Stateless Network Functions:
Breaking the Tight Coupling of State and Processing

Murad Kablan, Azzam Alsudais, Eric Keller, Franck Le
University of Colorado IBM
Networks Need Network Functions

- Firewall
- NAT
- Intrusion Prevention
- Load balancer

To protect and manage the network traffic
Networks Need *Agile* Network Functions

To match the agility of today’s (cloud) compute infrastructure
Network Agility -> Easy and Quickly to Use

- Seamless Scalability
- Failure Resiliency
- Instant Deployment

*Without Sacrificing Performance*
Virtual Network Functions to the Rescue?

Hardware Network Functions

Software Network Functions (Virtual Machines)
Same core architecture, same fundamental limit in agility
The Challenge is with The State

- **Firewall**: connection tracking information
- **Load balancer**: mapping to back end server
- **Intrusion Prevention**: automata state
- **NAT**: mapping of internal to external addresses
Example Problem 1: Failure

Redirect Traffic

P3
P2
P1

State

Flow

Lookup fails!!!
Example Problem 2: Scaling In and Out
Example Problem 3: Asymmetric / Multi-path

Flow1 (syn)

P1 syn

State

P2 synack

Lookup fails!!!
Other Solutions
Industry Approaches to Deal with State

**HA Pairs**

- Doubles cost, limited scalability, unreliable [Jain2009]

  ![Active and Backup HA Pairs Diagram](image)

**Don’t use state**

- e.g., Google Maglev
  - (hash 5-tuple to select backend).
  - Limited applications
Dealing with State: State Migration (for scaling)

Router Grafting [NSDI 2010], Split Merge [NSDI 2013], OpenNF [SIGCOMM 2014]

- When needed, migrate the relevant state
- Only handles pre-planned events
- High overhead to migrate state (e.g., 100 ms)
- Relies on flow affinity
Dealing with State: Check Pointing (for failure)

Pico Replication [SoCC 2013]
• Periodically checkpoint state (only diffs, and only network state)

Limitations:
• Quick recovery from failure
• High packet latency (can’t release packets until state check pointed)
Dealing with State: Deterministic Replay (for failure)

FTMB [SIGCOMM 2015]
- Log events so that upon failure we can re-play those events to rebuild the state
- Use periodic check pointing to limit the replay time
- Improves packet latency

Limitation:
- Long recovery time (time since last check point)
What is the root of the problem?
... Appliance mentality

Maintaining the Tight Coupling between State and Processing
Stateless Network Functions

- Re-designed as a distributed system from the ground up.
- Decoupling the state from the processing
Benefits of Decoupling State from Processing

Traditional Network Function
e.g., Firewall

- High overhead to manage state
- Relies on flow affinity
- Hard to achieve both resiliency and elasticity

Stateless Network Function
e.g., Stateless Firewall

- Seamless elasticity
- No disruption in failure
- Doesn’t rely on flow affinity
- Centralized state (simpler to manage)
Is this even possible?

We need to handle millions of packets per second
Why we can do this:

• Common packet processing pipeline has a lookup stage (so, per packet request to data store, but not lots of back and forth)

• Requests to data store are much smaller than packets (so, scaling traffic rates does not result in same scaling of data store)

• Advances in low-latency technologies (data stores, network I/O, etc.)
How State is Accessed

• Example for Load balancer

1\textsuperscript{st} Packet of flow
(Pick an available server)
• \textbf{1 Read} from Available table,
• \textbf{1 Write} to Assigned table

Every other Packet of flow
(look up assigned server)
• \textbf{1 Read} from Assigned table
System Architecture
StatelessNF
StatelessNF Architecture

- Network Function Host
- Data Store
- State
- Network Function
- Timeout Manager
- Controller
- Visualize/Modify
- Monitor/Manage
- OF Rules
- SDN Switch
- Traffic to network functions
• Low latency, etc.
• Also needs (or could use) support for timers, atomic updates, queues
Network Function Instances
High-Performance Network I/O

e.g., DPDK, netmap

To remote data store

Input

Output
Deployable Packet Processing Container

e.g., Docker

To remote data store

Pipe 1
NIC 1 → Pull → Queue 1 → Parse, Lookup, and Process → NIC 1

Thread 1

Thread 2

Pipe 2
NIC 2 → Pull → Queue 2 → Parse, Lookup, and Process → NIC 2

Thread 3

Thread 4

Pipe N
NIC N → Pull → Queue N → Parse, Lookup, and Process → NIC N

Thread Nx2-1

Thread Nx2
Optimized Data Store Client Interface

e.g., Batching, Buffer Alloc

Data Store Client Interface

Buffer Pool

Request Batching

Pipe 1
NIC 1  Pull  Queue 1  Thread 1  Parse, Lookup, and Process  Thread 2  NIC 1

Pipe 2
NIC 2  Pull  Queue 2  Thread 3  Parse, Lookup, and Process  Thread 4  NIC 2

Pipe N
NIC N  Pull  Queue N  Thread Nx2-1  Parse, Lookup, and Process  Thread Nx2  NIC N

To remote data store

Input

Output
Orchestration

- Failure handling – speculative failure detection (much faster reactivity)
- Scaling in and out – no need to worry about state when balancing traffic
Implementation

Network Functions (NAT, Firewall, Load balancer)
• DPDK
• SR-IOV
• Docker
• Infiniband to Data store (DPDK since paper)

Data store
• RAMCloud (Redis since paper)
• Extending with timer

Controller
• Extended FloodLight, basic policies for handling scaling and failure.
StatelessNF System Evaluation
Evaluation

Goal: in this extreme case architecture, can we get similar throughput and latency as other software solutions, but with better handling of resilience and failure?
Experiment Setup

Tests:
- Raw throughput, latency
- Handling failure
- Handling scaling in-out

Network Functions:
- Baseline Network Functions (state and processing are coupled)
- Stateless Network Functions (state and processing are decoupled)
Raw packets per second – lower until about 256 byte packets

Enterprise Trace – Stateless
Roughly matches Baseline

Note: similar to systems which have added support for scaling or failure
Latency

NAT (Firewall and Load balancer has slight less latencies)
Scaling In and Out

![Graph showing Goodput vs. Time for different scaling scenarios: Ideal, Baseline FW, Stateless FW.](Image)

- **Goodput (Gbps)**: Y-axis represents the goodput in Gbps, starting from 0 to 8.
- **Time (sec)**: X-axis represents time in seconds, ranging from 0 to 100.
- **Lines**:
  - **Ideal** (green line): Shows the ideal scaling scenario.
  - **Baseline FW** (orange line): Represents the baseline firewall performance.
  - **Stateless FW** (dark blue line): Demonstrates the performance of a stateless firewall.

The graph illustrates the performance differences between the Ideal, Baseline FW, and Stateless FW scenarios over time.
Handling Failure

![Graph showing time to complete vs number of download requests for different firewall configurations.]
Discussion and Future Work
• Date store scalability
  • Replace RAMCloud with other systems that report better throughput and lower latency (e.g., FARM, Algo-Logic)

• Reducing interactions with a remote data store
  • Integrate a set membership structure (e.g., a bloom filter) to reduce the penalty of read misses
  • Explore placement of data store instances (e.g., co-locating with network function instances)
Conclusions and Future Work

• Networks need agile network functions
  • Seamless scalability, failure resiliency, without sacrificing performance

• StatelessNF is a design from the ground up
  • Zero loss scaling, zero loss fail-over

• Main potential drawback... performance, but in this extreme point:
  • Throughput similar to other solutions
  • 100-300us added latency (similar to other solutions)

• Future work: Evolve data store design for network functions
Thanks!