IJDCST @Oct-Nov-2015, Issue- V-3, I-7, SW-01
ISSN-2320-7884 (Online)
ISSN-2321-0257 (Print)

# A Proactive Approach to Check the Health of Network

**Zarina Begum[1], Sayeed Yasin[2]**

**[1] M.Tech (CSE), Nimra College of Engineering & Technology, A.P., India.**

**[2]Head of the Department, Dept. of Computer Science & Engineering, Nimra College of Engineering & Technology, A.P., India.**

*Abstract*— Of late there is a mammoth increase in networks, and facing insurmountable complexities. Yet administrators rely on basic tools such as ping and traceroute to debug problems. We propose an automated and systematic approach for testing and debugging networks called "Automatic Test Packet Generation" (ATPG). ATPG reads router configurations and generates a device-independent model. The model is used to generate a minimum set of test packets to (minimally) exercise every link in the network or (maximally) exercise every rule in the network. Test packets are sent periodically, and detected failures trigger a separate mechanism to localize the fault. ATPG can detect both functional (e.g., incorrect firewall rule) and performance problems (e.g., congested queue). ATPG complements but goes beyond earlier work in static checking (which cannot detect live-ness or performance faults) or fault localization (which only localize faults given live-ness results). We describe our prototype ATPG implementation and results on two real-world data sets: Stanford University's backbone network and Internet2. We find that a small number of test packets suffices to test all rules in these networks: For example, 4000 packets can cover all rules in Stanford backbone network, while 54 are enough to cover all links. Sending 4000 test packets 10 times per second consumes less than 1% of link capacity. ATPG code and the data sets are publicly available.

***Keywords* — Data plane analysis, network troubleshooting, test packet generation.**

## I. INTRODUCTION

Networks are becoming larger and larger. And it is extremely hard to debug the problems emerged in networks. Every day, network engineers wrestle with router misconfigurations, 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 basic 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 100-Gb/s long-haul link may carry 100 000 flows) and are getting more complicated (with over 6000 RFCs, router software is based on millions of lines of source code, and network chips often contain billions of gates). It is a small wonder that network engineers have been labeled "masters of complexity" [1]. 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.

Troubleshooting a network is difficult for three 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. Third, there are many different programs, protocols, and humans updating the forwarding state simultaneously. When Alice uses ping and traceroute, she is using a crude lens to examine the current forwarding state for clues to track down the failure.

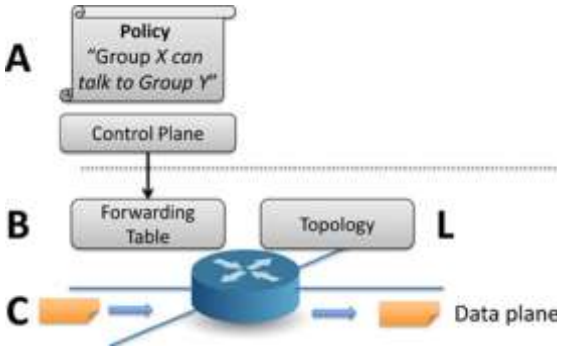


Figure 1 Static versus dynamic checking

Fig. 1 is a simplified view of network state. At the bottom of the figure is the forwarding state used to forward each packet, consisting of the L2 and L3 forwarding information base (FIB), access control lists, etc. The forwarding state is written by the control plane (that can be local or remote as in the SDN model [1]) and should correctly implement the network administrator's policy. Examples of the policy include: "Security group X is isolated from security Group Y," "Use OSPF for routing," and "Video traffic should receive at least 1 Mb/s." We can think of the controller compiling the policy (A) into device-specific configuration files (B), which in turn determine the forwarding behavior of each packet (C). To ensure the network behaves as designed, all three steps should remain consistent at all times, i.e., $A = B = C$. In addition, the topology, shown to the bottom right in the figure, should also satisfy a set of liveness properties $L$. Minimally $L$, requires that sufficient links and nodes are working; if the control plane specifies that a laptop can access a server, the desired outcome can fail if links fail. $L$ can also specify performance guarantees that detect flaky links.

Recently, researchers have proposed tools to check that A = B, enforcing consistency between policy and the configuration [2]. While these approaches can find (or prevent) software logic errors in the control plane, they are not designed to identify liveness failures caused by failed links and routers, bugs caused by faulty router hardware or software, or performance problems caused by network congestion. Such failures require checking $L$ for and whether $B = C$. Alice's first problem was with $L$ (link not working), and her second problem was with $B = C$ (low level token bucket state not reflecting policy for video bandwidth).

The main contribution of this paper is what we call an Automatic Test Packet Generation (ATPG) framework that automatically generates a minimal set of packets to test the live-ness 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 [3] 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 un-scalable for large networks such as PlanetLab [4].

Organizations can customize ATPG to meet 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. For example, a healthcare network may dedicate more test packets to Firewall rules to ensure HIPPA compliance.

## II. OUR SYSTEM

Based on the network model, our system 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, our system uses a fault localization algorithm to determine the failing rules or links.

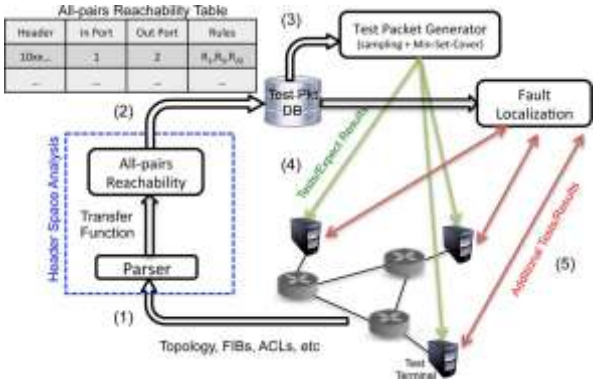


**Figure 2 System Block diagram**

Fig. 2 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 [5] to compute reachability between all the test terminals all rules (step 2). The result is then used by the test packet selection algorithm to compute a minimal set of test packets that can test (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 [5], steps 3–5 are new.

### *A. Test Packet Generation*

Algorithm: We assume 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.

*B. Fault Localization*

We 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. On the other hand, match faults are harder to detect because they only affect some packets matching the rule: for example, when a rule matches a header it should not, or when a rule misses a header it should match. Match faults can only be detected by more exhaustive sampling such that at least one test packet exercises each faulty region. For example, if a TCAM bit is supposed to be , but is "stuck at 1," then all packets with a 0 in the corresponding position will not match correctly. Detecting this error requires at least two packets to exercise the rule: one with a 1 in this position, and the other with a 0.

## III. IMPLEMENTATION

We implemented a prototype system to automatically parse router configurations and generate a set of test packets for the network. The code is publicly available [1].

*A. Test Packet Generator*

The test packet generator, written in Python, contains a Cisco IOS configuration parser and a Juniper Junos parser. The dataplane information, including router configurations, FIBs, MAC learning tables, and network topologies, is collected and parsed through the command line interface (Cisco IOS) or XML files (Junos). The generator then uses the Hassel [6] header space analysis library to construct switch and topology functions.

All-pairs reachability is computed using the multi-process parallel-processing module shipped with Python. Each process considers a subset of the test ports and finds all the reachable ports from each one. After reachability tests are complete, results are collected, and the master process executes the Min-Set-Cover algorithm. Test packets and the set of tested rules are stored in a SQLite database.

*B. Network Monitor*

The network monitor assumes there are special test agents in the network that are able to send/receive test packets. The network monitor reads the database and constructs test packets and instructs each agent to send the appropriate packets. Currently, test agents separate test packets by IP Proto field and TCP/UDP port number, but other fields, such as IP option, can also be used. If some of the tests fail, the monitor selects additional test packets from reserved packets to pinpoint the problem. The process repeats until the fault has been identified. The monitor uses JSON to communicate with the test agents, and uses SQLite's string matching to lookup test packets efficiently.

*C. Alternate Implementations*

Our prototype was designed to be minimally invasive, requiring no changes to the network except to add terminals at the edge. In networks requiring faster diagnosis, the following extensions are possible.

*Cooperative Routers:* A new feature could be added to switches/routers, so that a central ATPG system can instruct a router to send/receive test packets. In fact, for manufacturing testing purposes, it is likely that almost every commercial switch/router can already do this; we just need an open interface to control them.

*SDN-Based Testing*: In a software defined network (SDN) such as OpenFlow [27], the controller could directly instruct the switch to send test packets and to detect and forward received test packets to the control plane. For performance testing, test packets need to be time-stamped at the routers.

## IV. RELATED WORK

We are unaware of earlier techniques that automatically generate test packets from configurations. The closest related works we know of are offline tools that check invariants in networks. In the control plane, NICE [7] attempts to exhaustively cover the code paths symbolically in controller applications with the help of simplified switch/host models. In the data plane, Anteater [8] models invariants as Boolean satisfiability problems and checks them against configurations with a SAT solver. Header Space Analysis [5] uses a geometric model to check reachability, detect loops, and verify slicing. Recently, SOFT [9] was proposed to verify consistency between different OpenFlow agent implementations that are responsible for bridging control and data planes in the SDN context. ATPG complements these checkers by directly *testing* the data plane and covering a significant set of dynamic or performance errors that cannot otherwise be captured.

End-to-end probes have long been used in network fault diagnosis in work such as [10]. Recently, mining low-quality, unstructured data, such as router configurations and network tickets, has attracted interest. By contrast, the primary contribution of ATPG is not fault localization, but determining a compact set of end-to-end measurements that can cover every rule or every link. The mapping between Min-Set-Cover and network monitoring has been previously explored in [3] and [5]. ATPG improves the detection granuality to the rule level by employing router configuration and data plane information. Furthermore, ATPG is not limited to liveness testing, but can be applied to checking higher level properties such as performance.

Our work is closely related to work in programming languages and symbolic debugging. We made a preliminary attempt to use KLEE [11] and found it to be 10 times slower than even the un-optimized header space framework. We speculate that this is fundamentally because in our framework we directly *simulate* the forward path of a packet instead of *solving constraints* using an SMT solver. However, more work is required to understand the differences and potential opportunities.

## V. CONCLUSION

Testing liveness of a network is a fundamental problem for ISPs and large data center operators. Sending probes between every pair of edge ports is neither exhaustive nor scalable [30]. It suffices to find a minimal set of end-to-end packets that traverse each link. However, doing this requires a way of abstracting across device specific configuration files (e.g., header space), generating headers and the links they reach (e.g., all-pairs reachability), and finally determining a minimum set of test packets (Min-Set-Cover). Even the fundamental problem of automatically generating test packets for efficient liveness testing requires techniques akin to ATPG.

Network managers today use primitive tools such as ping and traceroute. Our survey results indicate that they are eager for more sophisticated tools. Other fields of engineering indicate that these desires are not unreasonable: 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. In fact, many months after we built and named our system, we discovered to our surprise that ATPG was a well-known acronym in hardware chip testing, where it stands for Automatic Test *Pattern* Generation [2]. We hope network ATPG will be equally useful for automated dynamic testing of production networks.

**REFERENCES**

[1] S. Shenker, "The future of networking, and the past of protocols," 2011 [Online].Available: http://opennetsummit.org/archives/oct11/shenkertue. Pdf

[2] M. Canini,D.Venzano, P. Peresini,D.Kostic, and J. Rexford, "A NICE way to test OpenFlow applications," in *Proc. NSDI*, 2012, pp. 10–10.

[3] Internet2, Ann Arbor, MI, USA, "The Internet2 observatory data collections," [Online]. Available: http://www.internet2.edu/observatory/ archive/data-collections.html

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

[5] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *Proc. NSDI*, 2012, pp. 9–9.

[6] "Hassel, the Header Space Library," [Online]. Available: https://bitbucket. org/peymank/hassel-public/

[7] M. Canini,D.Venzano, P. Peresini,D.Kostic, and J. Rexford, "A NICE way to test OpenFlow applications," in *Proc. NSDI*, 2012, pp. 10–10.

[8] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, "Debugging the data plane with Anteater," *Comput. Commun Rev.*, vol. 41, no. 4, pp. 290–301, Aug. 2011.

[9] M. Kuzniar, P. Peresini, M. Canini, D. Venzano, and D. Kostic, "A SOFT way for OpenFlow switch interoperability testing," in *Proc ACM CoNEXT*, 2012, pp. 265–276.

[10] N. Duffield, F. L. Presti, V. Paxson, and D. Towsley, "Inferring link loss using striped unicast probes," in *Proc. IEEE INFOCOM*, 2001, vol. 2, pp. 915–923.

[11] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. OSDI*, Berkeley, CA, USA, 2008, pp. 209–224.

**Authors:**

Student

Guide