STSM Report*
Modeling of software failures and failure propagation models in Software Defined Networking (SDN)

STSM Applicant: Petra Stojsavljevic Vizarreta
Supervisors: Poul Heegaard, Carmen Mas Machuca
Home Institution: Technical University of Munich, Germany
Norwegian University of Science and Technology, Norway

November 30, 2016

1 Motivation

The goal of this STSM was to i) study the new failure modes introduced by Software Defined Networking, and ii) provide a comprehensive model that captures complex dependencies introduced by network softwarization. The model will serve as a baseline for design of the strategies to prevent and mitigate technology related disasters, which is the umbrella topic of the RECODIS WG3 (Technology-related disasters).

2 Classification of failures in SDN

SDN in a novel network architecture concept of decoupling control and data plane, which brings high degree of programmability and potentially the lower deployment cost. Control plane logic in SDN networks is concentrated in logically centralized controllers (Fig. 1).

SDN is still in its infancy, and there are not many reports on its performance in wide scale deployments. Hence, we studied the reported problems and suggested feature enhancements in the existing open source projects on SDN controllers to get a grasp of the newly introduced failure modes. Here, we describe the most representative examples, for each layer.

*This STSM was a part of the CA COST Action CA15127 Resilient communication services protecting end-user applications from disaster-based failures (RECODIS) WG3 (Technology-related disasters) and was carried between 6th and 11th of November 2016

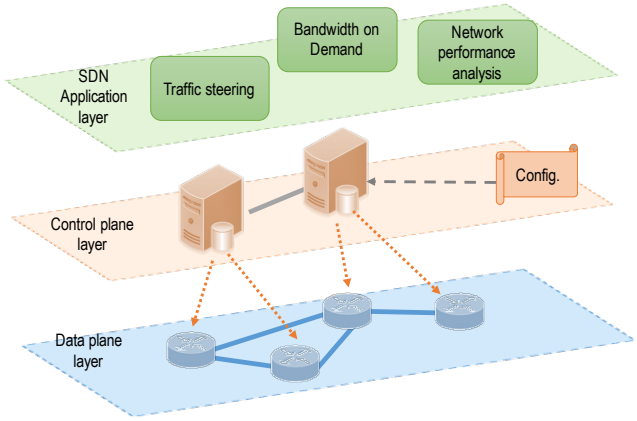

Figure 1: Layers in SDN

Note that malicious attacks and other security issues are omitted from this analysis. Good overview of the security threat vectors can be found in [KRV13].

2.1 SDN application layer

SDN applications provide a high level instructions to the SDN controllers, e.g. reservation of the bandwidth between the physical end points, which result in the reconfiguration of the underlying data plane, e.g. by adding the new flow rules. The problem occurs when two applications try to enforce two **conflicting policies**, e.g. install a flow using a forbidden port. Before the application of the new policy, the controller must apply the policy checker, to ensure it does not violate any of the already installed rules or violate other network invariants. Examples of **network invariant violations** include connectivity (creation of loops, black holes leading to silent drops and partitioning of the network), access policy violation, and lack of the isolation of network slices.

SDN applications can be implemented in the same name space as the controllers. In this case, a crash of the application may lead to a crash of the entire controller [CB14]. The **fate sharing** should be avoided by providing a proper isolation between the layers. The **application crash** during the implementation of the new policy may lead to a set of the inconsistent rules in the data plane [CB14]. The check-pointing and the rollback mechanism to the last working configuration should be supported by the controller.

2.2 Control plane layer

SDN controller implements all the control plane logic as software modules, e.g. path computation module or device manager. As such, the controller is the subject to **software aging** effects, such as the accumulation of the numerical errors and memory leaks. The authors in [LDBS15] argue that the speed of software ageing depends of the load of the controller. Software rejuvenation methods such as timely restart of individual processes or reboot of the whole machine may be used to mitigate the effect of such failures [GPTT95].

Software bugs cannot be avoided in a complex software projects. It is estimated that the professional programmer will introduce approximately 6 bugs per thousand lines of code (LOC). Prototype controllers, like Floodligt, have almost 100 thousand LOC, while complex project, like Open Daylight, can have more than a million LOC. This means that the system designer may expect anywhere between few hundreds to few thousands of bugs in the SDN controller software. An example of the software bug are is the incorrect implementation of Floyd-Warshal algorithm (path computation). Some bugs will be quickly detected during the normal operation, while others may be activated only by a very specific sequence of input parameters. The authors in [SWR+15] showed that by isolating minimum causal sequences that lead to the controller crash, debugging process can significantly speed up the debugging process.

The goal of the network control is to enforce invariants e.g. connectivity, access control, isolation and virtualization. Controller bugs may cause that some of the invariants to be violated [SWR+15]. The controller should be able to verify that the network invariants are not violated when inserting, modifying or deleting the rule [KZZ+13]. Disturbances during network updates may lead to problems like black holes, forwarding rules, link overload, incorrect packet destinations. The authors in [KPK14] provide a mechanism to roll-back to the last working configuration if not all acknowledgements are received.

SDN controllers rely on the multi-thread processing. Bugs caused by the concurrency issues, such **multi-thread data race condition**, are especially difficult to isolate and reproduce. **Memory handling issues** (memory leakage, buffer overflow, uninitialized read, buffer recycling) also may lead to the controller crush. Changing the environment (e.g. portion of the used memory block or repeating the multi-thread operation with different scheduling timers) may help with overcome such **non-deterministic bugs**. Such system recovery is in general faster than the system restart or reboot [QTSZ05].

More than one controller might be deployed in the network for the scalability and the resilience. The lack of the coordination may lead to unexpected failures. **Database deadlock** happened when the switch failed to connect, because two controllers tried to insert the key in the shared graph library (ONOS). Temporary loops and black holes may be created due to the time difference between the switch or link down/recovery event and the notification of the controller. This may be specially pronounced when the the controllers are physically and the updates are additionally delayed due to synchronization. **Operating on stale state data** in the shared database may lead to the suboptimal output

and the violation of the network invariants.

Byzantine fault tolerance relies on the controller replication. This typically requires $n \geq 3f+1$ controllers to tolerate f **corrupted controllers**. Corrupted controllers may be malicious, provide a wrong output due to the software bug or simply have its internal data structure corrupted [LLGN14, VCB$^+$13].

SDN controller typically runs as in a virtual machine deployed on the commodity hardware. The **failure of the underlying hardware, host OS or a virtualization layer** lead to the controller failure. **Configuration errors** (erroneous management system or human error) may happen, but are expected to occur less frequently than in the legacy systems.

2.3 Data plane layer

The forwarding devices (switches and routers) may be connected to more than one controller for the resiliency purpose. In such cases usually one controller has the master role, meaning it can change the configuration of the device(add and remove the flow entries), while other controllers, have the role of the slave and only receive the state updates. The other possibilities that all the controllers have the equal role, but this may lead to the **ownership conflicts**.

Overlapping flow entries may lead to a faulty operation. If the new forwarding rule that unintentionally has the higher priority than previously installed DoS rule, important security policy may be violated. Switch may become unresponsive due to the **flow table overflow**. Flow table overflow caused the load balancer crush. In high load scenarios it may happen that the data plane traffic interferes with link discovery packets, if the traffic priority is not properly configured.

The summary of the most relevant software-based failures in SDN is provided in the Tab. 1.

3 Modelling of software based failures

Our goal was to provide a comprehensive model of software based failures in SDN. In order to provide a suitable model let us first recapture the services provided by each layer, and interactions between different software modules.

The controller reacts to the events in data plane (e.g. periodic collection of the statistics, topology changes) and high level instructions coming from the SDN applications (e.g. bandwidth reservation, new security policy). SDN controller consists of several software modules providing the services to the application and data layers. The split of the functionalities might look different in different controller architectures, but basic modules always include the topology and statistics (maintaining the overview of the state of the network), flow rule manager (handling flow entries), path computation and policy manager. Some of the dependencies are illustrated in Fig. 3 as arrows, whose direction indicates triggering events and services provided by each module.

Table 1: Summary of the software-based failures in SDN

Fault	Activating event	Failure impact	Remediation
SDN application crush	External	Depends on design	Process isolation
Software ageing	Time	Controller crush	Restart/reboot
Software bug	Input parameters	Incorrect output	Debugging
Concurrency issues	Data race conditions	Incorrect output	Recovery
Memory handling issues	Memory allocation	Controller crush	Recovery
Corrupted controller	Majority corrupted	Incorrect output	Restart/reboot
Database deadlock	Ownership conflict	System crush	System redesign
Cooperation issues	Stale data	Incorrect output	–
Virt. layer failure	Hypervisor crush	Depends on design	Reboot
Switch not reachable	Failure or congestion	Depends on design	Repair

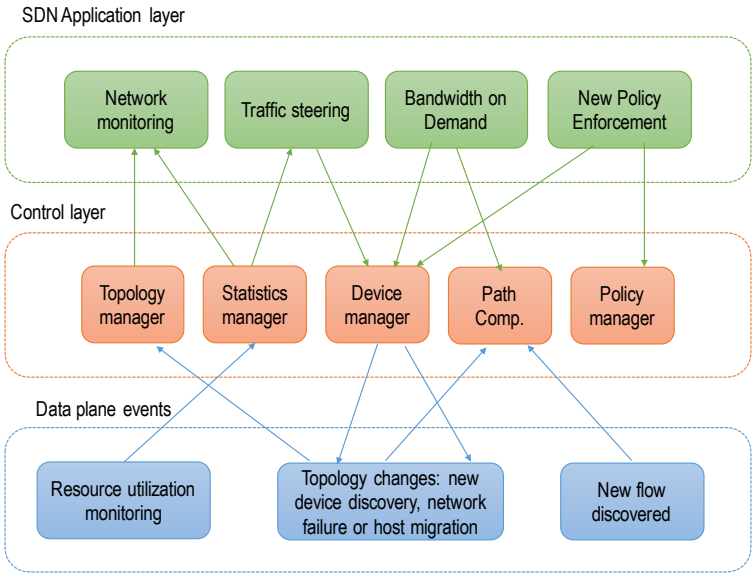

Figure 2: SDN controller reacts to the changes in the data plane and receives high level instructions coming from the application layer. The controller consists of several modules providing the services to the application and data plane layers.

3.1 Structural model: Reliability Block Diagram

Let us consider an SDN application that tries to install the new security policy (e.g. block the traffic coming from a particular port). The controller modules

involved in this service are Policy and Flow Rule manager. In order to provide high availability, three controllers are installed on different host servers in the cluster. At least one controller has be working properly in order to provide the service. We assume that all the switches have to be alive and reachable for the policy to be successfully installed.

The simplest way to illustrate this relationship structural model are the Reliability Block Diagrams (RBD) illustrated in the Fig. 3.

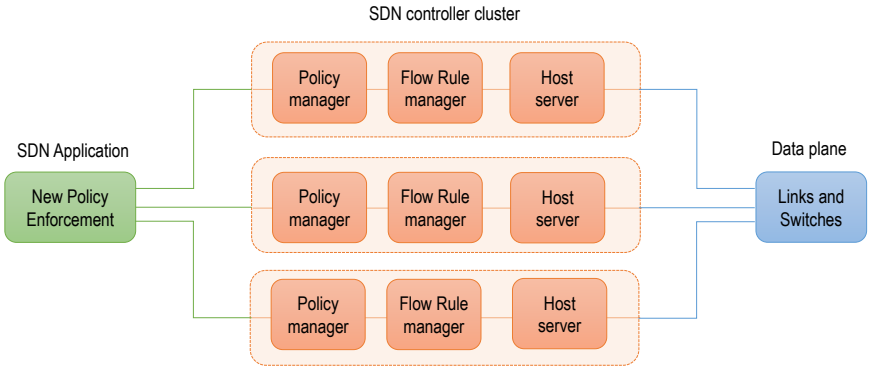

Figure 3: Reliability Block Diagram (RBD) for the New Policy Enforcement service. In order to be successfully installed, the application, at least one out of three controllers in the cluster, control plane links and all the switches have to be working properly.

$$A_{service} = A_{app} * A_{CP} * A_{DP}$$

$$A_{cp} = 1 - (1 - A_{PM} * A_{FRM} * A_{HS})^3$$

3.2 Dynamic model: Markov model

Availability of the host server and switches might be given in their product catalogues, but the availability of the software modules is difficult to estimate, because it depends on the specific operational environment, and many parameters that cannot be estimated before its deployment (e.g. how many bugs are left in the software after the testing).

Software modules have different failure modes, each one of them having a different failure intensity and different reparation rate. Based on the analysis on software-based provided in the previous section, we distinguish between four failure modes: deterministic software bug (BG) that can be only repaired by debugging, temporary errors (TE) due to the non-deterministic bugs, such as data race conditions, numerical errors (NE) due to the software ageing, and

hardware errors (HW) that happen less frequently, but take longer time to fix. The dynamics of these failures can be captured by Markov models (Fig. 4).

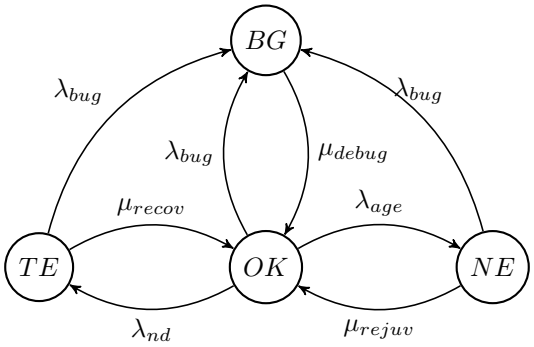

Figure 4: Markov model for different controller software modules. We distinguish four failure modes: deterministic software bug (BG), temporary errors (TE) due to the non-deterministic bugs, numerical errors (NE) due to the software ageing, and hardware errors (HW).

The steady state probabilities are:

$$\underline{p} = \left[p_{OK} p_{TE} p_{NE} p_{BG}\right]$$

Steady state probabilities can be found as the solution of the system of equations:

$$p_{OK} + p_{TE} + p_{NE} + p_{BG} = 1$$

$$\underline{p}.\underline{Q} = \underline{0}$$

$$\underline{Q} = \begin{bmatrix} -\lambda_{nd} - \lambda_{age} - \lambda_{bug} & \lambda_{nd} & \lambda_{age} & \lambda_{BG} \\ \mu_{recov} & -\mu_{recov} - \lambda_{bug} & 0 & \lambda_{bug} \\ \mu_{rejuv} & 0 & -\mu_{rejuv} - \lambda_{bug} & \lambda_{bug} \\ \mu_{debug} & 0 & 0 & -\mu_{debug} \end{bmatrix}$$

Availability of the software module can be expressed then as:

$$A_{SW} = p_{OK} = \frac{\mu_{debug}(\mu_{recov} + \lambda_{bug})(\mu_{rejuv} + \lambda_{bug})}{\lambda(\lambda_{bug} + \mu_{debug})}$$

where:

$$\lambda = \lambda_{bug}^2 + \lambda_{age}\lambda_{bug} + \lambda_{nd}\lambda_{bug} + \lambda_{bug}\mu_{recov} + \lambda_{age}\mu_{recov} + \lambda_{bug}\mu_{rejuv} + \lambda_{nd}\mu_{rejuv} + \mu_{recov}\mu_{rejuv}$$

There are two main drawbacks of using the Markov models in system dependability analysis. First, adding the complexity (e.g. state synchronization within the cluster) would lead to the state explosion, which makes it difficult to obtain the closed form solution. Second, Markov models a memoryless property of the underlying stochastic process, which does not always hold in the described system[1].

Next steps and future work

The models presented in this report will serve as the basis for our future work on the resilience of the SDN based networks.

The results of this collaboration will be published in the suitable conference. We identified RNDM 2017 - 9th International Workshop on Resilient Networks Design and Modeling as our target conference.

[1] Several Software Reliability Growth Models (SRGM) models have been proposed to model the time between software bug failures

References

[CB14] Balakrishnan Chandrasekaran and Theophilus Benson. Tolerating sdn application failures with legosdn. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, page 22. ACM, 2014.

[GPTT95] Sachin Garg, Antonio Puliafito, Miklós Telek, and Kishor S Trivedi. Analysis of software rejuvenation using markov regenerative stochastic petri net. In *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on*, pages 180–187. IEEE, 1995.

[KPK14] Maciej Kuzniar, Peter Peresini, and Dejan Kostić. Providing reliable fib update acknowledgments in sdn. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 415–422. ACM, 2014.

[KRV13] Diego Kreutz, Fernando Ramos, and Paulo Verissimo. Towards secure and dependable software-defined networks. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 55–60. ACM, 2013.

[KZZ+13] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P Brighten Godfrey. Veriflow: verifying network-wide invariants in real time. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 15–27, 2013.

[LDBS15] Francesco Longo, Salvatore Distefano, Dario Bruneo, and Marco Scarpa. Dependability modeling of software defined networking. *Computer Networks*, 83:280–296, 2015.

[LLGN14] He Li, Peng Li, Song Guo, and Amiya Nayak. Byzantine-resilient secure software-defined networks with multiple controllers in cloud. *IEEE Transactions on Cloud Computing*, 2(4):436–447, 2014.

[QTSZ05] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: treating bugs as allergies—a safe method to survive software failures. In *Acm sigops operating systems review*, volume 39, pages 235–248. ACM, 2005.

[SWR+15] Colin Scott, Andreas Wundsam, Barath Raghavan, Aurojit Panda, Andrew Or, Jefferson Lai, Eugene Huang, Zhi Liu, Ahmed El-Hassany, Sam Whitlock, et al. Troubleshooting blackbox sdn control software with minimal causal sequences. *ACM SIGCOMM Computer Communication Review*, 44(4):395–406, 2015.

[VCB+13] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. Efficient byzantine fault-tolerance. *IEEE Transactions on Computers*, 62(1):16–30, 2013.