RESEARCH ARTICLE OPEN ACCESS

Minimizing the Test Packet Failures by Applying Threshold on Test Packet Generation in Debugging and Network Testing

P. Anjaneyulu[1], T. Venkata Naga Jayudu[2]
[1](P.G student Dept of CSE, JNTUA University, Andhra Pradesh, India)
[2](Asst. Professor Dept of CSE, JNTUA University, Andhra Pradesh, India)

----------**********----------

Abstract:

It is very hard to debug networks. Every day network engineers struggle with router fiber cuts, misconfigurations, mislabeled cables, faulty interfaces, software bugs, intermittent links and additional reasons that are reason for networks to behave badly, or be unsuccessful completely. Network engineers chase down bugs by means of the most elementary tools like ping and trace route, and trail down origin causes by means of combination of perception and wisdom. Debugging networks is just becoming difficult as networks are getting larger and are getting more complex. This paper proposes an automatic testing and debugging procedure for verifying the various network conditions and to provide safe reaching of the packets to the desired destination. The paper uses the Automatic Test Packet Generation system for debugging and adds an enhancement to that system which restricts the number of test packets generated and removes the threshold violation over the file size limit. This reduces the time taken to debug the entire system.

Keywords — **Test Packet Generation, Network Troubleshooting, Data Plane Analysis.**

----------**********----------

I. INTRODUCTION

Operating a modern network is no easy task. Every day network engineers have to wrestle with misconfigured routers, fiber cuts, faulty interfaces, mislabeled cables, software bugs, intermittent links and a myriad other reasons that cause networks to misbehave, or fail completely. Network engineers hunt down bugs using the most rudimentary tools (e.g., ping, traceroute, SNMP, and tcpdump), and track down root causes using a combination of accrued wisdom and intuition. Debugging networks is only becoming harder as networks are getting bigger (modern data centers may contain 10,000 switches, a campus network may serve 50,000 users, a 100Gb/s long-haul link may carry 100,000 flows) and getting more complicated (with over 6,000 RFCs, router software is based on millions of lines of source code, and network chips often contain billions of gates). Small wonder that network engineers have been labeled "masters of complexity". Troubleshooting a network is difficult for good reasons. First, the forwarding state is distributed across multiple routers and firewalls and is defined by their forwarding tables, filter rules, and other configuration parameters. Second, the forwarding state is hard to observe, because it typically requires manually logging into every box in the network via the Command Line Interface (CLI). Third, there are many different programs, protocols, and humans updating the forwarding state simultaneously.

Facing this hard problem, network engineers deserve better tools than ping and traceroute. In fact, in other fields of engineering testing tools have been evolving for a long time. For example, both the ASIC and software design industries are buttressed by billion-dollar tool businesses that supply techniques for both static (e.g., design rule) and dynamic (e.g., timing) verification.

Modern computer networks can be divided into the

- Data plane and
- The control plane

The data plane consists of a number of interconnected switches; each contains forwarding rules that determine the flow of packets. For example, the forwarding rule in an Ethernet switch looks at a packet's destination MAC address, and decides its next port. On top of the data plane is the control plane that runs routing protocols such as OSPF or

BGP. The control plane populates the data plane with forwarding rules based on its global network knowledge.

The unfortunate realities of network operation make automatic, systematic data plane troubleshooting a necessity. However, there is more than one way to approach this problem. Moreover, different networks may call for different approaches. Any data plane tester design should answer the following three questions:

- **Method**: Do we only read and analyze forwarding tables (static analysis), or do we actually send out test packets to observe the network's behavior (dynamic analysis)?
- **Knowledge**: How much we know about the network under test? Do we know all the forwarding tables and topology, just part of them, or none of them?
- **Coverage**: Which network components do we cover, links or rules? How do we achieve 100% coverage? Is that even possible?

II. RELATED WORK

Let us start by solving the simple white box, dynamic testing problem:

Given all forwarding rules and network topology, what is the minimum set of test packets that can cover 100% of rules, links and interfaces? Moreover, how to use this set to localize and diagnose data plane problems? This kind of problems occurs frequently in today's network management. Consider two examples:

Example 1:

Suppose a router with a faulty line card starts dropping packets silently. Alice, who administers 100 routers, receives a ticket from several unhappy users complaining about connectivity. First, Alice examines each router to see if the configuration was changed recently, and concludes that the configuration was untouched. Next, Alice uses her knowledge of the topology to triangulate the faulty device with ping and traceroute. Finally, she calls a colleague to replace the line card.

Example 2:

Suppose that video traffic is mapped to a specific queue in a router, but packets are dropped because the token bucket rate is too low. It is not at all clear how Alice can track down such a performance fault using ping and traceroute.

Automatic Test Packet Generation (ATPG) is one answer to this dynamic testing problem:

ATPG automatically generates a set of packets to test the liveness of the underlying topology and the congruence between data plane state and configuration specifications. The tool can also automatically generate packets to test performance assertions such as packet latency. In Example 1 instead of Alice manually deciding which ping packets to send, the tool does so periodically on her behalf. In Example 2, the tool determines that it must send packets with certain headers to "exercise" the video queue, and then determines that these packets are being dropped.

ATPG detects and diagnoses errors by independently and exhaustively testing all forwarding entries, firewall rules, and any packet processing rules in the network. In ATPG, test packets are generated algorithmically from the device configuration files and FIBs, with the minimum number of packets required for complete coverage. Test packets are fed into the network so that every rule is exercised directly from the data plane. Since ATPG treats links just like normal forwarding rules, its full coverage guarantees testing of every link in the network. It can also be specialized to generate a minimal set of packets that merely test every link for network liveness. At least in this basic form, we feel that ATPG or some similar technique is fundamental to networks: Instead of reacting to failures, many network operators such as Internet2 proactively check the health of their network using pings between all pairs of sources. However all-pairs ping does not guarantee testing of all links, and has been found to be unscalable for large networks such as PlanetLab.

Organizations can customize ATPG tomeet their needs; for example, they can choose to merely check for network liveness (link cover) or check every rule (rule cover) to ensure security policy. ATPG can be customized to check only for reachability or for performance as well. ATPG can adapt to constraints such as requiring test packets from only a few places in the network, or using special routers to generate test packets from every port. ATPG can also be tuned to allocate more test packets to exercise more critical rules.

III. PROPOSED WORK

The main behind the proposed ATPG model is to restrict the number of packets generated so that the debugging time can be reduced to greater extent and the threshold over the file size limit will be removed so that the data file can be divided into equal groups among the minimum number of test packets.

3.1 Network Model:

ATPG uses the header space framework a geometric model of how packets are processed we described. In header space, protocol-specific meanings associated with headers are ignored: a header is viewed as a flat sequence of ones and zeros. A header is a point (and a flow is a region) in the $\{0,1\}^L$ space, where L is an upper bound on header length. By

using the header space framework, we obtain a unified, vendor-independent and protocol-agnostic model of the network1 that simplifies the packet generation process significantly.

3.2 Definitions

Packets: A packet is defined by a (port , header) tuple, where the port denotes a packet's position in the network at any time instant; each physical port in the network is assigned a unique number.

Router: A switch transfer function, T , models a network device, such as a switch or router. Each network device contains a set of forwarding rules (e.g., the forwarding table) that determine how packets are processed. An arriving packet is associated with exactly one rule by matching it against each rule in descending order of priority, and is dropped if no rule matches.

3.3 ATPG System:

Based on the network model, ATPG generates the minimal number of test packets so that every forwarding rule in the network is exercised and covered by at least one test packet. When an error is detected, ATPG uses a fault localization algorithm to determine the failing rules or links.

Figure 1 is a block diagram of the ATPG system. The system first collects all the forwarding state from the network (step 1). This usually involves reading the FIBs, ACLs and config files, as well as obtaining the topology. ATPG uses Header Space Analysis to compute reachability between all the test terminals (step 2).

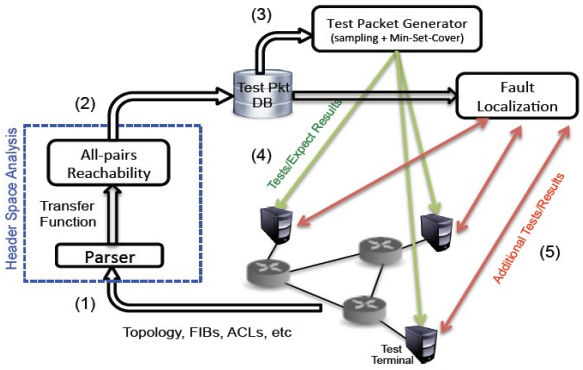

Fig 1: ATPG Architecture

The result is then used by the test packet selection algorithm to compute a minimal set of test packets that can test all rules (step 3). These packets will be sent periodically by the test terminals (step 4). If an error is detected, the fault localization algorithm is invoked to narrow down the cause of the error (step 5). While steps 1 and 2 are described in, steps 3 through 5 are new.

3.3.1 Test Packet Generation Algorithm:

A set of test terminals in the network can send and receive test packets. Our goal is to generate a set of test packets to exercise every rule in every switch function, so that any fault will be observed by at least one test packet. This is analogous to software test suites that try to test every possible branch in a program. The broader goal can be limited to testing every link or every queue. When generating test packets, ATPG must respect two key constraints: (1) Port: ATPG must only use test terminals that are available; (2) Header: ATPG must only use headers that each test terminal is permitted to send.
For example, the network administrator may only allow using a specific set of VLANs. Formally:
Problem1 (Test Packet Selection) For a network with the switch functions, {T1, ...,Tn}, and topology function, ¡, determine the minimum set of test packets to exercise all reachable rules, subject to the port and header constraints.

ATPG chooses test packets using an algorithm we call Test Packet Selection (TPS). TPS first finds all equivalent classes between each pair of available ports. An equivalent class is a set of packets that exercises the same combination of rules. It then samples each class to choose test packets, and finally compresses the resulting set of test packets to find the minimum covering set.

Generate all-pairs reachability table. ATPG starts by computing the complete set of packet headers that can be sent from each test terminal to every other test terminal. For each such header, ATPG finds the complete set of rules it exercises along the path. To do so, ATPG applies the all-pairs reachability algorithm described, on every terminal port, an all-x header (a header which has all wild carded bits) is applied to the transfer function of the first switch connected to each test terminal. Header constraints are applied here. For example, if traffic can only be sent on VLAN A, then instead of starting with an all-x header, the VLAN tag bits are set to A. As each packet pk traverses the network using the network function, the set of rules that match pk are recorded in pk. history.

3.3.2 Fault Localization:

ATPG periodically sends a set of test packets. If test packets fail, ATPG pinpoints the fault(s) that caused the problem.

Faultmodel

A rule fails if its observed behavior differs from its expected behavior. ATPG keeps track of where rules fail using a result function R. For a rule r , the result function is defined as

$$R(r, pk) = \begin{cases} 0 & \text{if } pk \text{ fails at rule } r \\ 1 & \text{if } pk \text{ succeeds at rule } r \end{cases}$$

"Success" and "failure" depend on the nature of the rule: a forwarding rule fails if a test packet is not delivered to the intended output port, whereas a drop rule behaves correctly when packets are dropped. Similarly, a link failure is a failure of a forwarding rule in the topology function. On the other hand, if an output link is congested, failure is captured by the latency of a test packet going above a threshold. We can divide faults into two categories: action faults and match faults. An action fault occurs when every packet matching the rule is processed incorrectly. Examples of action faults include unexpected packet loss, a missing rule, congestion, and mis-wiring.

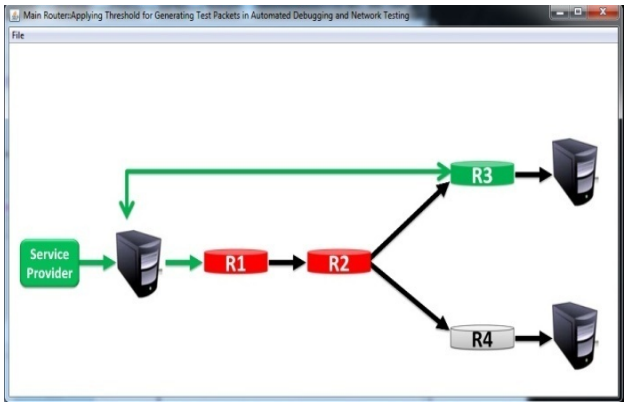

Figure 2: Router Selection Process

The above figure clearly depicts that the router selection process is done via checking the drop condition if the router is in drop state (off) the ATPG system automatically selects another router for data transferring.

IV. ANALYSIS

The system is tested only for the time taken to debug the entire network. As the proposed system claims to reduce the debugging time compared to the existing system we compare our results with the existing system.

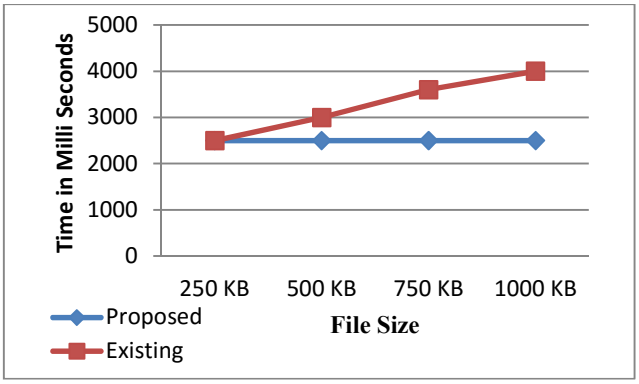

Figure 3: Debugging time Comparison

As the figure clearly depicts that the proposed system utilizes constant size for debugging this is due to that the number of test packets generated will be equal at all data sizes. Whereas the existing system generates more number of test packets with the growth of the data size which will clearly takes more debugging time.

V.CONCLUSION

In any complex system, accidents and failures are the norm rather than the exception. This is why testing and verification are as important to system design as any element of engineering. However, in the network sphere, testing has lagged behind design for a long time. The tools and methods described in this paper, along with other recent efforts from the network research community, demonstrate the power of rigorous, systematic, and automatic network testing. The paper proposed an ATPG system that generates only a minimal set of test packets and removes the threshold over the file size limitation. Doing this the proposed system clearly outperformed the existing system in the time taken to debugging.

REFERENCES

[1] Hongyi Zeng, Peyman Kazemian," Automatic Test Packet Generation", IEEE/ACM Transactions on Networking, April 2014, PP. 554 – 566.

[2] Scott Shenker. The future of networking, and the past of protocols. http:// pennetsummit.org/talks/shenker-tue.pdf.

[3] The Internet2 Observatory Data Collections. http://www.internet2.edu/ observatory/archive/data-collections.html.

[4] All-pairs ping service for PlanetLab ceased. http://lists.planet-lab.org/pipermail/users/2005-July/001518.html.

[5] Peyman Kazemian, George Varghese, and Nick

McKeown. Header Space Analysis: static checking for networks. Proceedings of NSDI'12, 2012.

[6] Franck Le, Sihyung Lee, Tina Wong, Hyong S. Kim, and Darrell Newcomb. Detecting network-wide and router-specific misconfigurations through data mining. IEEE/ACMTrans. Netw., 17(1):66–79, February 2009.

[7] Harsha V. Madhyastha, Tomas Isdal, Michael Piatek, Colin Dixon, Thomas Anderson, Arvind Krishnamurthy, and Arun Venkataramani. iplane: an information plane for distributed services. In Proceedings of OSDI'06, pages 367–380, Berkeley, CA, USA, 2006. USENIX Association.

[8] Hassel, the header space library.https://bitbucket.org/peymank/hassel-public/.

[9] The Internet2 Observatory Data Collections. http://www.internet2.edu/observatory/archive/ data-collections.html.

[10] Ajay Mahimkar, Zihui Ge, Jia Wang, Jennifer Yates, Yin Zhang, Joanne Emmons, Brian Huntley, and Mark Stockert. Rapid detection of maintenance induced changes in service performance. In Proceedings of the 2011 ACM CoNEXT Conference, pages 13:1–13:12, New York, NY, USA, 2011. ACM.

[11] M. Jain and C. Dovrolis. End-to-end available bandwidth: measurement methodology, dynamics, and relation with tcp throughput. IEEE/ACM Trans. Netw. 11(4):537{549,Aug. 2003.

[12] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: static checking for networks. Proceedings of the 9th conference on Symposium on Networked Systems Design & Implementation, 2012.

[13] R. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren. Ip fault localization via risk modeling. In Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation – Volume 2, NSDI'05, pages 57{70, Berkeley, CA, USA, 2005. USENIX Association.

[14] M. Kuzniar, P. Peresini, M. Canini, D. Venzano, and

[15] Kostic. A SOFT way for open ow switch interoperability testing. In Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies, CoNEXT '12, 2012.