Stateless TCP

Marcelo Abranches
marcelo.deabranches@colorado.edu
University of Colorado Boulder
Boulder, Colorado

Eric Keller
eric.keller@colorado.edu
University of Colorado Boulder
Boulder, Colorado

CCS CONCEPTS

• **Networks → Middle boxes / network appliances**; **End nodes**; • **Computer systems organization → Distributed architectures**.

KEYWORDS

Middleboxes, Elasticity, Resilience, Network State

1 INTRODUCTION

Increasingly, middleboxes have served as a means to introduce new capabilities into network to better secure, monitor, or optimize traffic. While these were traditionally deployed as physical appliances, industry is moving to more software based solutions. In reaction to the limitations of putting appliances inside of virtual machines, as with network functions virtualization, Stateless Network Functions (StatelessNF) [1] introduced a new concept where the processing in network functions can be decoupled from the state (which is stored in a remote data store). This enabled simultaneously solving the elastic scaling, fault tolerance, and other operational challenges such as software updates.

While follow on work has extended the concept to improve performance [2, 4], each (including StatelessNF) has focused on middleboxes. We propose that the same concept can be applied to network end points. Applications already take steps to persist state, but we propose going one step further and decoupling the state from processing of the TCP stack itself. In doing so, we enable applications to be truly elastic and resilient in a more seamless manner than depending on other applications handling the broken connections and re-sending requests in a graceful manner. While other works have enabled TCP migration capabilities [3], in building the TCP stack in a StatelessNF manner, we enable seamless handling of planned and unplanned events.

The impact of this can enable improved behavior for a number of applications, such as CDNs, HTTP servers, HTTP proxies, caches, BGP speakers, SSH servers, etc. For example, CDNs can benefit from Stateless TCP by avoiding the need to re-establish TCP connections, after failures or scaling events, which can improve latency. Also BGP routers (which use TCP as the transport on top of which BGP runs) can avoid costly session re-establishments and route re-advertisements after these events, enabling an always on router.

CoNEXT '19 Companion, December 9–12, 2019, Orlando, FL, USA

ACM ISBN 978-1-4503-7006-6/19/12…$15.00
https://doi.org/10.1145/3360468.3368182

While the concepts can be applied from the original StatelessNF work, doing so for a TCP stack does introduce some new aspects, which we address in this work. For example, there is new state we need to deal with in an efficient manner, namely the packet send and receive buffers. Also, there is a tighter coupling to the the end applications (which can be an opportunity and a challenge).

In the remainder of this extended abstract, we dig deeper into the challenges and provide an architectural overview of how we are overcoming some of them. We also discuss our current, preliminary work, toward the overall goal.

2 ARCHITECTURE

Figure 1 shows the high level design of Stateless TCP. Clients access TCP applications as usual without needing any modification. Each host runs a series of Stateless TCP stacks, and also applications and enpoints that use their services. The stateless TCP stack saves and updates its current state on a remote datastore. If the environment consists of multiple hosts that can host a service, clients may access services through a load balancer or through an SDN switch that can distribute traffic based on Openflow rules.

2.1 Controller

The Stateless TCP controller is responsible for monitoring the health and load of the machines that host the stack and the applications. It is also responsible for detecting failures on the TCP stack and on the applications. We rely on the services of a container orchestration platform for building and running our controller. As the controller has a global view of the current state of the infrastructure, we use it to setup the entities that distribute traffic to the Stateless TCP cluster (e.g. load balancers or SDN switches). The controller is also responsible to track which Stateless TCP instance is currently serving a given stream in a locking mechanism to avoid for example two instances trying to write state for a given flow during or after a scaling event.

2.2 TCP State

Stateless TCP saves TCP state on a remote datastore. This data is used to enable seamless TCP recoveries from failures and also enables cluster scaling events. As TCP works as a reliable stream oriented communication channel between applications, our stack must save the TCP state of each stream. Our system updates the current state of the TCP stack at the remote datastore at the end of the TCP stack processing for each received packet, and also just before sending a packet through the network. In order to not slow down the TCP stack, we designed a mechanism that allows Stateless TCP instances to read state data from a local cache during normal operation. But for coherent operation, Stateless TCP loads state from the remote datastore, if it is in recovery mode i.e., if the controller identifies a failure, and needs to restore a TCP stack. This

mechanism is also used during scaling-in/out events, and stream relocation events. We save and load TCP state on a remote key-value datastore using the hash of the 4-tuple of the connection as key, and state like acknowledgements and sequence numbers as values. During a relocation or recovery event if it is needed, Stateless TCP takes additional steps to set all the data structures, timers and buffers needed by the TCP stack, to enable coherent operation no matter in what state a flow currently is. We also save send and receive buffers on the remote datastore, in order to enable full TCP stack restore.

2.3 TCP Buffer Recovery

In order to allow correct protocol operation and to add useful features for applications, Stateless TCP provides two levels of buffer recovery. The first level does not require applications to be changed and it enables consistent protocol operation during recovery or relocation events. In the case of the receive buffer, recovery works by verifying if the last sequence of bytes read by the application is less than the last acknowledge number sent by the TCP stack. If so Stateless TCP will engage the first level buffer recovery mechanism, by reading buffer data from the remote datastore, where it is indexed by sequence numbers. After recovering data at the recovery buffer, Stateless TCP generates an epoll event that informs the application that there is data to be read. In the case of the sending buffer Stateless TCP verifies if data was passed to the TCP stack and was not sent by the packet I/O layer. In this case Stateless TCP will recover the send buffer using buffer data from the remote data store and generate an event to the packet I/O layer so that this data can be sent through the network.

The second level of recovery buffer leverages the fact that there is a copy of the TCP buffers at the remote datastore enabling the application to request data that it may previously have read, but the data was not persisted by the application. With this mechanism the application needs to inform our stack which bytes from the receive buffer an application has already consumed and persisted. So developers need to change applications to keep track of the bytes from receive buffer that already have been consumed by the application. This mechanism increases flexibility because a developer can choose to immediately commit the bytes that the application has received (e.g. before persisting its data at the application level), or it can delay the commit until important state on the application has been updated.

2.4 Offloader

We leverage the fact that the Stateless TCP only needs to read remote state and buffers during a recovery or relocation event to build an asynchronous state write mechanism that we call offloader. The offloader is used to accelerate writing state to remote datastore. For each active stream in Stateless TCP we create a lightweight queue that is stored in a hash table. Instead of writing state and TCP data directly to the remote datastore, Stateless TCP writes to the offloader, and a pool of threads is responsible for flushing these queues to the remote datastore. During a relocation or recovery event, Stateless TCP first queries the offloader to see if there is still data to be written to the remote datastore. If this is the case Stateless TCP will wait until the queue becomes empty before restoring the TCP stack.

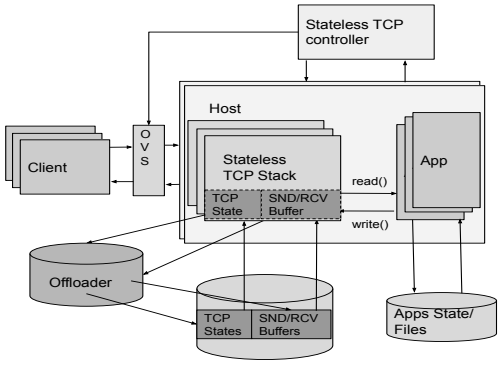

Figure 1: Stateless TCP high level design

3 PROTOTYPE AND FUTURE WORK

We are building stateless TCP on the top of mTCP and DPDK. We also use Redis as the remote datastore and built the offloader from scratch in C. Currently the controller is a simple Python program that install Openflow rules on OpenVswitch in order to balance load between Stateless TCP instances at the granularity of a stream. We are evaluating our stack using an HTTP server built on the top of Stateless TCP and using Apache Bench as our benchmark. At this point we are able to perform seamless recovery of TCP flows after induced Stateless TCP app failures, and we can also perform seamless relocation of streams through multiple instances of an application running on the top of Stateless TCP. We compared the performance of Stateless TCP with plain mTCP and we saw that Stateless TCP currently reduces throughput by less than 20%, but we still see room for improvements that could lower this gap. The first level of buffer recovery is already implemented, and we are working on the second level. We are also designing use cases like balancing load across clusters, live updates, live malware recovery, BGP resiliency, offloaded firewall/IDS and much more.

4 ACKNOWLEDGEMENTS

This work was supported in part by NSF Grants 1652698 (CAREER) and the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001

REFERENCES

[1] Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. 2017. Stateless network functions: Breaking the tight coupling of state and processing. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 97–112.

[2] Junaid Khalid and Aditya Akella. 2019. Correctness and Performance for Stateful Chained Network Functions.. In *NSDI*. 501–516.

[3] Alex C Snoeren, David G Andersen, and Hari Balakrishnan. 2001. Fine-Grained Failover Using Connection Migration.. In *USITS*, Vol. 1. 19–19.

[4] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. 2018. Elastic scaling of stateful network functions. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 299–312.