Fault tolerance in SDN

Leander Seidlitz, Cora Perner*
*Chair of Network Architectures and Services, Department of Informatics
Technical University of Munich, Germany
Email: leander.seidlitz@tum.de, clperner@net.in.tum.de

***Abstract*—Software Defined Networking (SDN) is based on decoupling the data and control plane of network devices. Switches handle packet forwarding in the data plane. A centralized controller offers a global network view to network applications and enables configuration through a single point. Paths through the network may be configured end-to-end, the centralized controller takes care of configuring the switches.**

While SDN offers high flexibility, fault-tolerance becomes an issue. The global view of the controller allows for fast failover in the data plane. Fault-tolerance in the control plane is a more complex problem. In order to correctly process incoming packets a controller must be available at any time. The fault-tolerance of the control layer is vital for the function of the network

This paper gives an overview of current approaches to fault-tolerance in the data as well as control plane.

***Index Terms*—software-defined networking, fault-tolerance**

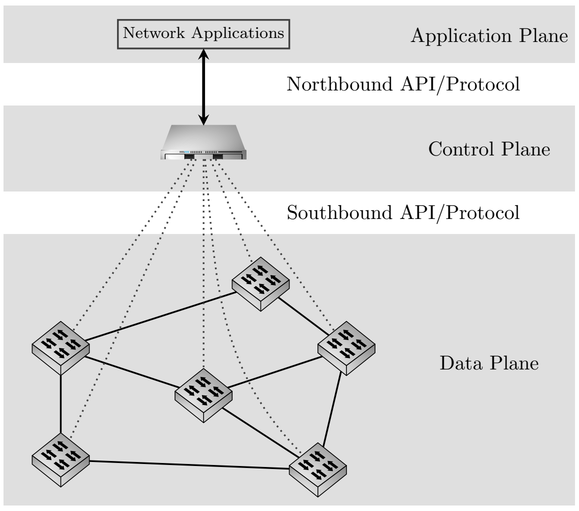

Figure 1: Structure of a SDN network.

1. Introduction

Software Defined Networking (SDN) offers greater flexibility than traditional network architectures. In traditional network topologies, control and data plane are distributed over the network. Routing protocols such as OSPF (Open Shortest Path First) [1] are used by the distributed entities in order to establish routes through a network. Network applications have to communicate with multiple devices in order to configurate the network.

In contrast, SDN splits up the data, control and application plane of a network, as depicted by Figure 1. While the data plane is responsible for packet filtering and forwarding, the control plane enforces policies and handles tasks as load balancing and multipath routing. A central controller configures the data plane by sending commands to the respective switches. It offers an abstracted view of the network topology and flows to the network applications in the application plane, enabling network configuration through a single controller. While the data and control plane take care of handling flows in the network, the application plane manages network policies using the global network view presented by the control layer.

The switches in the data plane rely on a Network Information Base (NIB) in order to handle packets. Packets arriving at the ingress port of a switch are matched to flows. The NIB specifies how to handle the packet, and whether to forward or drop it. Packets belonging to an unknown flow are forwarded to the controller. The controller decides how to handle the packet, for example by configuring a flow through commands to the respective switches. The flow configuration received by the switches through controller commands is saved in the NIB. Future packets of the established flow can be handled without consulting the controller.

Communication between the control plane and application plane is done through a northbound Application programming interface (API) or protocol. As of December 2019, there are no standarized APIs or protocols for the northbound interface, although Representational State Transfer (REST) APIs are widely used.

The control plane communicates with the switches through a southbound API or protocol. OpenFlow [2] is a common protocol for the communication between control and data plane.

While the separation of data and control plane offers flexibility, the centralized control plane creates a single point of failure. To ensure correct network function, the control plane must be available at any time.

The Fault Tolerance Problem. Networks are expected to operate without disruption, even in the presence of link or device failures. Faults in the network should be handled quickly and transparently, causing only minimal service interruption. The strict separation of data and control plane forces us to handle fault-tolerance for both planes separately. SDNs require approaches to fault-tolerance in two domains: The data plane, where switches or links can fail as well as the control plane, where controllers or the link between controller and switch may fail.

We will focus on fault-tolerance in the control plane. Section 2 gives an overview of fault-tolerance in the data plane. Approaches to fault-tolerance in the control plane are discussed in Section 3. Fault-tolerance in the application plane is not in the scope of this paper.

2. Fault Tolerance in the Data Plane

The data plane takes care of handling packet flows in the network. Flows are established by the controller through configuration of the individual switches. For reliable network operation the data plane must be resilent against link and switch failures. Failures have to be detected quickly and resolved by rerouting affected traffic on alternative links, restoring the networks functionality.

Still, basic network policies must not be violated. For example, traffic rerouted on a different path through the network should not be able to bypass a firewall. Fault-tolerance mechanisms therefore do not only have to regard the network topology but also its policies configured by the application layer.

The controller possesses global knowledge of the network topology and can therefore run centralized algorithms. These are potentially more efficient than distributed algorithms, such as the *rapid spanning tree protocol* [3], which only have limited information about the network.

Approaches to focusing on data plane fault tolerance, such as FatTire [4], the approach of Paris et al. [5] or CORONET [6], have to ensure resilence of the data plane against failures without introducing large overhead.

2.1. Reacting to Topology Changes

The structure of SDN based networks is not static. Links in the network are removed and established, constantly changing the networks topology. While traffic should usually take a near optimal path through the network, this path may fail. An optimal failover algorithm would choose the next optimal path, but calculating this path can be expensive and therefore time-consuming for large networks. Restoring the network function by applying a suboptimal path outweighs path optimization and is acceptable. A suboptimal path can be optimized after network function is restored.

Traffic traversing the network on an suboptimal path causes overhead. While rerouting the traffic to a lower-overhead path may lower the costs of traversing the network by finding a better path. Reconfiguring the network introduces overhead as well. Approaches to fault-tolerance should only change existing paths if the benefit of rerouting traffic is larger than the overhead caused by reconfiguring.

2.2. Minimizing Overhead

Paris et al. [5] present an approach that finds a balance between optimal paths and the frequency of reconfiguration. They divide their approach into two sub-mechanisms: Firstly, rapid handling of failures by rerouting to alternative paths and secondly a mechanism for path optimization.

Restoring Paths. After a link or device failure backup paths are calculated on demand. The priority is to quickly find an alternative path, which is allowed to be sub-optimal. The path is calculated based on a shortest-path algorithm. Restoring the path is vital for the networks function, the path optimization is taken care of by another mechanism.

Optimizing Paths. Optimization of network paths is done by a mechanism Paris et al. call *Garbage Collection of network resources*. Periodically flow allocations in the network are analyzed and optimized. An iterative algorithm converging to the optimal solution is used. As new links become available and failed links are repaired, the garbage collection may reroute traffic, should network changes open up shorter paths. Rerouting is only done in case that the optimization is larger than the overhead caused by the necessary network reconfiguration.

In a static network the paths would converge to the optimal solution. Failed links and devices introduce sub-optimal paths and therefore overhead, moving further away from the optimal solution.

2.3. OpenFlow Action Buckets

OpenFlow 1.3 [7] introduced the concept of action buckets. An action bucket groups a number of rules, the bucket itself is bound to conditions based on switch state, such as the status of a link.

Action buckets allow creating conditional forwards such as deactivating a set of rules as a link fails. The buckets are prioritized. Packets are matched with the rules in the highest bucket which conditions are met. This allows specifying precomputed backup paths that become instantly active when a link fails. FatTire [4] makes use of OpenFlow action buckets.

The FatTire Language allows the definition of network paths as regex-like expressions. Paths in the network are specified end-to-end, the necessary degree of fault tolerance can be specified for each path. The FatTire compiler then calculates the hops through the network for path realization as well as possible backup paths. The result is an OpenFlow configuration that can be applied to the individual switches. As links fail the precomputed backup paths become active. Repaired links are instantly reused.

2.4. Fault-Tolerant Controllers

The solutions presented above depend on the controller being available at any point. Nevertheless, controller failures are possible and must be handled. A failed control plane leaves the network in a headless state. Events such as incoming packets belonging to unknown flows cannot be handled without a controller. Therefore, a fault-tolerant control plane is vital. In the following section we will present approaches to a fault-tolerant control plane.

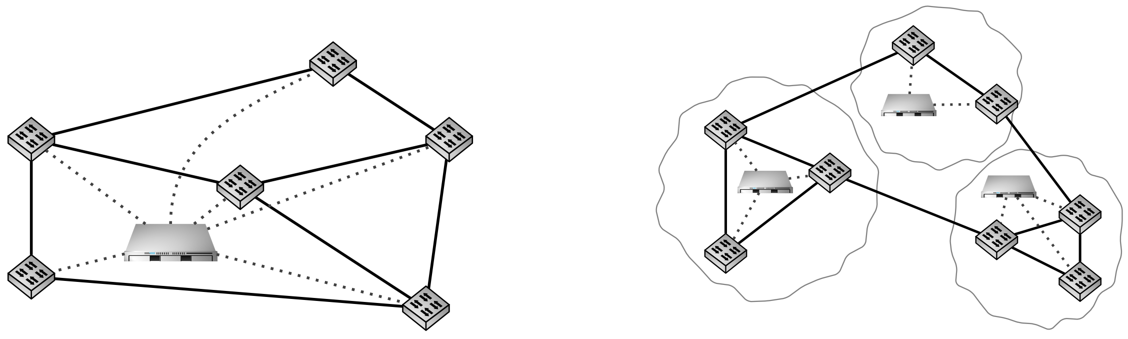
(a) Centralized (b) Distributed

Figure 2: SDN Control Plane Topologies.

3. Fault-Tolerance in the Control Plane

While fault-tolerance in the data plane is essential, the data plane is dependent on the controller. The control plane is vital for the operation of the network as it handles tasks such as deciding how to handle unknown flows as well as the communication with the application layer. The controller must be operational at all times to ensure correct network operation. SDN builds on a centralized controller. This controller presents a single point of failure. Introducing redundancy to the control plane allows for a fast failover in case of a controller fault. In the following we present approaches to fault-tolerant controllers.

3.1. Control Plane Topology

SDN control planes either have a single logically centralized controller (Figure 2a) or are constructed in a distributed topology (Figure 2b). A distributed topology has multiple controllers, each handling a separate domain of a network. Distributed typologies are mostly found in large networks in which a single controller cannot handle the load. Multiple parallel operating controllers offer the possibility of one controller taking over another part of a network as its controller fails.

The solution to fault-tolerance in a distributed controller scheme can be found in approaches such as HyperFlow [8] or ECFT [9]. In the following we will focus on more traditional SDN based networks with a centralized controller.

Logically Centralized Control Planes. In a logically centralized control plane only one controller is active at any time. This controller takes charge of all decisions in the network such as configuring the switches. Fault-tolerance in a logically centralized scheme is commonly achieved through a master-slave approach.

A master controller operates the control plane, slave controllers passively mirror the master controllers state. As the master controller fails, one of the slave controllers becomes master and takes over.

The failover time is critical. The network is inpoerable as long as the controller is not available.

3.2. Ravana: A Master-Slave Approach

In the following we discuss master-slave approaches to fault-tolerance in the control plane on the example Ravana [10]. In a master-slave topology, multiple controllers are present. One controller serves as master controller, controlling the network. Slave controllers mirror the master, and take over if the master fails.

Ravana [10] is an approach ensuring fault-tolerance for the controller, the communication between controller and backup controllers as well as for the communication between switches and the controller. It was proposed by Katta et al. A solution to fault-tolerance must fulfill the following requirements:

A) Total event ordering
B) Exactly-once event processing
C) Exactly-once execution of commands
D) Consistency under switch and controller failures

In the following we analyze Ravana in regard to these requirements.

A) Total event ordering. In a master-slave approach the slaves must have the same view of the network as the master. Each replica of the master builds its state independently, based on the stream of events received. In order to keep the state equal over all replicas the order of events processed must be the same for all controllers. Inconsistent event ordering may be caused by different latencies between switches and controllers.

There are two approaches to keeping the master and its replicas in sync:

1) Let the switch broadcast events to all controllers
2) Replicate the event at the master controller before processing it

Regarding to the first approach, ensuring the same order of events at each controller is challenging. A total order would require the controllers to synchronize the events received, posing a large overhead.

Replicating the event at the master controller before processing ensures the order of events received. It is the same for all controller replicas as the master controller defines the order. In Ravana switches send events to only the current master controller. The switches use an event buffer to prevent lost messages in case of a master controller failure.

By performing event replication at the master controller Ravana ensures a total event order at all replicas.

B) Exactly-once event processing. An approach to fault-tolerance must ensure that every event sent by a switch is processed exactly once. Events must not be lost nor processed repeatedly. The delivery of messages can be ensured by sender-side buffers, repeating the transmit if the receiver has not acknowledged the message.

If a packet causes an event the switch sends the event to the current master controller and additionally saves the event in a local buffer. Events received by the master are replicated to the slave controllers before the master processes them. The slaves hold back the event from their application layer until the master has successfully processed the event. After successful processing of the event, a confirmation message is sent to the switches and slaves. The switch clears the event from its buffer, the slaves can now safely release the event to the application. If the master fails during event processing, the switch can retransmit the event from its event buffer to the new master.

Unique messages IDs and receiver side filtering guarantee that messages are processed at most once.

C) Exactly-once Execution of Commands. As commands from the controller may not be idempotent, we must ensure that they are executed exactly once by a switch. A command buffer at the controller and acknowledgments by the switches, analogous to the switch-side event buffer, ensure that a switch receives and successfully executes a command.

The controller buffers commands sent to the switch, and deletes them from the buffer when the switch acknowledges execution of the command. The replicas of the controller are informed about the command buffer and the status of the commands sent. In case of the master controller crashing after sending a command but before processing the acknowledgment by the switch, the new master controller will find an incomplete command execution in the command buffer. It will resend the command. As commands have unique IDs, the switch will filter the command (ensuring at-most-once execution) but resend the acknowledge to the new master. The master controller will mark the command as successfully executed and replicas are informed.

D) Consistency under Switch and Controller Failures. Switches retransmit events and acknowledge commands. As long as the controller does not fail, we can handle switch failures the same way as single-controller SDN do: Relay the decision to the control application in the application layer. The network application will then decide how to reroute traffic and send appropriate commands to the controller.

Combined switch and controller failures pose a more complex problem. Should the master controller fail before finishing the processing of an event, a slave controller will take over. A switch that has sent an event has therefore not received an acknowledge yet. If this switch fails during this failover, it cannot retransmit the event to the new master. Still, the new master controller has received a copy of the event from the old master as events are replicated before processing. It sees the unfinished event in his buffer and can process it, even without the switch retransmitting. After this, we handle the switch failure as regular switch failure.

We conclude that Ravana solves the requirements needed for reliable fault-tolerance in the control plane. We now discuss how the failed controller is replaced in distributed and centralized topologies.

3.3. Controller Failover

In a distributed control plane topology a controller is responsible for a set of switches. HyperFlow [8] or ECFT [9] are approaches to fault-tolerance in distributed schemes. Both approaches use a similar approach to replacing the failed controller. The switches the failed controller was responsible for are split up and assigned to other controllers. This is done based on metrics such as the delay between the respective switch and controller as well as the controller load. In order to prevent cascading failures controllers must not be overloaded.

In a master-slave topology the slave controllers have to decide on who becomes the new master. Ravana [10] solves this problem by letting the switches contend for a distributed Zookeeper [11] lock. The controller that obtains the lock becomes the new master. The new master then informs the switches of the change in master controller.

3.4. Interfacing with the Application Layer

In a traditional SDN topology, the application layer interfaces with a single controller. A fault-tolerant approach should be observational indistinguishable from a single controller SDN: the system should behave in the same way as a fault-free single controller system would.

Additionally, controller redundancy and failover should be transparent for the application layer. This enables network applications to interface with fault-tolerant control layers without the need of rewriting.

4. Conclusion and Future Work

While fault-tolerance in the data plane seems mostly to be an optimization problem, fault-tolerance in the control plane is a more difficult problem to solve. Concerning the data plane, the global knowledge of the controller allows fast re-routing of traffic in case of failed links and switches. In the control plane, failures are more difficult to handle.

We presented requirements, solutions to fault-tolerance in the control plane must meet, and how approaches can fullfil them. Ravana [10] is a promising master-slave approach to fault-tolerance in the control plane. It offers transparent fault-tolerance and fast failover between controllers. Ravana does require extension of the OpenFlow protocol, which may hinder its acceptance.

Future extensions to master-slave schemes for fault-tolerance in the control plane may base on Ravana and extend it. Current protocols fail at state-replicating multi-threaded control applications as well as handling byzantine faults. These are tasks to be solved by future approaches to fault-tolerance in SDNs.

References

[1] J. Moy, "OSPF Version 2," RFC Editor, RFC 2328, Apr. 1998. [Online]. Available: https://tools.ietf.org/html/rfc2328

[2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, p. 69, Mar. 2008. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1355734.1355746

[3] IEEE Standards Association, "IEEE Standard for Local and metropolitan area networks–Bridges and Bridged Networks," IEEE, Tech. Rep., May 2018. [Online]. Available: http://ieeexplore.ieee.org/document/6991462/

[4] M. Reitblatt, M. Canini, A. Guha, and N. Foster, "FatTire: declarative fault tolerance for software-defined networks," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking - HotSDN '13*. Hong Kong, China: ACM Press, 2013, p. 109. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2491185.2491187

[5] S. Paris, G. S. Paschos, and J. Leguay, "Dynamic control for failure recovery and flow reconfiguration in SDN," in *2016 12th International Conference on the Design of Reliable Communication Networks (DRCN)*. Paris: IEEE, Mar. 2016, pp. 152–159. [Online]. Available: http://ieeexplore.ieee.org/document/7470850/

[6] Hyojoon Kim, M. Schlansker, J. R. Santos, J. Tourrilhes, Y. Turner, and N. Feamster, "CORONET: Fault tolerance for Software Defined Networks," in *2012 20th IEEE International Conference on Network Protocols (ICNP)*. Austin, TX, USA: IEEE, Oct. 2012, pp. 1–2. [Online]. Available: http://ieeexplore.ieee.org/document/6459938/

[7] Open Networking Foundation, "OpenFlow Switch Specification," Open Networking Foundation, Tech. Rep. Version 1.3, Jun. 2012. [Online]. Available: https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf

[8] A. Tootoonchian and Y. Ganjali, "HyperFlow: A Distributed Control Plane for OpenFlow," in *INM/WREN'10 Proceedings of the 2010 internet network management conference on Research on enterprise networking*, Apr. 2010, p. 6.

[9] W. H. F. Aly and A. M. A. Al-anazi, "Enhanced Controller Fault Tolerant (ECFT) model for Software Defined Networking," in *2018 Fifth International Conference on Software Defined Systems (SDS)*. Barcelona: IEEE, Apr. 2018, pp. 217–222. [Online]. Available: https://ieeexplore.ieee.org/document/8370446/

[10] N. Katta, H. Zhang, M. Freedman, and J. Rexford, "Ravana: controller fault-tolerance in software-defined networking," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research - SOSR '15*. Santa Clara, California: ACM Press, 2015, pp. 1–12. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2774993.2774996

[11] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free coordination for Internet-scale systems," *Proceedings of the 2010 USENIX Annual Technical Conference*, p. 14, Jun. 2010.