Attendre: Mitigating Ill Effects of Race Conditions in OpenFlow via Queueing Mechanism

Xiaoye Sun Apoorv Agarwal T. S. Eugene Ng

Rice University
Houston, Texas, USA

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols; C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design

Keywords

OpenFlow, protocol design, verification

1. INTRODUCTION

According to the specification of the OpenFlow protocol, whenever a flow table entry is absent for any arriving data packet, a packet-in message is sent to the controller. This behaviour results in various types of race conditions in accord with various packet and message orderings. Increased forwarding delay of data packets, increased complexity in performing software verification, and increased load on switch and controller processors are the ill effects of these races.

We identify 3 types of races inherent in the OpenFlow protocol. The **Type 1** race is between data packets sent by a host and the command messages sent by the controller. Consider the scenario in which a host has sent two data packets sequentially. A switch, on receiving the first data packet, finds out that there is no entry matching the packet's attributes, so according to the default action, a packet-in message is sent to the controller. The controller processes this packet-in message and sends out a flow-mod message combined with a packet-out message (denoted by flow-mod/packet-out) to the switch.

The state diagram of a system experiencing the Type 1 race is shown in Figure 1. The q_0 state is when a switch has sent a packet-in message to the controller on arrival of the first data packet. Now, the next event at the switch is either the second data packet from the same flow, or the flow-mod/packet-out message (or command for short) from the controller. Consider case 1 when the switch receives the command before the data packet. In this case, the switch inserts an entry in its flow table and processes the next data packet from this flow according to the installed entry. This scenario causes the system state to change along $q_0 \to q_1 \to q_2$. Consider case 2 when the second data packet arrives at the switch before the command message. In this case, the switch will sent another packet-in message to the controller. The switch on receiving flow-mod/packet-out from the controller corresponding to this second packet-in will process the second packet. This makes the system state to change from $q_0 \to q_3 \to ... \to q_2$.

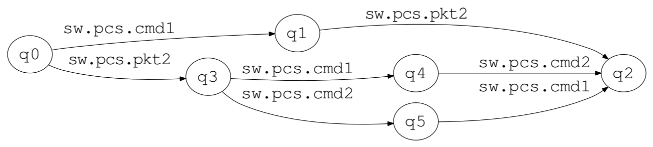
$$\begin{tikzcd}
 & q1 \arrow[rrr, "sw.pcs.pkt2"] & & & q2 \\
q0 \arrow[ur, "sw.pcs.cmd1"] \arrow[r, "sw.pcs.pkt2"'] & q3 \arrow[r, "sw.pcs.cmd1"] \arrow[dr, "sw.pcs.cmd2"'] & q4 \arrow[urr, "sw.pcs.cmd2"] & & \\
 & & q5 \arrow[uurr, "sw.pcs.cmd1"'] & &
\end{tikzcd}$$

Figure 1: Race types 1 & 2 state diagram.

Figure 1 also illustrates the **Type 2** race, which is a race between different command messages issued by the controller for a particular switch. This is possible when the controller is *multi-threaded*. A multi-threaded controller can have multiple threads processing several requests which may be contending for resources. This causes threads to finish their tasks in an arbitrary order. Branching at state q_3 in Figure 1 happens because of this reason. The branch $q_3 \to q_4 \to q_2$ corresponds to the state when the controller is able to process the packet-in for the first packet before processing the packet-in for the second packet. The other branch $q_3 \to q_5 \to q_2$ happens when the controller processes them in reverse.

The **Type 3** race (not depicted due to space limit) is experienced by the network when different command messages are sent by the controller to different switches, such as when a "route flow" controller configures a network path $S_1, S_2, ...$ Consider a case when a data packet arrives at switch S_{n+1} earlier than the command message meant for it. This is possible when the command message arrives at switch S_n before the arrival of the command message for switch S_{n+1}. In this scenario, a packet-in message will be sent by switch S_{n+1}, leading to a series of extra state transitions compared with a scenario in which the command message arrives at switch S_{n+1} before the data packet.

Problems due to the Race Conditions

(a) **Increased complexity in software verification:** The identified race conditions lead to a serious state space explosion problem in controller software verification because the verifier needs to exercise all the possible state transitions. Figure 2 shows the numbers of state transitions (hollow markers) generated during the verification of an OpenFlow "route flow" application for various number of switches and packets. The data are collected from NICE – an OpenFlow controller software verifi-

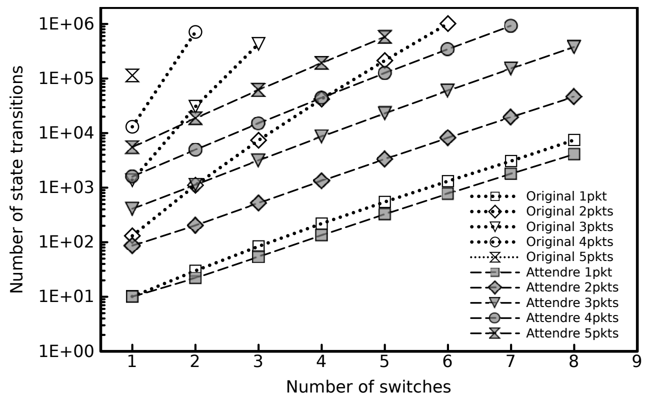

Figure 2: Number of state transitions incurred by NICE verification under various scenarios for original OpenFlow and Attendre.

cation tool[1]. It is important to note that NICE assumes a single-threaded controller, therefore the figures are highly conservative. If a multi-threaded controller is assumed, the occurrences of Type 2 races will cause further explosion in the state space.

(b) **Increased forwarding delay of packet:** The races increase the forwarding delay of a packet as whenever a switch receives a data packet without a matching entry in the flow table, a packet-in is sent to the controller. In the worst case, this could happen at every hop in the path. This increases the forwarding delay of the data packet significantly since in some cases the command message may arrive at the switch just after a packet-in is dispatched for the controller.

(c) **Increased processing load on switches and controller:** The extra packet-in, flow-mod/packet-out, and flow-mod messages caused by the races increase the processing load on switches and the controller. When a packet-in is sent, it is possible that the controller has already processed another packet-in for the same flow and the command message is on its way to the switch. Processing related to subsequent packet-in messages is thus redundant.

2. ATTENDRE MECHANISM

Attendre is used to allay the ill effects of the race conditions in an OpenFlow network by avoiding sending redundant packet-in messages. The core idea of Attendre is that if the switch is expecting a command message, the incoming packets that can be handled according to this command message should be queued at the switch buffer until the command comes.

To achieve this, each flow table entry should be tagged with a **version number** which is issued by the controller platform and sent to the switch together with the flow-mod message. The version number is a unique identifier used to differentiate old and new entries. The command message to a switch S_n should carry an additional match field and the version number of the flow table entry for the next hop switch S_{n+1}. This occurs when the application on the controller intends to install a forwarding path for the packet. The match field and version number for the next switch S_{n+1} will be piggybacked by the first data packet handled by the command message and carried to S_{n+1}. A new **WAIT** action is also defined in Attendre. The packets matching the entry with WAIT action type will be inserted into the switch buffer and a buffer ID will be associated to the entry. An entry with this WAIT action type will be inserted to the flow table when a packet-in is sent from a switch. A WAIT entry can also be inserted when a switch receives a data packet carrying the match field and version number information, and the corresponding command message has not been processed by the switch. The WAIT entry will be removed or replaced after the switch receives the corresponding command message. At the same time, the packets queued in the buffer associated with the being eliminated WAIT entry will be dequeued and processed according to the actions of the command.

3. BENEFITS TO VERIFICATION

We have integrated the Attendre mechanism into NICE by modifying its network component models accordingly. We compare the number of state transitions incurred by original OpenFlow and Attendre.

For simplicity, the network topology used is a simple chain. Two hosts, a sender and a replier, locate at the ends of the chain. The sender sends packets to the replier and will receive the same number of reply packets. The application on the controller is a “route flow” application that installs entries along the chosen route for the forwarding of the packets. The verification is done for different numbers of injected packets and switches along the path.

The numbers of state transitions generated during the model checking for different experiment configurations (upto eight switches and one million state transitions) are shown in Figure 2. When the number of packets or switches increases, the number of states transition increases exponentially. With one injected packet (square markers in Figure 2), Attendre could save the number of state transitions by about 50% if there are eight switches on the path. If there is more than one packet, the number of state transitions can be reduced by several orders of magnitude, since in this case, the following packet could queue at the switch which results in far more state transition reduction. For example, when there are six switches along the path and two packets injected (diamond markers in Figure 2), the state transitions could be reduced by two orders of magnitude. Note that NICE does not model the Type 2 race, it assumes a single-threaded controller. The results in Figure 2 for OpenFlow without Attendre are expected to become much worse if Type 2 race is accounted for. In other words, the actual benefit of Attendre is even larger than Figure 2 shows. It is not hard to see that if Attendre is adopted, far more complex network topologies and applications can be successfully verified for a given amount of computing resources.

4. REFERENCES

[1] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford. A nice way to test openflow applications. In *NSDI'12: Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'12, pages 127–140, Berkeley, CA, USA, 2012. USENIX Association.