Carbon Copy - State Management anf Fault Tolerance

Arijit Pramanik & Kanak Agarwal
150020094 & 150050016

May 2019

1 Introduction

In recent times, stateful programmable planes are on the rise, and network functionality is being shifted from the end-hosts or the controllers to the data-plane to take advantage of line-rate processing. This now, not just involves migration of flows on network updates but also consistent migration of associated states to ensure correct network behaviour. Keeping the required states at the switch with subsequent maintenance will allow us to fully leverage the line-rate processing on the data plane.

Reads on the switch can occur at near line-rates without any control plane intervention. However, writes need to be done by the control plane by the root controller to ensure consistent state updates across all switches. In the case of performing writes at the switch itself, such fault tolerant update guarantees are no longer there. It might be that every switch updates and maintains a different set of states with possible similarities. But in case of network failure or extremely heavy traffic being incident on one of the switches, we might need to re-route the flows to another switch which might not have the associated states required for routing.

Hence, fault tolerance guarantees need to be provided for writes at the switch level. This is also motivated by the fact that consistent updates by the root controller involve much higher latencies. Hence, we propose to update suhc states by means of a local controller residing on and local to the switch itself, keeping in mind the necessities of fault-tolerance that most modern applications have. To provide fast switchover and consequent recovery and re-routing of the flows, we provide both synchronous and asynchronous models of our fault tolerance. We provide a comparison along with existing methods of fault tolerance.

Also, every P4 program nowadays needs guaranteed fault tolerance. In order to provide an API for this to programmers and abstract this out, we also propose to generate a generalized modular version of a fault-tolerance model on P4, given the programmer requirements. All of the above calls for a new fault-tolerance module on P4 and thorough comparison with existing architectures for the same.

2 Related Work

- A basic version of providing fault tolerance is through the root controller. Since the root controller in a network is aware of the entire topology and is assumed to not susceptible to failures, the states to be pushed onto the switches re realised by pushing them through this root controller itself. Since, the root controller is located on a distant remote machine, this induces extremely high latencies, which however comes with the guarantee that all backup switches will have exactly the same states as those that are present on the secondary switch. (Ref. to presentation for working model)

- A control-plane mediated version is also in existence. This processes all such writes/updates only via the control plane. Any such state update request is sent to the local controller of primary that updates the corresponding state and after ensuring it is updated, sends a packet containing the same state info to be updated to the secondary switch. The secondary switch now invokes its local controller to update the corresponding state and after ensuring the update, the controller sends back an ACK which is forwarded by the secondary to the primary switch. This is again sent to the primary's local controller which upon this ACK message from secondary, generates another ACK and sends it back

signalling a successful update. Since, this is a controller mediated version, all communication in the form of packets is carried out between controllers. (Ref. to presentation for working model)

- We also came across [1] which proposes a generalized state-management framework for consistent state migration in data planes. This however, demarcates every state as being hard (needs to be migrated) or soft(no need of migration owing to easy reconstruction from the flow itself). The state migration in such a setup is carried out by augmenting such states to love traffic from the source to the destination switch where these states need to be migrated. In such a framework, the destination switch will update its values accordingly. However, this differs from our model in the sense that :
  - These states are migrated explicitly on request and not on-the-fly. It notifies the central controller for re-routing the flows upon successful state synchronization of the 'hard' states. So, this is a reactive framework rather than being pro-active.
  - There is no guarantee provided for a state update. In that sense, there is no ACK/reply from the destination switch which proclaims that all the corresponding states have been updated successfully and accurately.

3 Problem statement

Our goal hereby is to construct a fault-tolerance model for state maintenance on switches that can take advantage of the near line-rate processing of the data plane in switches while at the same time, ensuring write consistency across both the primary and the backup switch. This now implies that the key-value store that is now on the switch effectively acts as a cache with the usual line-rate speed for read operations. We also desire to provide the API/abstractions for fault-tolerance on P4 as per programmer requirements.

4 Experiments

4.1 The generic setup

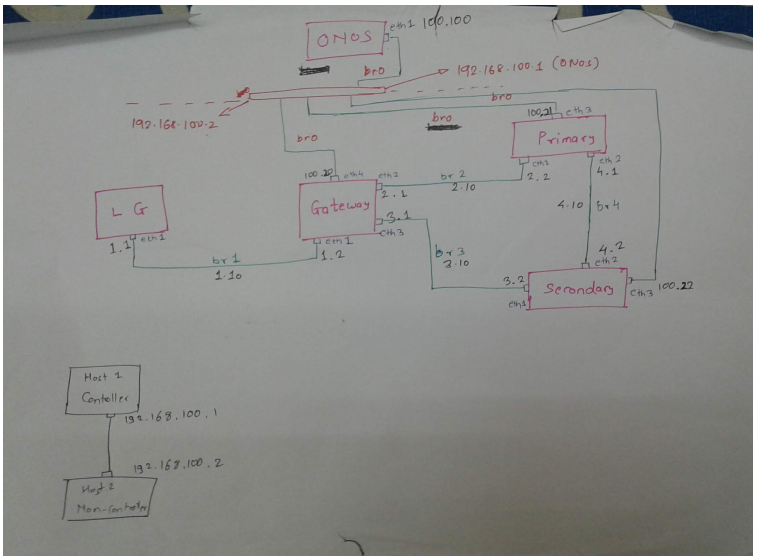

Figure 1: The fault-tolerance model setup consisting of primary, secondary, gateway and load generator switches on one of the machines with the root controller on the other *(Drawing Courtesy : Rinku Shah)*

This consists of the load generator on a container, with the gateway on another container for forwarding packets from the load generator, based on the type of the packet. We have set up two containers running

the primary and secondary switches along with a local ONOS v1.13.0 controller on both these containers which provides the functionality of pushing rules onto the switches. On a separate machine, we have a root ONOS controller running, which keeps monitoring the primary and secondary switches and is responsible for pushing the initial table entries onto the gateway and for mediating the switchover after detecting a switch failure. Note here, that the P4-16 switches used for our experiments are software switches. The packet generator used here is a multi-threaded raw packet generator that runs in blocking mode. The switches are bound to a limited number of CPUs using `taskset` so as to have good CPU utilization by the switches in order to reach throughput saturation(A condition in which an increase in the number of threads will no longer increase throughput but increase packet processing latencies). We have used a variable number of threads in our experiment for this. There is also an inherent setting for wait latencies between packets. Each thread works on a disjoint set of 100 keys.

The packets that have been used in this experiment are of the form `[Ethernet | IP | UDP | Payload]` where payload is of the form `[type_sync | key | value | version]`, where type_sync denotes whether it is a read, read-reply, write or a write-reply packet. Key and value simply characterise the state that we want to update for a read, while key signifies the state we want to read. For read-reply, the value field will hold the corresponding value, while for a write-reply, the entire payload is preserved as it was for the corresponding write packet.

As per the experimental setup outlined above, we run our packet generator and measure the throughput for incoming packets at the load generator itself. The packets are initially forwarded to the primary from gateway. Shortly afterwards, the primary switch process is killed to trigger a failover and corresponding read-write throughput and latencies at the secondary switch are subsequently obtained. Both of them operate within the synchronous version and asynchronous version setup, with the former guaranteeing fault-tolerant updates and the latter being leveraged for very fast applications without any need for the above.

4.2 The tolerant three and the faulty fourth

We now describe different variants of our setup (Please refer to the presentation and live demo) :

- **CC-sync** : The setup is as described in 4.1. Here, the state to be updated has its info sent to the primary switch. The primary switch sends this packet to its ONOS local controller encoding it in a packet-in message for update of corresponding state, while simultaneously cloning the received packet and sending it over to secondary switch. The secondary switch now sends this to its local ONOS controller, which then updates the sate on secondary switch accordingly and sends back and ACK message to the primary switch. The ACK being sent only when the state has been updates ensures backing up of state info at the secondary. The primary switch will now just forward this ACK back to the gateway from where the write request came. As usual, read requests are handled in the switch's data plane itself.

- **CC-async**: The setup is as described in 4.1. The difference from the above is the nature of the ACK response that is sent back to the gateway requesting the write. The state to be updated has its info sent to the primary switch. The primary switch sends this packet to its ONOS local controller encoding it in a packet-in message for update of corresponding state, while simultaneously cloning the received packet and sending it over to secondary switch. The secondary switch now sends this to its local ONOS controller, which then updates the sate on secondary switch accordingly. Meanwhile, after ensuring that the state has been updated on the primary switch accurately, the primary switch sends back an ACK to the gateway itself, without

- **CC-CP**: The packet containing info about the state to be updated is forwarded to the primary switch. The primary switch puts this as data into a packet-in message and sends it to its local controller of primary switch. The primary local controller pushes the state (guaranteed through ONOS API's) and the same state info is sent to the secondary switch if state is updated successfully at primary switch. Secondary switch sends this as packet-in message to its local controller that pushes the state and generates an ACK if it is successful. The ACK is sent back to the primary which sends it as a packet-in message to the primary's local controller, which then sends back again an ACK message, This ACK message is now forwarded by the primary switch to the gateway requesting the update.

- **CC-basic** : This is the root controller setup, consisting of the same old primary, secondary switches without the local controller, and the gateway and the load generator switches with the root controller on a distant remote machine. Here, the packet containing info about the state to be updated is forwarded to the primary switch. Primary switch now encodes it in a packet-in message and sends it to the root controller that maintains a *concurrent hashmap* and updates the same state on both of the switches. The root controller after successfully pushing these entries generates an ACK message and sends it to the primary switch, which then forwards it back to the gateway.

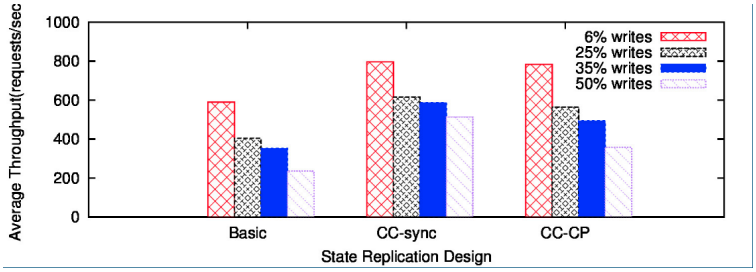

Figure 2: Comparison of different designs : Throughput vs the % of writes in a fixed number of sent packets

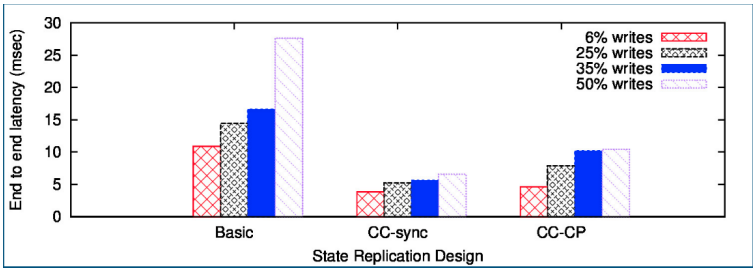

Figure 3: Comparison of different designs : Latency vs the % of writes in a fixed number of sent packets

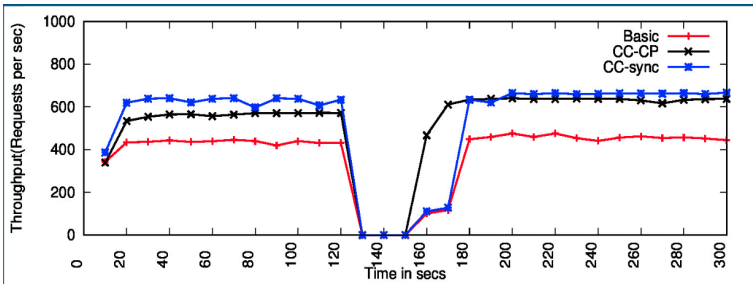

Figure 4: The fault-tolerance model CC-sync : simulation graph of throughput vs. time with failover

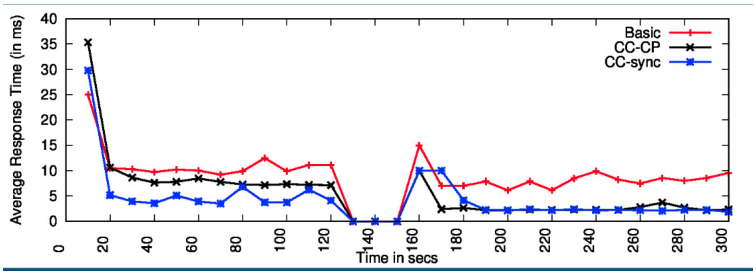

Figure 5: The fault-tolerance model CC-sync : simulation graph of average response time vs. time with failover

5 Conclusion

Read operations occur very fast but since write operations should have inherent fault tolerance, performing a write at a switch induces a much higher latency. So higher latencies and lower throughputs for higher write:read ratios are observed. The increase in latency does not show a linear increasing trend. CC-sync has no added latency overheads, unlike sending it to local controller at every step (for CC-CP) which introduces minimal latencies and sending it to root controller (CC-basic) which has a very high latency. For the failover, switchover time is same(mediated by root controller) and respective time to reach saturation is almost the same (theoretically should be the same). A similar trend shows up for the response time per packet. Since, the values just after the failover have been interpolated, there can be some minimal error. In all 3 cases, the respective setup reaches its throughput saturation before and after the failure of primary switch (zero state loss). Post-failover involves only read-write operations at the secondary switch without any need for state replication. So CC-CP saturated throughput becomes almost the same as CC-sync saturated throughput.

6 Future Work

The current work focuses on full table (set of states) replication. However, the states can be segmented into hard and soft states, out of which only hard states need to be migrated. Such partial table replication has not been implemented. Also, we have not implemented on-the-fly segmentation of states into the above two groups.

Other models of fault tolerance like 1:1 (a secondary switch for every primary), N:1 (a secondary switch for N primary switches) also can be potentially developed into P4 modules/programs.

References

[1] Shouxi Luo, Hongfang Yu, and Laurent Vanbever, "Swing state: Consistent updates for stateful and programmable data planes," in *Proceedings of the Symposium on SDN Research*. ACM, 2017, pp. 115–121.