Technical Disclosure Commons

Defensive Publications Series

August 06, 2018

SOFTWARE DEFINED NETWORKING CONTEXT DISSOCIATION

Kevin Sheeran
Jerry David
Ramesh Mishra
Anand Pulicat Gopalakrishnan

Follow this and additional works at: https://www.tdcommons.org/dpubs_series

Recommended Citation
Sheeran, Kevin; David, Jerry; Mishra, Ramesh; and Gopalakrishnan, Anand Pulicat, "SOFTWARE DEFINED NETWORKING CONTEXT DISSOCIATION", Technical Disclosure Commons, (August 06, 2018)
https://www.tdcommons.org/dpubs_series/1389

This work is licensed under a Creative Commons Attribution 4.0 License.
This Article is brought to you for free and open access by Technical Disclosure Commons. It has been accepted for inclusion in Defensive Publications Series by an authorized administrator of Technical Disclosure Commons.
SOFTWARE DEFINED NETWORKING CONTEXT DISSOCIATION

AUTHORS:
Kevin Sheeran
Jerry David
Ramesh Mishra
Anand Pulicat Gopalakrishnan

ABSTRACT
Context dissociation techniques are described herein for Software Defined Networking (SDN) applications. This prevents a poor user experience wherein the user is forced to look at a spinner until the operation is complete.

DETAILED DESCRIPTION
Today, it is common for user interfaces that sit at the end of the software stack to configure hardware. Network administrators are accustomed to using applications that do not use spinners that block the user interface. This is because most servers simply need to update a database. In the Software Defined Networking (SDN) context, the network devices and configurations need to be pushed first. This scale can become astronomical for large enterprises.

Adding or removing commands on multiple devices (i.e., configuration) from a central SDN application takes too much time. This creates a poor user experience as the user is presumably forced to look at a spinner until the operation is complete.

The client and user interface (UI) may be used interchangeably. As illustrated in Figure 1 below, the SDN application may send a POST request to the server. The request may contain a device list as well as the configurations per device. The POST payload itself may be arbitrary. At this point the user interface will show a spinner to the user.

The application may then start periodically polling the server for an updated response. The server’s first job is to validate that the specifics of the arbitrary request makes sense. This is the validation step.
If the validation step is successful, the server may store the business logic in the database. As illustrated in Figure 2 below, the server may then set an arbitrary string letting the client know that the validation was successful and that the application intends to program these devices. When the client receives this message during the periodic polling process, the user interface may re-render, meaning the spinner may stop and the devices may have some updated indication (e.g., a color change). Depending on the network connection, the spinner may appear instantaneously since the validation code on the server side may be fast.

![Figure 2](image)

If the validation step failed, the business logic may not have been coherent. In this case, the server may set the error message. The client may receive the validation error message during the polling process. This may prompt the spinner to stop and show the error on the user interface.

At this time the user may click where desired without having to look at the spinner. However, the client may continue periodically polling the server since the server may now try to configure the devices. This is the context dissociation, and is illustrated in Figure 3 below.

![Figure 3](image)

Each device may either accept or reject the configuration with arbitrary commands. The server may keep track of the result of each configuration attempt. The server may again
update the information in the polling message to make the user interface aware of the current configuration status. This enables immediately showing per device updates to the user.

Figure 4 below illustrates the asynchronous behavior of configuring multiple devices at once. A configuration from the server to a device is denoted as a green vector. A failed configuration is denoted as a red vector. As shown, two devices have been successfully configured and three devices are currently under configuration.

As illustrated in Figure 5 below, all of the devices have successfully received the configurations from the server. At this point, the application has all the information it needs, and thus the polling will stop.

Figure 6 below illustrates a situation in which a device configuration fails. The application has all of the information it needs, and the polling may then stop. In this case, however, the user interface parses an error on device X from the periodic polling, and the user interface may display the error to the user.
Figure 6 illustrates provisioning with context dissociation.

Figure 7 illustrates an example spinner.
In summary, context dissociation techniques are described herein for SDN applications. This prevents a poor user experience wherein the user is forced to look at a spinner until the operation is complete.