we investigate the optimal values for batch size and buffer size in our system.

1) Small vs large batch size. Which one and why?: As our system processes packets in batches, we first aim to determine the optimal batch size for our system operation.

We start with a small batch size of 10 packets and send the traffic at 10,000 packets/second to the system. When the batch size is very low, frequent local state sharing increases wait time in the input buffer resulting in increased latency. As we increase the batch size, latency decreases. But after a certain batch size limit (50 packets), DEFT's latency starts to increase as packets now have to stay in the buffer for a longer period of time. Hence, we determine the optimal batch size for DEFT is 50 at which point the system achieves its lowest latency of 2.41 ms. We use this batch size in all successive experiments. An illustration of this behavior can be seen in Figure 2a.

We then try to find the effect of packet rate on this optimum batch size. Increasing this traffic from 8,000 to 10,000 packets/second increases the rate at which DEFT processes them from 7,950 to 9,906 packets/second respectively. This trend in the increase in throughput retains till the packet rate exceeds 27,000 packets/second, beyond which the packet processing rate does not seem to increase linearly as before. Figure 2b illustrates this behavior graphically.

2) Buffer size is theoretically infinite. But do we really need it?: We next evaluate the significance of input buffer size and how it impacts packet drop in DEFT.

With a input buffer size equals to the batch size (50 packets), when we send the traffic at 14,000 packets/second, the system drops 257 packets out of the 50,000 packets it processed (about 0.51%). DEFT manages to process the whole traffic without dropping any packets at a buffer size equals 5 times of the batch size. This concludes that 5 batches of packets per input buffer are sufficient for the system to encounter zero packet loss. Figure 2c illustrates this behavior.

From Figure 2c, we can also see that with an increase in buffer size for any given packet rate, the number of packets dropped starts to reduce. This is because more packets can be accommodated in the buffer. Additionally, for any given buffer size, with an increase in the packet rate, the number of packets dropped starts to increase.

B. Global State Update

In this section, we showcase DEFT's robustness in handling global state updates and illustrate how the system performs under the burden of running consensus algorithms of greater magnitude.

1) How frequently should we perform global updates?: When the input traffic is 6,000 packets/second with a single consensus per batch, the system does not seem to get impacted at all and processes the traffic at around 6,000 packets/second. A significant change in throughput is seen (5,484 packets/second) when the traffic is raised to 10,000 packets/second with a global update frequency of 10 per batch. Figure 2d illustrates this behavior in detail.

2) How well does DEFT tackle a heavy consensus?: We examine DEFT's behavior at a micro-level, testing its resilience to heavy global state updates. Under normal conditions, DEFT exhibits a latency of around 2.7 ms. When subjected to 100 consecutive global state updates, it experiences a brief drop in throughput to 1.9 kpps for 100 ms, followed by a rapid recovery to its normal processing rate of 10 kpps. This highlights DEFT's ability to quickly bounce back from temporary latency and throughput challenges caused by intense global state updates. We illustrate this behavior through a timelapse graph in Figure 3.

VIII. Discussion

We tested the scalability and performance of DEFT in scenarios like amping up the packet sending rate, increasing the rate of global updates, and imposing heavy global updates to stress test the system.

We found the following optimal values for the system under the current experimental setup:

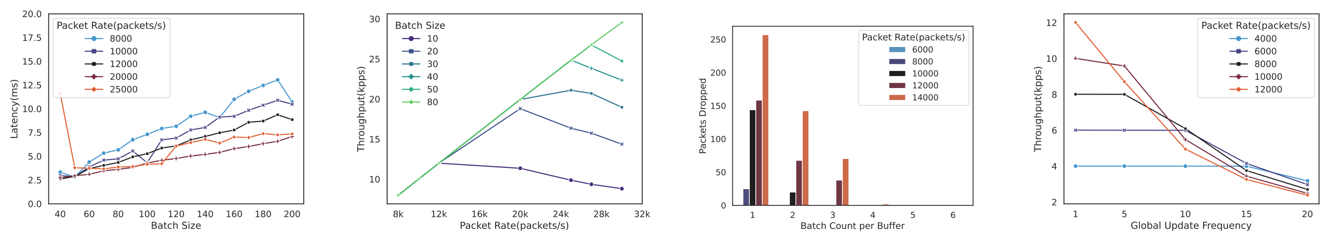
- We found that a batch size of 50 is optimal for DEFT. Decreasing the batch size from this increases the rate of local state updates and increasing it makes the packets wait in the buffer for a longer period of time.
- The optimal input buffer size for DEFT is 5 batches of packets. Keeping the buffer size to any lower value incurred packet loss in the system.
- For a moderate packet rate (4,000-6,000 packets/second), DEFT's throughput and latency are invariant of global state update frequency.

IX. Conclusion and Future Work

While the relevant research on state management systems focuses on either fault-tolerance or elastic scaling, we have designed a complete state management system, DEFT, that is fault-tolerant and supports elastic scaling. DEFT achieves strong consistency on global state updates in a distributed manner. DEFT also guarantees loss-free and order-preserving state migration and failure recovery. Our experiments show that DEFT achieves considerably high throughput under several model conditions. We also observe that DEFT can achieve strong consistency with minimal performance overhead.

In this work, our primary objective was to bring down the design goals under one roof when dealing with both elastic scaling and fault-tolerance and present mechanisms to deal with states without any central dependency in a comprehensive manner. However, we understand that there are still ways to improve, which we leave as future work. Here are some of the ways.

1) We want to integrate practical and commercially available NFs like Bro IDS [17], and PRADS [18] in our implementation.
2) Existing systems have different design goals and properties. We want to develop a generalized and streamlined



(a) (b) (c) (d)

Fig. 2: (a) Increasing batch size reduces latency up to a certain threshold. (b) Batch sizes and packet rates linearly affect throughput until a threshold. (c) Change in input buffer size affects the number of dropped packets. (d) With increasing packet rates, global updates have a more pronounced impact on throughput, with a lesser effect at moderate rates.

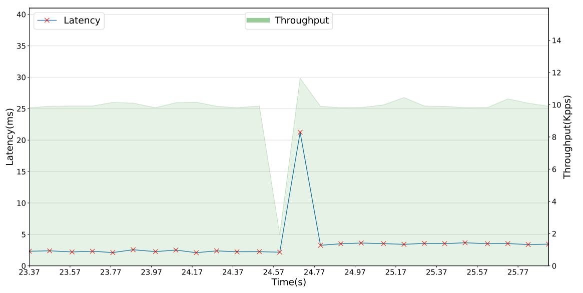


Fig. 3: The figure shows the impact of a heavy global update on DEFT in a granular timescale.

evaluation mechanism to test performance of one system against another.

3) Our current configuration has global states shared among all the participants. However, all global states may not be associated with every participant. As such, it adds unnecessary overhead by having more participants than required. We can resolve this issue by having a cluster-based state update for the global states. In this cluster-based design, only the NFs concerned with a given state will be included in the cluster for that state.

References

[1] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, "Opennf: Enabling innovation in network function control," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 163–174, 2014.

[2] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, "Split/merge: System support for elastic execution in virtual middleboxes," in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013, pp. 227–240.

[3] M. Kablan, A. Alsudais, E. Keller, and F. Le, "Stateless network functions: Breaking the tight coupling of state and processing," in *Proccedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 97–112.

[4] S. Woo, J. Sherry, S. Han, S. Moon, S. Ratnasamy, and S. Shenker, "Elastic scaling of stateful network functions," in *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018, pp. 299–312.

[5] S. Rajagopalan, D. Williams, and H. Jamjoom, "Pico replication: A high availability framework for middleboxes," in *Proceedings of the 4th annual Symposium on Cloud Computing*, 2013, pp. 1–15.

[6] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. Martins, S. Ratnasamy, L. Rizzo *et al.*, "Rollback-recovery for middleboxes," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, 2015, pp. 227–240.

[7] S. G. Kulkarni, G. Liu, K. Ramakrishnan, M. Arumaithurai, T. Wood, and X. Fu, "Reinforce: Achieving efficient failure resiliency for network function virtualization based services," in *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, 2018, pp. 41–53.

[8] M. Vukolic, "Eventually returning to strong consistency," *IEEE Data Eng. Bull.*, vol. 39, no. 1, pp. 39–44, 2016.

[9] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *Computer*, vol. 29, no. 12, pp. 66–76, 1996.

[10] P. J. S., "Libckpt : Transparent checkpointing under unix," *Proceedings of the Usenix Winter Technical Conference*, pp. 213–223, 1995. [Online]. Available: https://cir.nii.ac.jp/crid/1573387449739290752

[11] C. Cachin, S. Schubert, and M. Vukolić, "Non-determinism in byzantine fault-tolerant replication," *arXiv preprint arXiv:1603.07351*, 2016.

[12] G. Samaras, K. Britton, A. Citron, and C. Mohan, "Two-phase commit optimizations in a commercial distributed environment," *Distributed and Parallel Databases*, vol. 3, no. 4, pp. 325–360, 1995.

[13] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: measurement, analysis, and implications," in *Proceedings of the ACM SIGCOMM 2011 Conference*, 2011, pp. 350–361.

[14] R. Potharaju and N. Jain, "Demystifying the dark side of the middle: a field study of middlebox failures in datacenters," in *Proceedings of the 2013 conference on Internet measurement conference*, 2013, pp. 9–22.

[15] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin, and A. D. Satria, "What bugs live in the cloud? a study of 3000+ issues in cloud systems," in *Proceedings of the ACM symposium on cloud computing*, 2014, pp. 1–14.

[16] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. Eliazar, "Why does the cloud stop computing? lessons from hundreds of service outages," in *Proceedings of the 7th ACM Symposium on Cloud Computing*, 2016, pp. 1–16.

[17] V. Paxson, "Bro: a system for detecting network intruders in real-time," *Computer networks*, vol. 31, no. 23-24, pp. 2435–2463, 1999.

[18] Passive real-time asset detection system. [Online]. Available: https://github.com/gamelinux/prads