Anolis: A Scalable, Fault-Tolerant Load Balancer for Stateful Network Functions

Paper #63, 12 pages

Abstract

Layer-4 load balancers deployed in an ISP's network cloud need to provide the affinity of inbound and outbound connections to stateful network functions. Anolis is the first high-performance system that meets this goal in the presence of failures of load balancer nodes and concurrent elastic scaling of the load balancer and network function nodes. Its design integrates packet forwarding with replication to efficiently replicate each connection's entries at a chain of Anolis nodes. A key challenge in designing Anolis is to elastically scale the system by managing an ensemble of these chains overlaid on a consistent hashing ring. Anolis's replication protocol meets this challenge and provably guarantees connection affinity despite failures, additions or removals of Anolis nodes. In experiments with a DPDK-based implementation, Anolis's incurs a marginal latency of 15 us per replica, supports 2 M new connections/second with 14 nodes, and maintains all TCP sessions during reconfigurations.

1 INTRODUCTION

Stateful network functions (NFs) need to support massive amounts of state and line rate forwarding performance at low latency. Their state management is likely to grow even more challenging given that Internet-connected devices are expected to grow to several tens of billions over the next decade [14]. Hence, a key technical requirement for future scaling of these stateful NFs is a massive, high-performance flow table that supports several hundred million entries.

Large flow tables are more practical to implement in software than in hardware because of 100s of GBs of RAM on servers. In comparison, a state-of-the-art hardware switch supports just 100 MB of SRAM [8]. However, software-based NFs have relatively low per-node forwarding performance of tens of Gbps, which necessitates a scale-out deployment of stateful NFs and their constituent flow tables. A scale-out design brings forth the need to support seamless elasticity as well as fault tolerance of nodes.

For an ISP's network cloud, we consider the design of one such NF that has received a lot of attention recently – a stateful L4 load balancer. The first category of these efforts consists of cloud L4 load balancing schemes, which seek to maintain connection affinity via *local packet forwarding* [2, 10]. As we discuss in Section 7, schemes in this category fail to meet the affinity requirements of an ISP's network cloud, especially during failures and concurrent elastic scaling of load balancers and NFs. The second category consists of schemes for *distributed state management* in NFs on top of which L4 load balancers can be built. While these schemes can meet either elasticity [3, 5, 12] or fault-tolerance [13] or both [7], most of these systems demonstrate scenarios with few 10K flows and hence their scalability even to a million flows is largely untested (Section 7).

In this paper, we present a new load balancer Anolis that meets all of the above goals for an ISP's network cloud. It guarantees affinity for connections originating towards the NF (inbound) or from the NF (outbound) despite concurrent changes in the set of Anolis and NF nodes. It tolerates a configurable number of failures at a cost that is in proportion to the number of failed replicas. Elastic scaling of its nodes linearly increases the forwarding throughput as well as the connection table size. Finally, it provides low, predictable per-packet latency commensurate with the hardware capability (e.g., up to tens of microseconds on commodity servers and NICs [16]), a throughput of several million packets per second per core, and a flow table that can scale up to the allocated memory at a server. These capabilities make it useful for deployment in an ISP's network cloud as a front-end load balancer to safely connect stateless network devices such as hardware routers to stateful NFs.

A key principle in Anolis's design is to *integrate forwarding with replication* unlike above efforts that separate local packet forwarding from distributed state management. A packet, along with some replication metadata, is forwarded along the Anolis nodes where its connection state is to be replicated. At each node, the packet is read from the incoming queue, processed locally at a predictable computation cost and latency, and forwarded to the next node. This idea of "chain replication" was originally proposed in 2004 [15] and has recently been used for in-network state replication in hardware switches [6]. Our work applies it to enable flow table replication in software data planes while maintaining high-performance run-to-completion packet processing at individual nodes.

We extend prior work on chain replication in three ways by leveraging the fact that a load balancer's flow entry does not need to change the NF assigned to a connection during its lifetime. First, we show that fully replicated entries can be safely cached without violating affinity. Second, we enable arbitrary chain reconfigurations (node additions and removals) in completely non-blocking manner. Prior work (e.g., Netchain) blocks writes and reads to avoid inconsistent

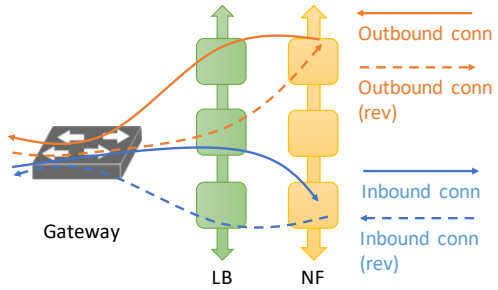


Figure 1: Load balancing of stateful NFs requires both inbound and outbound connection affinity in presence of changing sets of LB and NF instances.

entries during reconfigurations. But, we allow the inconsistencies to happen during query processing and let our sync protocol correct any inconsistencies as long as a single common node exists with the previous chain. Third, we relax the reliable FIFO channel assumption in the chain replication protocol to reduce the overhead of the TCP stack during query processing.

Our paper makes the following contributions:

(1) We outline the requirements of a network cloud load balancer including inbound and outbound affinity. We show that an unreplicated distributed flow table with flow caching can ensure affinity with a small performance impact (Sections 2 & 3).

(2) We present a new chain replication protocol to replicate L4 load balancer flow tables. It supports completely non-blocking chain reconfiguration by means of state synchronization operations that are computed and orchestrated by a controller (Section 4).

(3) We evaluate a DPDK-based prototype to quantify the trade-off between replication and performance, the impact of horizontal scaling and TCP behavior during reconfiguration using a stateful traffic generator on multiple testbeds (Sections 5 & 6).

Our experiments show that the marginal latency of adding a chain node is 15 us per replica and is incurred only for the first packet in a flow or once every keep-alive interval. Our prototype can achieve up to 4 Mpps per node forwarding throughput per node for longer flows and can linearly scale the throughput and flow table size in experiments with 14 nodes. We show a peak performance of close to 2 million new connections per second and 120 M concurrent flow table entries. In an experiment with TCP traffic, Anolis completely avoids disruption to any connection despite reconfigurations.

2 NETWORK CLOUD LOAD BALANCING

We consider an ISP's network cloud datacenter consisting of a stateless gateway router that connects to the external network (Figure 1). The gateway router routes incoming traffic from the external network to a set of L4 load balancers. It performs equal cost multipath (ECMP) among the load balancer nodes based on a unique connection key, e.g., the hash value of an incoming packet header's 5-tuple. It routes traffic received from the load balancer to the external network. The L4 load balancer sends traffic to a set of stateful network function instances (e.g, NAT, or Proxy). The set of network function instances is not fixed and changes according to input traffic and scaling policies.

A connection is uniquely identified by its 5-tuple. New connections, i.e., packets with a unique five-tuple, may be initiated from the network function to be sent to the external network, or may be initiated from the external network towards the network function. Once initiated, a connection may send traffic in both directions. Network functions require *connection affinity*, i.e., all packets in a connection in both directions must be processed by the same network function instance. Failure to do so results in the connection being dropped and causes significant user-visible disruptions.

2.1 Design goals

An L4 load balancer must meet these design goals to be practicable for an ISP's network cloud:

1. Outbound connection affinity: If a connection is initiated by a stateful NF node to the external network, all packets in the reverse flow of the connection must be routed to the same NF node.

2. Inbound connection affinity: If a packet of an inbound connection is routed to an NF node, all subsequent inbound packets must be routed to the same NF node.

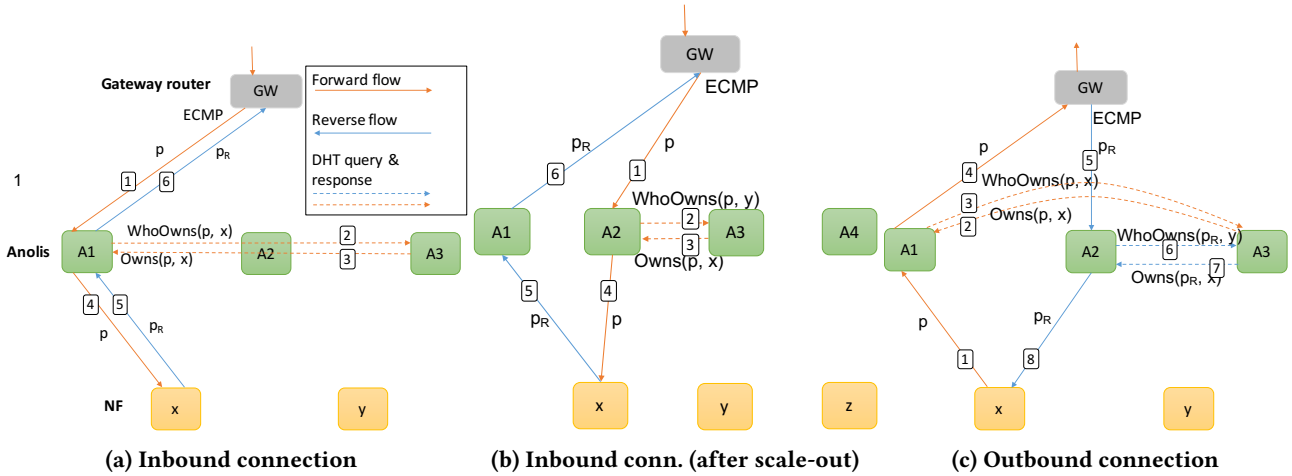
3. High throughput: The system must achieve a high forwarding throughput in terms of packets per second.

4. Horizontal scaling: The system should support a linear increase in the number of connections and forwarding throughput by increasing the number of load balancer nodes.

5. Reconfigurability: A planned increase or unplanned failures in the number of load balancer nodes up to a configured number of replicas should not interrupt packet processing or violate connection affinity.

3 ANOLIS CONNECTION AFFINITY

Anolis is a stateful load balancer that maintains connection entries in a distributed flow table. A distributed table enables any Anolis node to create or read any connection entry. This section shows how our approach can help meet its design goals of inbound and outbound connection affinity. We show how caching of flow entries enables efficient packet forwarding in the common case. Section 4 shows how Anolis supports the goals of scaling and reconfiguration.



(a) Inbound connection (b) Inbound conn. (after scale-out) (c) Outbound connection

Figure 2: Anolis preserves both inbound and outbound affinity despite concurrent scaling of NF and Anolis nodes.

3.1 System overview

Figure 2 illustrates a typical deployment of Anolis nodes (also called Anolis peers) as a load balancing element between a stateless gateway router and a changing set of stateful NF nodes. An Anolis node can forward traffic to any of the NF nodes. But, an NF node configures one of the Anolis nodes as its default gateway to transmit all its traffic. If that Anolis fails, the NF updates the default gateway to another Anolis node. A centralized controller (not shown in Figure 2) maintains the list of current Anolis nodes and communicates this list to the gateway, the NFs, and all Anolis nodes.

Anolis nodes implement a distributed flow table amongst themselves to store connection state. The main operation this table provides is an asynchronous call *WhoOwns(p, nf)* where p is the packet and nf is the identifier (ID) of the NF proposed to handle this connection. If the flow table has no entry corresponding to the connection represented by p, nf is selected to be the processing node for this connection and a response *Owns(p, nf)* is returned to the calling node. If the distributed table has already stored nf' to be the processing node for this connection, a response *Owns(p, nf')* is returned to the calling node. Executing WhoOwns may transmit queries to other nodes. But, Anolis nodes cache responses so that repeated WhoOwns queries for the same connection return the cached value.

3.2 Inbound and outbound affinity

Figure 2a discusses the case of an inbound connection. The gateway sends the first packet of a new connection to an Anolis node A_1 based on the packet's hash value. A_1 makes a WhoOwns query to its local flow table to find that an entry does not yet exist for this connection. Since an entry may still exist for this connection at another Anolis node, the *WhoOwns* query is sent to A_3 after selecting an NF node x as the owner. On receiving a *Owns* reply, the entry in the local flow table at A_1 and the packet included in the reply is forwarded to the NF x. Note that subsequent packets in the forward direction do not require a distributed query to A_3 and are directly forwarded to NF x. Packets in the reverse flow for this connection from NF x reach A_1 as A_1 is configured as the default gateway for NF x. Anolis associates the reverse flow to the forward flow from the entry in its flow table and forwards the traffic to the gateway.

A scale-out event can cause an existing inbound connection's forward flow to be routed to a different Anolis node. As shown in Figure 2b, A_2 receives the packet after the addition of the node A_4. A_2 queries the distributed flow table to obtain the NF ID to route the packet to, and sends the packet to the correct NF node x. A_2 also caches the flow entry locally to avoid distributed queries for subsequent packets. Note that the route for the reverse flow from the NF is unchanged, and it is routed via the Anolis node A_1.

In Figure 2c, an outbound connection from the NF x is routed via A_1 configured as the default gateway of x. The first packet of this connection requires Anolis to create a new entry storing the NF x as the owner for this connection. The WhoOwns query reaches A_3, where A_3 ascertains it is indeed a new connection and inserts the new entry. The Owns reply is sent to $A1$, which then forwards the packet contained in the reply.

The reverse flow of an outbound connection is equally likely to be sent to any of the Anolis nodes due to ECMP routing by the gateway. As shown in Figure 2c, if the Anolis node receiving the flow (A_2) is different from the one that handled the forward flow (A_1), A_3 sends a WhoOwns query to obtain the NF ID and then sends the packet to the correct

NF. Again, responses are cached locally to avoid distributed queries for subsequent packets.

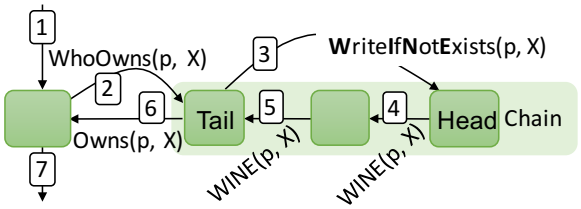


Figure 3: Anolis integrates forwarding & replication.

4 ANOLIS REPLICATION PROTOCOL

Anolis's replication protocol enables completely non-blocking reconfiguration of an ensemble of chains on a consistent hashing ring. We first introduce our approach and illustrate its challenges, present our terminology and assumptions, describe the protocol, and prove that it's non-blocking reconfigurations maintain connection affinity.

4.1 Replication approach and challenges

Our approach integrates forwarding with replication using chain replication, which enables run-to-completion packet processing at each node. Applying chain replication to our problem, the WhoOwns query is first sent to the tail node as shown in Figure 3 which returns the next hop NF if an entry exists for the connection. Otherwise, the tail node starts an insert request for the connection via the head node. The connection's entry is then written at all the nodes in the chain including the tail node. The tail node forwards the response to the sending Anolis node, which then sends to the next hop NF.

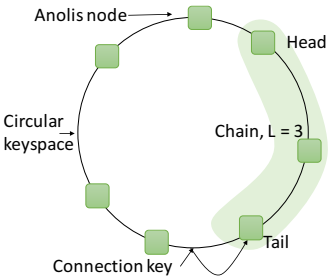


Figure 4: Combining chain replication with DHT.

In principle, we can horizontally scale chain replication by creating multiple chain instances and assigning parts of the keyspace to these chains using consistent hashing. But, the challenge is to reconfigure Anolis's chains upon failure, addition or removal of nodes while providing the consistency required for this problem. While a similar problem is addressed in FAWN [1, 11] and Netchain [6], both these systems block write and sometimes reads to synchronize state to new nodes. Since such blocking reconfigurations may cause long disruption in a production network, we design a new non-blocking reconfiguration protocol for our load balancer that addresses these challenges:

(1) Divergent views of Anolis peer list: A new peer list is first determined by the controller and then propagated to all nodes in the system. In this time window, nodes can have divergent views of the peer list. As a result, nodes can incorrectly compute the head or tail nodes for a particular connection's key, and thereby can make incorrect forwarding and replication decisions.

(2) Concurrent chain reconfigurations: Chains must support concurrent reconfigurations at arbitrary positions in the chain, e.g., insert the new node at the 2nd chain node and remove the 4th chain node. To do so, a chain must correct any inconsistent or missing entries at its nodes as a result of these changes.

(3) Cross-chain operations: The number of unique chains in the system is equal to the number of Anolis nodes. Hence, the addition (removal) of a node adds (removes) one more chain to (from) the system. These reconfiguration events require performing cross-chain operations including splitting and merging of chains without violating the affinity or fault-tolerance goals of the system.

The protocol we present next addresses these challenges.

4.2 Protocol preliminaries

The chain nodes that store a connection's entry (Figure 4) are computed based on three parameters: the unique connection key derived from the 5-tuple of each packet header, the node ID of Anolis nodes, and the length **L** of the chain. The tail of a connection's chain is computed by mapping the connection's key (5-tuple) to the circular keyspace and subsequently to an Anolis node using a consistent hash function. The head of the chain is the (L-1)-th predecessor of the tail node on the ring. A connection's entry may additionally be cached at other nodes that are forwarding packets of that connection.

The controller assigns unique **node IDs** to nodes and unique **version IDs** to Anolis peer lists upon each reconfiguration event (node additions or removals). The controller executes a *reconfiguration* in two phases. First, it sends the new peer list to all Anolis nodes, the NFs, and the gateway. Second, it initiates a set of **sync** operations so that flow table entries for all keys are replicated at the corresponding chain nodes. If all the syncs complete before further reconfigurations occur, the current version ID is marked as **synced**.

The sync operation is subject to the existence of a **first common node (FCN)** for a chain. In Figure 5, consider the list of chains for a key across all peer list versions starting from the last synced version (version 6) up to the current

```
1  WriteIfNotExists(pkt, nf, fwdNode, lastNode):
2  if lastNode = null & myId != head(nodeList, pkt, L), return
3  if myId or lastNode not in chain(nodeList, pkt, L), return
4  if (nf' <- table.read(k)) = null, table.insert(pkt, nf),
       else if nf' != nf, nf <- nf'
5  send WriteIfNotExists(pkt, nf, fwdNode, myId) to nextInChain
       (nodeList, pkt, L, myId) if exists; else send Owns(pkt,
        nf) to fwdNode
```

Figure 6: Query protocol: Listing for WriteIfNotExists

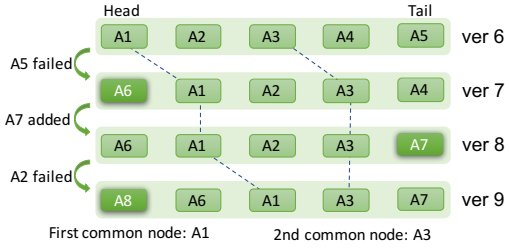


Figure 5: Identifying common nodes for a chain across peer list versions.

version (version 9). FCN is defined as the first node in the first chain (starting from the head node) that is present in every subsequent chain in this list. By this definition, A_1 is the FCN in this example.

Assumptions: Anolis meets its stated design goals under the following assumptions.

Keep-alive: A connection successfully transmits a packet once every keep-alive interval that is on the order of minutes.

Reconfiguration: From one synced version of the peer list to its next synced version, there always exists a first common node for any key in the system. For example, a failure of all but one node in a chain is permissible as long as there is enough time for the system to reach a synced version.

Failure: Anolis nodes follow crash failure semantics, where the controller is unable to reliably detect whether an Anolis node has failed. The controller can log its execution state persistently. Upon a failure, the controller eventually restarts and recovers its execution state.

Memory: A node's local flow table has sufficient capacity to store all the chain entries assigned to it. If a node is running out of space in its table, the controller is able to trigger out a scale-out action to reduce the entries assigned to that node.

4.3 Protocol descriptions

Anolis's distributed flow table design is comprised of three protocols. The *query protocol* implements the WhoOwns and Delete queries (§4.3.1). The *sync protocol* defines the operation of Anolis nodes in response to reconfigurations (§4.3.2). The *controller protocol* defines the actions of the Anolis controller to maintain peer lists and orchestrate sync operations (§4.3.3).

4.3.1 Query protocol. We continue the discussion of WhoOwns execution from Section 4.1 and additionally focus on deletions and changes needed to support chain reconfigurations changes?. Unlike the other two protocols, the query protocol does not assume that its messages are sent over reliable FIFO channels which further simplifies its implementation.

Insertion: Figure 6 shows the pseudo-code for inserting a connection entry. The tail node starts insertion with the message WriteIfNotExists to the head node. If the insertion does not begin with the head node, the head node drops the packet. If a packet is forwarded to or from a non-chain member, the receiving node drops the packet. Thus, replication is aborted if the packet reaches a non-chain node due to reconfigurations. If a node has an entry for the packet's connection with a different NF ID nf', then the parameter nf is changed to nf' prior to forwarding. It is necessary to do so since a prior packet for this connection may have been forwarded to nf'. WriteIfNotExists is forwarded through the chain to the tail which sends the response to the fwdNode.

Caching: If WriteIfNotExists executes successfully till the tail node, the entry at the tail node can safely be cached. For subsequent WhoOwns queries for this connection, Anolis's is expected to always return the NF ID stored at the tail node. Further, Anolis guards against using an entry at a non-tail node to forward packets. This is accomplished by adding a single cache-able bit to each connection's entry. This bit is set to 1 in the entry stored at the tail node or if the entry is obtained from the tail node in response to a WhoOwns query. Otherwise, this bit is set to 0 at other nodes, implying that those entries cannot be used to forward packets.

Deletion: Anolis treats flow entries as soft-state to handle their deletion. Deletion is implemented using periodic scans of the flow table at each node at equal-length *deletion intervals*, which is a fraction of the keep-alive interval for Anolis. Each node independently maintain counter c which represents the current period. Each flow entry at a node also keeps a counter which represents the period when this flow entry was last used. A flow entry is deleted if it was last used at least one keep-alive interval ago.

The last used cycle for a flow entry at chain and other nodes is updated as follows. When a node receives an external packet, it checks if the counter in the local flow entry (if any) matches the current period. If the local flow entry was last used in a previous period, the packet is forwarded through its chain as if it is the first packet of a flow. This allows the chain nodes to update their flow table entries upon arrival of packets every keep-alive interval. To ensure that last used counter are updated at least once every keep-alive

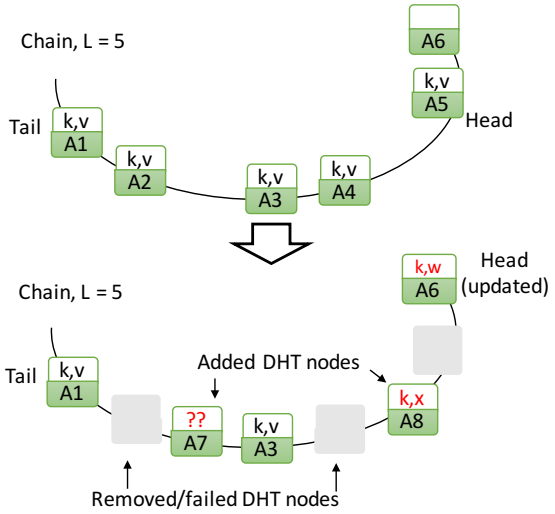


Figure 7: Reconfiguration can cause missing entries (at A7) and inconsistent entries (at A8 and A6).

interval, the deletion interval is set to half of the keep-alive interval or lower.

4.3.2 Sync protocol. This protocol transfers state to new chain nodes and corrects data inconsistencies due to reconfigurations. Figure 7 gives an example of reconfiguration and how it affects the data at chain nodes. Prior to reconfiguration, all L (=5) chain nodes have the same NF ID v for the connection key k. During a sequence of reconfigurations, nodes A2, A4 and A5 from the original chain either failed or were removed. Two new Anolis nodes A7 and A8 joined the chain. An existing Anolis node A6 became the new head of the chain. After these changes, A7 is missing an entry for the key k. More problematically, the nodes A6 and A8 have NF IDs x and w that are different from the original NF ID v. Such inconsistencies can arise since these nodes did not have any entry for the key and hence accepted the values proposed in the WriteIfNotExists queries. Now, the task of the sync protocol is to restore the entry (k,v) at all chain nodes.

The sync process is initiated by a message sync(syncId, kLow, kHigh) from the controller to an Anolis node. SyncID is assigned by the controller and it uniquely identifies this sync operation from others. All keys in the range kLow (exclusive), kHigh (inclusive) were part of the same chains in all reconfigurations since the last synced version ID. Further, this Anolis node is the first common node (FCN) in all of those chains. In the above example, the third node in the chain is the FCN. The FCN responds to the sync message by broadcasting all entries in its flow table in this key range to all other chain nodes. The recipient chain nodes insert these entries in their tables overwriting the existing entries if any, and then send an acknowledgment to the FCN. Upon receiving all acknowledgments, the FCN confirms to the controller that the sync is complete. This technique, while expensive, does achieve the desired goal for the above example.

But, the technique does not resolve inconsistencies if the FCN does not have an entry for a connection. In the above example, if the FCN did not have the entry (k,v) to begin with, it would not be able to resolve the conflicting (k,w) and (k,x) entries at the first two nodes. The algorithm below resolves these inconsistencies using two rounds of messages between the FCN and the nodes prior to it in the chain.

Sync algorithm

(1) FCN broadcasts entries in the key range (kLow, kHigh) to all other chain nodes.
(2) Chain nodes insert these entries in their flow table overriding existing entries and send acknowledgment to the FCN.
(3) Chain nodes preceding the FCN send to FCN the set of entries in this key range that exist in their table but are not reported by the FCN.
(4) If the FCN receives an entry (k, v) and does not have an entry for k, it inserts (k, v) to its table and broadcasts (k, v) to all nodes prior to FCN. Otherwise, FCN reads its entry (k, v') from its table and broadcasts (k, v') to all nodes prior to FCN.
(5) Nodes insert the entries received from FCN in step (4) and send acknowledgments to the FCN.

Optimizing step (1): We can significantly reduce the overhead of state broadcast in step (1) by distributing this task among multiple nodes and selecting the recipient nodes carefully. In addition to the FCN, any subsequent chain node that is common to all reconfigurations of the chains participates in state transfer, e.g., the 3rd and 5th chain nodes in Figure 7. This group of common nodes sends their entries in the key range (kLow, kHigh) to other nodes outside the group. A common node sends entries to the preceding nodes in the chain up to the next common node. For example, tail sends its state to node a6 and a3 sends its state to a1 and a4. The last common node also sends its entries to any succeeding nodes in the chain. This optimized mechanism requires the controller to send separate sync messages to each common node with the specific broadcast instructions. For ease of exposition, we discuss only the above non-optimized sync algorithm in the discussion of the controller protocol.

4.3.3 Controller protocol. Figure 8 shows the controller state machine. There are three events that trigger the state machine: a reconfiguration, an ACK from an Anolis node that it has received the new peer list sent by the controller and a confirmation that a sync operation is complete from the corresponding FCN. We discuss the controller's actions in each of these events.

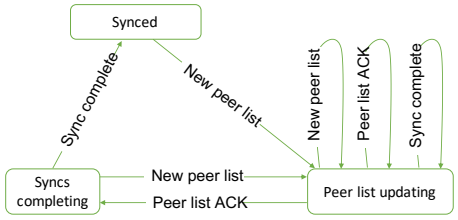


Figure 8: Controller state machine

```
1  ComputeSync(allPeerLists, syncedVersion, curVersion,
       ongoingSyncs, nodesAdded, nodesRemoved):
2  newSyncs <- [], cancelSyncs <- []
3  selectPeerLists <- allPeerLists[syncedVersion:curVersion]
4  allIDs <- unique nodeIDs in selectPeerLists in sorted order
5  for every key range (allIDs[i], allIDs[i+1]):
6    chainList <- [chain(peerList, allIDs[i], L) for
         selectPeerList in selectLists]
7    if chain(peer_list[curVersion], all_ids[i], L) has no
         common node with nodesAdded or nodesRemoved, continue
8    newSyncs.add([syncID++, allIDs[i], allIDs[i+1], FCN(
         chainList)])
9    if (allIDs[i], allIDs[i+1]) overlaps with key range of any
          [syncID', kLow, kHigh, fcnID] in currentSyncs,
         cancelSyncs.add([syncID', kLow, kHigh, fcnID])
10 return cancelSyncs, newSyncs
```

Figure 9: Controller protocol: Listing for ComputeSync

A new peer list is determined as a result of unplanned node failures detected by the controller or planned additions and removals of Anolis nodes by an administrator. This event takes the controller to the "peer list updating" state from any state. The first action by the controller is to increment the version of the peer list. It then computes two lists of sync operations. One list describes the new sync operations as a result of the reconfiguration. The other is a list of the canceled sync operations from any previous reconfiguration that are overridden by the new sync operations. The algorithm to compute these list is discussed later in the section. The controller sends the new peer list and the list of canceled sync IDs to the new Anolis peers.

The controller remains in the same state until all new peers have ACK-ed the new peer list. Upon that event, the controller initiates the new sync operations by sending messages to the corresponding FCNs. It enters the "syncs completing" state to wait for the completion of all ongoing syncs. The controller receives confirmations of sync completions from the corresponding FCNs. If all ongoing syncs are completed and the system is in "syncs completing" state, the controller enters the "synced" state. It marks the current version ID as synced and informs Anolis peers of the synced version ID.

ComputeSync (Figure 9) outputs the aforementioned lists of new and canceled syncs. It considers peer lists from the synced version to the current version (3). It computes a list of nodes all unique node IDs among these lists in sorted order (4). For each key range between successive node IDs, it computes the list of chains in these peer lists (5,6). A sync operation is necessary for this key range if any of the added nodes exist in the most recent chain in the list or if the any of the removed nodes were a part of the immediately preceding chain in the list (7). In these cases, the FCN for the list of chains is computed and the corresponding sync operation is added to the list of new syncs (8). Further, if the key range of any of the ongoing syncs overlaps with the key range of this sync, those syncs are canceled.

4.4 Proof of connection affinity

The proof presented here focuses on the Anolis query and sync protocols. We consider the time interval between one synced version of peer list to the next synced version as viewed by the controller. We prove that in any such interval for any key in the system, the following property holds.

Property 1: If a WhoOwns query on a key K returns a value V, then any subsequent WhoOwns will either not return a value or return the same value V.

Tuple notation: We denote the values stored with a key K at all its chain nodes using an L-tuple. The value at the head node is at the start of the tuple and the value at the tail is at the end of the tuple. If a node does not have an entry for this key, it is represented using 0 (read as null) in the tuple.

We define the following three tuples as *good tuples*. In tuple 1, no node has an entry for key K. In tuple 2, a prefix of nodes all have the same value V and the remaining nodes have no entries. In a chain of length L, there are (L-1) good tuples of this type. In tuple 3, all nodes have the value V.

$$\text{Tuple1} : (0, \cdots 0). \ \text{Tuple2} : (V, \cdots V, 0, \cdots 0). \ \text{Tuple3} : (V, \cdots V)$$

Proof structure: In between one synced version and the next, the system undergoes three phases: a period of no reconfigurations, a period of one or more reconfigurations and a period with no reconfigurations where the sync protocol executes and completes. We prove that if the system starts from a good tuple state at the beginning of the first phase then Anolis satisfies Property 1 during the three phases and returns to a good tuple state at the end of the third phase. We assume that the system starts in a tuple 1 state for key K, and is at the beginning of the first phase.

Claim 1: If the system starts with a good tuple state and no reconfigurations occur, Property 1 is satisfied.

Proof of Claim 1: The system starts with tuple 1 and V be the first value inserted at the head node. Then, all subsequent nodes will store no other value except V for the key K. Even if a conflicting WriteIfNotExists(K, V') is executed, the head

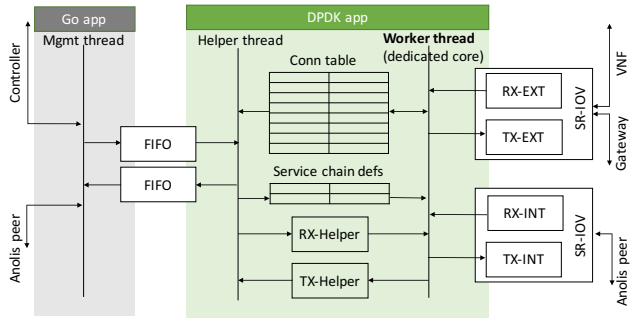


Figure 10: Anolis worker thread performs packet forwarding at line rate while delegating non-critical tasks to the helper and management threads.

node will reject the conflicting value V' and propagate the value V to the rest of the chain. Starting from a tuple 1 state, the system can only move to tuple 2 or a tuple 3 state.

A similar reasoning shows that a system in tuple 2 state remains in tuple 2 state (with an increasing number of non-null values) or moves to a tuple 3 state. A system in tuple 3 state always remains in a tuple 3 state. In all cases, the only value ever returned by the system is the initial value at the head node. Hence, the system satisfies Property 1.

Claim 2: Anolis can satisfy property 1 during a series of reconfigurations if (1) it was in a good tuple state right before the first reconfiguration and (2) there exists an FCN across all reconfigurations of the chain.

Proof of Claim 2: If the FCN does not have an entry, then no WhoOwns has been executed yet, which trivially satisfies Property 1. If the FCN has an entry (K, V), any subsequent nodes in the chain will have either no value for K or the same value V. Any WhoOwns query could only have returned the value V at the FCN, thereby satisfying Property 1.

Claim 3: After a series of chain reconfigurations, Anolis's sync algorithm restores the system to one of the good tuple states if there exists an FCN across all chain reconfigurations.

Proof of Claim 3: We consider three cases depending on the values at the FCN and the nodes prior to it at the start of the sync operation.

Case 1: FCN has a value V for key K. Step 1 and Step 2 result in that value being copied to all chain nodes. The system reaches a tuple 3 state.

Case 2: FCN does not have an entry for key K but there is an entry for K at one or more nodes prior to it. The FCN must receive an entry for K in step 3. In step 4, the FCN must accept the first value received for key K and send that entry to all nodes prior to it. As a result, all chain nodes up to the FCN must have the same value for the key and the system reaches tuple 2 or tuple 3 state.

Case 3: No nodes have an entry for key K. Sync has no effect and the system remains in a tuple 1 state.

Thus, the sync algorithm restores the system to one of the good tuple states in all cases, satisfying Property 1.

5 IMPLEMENTATION

Anolis's implementation seeks to strike a balance between performance and ease of implementation. To this end, it leverages many of the benefits offered by its design including integrated forwarding and replication, scale-out deployment, and reliability needs (or lack thereof) on message delivery. Our implementation efforts consist of the Anolis application discussed below and a standalone controller application written in Java that uses Zookeeper for state persistence.

Overview: Our implementation is split into a high-performance DPDK application for packet forwarding and a Go application for management tasks such as reconfigurations, addition of load balancing rules, and sync operations. The DPDK application consists of a worker thread that implements line-rate forwarding and a helper thread to which less performance-critical tasks are delegated. The helper thread communicates with the Go app via a pair of FIFOs and with the worker thread via DPDK message buffers (RX-Helper and TX-helper). The worker thread communicates with Anolis peers via the internal interface (RX-INT and TX-INT) and with the gateway and NFs via the external interface (RX-EXT and TX-EXT). This separation of interfaces is for de-multiplexing of DHT and external messages. The worker thread's connection table uses DPDK's RTE_HASH data structure, whose size is determined by the memory available for the Anolis instance. The helper thread populates two key configuration related data structures – load balancing rules and peer lists – for use by the worker thread.

Query protocol: Efficient implementation of the query protocol is key to Anolis's performance. Hence, its implementation by the worker thread is on top of the DPDK library bypassing the kernel, is run on a dedicated CPU core and is assigned SR-IOV network interfaces for direct access to underlying NIC hardware. We have employed a suite of code optimizations – packet batching, zero copies of packet data, use of specialized DPDK APIs for fast metadata copies, reusable header templates for DHT queries and responses, and cache alignment of connection table entries – to keep the per-packet forwarding cost down to 100-300 CPU cycles. We further limit any overhead of TCP/IP network stack by implementing Anolis strictly as an L2 device. Since the query protocol does not impose reliable FIFO semantics on messages, the worker thread can efficiently forward packets to Anolis peers, to the gateway or to NFs directly using their MAC addresses.

Sync protocol: The implementation of the sync protocol requires reliable message delivery as well as consideration of concurrent access to the connection table, which supports

thread-safe reads but not writes. The transfer of a range of keys from an Anolis node to another is implemented as follows. The Go app relays the controller's sync message to the helper thread. The helper thread safely reads entries from the connection table in the assigned key range and sends them to the Go app. The Go app reliably transfers this state to the Go app at the destination Anolis node over TCP. The state then reaches the helper thread at that node. However, only the worker thread can insert it to the connection table. Hence, it receives this state from the helper thread over the RX-helper buffer and inserts it into the connection table.

Since writes are not thread-safe, deletion of flow entries is implemented by the worker thread. A thread-safe hash-table can improve the performance of the worker thread.

6 EVALUATION

Our evaluation addresses these question using a variety of experiments on network testbeds. (1) What is the tradeoff between Anolis's replication and forwarding throughput and latency? (2) How does Anolis perform with flows of varying length? (3) Does Anolis's performance scale linearly with the number of nodes? (4) What is the impact of Anolis node failure on end-to-end performance?

6.1 Experimental setup

Our testbed consists of 4 servers each with two Intel Xeon E5-2470 CPUs (2.3GHz) and 64GB RAM connected via a switch with 10G and 40G network ports. All experiments except the one in Section 6.5 use this testbed.

T-Rex traffic generator: T-Rex provides stateful traffic generation capabilities at scales of millions of connections. In its default mode, it takes as input a packet capture from a sample flow in the form of a pcap file, a parameter connection per second (**CPS**) and a range of client and server IP addresses. It generates CPS new flows each second like the sample flow with unique 5-tuple values taken from the specified IP ranges. The packet arrival times can be replayed from the pcap file itself. Hence, it can generate continuous arrival and departure of flows at a scale high enough to test Anolis's performance limits. We setup TRex on a server with a pair of 40G ports, one of which serves as the external traffic endpoint and another serves as the endpoint of the network function being served.

Gateway router: We emulate a datacenter gateway switch using VPP – a high-performance software router. VPP provides hash-based ECMP forwarding among the set of Anolis nodes. The set of nodes can be changed at any time during or across experiments. We setup VPP on a physical server with a pair of 40G ports, and configured it to receive traffic from TRex's external port, and forward it to Anolis. Both T-Rex and VPP are provided sufficient CPU cores so that their performance exceeds that of Anolis in any experiment.

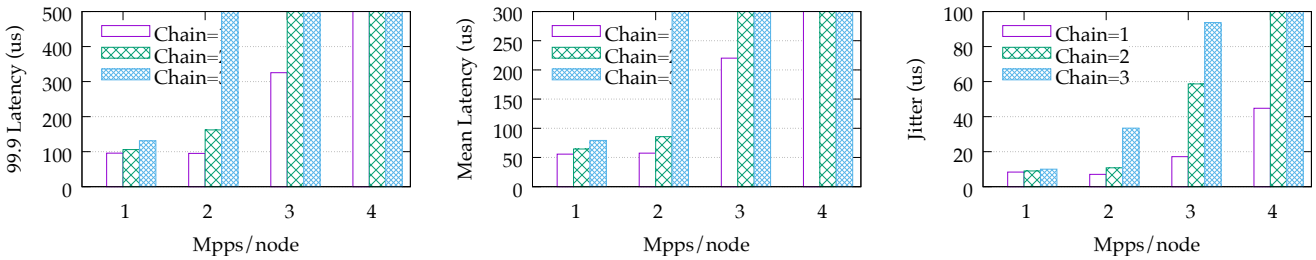
Anolis: Each Anolis node is provided a pair of 10G ports for external and internal traffic respectively, a single CPU core dedicated for running the worker thread, and sufficient huge page memory to store the entire flow table in its steady state (e.g., a 50M flow table requires 4GB memory). We run up to four Anolis nodes on a pair of servers each with 4X10G ports respectively. Additional experiments on Cloudlab test Anolis scalability up to 14 nodes.

6.2 Marginal cost of replication

We evaluate the marginal cost of creating additional chain replicas in terms of latency and forwarding throughput measured in packets/second. We experiment with sample UDP flows consisting of minimum sized (64B) packets only. We use the parameter packets per flow (**PPF**) to denote the length of a flow. We conduct two sets of experiments with PPF = 10 and PPF = 100 and present the results in Figure 11 and Figure 12 respectively. For each PPF, we generate traffic at the rates of 1, 2, 3 and 4 Mpps per Anolis node and measure latency for with Anolis chain lengths of 1, 2 and 3. We measure latency using trace packets that are forced to traverse the chain instead of using a cached response. Thus, we report the latency experienced by the first packet of each flow.

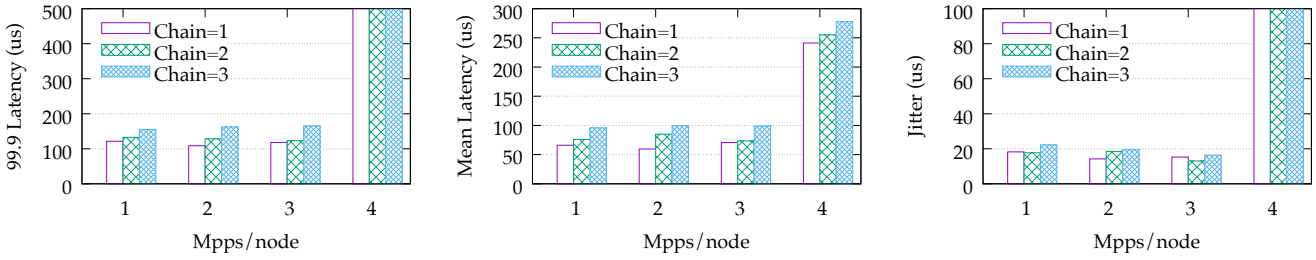
At lower traffic, we find that increasing the number of chain replicas from 0 to 3 causes a gradual increase in latency with each addition replica. In figure 12, the mean (99.9 percentile) latency for 3 Mpps traffic increases from 60 us (120 us) to 90 us (160 us) as we proceed from a 1x to a 3x replicated flow table. Jitter remains nearly uniform (Figure 12c) even on increasing chain length. This finding suggests that the latency cost of chain replication is similar to the per-hop forwarding cost of typical DPDK applications. As this latency is incurred only for the first packet of a flow (or once every keep-alive interval = 1 min), it could be considered a worthwhile tradeoff for other benefits such as elasticity and fault tolerance.

At higher traffic, we find that additional chain replicas can reduce the throughput of the system which is reflected in terms of higher latencies. This effect is much more pronounced for PPF=10 than for PPF=100. In Figure 11b (PPF=10), mean latency at 2 Mpps is significantly higher for chain=3 because of its lower overall throughput. We find that for a longer flow of PPF=100, additional chain replicas appear to cause a smaller reduction in throughput. In Figure 12b, we find that all chain lengths have comparable latencies up to 4 Mpps, which is close to the system throughput in this experiment. Thus, we find that for flows with around 100 packets, Anolis's replication has a small effect on throughput as well as latency.



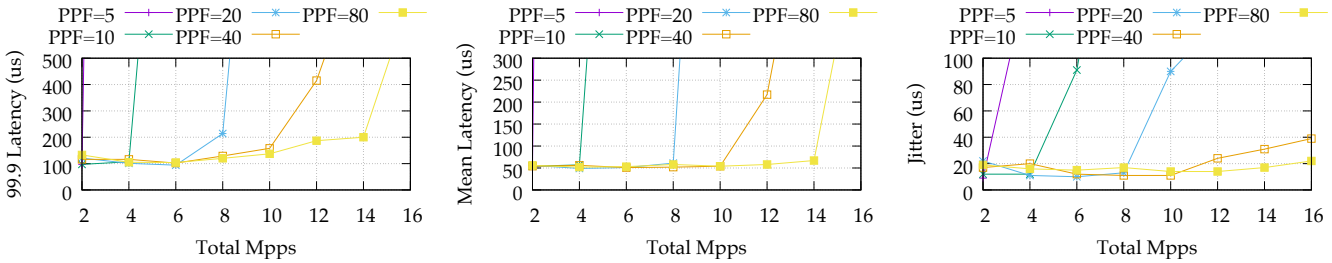
(a) 99.9 percentile latency **(b) Mean latency** **(c) Jitter**

Figure 11: Chain replication with packets per flow (PPF) = 10



(a) 99.9 percentile latency **(b) Mean latency** **(c) Jitter**

Figure 12: Chain replication with packets per flow (PPF) = 100



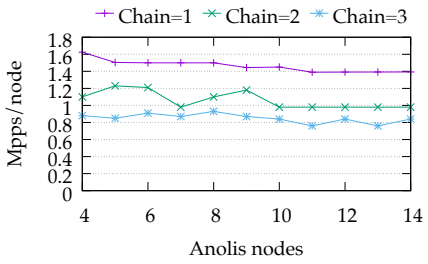
(a) 99.9 percentile latency **(b) Mean latency** **(c) Jitter**

Figure 13: Packets per flow

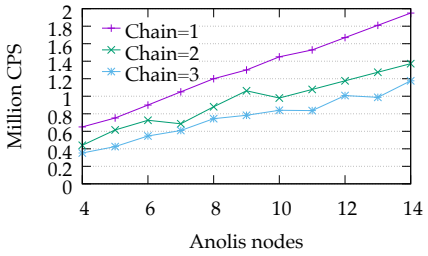
6.3 Scaling up flow tables

In this experiment, we further explore Anolis's performance for short flows with less than 100 packets, where the throughput impact of replication is more pronounced. For a chain length of 2, we profile throughput vs. latency curves for flows with PPF=5 to PPF=80 and show results in Figure 13. A sudden spike in latency for a given PPF indicates that throughput has been reached. By this indicator, PPFs of 5, 10, 20, 40 and 80 respectively achieve a total throughput (across 4 nodes) of at least 2 Mpps, 4 Mpps, 8 Mpps, 10 Mpps, and 14 Mpps respectively. However, latency seems comparable for all PPF values at loads below the system capacity.

Several factors explain the reduction in packet throughput for shorter flows. First, the fraction of internal packets used for replication increases sharply. A new flow creates 3 additional packets for replicating flow entry at 2 chain nodes – which is less than 4% of packets for a flow of length 80 but 60% of packets for a flow of length 5. Second, these additional packets result in memory writes which typically have lower throughput than memory reads. Third, smaller PPFs result in larger flow tables which require a larger fraction of CPU resources for garbage collection. We look to explore more efficient garbage collection techniques in the future to improve performance for short flows. If the workload is heavily dominated by short flows, Anolis supports horizontal scaling to meet input traffic demand as demonstrated next.



(a) Throughput scaling



(b) Connection scaling

Figure 14: Horizontal scaling

6.4 Horizontal scaling

We demonstrate Anolis's horizontal scaling capabilities using a 14 server testbed on Cloudlab. These servers were connected in a star topology with a 10G link. While each server only has a single physical port, we use SR-IOV support on that port to configure four virtual functions (VFs). First two VFs on each server are allocated to a Trex instance and next two VFs are allocated to a single Anolis instance. Anolis and Trex instances run on dedicated CPU cores. Each TRex instance sends and receives test traffic only from the Anolis instance on the same server. Anolis instances on all servers communicate via their internal interfaces over the Cloudlab network fabric.

We experiment with flows having a small PPF of 10 to ensure that inter-Anolis traffic is a non-trivial fraction of overall traffic. Using a binary search at various input traffic loads, the experiment measures the peak throughput for which total packet loss across all nodes remains below 1% over a two-minute duration. We measure throughput in Mpps for 4 to 14 Anolis nodes for chains of length 1, 2 and 3. Figure 14 shows the per-node packet throughput and the total number of new connections per second (CPS) in this experiment.

Figure 14a shows that Anolis throughput scales in a near-linear manner with the number of nodes. This behavior is the expected outcome given that all data plane functions in Anolis are fully decentralized and hash-partitioned among the nodes. The reduction of up to 0.2 Mpps in throughput with scaling could be explained by the load imbalance in the distributed hash table implemented among Anolis nodes. The linear scaling of throughput implies that the number of connections also scale linearly as highlighted in Figure 14b. In this experiment, Anolis supports a maximum packet throughput of 19.5 Mpps, or equivalently 1.95 million new connections per second. Since Anolis has a keep-alive of one minute in this experiment, this translates to roughly 120 million concurrent entries being maintained in the distributed flow table.

Statistic	Value
Client TCP connection attempted	3990780
Client TCP connection established	3990780
Server TCP connection established	3990780
Client TCP connection closed	3990780
Server TCP connection closed	3990780
TCP bytes sent by server	128080093320
TCP bytes received by client	128080093320
TCP bytes sent by client	993704220
TCP bytes received by server	993704220

Table 1: TCP statistics during Anolis reconfiguration.

6.5 Anolis reconfiguration

We demonstrate Anolis's ability to maintain connections despite node failures and node additions in the middle of ongoing flows. We run T-Rex in the **advanced stateful mode**, which runs a full-featured TCP stack. Hence, it can be used to measure the end-to-end behavior of TCP flows such as dropped vs. completed connections. In order to test if Anolis maintains connection affinity to a network function instance, we run 4 TRex "server" ports on the same 40G physical port using SR-IOV. In this experiment, TRex clients download a 30KB file at a rate of 66K connections per second for the one-minute duration of the experiment. These parameters are chosen so that, despite reconfigurations, Anolis nodes have sufficient network bandwidth to forward the traffic.

We perform the following reconfigurations of Anolis nodes. We begin with four Anolis nodes running a configuration with a chain of length = 2. Ten seconds into the experiment, we remove two nodes at alternate positions in the consistent hashing ring and update the peer lists at Anolis nodes and the gateway router. Thirty seconds later, we add two more nodes at the same positions in the ring, update the peer lists and do no reconfigurations thereafter. We tabulate a number of statistics from this experiment in Table 1.

These statistics show that Anolis failures cause no disruptions of ongoing connections. First, each connected attempted by a client is successfully established at client and server and is successfully closed by both client and server. Second, the total number of bytes sent by the server is equal to the bytes received by the client and vice versa. For further

validation, we repeated the experiment without any reconfiguration and found all these statistics to be identical. This experiment shows Anolis's ability to route packets to correct network function despite addition or removal of nodes in the middle of the connection.

7 RELATED WORK

Software load balancers: Ananta [10] pioneered the idea of a scale-out load balancer. While their paper alludes to the idea of a DHT to handle load balancer elasticity, they do not explore it further due to "reduced complexity and maintaining low latency". Our work shows that the additional latency is just tens of us on today's hardware and further addresses fault tolerance as well. Duet[4] is a hybrid hardware-software load balancer, but in a deployment with frequent server pool changes, it would revert to a pure software approach like Ananta to maintain affinity [8]. Scalebricks [17] uses a fully replicated, compact FIB that can handle asymmetric forward and reverse paths. But, it is optimized for a relatively static pool of connections and does not address the issue of concurrently updating the FIB while forwarding packets. In comparison, Anolis handles continuous arrivals and departure of flows and horizontally scales the flow table without full replication. Maglev [2] adopts a scale-out design that can provide inbound affinity in most scenarios, except when both the set of Maglev nodes and application server change during a connection's lifetime. But, it does not provider outbound affinity not fault tolerance. Beamer [9] provides a stateless approach to maintain affinity by reusing connection state available at application servers. But, in an ISP's network cloud would require significant modifications to network functions to adopt a Beamer-like approach which would be challenging given a large number of unique network functions operating in such a cloud.

Hardware-based state management: SilkRoad [8] implements L4 load balancing in P4 switches and offers terabits of forwarding throughput. Due to limited amounts of SRAM (100 MB) on modern ASICs, it is limited to a flow table of 10M entries despite a highly optimized data structure. It does not support elastic scaling of a switches flow table entries to other switches. Hence, it would be unable to support connection affinity during network updates that remove the designated load balancer switch from the path of an existing connection. NetChain [8] uses chain replication and partitions its key-space among multiple chains similar to Anolis. It addresses some of SilkRoad's elasticity limitations. But, Anolis's chain reconfiguration is completely non-blocking whereas NetChain blocks writes and sometimes reads for reconfiguration. Further, being a hardware-based solution, it inherits its disadvantages, namely limited memory, availability for specialized programmable switches, and necessary technical expertise to build and operate such networks.

NF state management: OpenNF [5] proposed the idea of migrating state across network functions for elasticity but its controller-based approach could become a bottleneck for elastic scaling. Split/Merge [12] supports splitting traffic among multiple VMs for scaling but does not support fault-tolerance. Conversely, FTMB [13] provides fault tolerance for general NFs but not elasticity. Unlike Split/Merge and FTMB, Anolis provides an integrated solution for fault tolerance and elasticity using chain replication in an L4 load balancer. StatelessNF [7] decouples NFs into a stateless processing component along with a stateful data store layer implemented using RAMCloud. Their approach can leverage RAMCloud to provide both elastic scaling and fault tolerance. But, its practicality is limited because it lacks widespread adoption by ISPs and their NF vendors today.

S6 [3] maintains state using a distributed shared object model to support elastic scaling. Their DHT-based lookup approach for locating remote objects is similar to ours. Unlike Anolis, S6 does not provide fault-tolerance. Their approach can support concurrent updates to shared state and is applicable to a broad class of NFs running on top of the Linux kernel. But, Anolis's custom replication protocol implemented using kernel-bypass techniques is expected to provide a significantly higher performance. For example, their experiments test performance at a maximum rate of 1 Mpps and with at most a few 10K flows.

8 CONCLUSIONS

ISPs network clouds have unique requirements, e.g. guarantees on both inbound and outbound affinity, that are not always met by solutions used in general purpose cloud platforms. There is a need for new distributed systems that can meet these requirements while extracting the maximum performance out of the underlying hardware. This paper is an effort in that direction. We presented the first L4 load balancer system that guarantees affinity during failures while delivering a high-performance design and the ability to elastically scale the system. Guided by our principle of integrated forwarding and replication, we leverage classic distributed systems ideas such as chain replication and consistent hashing. Further, we enhance them by developing a new replication protocol that supports seamless chain reconfiguration. In the future, we are looking to generalize these ideas and apply them to other stateful network functions.

REFERENCES

[1] David G Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. Fawn: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 1–14. ACM, 2009.

[2] Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *NSDI*, pages 523–535, 2016.

[3] Shinae et al. Elastic scaling of stateful network functions. In *NSDI*, 2018.

[4] Rohan Gandhi, Hongqiang Harry Liu, Y Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. Duet: Cloud scale load balancing with hardware and software. *ACM SIGCOMM Computer Communication Review*, 44:27–38, 2015.

[5] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. Opennf: Enabling innovation in network function control. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 163–174. ACM, 2014.

[6] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *NSDI*, 2018.

[7] Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. Stateless network functions: Breaking the tight coupling of state and processing. In *NSDI*, pages 97–112, 2017.

[8] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 15–28. ACM, 2017.

[9] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. Stateless datacenter load-balancing with beamer. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, volume 18, pages 125–139, 2018.

[10] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, et al. Ananta: Cloud scale load balancing. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 207–218. ACM, 2013.

[11] Amar Phanishayee. Chaining for flexible and high-performance key-value systems. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE, 2012.

[12] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. Split/merge: System support for elastic execution in virtual middleboxes. In *NSDI*, volume 13, pages 227–240, 2013.

[13] Justine Sherry, Peter Xiang Gao, Soumya Basu, Aurojit Panda, Arvind Krishnamurthy, Christian Maciocco, Maziar Manesh, João Martins, Sylvia Ratnasamy, Luigi Rizzo, et al. Rollback-recovery for middleboxes. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 227–240. ACM, 2015.

[14] Statista. Iot number of connected devices worldwide. https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/.

[15] Robbert Van Renesse and Fred B Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, volume 4, pages 91–104, 2004.

[16] VMWare. Deploying extremely latency-sensitive applications in vmware vsphere 5.5. https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/latency-sensitive-perf-vsphere55-white-paper.pdf.

[17] Dong Zhou, Bin Fan, Hyeontaek Lim, David G Andersen, Michael Kaminsky, Michael Mitzenmacher, Ren Wang, and Ajaypal Singh. Scaling up clustered network appliances with scalebricks. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 241–254. ACM, 2015.