Automatic Failure Recovery for Software-Defined Networks

Maciej Kuźniar[†][*] Peter Perešíni[†][*] Nedeljko Vasić[†] Marco Canini[♯] Dejan Kostić[‡]
[†]EPFL [♯]TU Berlin / T-Labs [‡]Institute IMDEA Networks
[†]<name.surname>@epfl.ch [♯]m.canini@tu-berlin.de [‡]dkostic@imdea.org
[*] These authors contributed equally to this work

ABSTRACT

Tolerating and recovering from link and switch failures are fundamental requirements of most networks, including Software-Defined Networks (SDNs). However, instead of traditional behaviors such as network-wide routing re-convergence, failure recovery in an SDN is determined by the specific software logic running at the controller. While this admits more freedom to respond to a failure event, it ultimately means that each controller application must include its own recovery logic, which makes the code more difficult to write and potentially more error-prone.

In this paper, we propose a runtime system that automates failure recovery and enables network developers to write simpler, failure-agnostic code. To this end, upon detecting a failure, our approach first spawns a new controller instance that runs in an emulated environment consisting of the network topology excluding the failed elements. Then, it quickly replays inputs observed by the controller before the failure occurred, leading the emulated network into the forwarding state that accounts for the failed elements. Finally, it recovers the network by installing the difference ruleset between emulated and current forwarding states.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Network operating systems; C.4 [**Performance of Systems**]: Reliability, availability, and serviceability

Keywords

Software Defined Network, Fault tolerance

1. INTRODUCTION

To ensure uninterrupted service, Software-Defined Networks (SDNs) must continue to forward traffic in the face of link and switch failures. Therefore, as with traditional networks, SDNs necessitate *failure recovery*—adapting to failures by directing traffic over functioning alternate paths. Conceptually, this requires computing the new forwarding state in response to a failure event, similarly to how a link-state routing protocol re-converges after detecting a failure. In an SDN however, this computation takes place at the SDN controller, based on its current view of the network.

To achieve higher availability, this computation could be done in advance to determine backup paths for a range of failure scenarios. With the appropriate hardware support[1], the backup paths can be preinstalled to enable switches to quickly adapt to failures based just on local state.

Regardless of whether the recovered forwarding state is computed in response to a failure or precomputed in advance, failure recovery requires the presence of extra software logic at the controller. In particular, in today's modular controller platforms such as POX, Floodlight, *etc.*, each individual module (*e.g.*, access control, routing) potentially needs to include its own failure recovery logic [6]. This leads to controller modules that are more difficult to develop, and potentially increases the chance of bugs [1]. Avoiding this problem by relying on simple recovery logic like timeouts or deleting rules forwarding to an inactive port is inefficient [6] and may introduce forwarding loops in the network [4].

In this paper, we propose AFRO (Automatic Failure Recovery for OpenFlow), a system that provides automatic failure recovery on behalf of simpler, failure-agnostic controller modules. Inspired by the successful implementation of such a model in other domains (*e.g.*, in the large-scale computing framework MapReduce [2]), we argue that application-specific functionality should be separated from the failure recovery mechanisms. Instead, a runtime system should be responsible for transparently recovering the network from failures. Doing so is critical for dramatically reducing the chance of introducing bugs and insidious corner cases, and increasing the overall chance of SDN's success.

AFRO extends the functionality of a basic controller program that is only capable of correctly operating over a network without failures and allows it to react to changes in the network topology. The intuition behind our approach is based on a straightforward recovery mechanism, or rather a solution that sidesteps the recovery problem altogether: After any topology change, we could wipe clean the entire network forwarding state and restart the controller. Then, the forwarding state gets recovered as the controller starts to install forwarding rules according to its control logic, initial configuration and the external events that influence its execution.

HotSDN'13, August 16, 2013, Hong Kong, China.
ACM 978-1-4503-2178-5/13/08.

[1]In OpenFlow, the FastFailover group type is available since version 1.1 of the specification.

However, because all rules in the network need to be reinstalled, such a simple method is inefficient and has significant disadvantages. First, a large number of packets may be dropped or redirected inside the `PacketIn` messages to the controller, which could be overwhelmed. Second, the recovery speed will be adversely affected by several factors, including the time to reset the network and controller, as well as the overhead of computing, transmitting and installing the new rules.

To overcome these issues, we take advantage of the "software" part in SDN. Since all the control decisions are made in software running at the controller, AFRO repeats (or predicts) these decisions by simply running the same software logic in an emulated environment of the underlying physical network [3, 7]. During normal execution, AFRO simply logs a trace of the events observed at the controller that influence its execution (for example, `PacketIn` messages). In case of a network failure, AFRO replays at accelerated speed the events prior to the failure within an isolated instance of the controller with its environment—except it pretends that the network topology never changed and all failed elements never existed since the beginning. Finally, AFRO recovers the actual network forwarding state by transitioning it to the recovered forwarding state of the emulated environment.

2. AUTOMATING FAILURE RECOVERY

AFRO works in two operation modes: *record mode*, when the network is not experiencing failures, and *recovery mode*, which activates after a failure is detected. The network and controller start in a combined clean forwarding state that later is gradually modified by the controller in response to `PacketIn` events. In record mode, AFRO records all `PacketIn` arriving at the controller and keeps track of currently installed rules by logging `FlowMod` and `FlowRem` messages. When a network failure is detected, AFRO enters the recovery mode. Recovery involves two main phases: *replay* and *reconfiguration*.

During replay, AFRO creates a clean copy of the original controller, which we call Shadow Controller. Shadow Controller has the same functionality as the original one, but it starts with a clean state and from the beginning works on a Shadow Network—an emulated copy of the actual network that is modified by removing failed elements from its topology. Then, AFRO feeds the Shadow Controller with recorded `PacketIn` messages processed by the original controller before the failure occurred. When replay ends, the Shadow Controller and Shadow Network contain a new forwarding state.

At this point, reconfiguration transitions the actual network from the current state to the new one. This transition requires making modifications to both the controller internal state as well as forwarding rules in the switches. First, AFRO computes a minimal set of rule changes. Then, it uses an efficient, yet consistent method to update all switches. Finally, it replaces the controller state with the state of Shadow Controller and ends the recovery procedure. In case another failure is detected during recovery, the system needs to interrupt the above procedure and restart with another Shadow Network representing the current topology.

2.1 Replay

Replay needs to be quick and efficient. While AFRO is reacting to the failure, the network is vulnerable and its performance is suboptimal even if there is fast failover implemented. Further, consecutive failures may cause issues that even failover can not handle. To improve the *replay* performance, we use two orthogonal optimizations. First, we filter `PacketIn` messages and choose only those necessary to reach the recovered forwarding state. Then, we parallelize the *replay* based on packet independence.

2.2 Network Reconfiguration

After the replay ends, AFRO needs to efficiently apply the changes without putting the network in an inconsistent state. The reconfiguration requires two steps: (i) modifying rules in the switches, and (ii) migrating the controller to a new state.

For rules in the switches we need a consistent and proportional update mechanism. If the size of an update is much bigger than the size of a required configuration change, it offsets the advantages of AFRO. Therefore we propose using a two-phase commit mechanism of per-packet consistent updates that is very similar to the one presented as optimization in [5]. We start modifying rules that will not be used and use a barrier to wait until switches apply a new configuration. At this point we install rules that direct traffic to the previously installed, new rules.

The transition between the old and new controller can be done either by exposing an interface to pass the entire state between the two controllers, or by replacing the old one altogether. In the second case, the Shadow Controller simply takes over the connections to real switches and takes responsibility of the original controller.

2.3 Prototype

We implemented a prototype of AFRO working with a POX controller platform. POX allows programmers to extend the controller functionality by adding new modules. Therefore, it is possible to introduce AFRO requiring no modifications to controller logic.

3. REFERENCES

[1] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford. A NICE Way to Test OpenFlow Applications. In *NSDI*, 2012.

[2] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1), 2008.

[3] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. Reproducible Network Experiments Using Container-Based Emulation. In *CoNEXT*, 2012.

[4] P. Perešíni, M. Kuźniar, N. Vasić, M. Canini, and D. Kostić. OF.CPP: Consistent Packet Processing for OpenFlow. In *HotSDN*, 2013.

[5] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for Network Update. In *SIGCOMM*, 2012.

[6] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester. Enabling Fast Failure Recovery in OpenFlow Networks. In *DRCN*, 2011.

[7] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann. OFRewind: Enabling Record and Replay Troubleshooting for Networks. In *USENIX ATC*, 2011.