Minimizing Downtimes: Using Dynamic Reconfiguration and State Management in SDN

Arne Schwabe
University of Paderborn
Email: arne.schwabe@uni-paderborn.de

Elisa Rojas
Telcaria Ideas S.L.
Email: elisa.rojas@telcaria.com

Holger Karl
University of Paderborn
Email: holger.karl@uni-paderborn.de

Abstract—**Software-Defined Networks (SDN) are constantly evolving and so is their software. One of the key advantages of SDN over traditional networks is the ability to rapidly develop and deploy new features. However, updating them often requires restarting the SDN controller and causes network downtime.**

In addition to such planned updates, unforeseen, accidental downtime is also a risk for SDN networks. While commercialized SDN controllers are adding mechanisms to deal with both planned and accidental downtime, they still are not competitive with conventional approaches, which typically use redundant hardware and special software to address these problems.

In this paper we investigate how these two challenges to SDN can be addressed with dynamic reconfiguration and show how the state of the network can be managed by reconfiguration. Finally, we present a proof-of-concept implementation of our approach.

I. Introduction

Conventional networks consist of dedicated hardware switches and proprietary software (firmware) running on them. The choice of software is limited; most times only one software image is available. Even if multiple software versions are available, the choice is usually small and license-driven (like "basic IP" and "advanced services"); changing licensing is rare. Software upgrades also happen rarely and are supported by specialized routines so that service is not interrupted. Typically, software is upgraded on a standby supervisor engine; then state is transferred; then the active engine is switched.

In this world of fixed components and monolithic software images, the need or even potential to change the running software does not exist. The user cannot simply exchange one part of the software, be it for the addition of features or to solve a problem with the code of that part of the software.

In stark contrast to that, SDN controllers, rather than having a monolithic software image, are typically composed of multiple modules (also called "apps"). Changing the composition of the modules –or also the modules themselves– is much more dynamic than in conventional scenarios. For example, since modules may come from different vendors, stability might be different and sometimes not ideal, requiring frequent tests and reconfigurations. Also, the interaction of the module composition might not have been tested at all and have issues that only manifest in this specific scenario. A controller restart, without a scheduled maintenance, to change or reinitialize the software is in most environments not an acceptable solution as it causes network downtime and intermediate failures.

Instead of treating the volatility of SDN controllers as a risk and a problem, we believe that this is a strength that can be capitalized on by introducing dynamic reconfiguration at runtime as a core component in an SDN deployment. Rather than having to stop/start the whole system, dynamic reconfiguration enables to only restart the modified parts of the system and can minimize the effects of restarts even further.

In this paper, we consider existing solutions and approaches to dynamic reconfiguration. First of all, we analyze related work (Section II). Secondly, we collect requirements of a dynamic reconfiguration in the context of SDN (Section III). We discuss how to fulfill these requirements by showing how malfunctioning modules can be detected (Section IV-D) and handled (Section IV-E). From the behavior in the unexpected case we develop dynamic reconfiguration behavior for a scheduled or manual reconfiguration in Section IV-F. After that, we present our specification language for dynamic reconfiguration in Section V and evaluate our implementation in Section VI and give a final conclusion in the last section.

II. Related work

SDN controllers help to break monolithic software into pieces or "apps". Controllers are usually single-language design and to synchronize their apps, they leverage dynamic frameworks already developed for their specific language.

The **OSGi** [?] technology componentizes software modules and applications, for the the Java programming language. OSGi allows applications or components (so-called *bundles*) to be remotely installed, started, stopped, updated, and uninstalled without requiring a reboot, which is not feasible in standalone Java environments. For example, Apache Felix [?] implements the OSGi framework and service platform in a basic form. It is extended by **Apache Karaf** [?], providing some additional features on top of a standard OSGi implementation, such as folder-based hot deployment, remote SSH access to the console, or centralized logging. Additionally, iPOPO [?], based on Pelix, is an OSGi dynamic service platform written in Python, and Apache Celix [?] is an implementation of the OSGi specification adapted to C. These OSGi frameworks provide the programmer an environment to implement mechanisms that act on updating or crashing of a component.

Currently, there are two SDN frameworks based on OSGi and Karaf: OpenDaylight (ODL) [?] and Open Network Operating System (ONOS) [?]. **ODL** is based on a Model-Driven Service Abstraction Layer (MD-SAL) architecture, which lets its different components – called *projects* – share information. ODL lacks an API to share common structures for dynamic reconfiguration of the network; e.g. if an ODL project needs to save information about flow entries in the network before being restarted, it is itself responsible for saving them. **ONOS** follows a well-defined architecture with specific APIs. For example, it is possible to obtain information about flow entries in the network and if a module is uninstalled, the flows related to it will disappear in the network as well. Although ONOS handles dynamic configuration for flow entries, it is still very oriented towards its own modules and is Java-oriented. To add external functionality, we should redevelop it to fit the language and architecture of the specific SDN framework. Moreover, these frameworks still lack a way to detect malfunctioning modules.

The authors in [**?**] analyze the requirements of a memory management tool to optimize system performance for five SDN controllers, which is one of the keys for smooth dynamic reconfiguration. It is a good approach but it does not address all the requirements for dynamic reconfiguration.

Research how to handle state on network update exists, albeit without focusing on reconfiguration of the whole system [**?**], [**?**]. While we focus on the state handling between different apps, their ideas and concepts can still be applied on top of our concepts for handling the state while reconfiguring the system.

III. Requirements for dynamic reconfiguration

We define the reconfiguration of the SDN network here as the possibility to replace, reload, add or remove a module in the currently running configuration without restarting or affecting the reminder of the SDN software. In this section, a module is a synonym for an SDN controller app, but it could be more general and even correspond to an entire controller.

The reasons for dynamic reconfiguration can be placed into two categories: (1) planned reconfigurations (moving the SDN controller to different hardware/software, changing the context: reconfiguring the services and resources based on changes in user needs, business goals, and/or environmental conditions) and (2) unplanned reconfigurations. Supporting both categories is equally important and implies three key requirements.

During a reconfiguration (e.g., implying a controller restart), the controller might not be able to process requests, impeding network traffic. This is clearly undesirable. Hence, the **first requirement** is to minimize the time for reconfiguration itself.

The reconfiguration itself needs to be triggered by an event. This is trivial for a planned reconfiguration. But for an unplanned configuration, it will be caused by detecting that the system is not working as intended: a part of the system is malfunctioning or has crashed. This leads to a **second requirement**: detection of malfunctioning parts or modules.

The network should be disrupted as little as possible when a reconfiguration takes place; therefore, probably the most important aspect of dynamic reconfiguration is the transition from the old state to the new state. Thus, the **third requirement** is to gracefully handle the state during the reconfiguration.

IV. Design

A. *External or internal?*

We identified two approaches of implementing the dynamic reconfiguration of modules in an SDN network (see Figure 1).

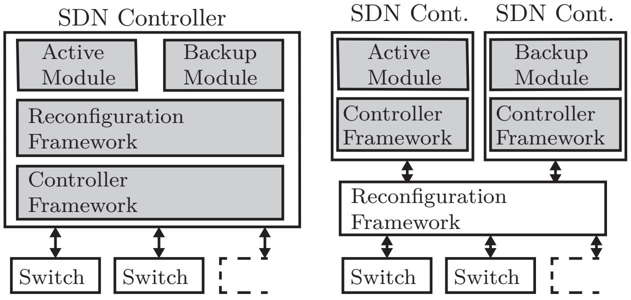

Fig. 1. Simplified comparison of approaches to implement reconfiguration as SDN controller component (left) and as independent component (right)

The first one keeps the SDN controller as the central instance in the SDN stack and integrates reconfiguration mechanisms in the controller. The second one has a separate entity that acts as central component to handle the actual controller and to control the restarting and reconfiguration of the controller and its apps. These approaches are orthogonal to whether the SDN controller is implemented in a distributed manner or not.

In the controller-based approach, the controller's own mechanisms can be extended and the dynamic reconfiguration can be tightly coupled with the controller's features. In practice, it is difficult to maintain such an approach when the underlying controller framework keeps developing. The approach with a separate entity may make the integration with the SDN controller harder but offers more flexibility and possibilities. In particular:

- Different modules can be separated into individual processes, which in turn improves reliability.
- The reconfiguration process becomes controller-framework-agnostic. This allows to extend reconfiguration even when multiple controllers work in parallel under the control of this central component, which can then also do composition of multiple modules (see NetIDE [**?**], [**?**]).
- Composition mechanism for multiple controllers (like CoVisor [**?**] or OpenVirtex [**?**]) can be integrated.
- Radical reconfigurations are possible since the SDN controller is no longer an entity that cannot be stopped.

There is ample evidence for the feasibility of such an external approach. Using a small external coordinator to improve the reliability of the whole system is also used in high availability systems in which multiple instances of a software, or even different software/hardware implementations, are used; the external component monitors the instance and selects a working instance to control the whole system (e.g., [**?**]).

In this paper, we are hence looking in detail at the second approach since its greater flexibility facilitates more powerful reconfiguration. We expect the line between these approaches will be blurred in the future by distributed SDN controllers that are a hybrid of the two approaches and incorporate a distributed version of the reconfiguration framework.

B. *Granularity*

As we base our design around the idea that the SDN network is controlled by multiple modules that run on one or more controllers, we have to also consider this for the central component's design. On the coarsest granularity, a module is a whole SDN controller. Enhancing the SDN controller to communicate more information about its loaded applications and annotating the network commands with an identifier of the application allows the central component to treat the individual applications of a controller as individual modules, and allows reconfiguration to treat them as individual modules instead of a having to treat a whole controller as module.

C. Specification of (Re-)Configurations

Reconfiguration at run-time requires that developers or deployers of an SDN system are able to specify the configuration somehow. First, this specification needs to be available at deployment time when an SDN controller along with is modules is brought online. Later on, the configuration might change during run-time; such changes then are reflected (if necessary) by a reconfiguration action.

Therefore, this specification language should support not only static configuration as such, but also the **re**configuration steps from one configuration to another. Ideally, the specification should only require reconfiguration specifications that cannot be calculated automatically from the differences between an old and a new configuration specification.

D. Detecting malfunctioning modules

One of the events for reconfiguration is handling malfunctioning components. For handling these run-time problems, we have to go through three steps. Firstly, we have to detect the problem to trigger the reconfiguration. Secondly, we need to choose one configuration that alleviates the problem. Thirdly, we perform a run-time reconfiguration.

Detecting a problem is generally hard (and strictly speaking, not even computable, cp. halting problem). Instead of checking if a module ceased to work, we hence decided a pragmatic approach that checks the *liveliness* of the deployed modules.

Malfunction of deployed modules can manifest in different ways. The most classic and basic form of malfunction is a *dead module* that crashes and ceases to respond to requests altogether. This can be detected using a simple heartbeat or timeout mechanism. For instance, if the module is using the OpenFlow protocol, the switch will periodically send `EchoReq` messages to the module. If the module is still alive, it will respond with an `EchoRes` message. Missing `EchoReq` responses from the module indicate a dead module. If the switch does not request `EchoReq` messages or the `EchoReq` message interval is too long for the required reaction time, the central component will send additional `EchoReq` messages to the module.

Apart from these implicit events of missing heartbeats, also explicit events can trigger a module to be marked as dead. The module can send an exit or failure message that indicates the module is no longer available (in the sense of a fail-stop error model). This message can also be intended, e.g. because the user explicitly stopped the module. Modules might also be restarted by external mechanism. In this case, the arrival of a new module with the same name and configuration is another event that marks it as dead and to be replaced by the new one.

Figure 2 shows a simple example of an active module with a standby module. After the active module stops to send keepalive packets, it is marked as "dead" and the central component switches over to the standby module.

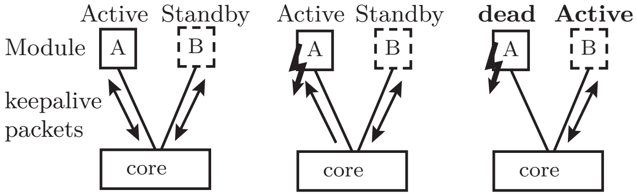

Fig. 2. Switchover between two modules after crash of primary module

Similar to the *dead* module, but harder to detect, is a *partially dead module* or *malfunctioning module* that still responds to some events or sends invalid or erratic messages. As an example, think of an ARP handler that uses multiple threads whose main thread, handling the `EchoReq`, is still alive and therefore the module is not marked as dead, but ARP requests are not answered anymore. To check liveliness in this scenario, the central component needs not only to check the general liveliness of the module but also the liveliness of a specific function, either by passive observation (e.g. ARP requests to the module should be followed by ARP answers) or by active checks (e.g. sending probe ARP requests).

Using external checks to monitor the network status and its components is already implemented in most data centers with specialized network monitoring software like Icinga [**?**].

E. Handling malfunctioning modules

To reach the goal of run-time reconfiguration with as little downtime as possible, malfunctioning modules also have to be handled fast. This requires an automatic solution that does not depend on manual intervention of an operator. But since not all modules that are failing are the same, the dynamic reconfiguration process needs some hints (from module developer or deployer) on how to handle the module. These hints can be provided offline, e.g., at deployment time.

For a malfunctioning module, we categorize the actions that can be taken into the following categories:

1) Repair the original module.
2) Replace the module with another module/configuration.
3) Control possible damage; do not restore the full functionality, but try to limit the impact of the malfunction.

Repairing the original module is usually accomplished by restarting the affected module (and potential dependencies). It differs from the usual "restart the controller" scenario since the central instance still manages the state while the module is restarting. Replacing the module is an option if an alternative module is available that can accomplish the same task, but perhaps not as well as the primary module. An example would be one sophisticated forwarding module that selects the forwarding paths using advanced utilization-based algorithms; the module has as backup a simpler forwarding module that only selects a path randomly among the shortest paths. Controlling possible damage is an option to contain a malfunctioning module without affecting the rest of the network.

For each of these options we have to treat the original module and its state, which can be broadly categorized into three categories:

1) the *internal state* of the module,

2) the observable and *controllable state* of the module,
3) the state of the module in external systems.

Let us consider an example of what these states are for a simple forwarding module. The internal state of the module is everything that the module holds in its own memory, like the path decisions made or internal statistics. The observable and controllable state consists mainly, but is not limited to, the installed flow rules. These rules can be tracked by the central component via the interaction of the module over its south-bound interface. In the reconfiguration event, this state be changed and controlled, e.g. removing the installed flow rules. The last state is the state in external system. In this example, the forwarding module could have answered the queries for the default gateway IP address with a specific MAC address, which created an entry in the ARP table of the connected hosts or, in other words, a state in an external system.

For a malfunctioning module, its internal state has to be considered as invalid or lost. If some of it is important the module needs to have its own mechanism to checkpoint and restore this state. Most modules will already have most of this implemented as restarting a module without dynamic reconfiguration also requires reading in configuration values and other persistent data stores.

The observable and controllable state is either kept or removed. For a more advanced scenario, also a combination is conceivable, e.g. removing or keeping only flow mods matching a certain pattern. There is no general rule to decide whether keeping or removing the state is preferable. This also depends on the function of a particular module. For example, when replacing a forwarding module, it is better to keep the old state and let the rules be slowly replaced with new rules.On the other hand, a firewall requires a consistent set of rules and mixing flow rules from different firewall applications might create unforeseeable problems, which means that we should provide a clean state on switchover. Hence, we again foresee an option for the module developer to specify the behavior desired from the central component.

In some instances (e.g. the mentioned forwarding module) it might be the best solution to keep the existing forwarding paths active to minimize network disruption. The failing module might not even be under the control of central component and it can only wait for the module to (hopefully) recover.

F. Adding/removing modules

When adding or removing a module, the behavior is slightly different from handling an exception. For a planned dynamic reconfiguration, the user of the SDN network will supply a configuration that supersedes the old one.

To accept the new configuration, the run-time system has to make sure that the new configuration is valid and that switching to it does not cause additional downtime. To clarify the verification of the new configuration, we only require a validation that all prerequisites of a new configuration are fulfilled. Checking and activating the new configuration has to go through the following steps:

- Checking the syntax of the new configuration.
- Checking if all (required) modules for the new configuration are present or, if necessary, wait for the new modules to connect.
- For removed modules, their state has to be handled. This is almost identical to the exception case. But since a module is usually removed intentionally, the default, unless specified otherwise, is to remove as much of its state as possible.
- Replace the old configuration with the new configuration.

V. Specification language

For the specification language, we extend here an XML-based specification language that we proposed earlier [**?**] with composition of modules into complex network applications.

For reconfiguration, the run-time configuration has two basic blocks: the **Modules** and the **ExecutionPolicy** blocks. Listing 1 provides an example of the specification language.

```
<RuntimeConfiguration>
  <Modules>
    <Module id=”firewall”>
      <liveliness type=”ofPing”/>
      <recovery type=”restart” />
    </Module>
    <Module id=”fwd”>
      <liveliness type=”timeout”>3000</liveliness>
      <liveliness type=”plugin”>eu.netide.arpchecker</liveliness>
      <recovery type=”replace”>slowfwd</recovery>
    </Module>
    <Module id=”slowfwd”>
      <recovery type=”ignore” />
    </Module>
  </Modules>
  <ExecutionPolicy>
    <ModuleCall id=”firewall” dpid=”2_7_9”/>
    <ModuleCall id=”fwd” dpid=”1_3_4”>
  </ExecutionPolicy>
</RuntimeConfiguration>
```

Listing 1. (Condensed) Specification Example

The **Modules** block specifies the modules with their behavior vis-à-vis reconfiguration using these attributes:

`id`: Unique identifier for each module.

`liveliness`: One or several tests that must be (all) passed for module to be considered alive. Some are available as internal checks, e.g. OpenFlow ping or failure to respond to `PACKET_IN`s in a given time (`timeout`). Sophisticated checks can be specified by external classes.

`recovery`: It specifies the action performed when the liveliness tests fail. Available options are: restart the module, ignore the module henceforth (do not accept any input from or forward information to the module), or to replace it, in which case the module is ignored and instead another module whose identifier is specified here is used in its place. In effect, the replacement's module ID is used instead of the replaced module's ID in the ExecutionPolicy. We also provide the possibility of having a custom class handle the recovery, similar to the liveliness check.

`state`: How to handle the module state. So far, we implemented `remove|keep|keepUntilRecover` where the last option is a hybrid that keeps the rules until the module is replaced or recovered. Bespoke state handling is easy to conceive and integrate.

The **ExecutionPolicy** block specifies the composition. Since we focus on the reconfiguration, we keep that section brief

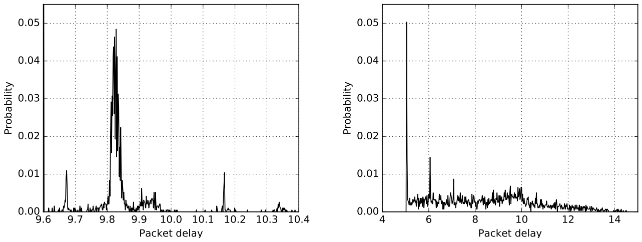

Fig. 3. Recover time seen from an application, left 0.2 s and right 20 s interval

here and provide a way to map modules to switches. The **ModuleCall** statement links the DPID of a switch with the Module to call if an event from this switch arrives.

VI. Evaluation

A quantitative evaluation is difficult and the implementation specific metrics (number of packets lost, time for to detect a malfunction) have not much significance to prove the concept. We therefore focus on a proof-of-concept evaluation.

We used a Ryu [**?**] learning switch. For a reconfiguration, we start a second instance and load a new configuration that directs all traffic to the new switch. The new configuration was applied instantly with no measurable delay or packet loss.

To test an unplanned reconfiguration we used the same scenario with dead module timeout of 10000 ms. We send a Unix SIGSTOP signal to the first controller to simulate a non-responding controller. When that happens the (queued) answer of the backup controller were send to the network.

The detection time is naturally dependent of the frequency of packets. The smaller the packets inter-arrival time, the quicker a non response is detected. The OpenFlow heartbeat was 5 s in our test setup. We ran the standard ping command and used its reported latency. The result for ping with an interval of 0.2 s in shown in Figure 3. Since the ping interval is much shorter than the heartbeat interval, most timeouts are triggered by ping itself and most responses are therefore also in the 10 s timeout with other peaks in 0.2 s intervals. The main peak being at 9.8 s rather than 10 s is an artifact of ping's RTT reporting.

With a 20 s ping interval, timeouts occurring from the heartbeat interval (5 s) are more significant and result in observed ping latency distribution from 5 to 15 s. The larger interval allows ARP timeouts to happen which add additional packets on the network which on themselves trigger timeouts, of the 10 s the 5 s from ARP *not* are included in ping's RTT .

The results have some minor irregularities, which also are expected in an emulated experiment setup like MiniNet, but otherwise confirm what is expected for a timeout based system, which shows that our approach is working.

VII. Conclusion & Future work

We are confident that our proposed reconfiguration methods and our implementation of the proposed mechanisms are an essential step for making SDN controllers and networks more resilient and further reduce the planned and unplanned down times in SDN networks.

The current prototype only uses a single central instance for controlling the dynamic reconfiguration. We will look into developing a distributed system to further reduce downtime and increase resilience.

Almost all of the concepts in this paper do not require a specific SDN protocol like OpenFlow. The existing implementation could be extended to cover other non-OpenFlow SDN protocols without conceptual difficulties.

Acknowledgments

Work presented here has been partially sponsored by the European Union through the FP7 project NetIDE, grant agreement 619543.