Tempus: Probabilistic Network Latency Verifier

Anonymous Author(s)

ABSTRACT

To combat problems and bugs that a given network might have, network verifiers have emerged as one of the promising solutions. State-of-the-art network verifiers mainly focused on evaluating qualitative properties under various scenarios of network failures, such as reachability under k-link failure. However, as modern networks evolved and performance need becomes more stringent – often expressed in terms of Service Level Agreement (SLA) – there is a need to evolve network verifiers to also reason about quantitative performance properties. Works in quantitative network verifiers that has arose in recent years mainly focused on one side of the network performance metric: bandwidth and load violation properties. Questions about the other side of network performance metric, latency, were left unanswered.

In this work, we introduce a verifier framework, Tempus, that can answer the probability of a given temporal property being true given latency distributions of individual links and nodes in the network. Early evaluation shows that Tempus can verify bounded reachability property – the probability that the average delay of packets traversing from a source to destination node is below T time unit – by only adding a fraction of the state exploration overhead introduced by the qualitative verifier it's built upon.

1 INTRODUCTION

Modern networks consisted of various distributed protocols such as OSPF and BGP that exchange routing information so that a packet can reach their intended destination efficiently, even in the event of a component failure. Due to their configuration intricacies however, these protocols are notoriously hard to get right. It is hard for a network engineer to reason about whether their configured network fulfilled some intended property in various possible states of the network. Thus, they are left between the choice of accepting the reality of Murphy's Law or getting a tool to help them in this task, namely, network control plane verifier.

Popular control plane verifiers, like ARC [1], are formulated to answer questions about a given property deterministically. In other words, given a set of states of the network (e.g. k-link failures) the verifier are expected to give a yes or no answer about the property (e.g. two nodes are reachable). While useful to some extent, this kind of models are sometimes too restrictive since network operators are usually able to tolerate small fraction of failure. For example, a given network might have an availability SLA of 99.999%. This kind of probabilistic properties have been the focus of a more recent works like NetDice [2].

The network property itself can also be divided into two kinds. Quantitative properties like reachability and loop existence, are the common properties that are studied by most of the existing verifiers. More recent work, like QARC [3], has also explored qualitative properties like link bandwidth violation for a given traffic.

In this work, we're exploring the other side of the network performance metric: latency. Certain network deployment often necessitates some latency requirement such as an ISP that has latency SLA [4] or deployments of Time-Sensitive Networking (TSN) [5]. We proposed a verification framework to probabilistically verify the latency property of packets traversing from a source to destination node under various failure scenarios, by using latency information of each components in the network.

2 OVERVIEW

To be written

3 ON LATENCY MODELING

3.1 Latency as Path Property

We began our study by first pondering about a basic construct: modeling the end-to-end latency of a single packet. In a packet-switched network, a packet is sent from one transmiting host to one receiving host through the many components of the network (e.g. links, routers, firewall) and each of those components might introduce some latency into the packet transmission process. Figuring out which of those components will actually introduce latency into the packet in question, and by how much, is the next logical step that we must figure out.

Obviously, a given packet doesn't need to visit all network components to reach its destination host. A network engineer will configure the network in such a way that a packet will only need to be routed via a specific subset of its components. That specific subset is dictated by two things: how those components are connected together (i.e. topology) and how the control plane protocols are configured to route a packet (e.g. routing protocol, ACL). Based on a variety of these setups, nodes in the network will form a forwarding table to route a given packet appropriately.

The goal of a classical control plane verifier then, is to use this forwarding table in some form to verify certain properties. However, looking at the forwarding table alone will

not give us a conclusive result regarding which components are going to be visited by a given packet, making verification of latency properties less clear. For example, the network might be configured with a load-balancing protocol in which a packet departing from a source host might take multiple different *paths* (with certain probability weights) to arrive at the destination host, possibly resulting in a different end-to-end latency measurement.

Therefore, we argue that when it comes to analyzing end-to-end latency, a network path should be the primary unit of reasoning, rather than forwarding table. By being more specific about our unit of reasoning, we could answer verification questions more clearly, and we could design our verifier more efficiently since we could use it to represent multiple different forwarding tables that shares the same path.

3.2 Relation to Classical Verifier

Before we analyze the latency of a given packet that propagates through a certain network path however, we must make sure that that path exists in the first place. We note that latency is a property that only make sense after connectivity between two hosts has been established. In other words, if two nodes in a network aren't even functionally connected (e.g. physical link failure, ACL policy), then the latency between them will *always* be infinite, making the verification task trivial.

Fortunately, there are a rich body of work in the network verification literature regarding functional reachability under failure. We could then design our verifier on top of an existing classical control plane verifier. We use it to verify reachability property, and only if the reachability property is fulfilled, we would verify whether the latency between two hosts fulfilled some additional condition.

3.3 Latency Distribution

Up to this point, we have talked about measuring the latency of a single packet by figuring out its path; analyzing which exact components it has traversed through. However, when we try to generalize this framework and ask questions about the latency of multiple packets, it is apparent that the path alone is not a determinative information, as the latency of two packets propagating through the same path might be different due to a multitude of factors.

The natural extension to the framework then, is instead of representing latency of a path with a single number, we instead represent it with a continuous random variable that signifies the possible delay that a given packet traversing through that path might have. This random variable will have a distribution that marginalizes over all other factors other than the path.

We can then use this latency distribution to verify some temporal properties in a probabilistic manner. For example, we could verify the probability that a packet will be delivered in under a time unit by taking the CDF of the distribution.

3.4 Path Decomposition and Convolution

Expand on these points:

- How do we get the path latency distribution? Component-wise measurement data
- What measurement? link delay and queuing delay → not all delays are significant
- What to do with it? Simulation → lack of error guarantee
- Convolution
- Independence?

4 ENCODING

We will start with the assertion that latency is a property that only make sense after connectivity between two nodes has been established. In other words, if two nodes in a network aren't connected (due to physical failure or ACL policy), then the latency between them will *always* be infinite. Because of this, we divide the problem of latency verification into two parts: verifying that two nodes are reachable (*functional* property) and only if the functional property is fulfilled, we would verify whether the latency between two nodes fulfilled some additional condition (*temporal* property)

4.1 Topology Graph

For functional property verification, NetDice [2] has laid the way for verifying reachability between two nodes under failure in a probabilistic and efficient manner. In this framework, the physical network is encoded in an edge-labeled directed graph $G_t = (V_t, E_t)$ where V_t represents the routers in the network and E_t represents the connectivity between a pair of source and destination router. The function $r : E_t \rightarrow \{x \in \mathbb{R} \mid 0 \leq x \leq 1\}$ is the edge-label that represents the failure rate of a given connectivity link.

On top of this topology, we could define additional information that would be used by a routing protocol to determine the valid path(s) between two nodes $src, dst \in V_t$ given a particular link failure scenario. In OSPF for example, we define a function $w_{ospf} : E_t \rightarrow \mathbb{N}$ as the edge-label that represents the positive weight of a link.

4.2 Latency Graph

There are many source of latency in a network. The principal challenge for us, then, is to determine the appropriate level of abstraction; detailed enough in order to accomplish our verification goal correctly but abstract enough for the

problem to remain tractable. We settle on modeling two significant sources of latency: propagation and queuing delay.

Propagation delay is the latency that is introduced by the links in the network, which is independent of the traffic load in the system. Queuing delay is the latency that the queuing process in the node introduced. Unlike propagation delay, queuing delay might be dependent on load in the system, since the more packets there are in the queue, the more delay the node will introduce to a subsequent packets.

To take both of these forms of delay into account, we've decided to model the latency of each components with a univariate continuous random variable. For propagation delay, the semantic of this random variable is relatively straightforward: it is the distribution of latency that a given link will introduce. For queuing delay however, this random variable represents the delay that a given queuing process will introduce, marginalized over various traffic pattern that a given network state might have resulted.

To convey this additional notion of latency distribution, we form a second edge-labeled directed graph (called a latency graph) $G_l = (V_l, E_l)$. For each network node $v_i \in V_t$, we want to "expand" this node to be able to represent the inner working of the queuing process within each node.

- Each v_i is expanded into $v_i^{in}, v_i^{out-1}, ..., v_i^{out-n} \in V_l$ where n is the amount v_i's outward neighbors.
- Every inward edge that v_i has in G_t will all get directed to v_i^{in}.
- Every outward edge that v_i has in G_t will be sourced from one of the v_i^{out-j} instead.
- Add additional edges e_i^j that connects from v_i^{in} to each v_i^{out-j}.
- Finally, we label each $e \in E_l$ with the random variable as the latency distribution.

Semantically, v_i^{out-j} represents a port, normal links that are copied from G_t represents a link that has a propagation delay, and newly introduced links e_i^j represents an output queue in node v_i that has a queuing delay.

5 VERIFICATION

From these two graphs, we then do the verification in the following way:

5.0.1 Functional Verification. Let a network state s_i be defined as a 2-tuple (U, D) where U is a set of links that are alive / up and D is a set of links that are dead / down. We then define a set S that represents all possible network states in a given topology. Our goal is to identify a subset of S where each element is a network state that can connect the source and destination node given a convergent behavior of a routing protocol.

This framework fits nicely with the framework of probability theory, where S represents the sample space and the power set of S represents the possible events, one of which is the one we care about: the event that a packet from a source node can reach a destination node. The way we assign probability to events is to simply do a summation over the product of all the failure and success rate of each links in the network state.

The straightforward and naive way to compute the probability of this event is to iterate over all the network state. We then determine whether that network state can fulfill said property and which path it will take. If yes, then we would compute the probability from the product of the link's failure and success rate, and add it to the running sum. If no, then we could skip the computation altogether.

While natural, this brute-force algorithm is obviously doesn't scale since it will take 2^n iteration (n as the number of links in the topology). NetDice [2] solves this issue by noticing that we could skip over some events that are guaranteed not to change the convergent paths of the current network state, by marginalizing over *cold edges*.

In NetDice, we start from a perfect network state (i.e. $s_i = (U, D)$ where $D = \emptyset$). We then determine whether that network state can fulfill said property and which path it will take, similar to the brute-force method. If no, then it is safe to say that the source node can never reach the destination node in all network state. If yes, then we could use the logic of the routing protocol to determine the set of cold edges, (i.e. edges whose failure won't change the convergent paths). Edges that are not part of this set is called *hot edges* $h_i = (U, D)$. Instead of using s_i to compute the probability of the current state, NetDice only use the set of hot edges to compute the probability of them being alive ($h_i = (U, D)$ where $D = \emptyset$). This in effect computes the probability of multiple network state at once.

Next, instead of examining other network state at random, we instead iterate over the links in h_i. By definition, if we fail a link that belongs to h_i, then the convergent paths will change. To cover all of those possibilities efficiently, NetDice iterate over all the links in h_i, disabled them, reenabled them, and move to the next edge to do the same. If $h_i = ((e_1, e_2, ..., e_j), \emptyset)$, then we would enqueue $(\emptyset, (e_1))$, $((e_1), (e_2))$, $((e_1, e_2), (e_3))$, ..., $((e_1, e_2, ..., e_{j-1}), (e_j))$. Thus, instead of exploring 2^j events, we will enqueue j events instead.

NetDice's exploration will effectively form a tree, where the root is the event where we have empty hot edges, and the leaves are where the hot edges are full.

5.0.2 Temporal Verification. Since we assign each links and output queue in the network with a certain latency distribution, computing the probability of the temporal property

involves convolving the the paths we got from functional verification and measuring the statistical property of the resulting distribution. However, while NetDice has shrunk the exploration space compared to brute-force iteration, for the purpose of temporal verification, further optimization can be done.

The first is what we call **hot-edges union**. The idea is, while each hot-edges events has the same convergent paths, it is not guaranteed to be unique. Two or more of these events can have the same convergent paths. Therefore, we could merge these two events and compute the convolution for a given path once.

6 PROPOSED EVALUATION

On the correctness side, we want to validate whether the result of our verifier is equivalent to another verifier that has lower fidelity (i.e. functional property only) if we set the temporal property to be virtually unconstrained (e.g. bounded reachability with a very high T).

On the performance side, we will evaluate the verifier by measuring the amount of additional overhead that the temporal verifier would introduce, measured by the amount of states that need to be re-explored (in the second step) and / or the amount of convolution operation performed. Early result on Highwinds [6] network shows that out of 2344 states produced by step 1, only 459 got re-explored on step 2.

REFERENCES

[1] A. Gember-Jacobson, R. Viswanathan, A. Akella, and R. Mahajan, "Fast control plane analysis using an abstract representation," in *Proceedings of the 2016 ACM SIGCOMM Conference*, pp. 300–313, 2016.

[2] S. Steffen, T. Gehr, P. Tsankov, L. Vanbever, and M. Vechev, "Probabilistic verification of network configurations," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pp. 750–764, 2020.

[3] K. Subramanian, A. Abhashkumar, L. D'Antoni, and A. Akella, "Detecting network load violations for distributed control planes," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 974–988, 2020.

[4] "Verizon global latency and packet delivery sla." https://www.verizon.com/business/terms/global_latency_sla/. Accessed: 2022-09-20.

[5] "Tsn." https://1.ieee802.org/tsn/. Accessed: 2022-09-20.

[6] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan, "The internet topology zoo," *IEEE Journal on Selected Areas in Communications*, vol. 29, no. 9, pp. 1765–1775, 2011.