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 said 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 Path 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

The final question that we had to decide on was how do we actually model path latency distribution based on real component measurements. Considering the complexity of factors that might determine the path latency distribution and the availability of measurement data, we settle on the assumption that the path latency distribution is composed of multiple independent distribution that corresponds to the latency each components in the path might introduce.

Since not all components in the path will introduce latency with a non-negligible value, we specifically choose to model two source of latency in the path's components which we deem significant: **link propagation latency** and **queuing latency**.

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.

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.

In order to obtain the overall path latency distribution from these per-component latency distributions, we do a convolution operations over all the relevant component's latency distributions. Since not all distributions can be convolved in a closed-form fashion, we use a numerical convolution technique with a guaranteed error bound. We initially consider a Monte-Carlo simulation in order to approach numerical convolution, but the lack of a general technique to measure error made us opt for the formerly mentioned method.

4 ENCODING

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 nodes in the network and E_t represents the functional connectivity between a source and a destination node. A physical link is then represented as a pair of symmetrical edges that shares the same source and destination node but with opposite direction (e.g. $v_1 \rightarrow v_2$ and $v_2 \rightarrow v_1$).

To represent the component's random failure, NetDice defined two failure rates. The first failure rate is the chance that a given physical link in the network will go down. The second failure rate is the chance that a given node in the network will go down. These rates are shared between all links / nodes in the network. Internally however, the node failure rate will be translated into the link failure rate with a Bayesian Network model, since a node failure can be modeled with the failure of each links connected to it.

We refine NetDice's model slightly by allowing each links to have different failure rate. We label each edge in E_t with a function $r : E_t \rightarrow \{x \in \mathbb{R} \mid 0 \leq x \leq 1\}$ that represents the failure rate of a given physical link. As a consequence, two symmetrical edges (i.e. two edges that shares the same node pairs but with opposite direction) will also share the same failure rate, and will be disabled in a coupled fashion.

4.2 Routing

On top of this topology, NetDice also defined additional informations 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. These routing informations would also be used by their optimization algorithms to reduce the amount of states that are going to be explored.

NetDice implemented iBGP and OSPF (with ECMP) routing protocols. They encode the relevant routing information by assigning some labels to the vertices or edges in the topology. 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. They would later be used to compute the convergent paths of a given network state (5).

4.3 Latency Label

Similarly, we could also use this edge-labeling technique to represent latency. As mentioned in 3.4, our model will encode two kinds of latency: link propagation and queuing in the node. We will encode these latencies by equipping the model with two additional labels.

Encoding link propagation latency is fairly straightforward. We define a function $l_p : E_t \rightarrow \mathcal{D}$ where $\mathcal{D}$ is a set of continuous univariate distribution that has a minimum value of 0. This distribution signifies the time it would take for the respective physical link to transmit a packet from one end to another. Because of this, just like r, two symmetrical edges will share the same distribution.

To encode node queuing latency, we first note that the queuing mechanism in modern switches usually resides on the output port. Since a port in a switch is only connected to one other port, we could effectively assign the latency to the connectivity between switches. To do this, we define a function $l_q : E_t \rightarrow \mathcal{D}$ where $\mathcal{D}$. Unlike l_p however, two symmetrical edges will have different distribution since they represent two different output queues.

5 VERIFICATION

From a labeled graph that encodes a given network, we will then do functional verification and temporal verification in the following way:

5.1 Functional Verification

For functional verification, we will essentially do the exploration technique employed by NetDice. However, we made some modifications to the algorithm in order for it to remember and pass necessary informations to the temporal verification stage. To contextualize our modification, we will briefly describe the description of NetDice's algorithm in three parts: Network States, Equivalence Classes, and Functional Exploration. For a more complete complete description of NetDice, please refer to the original paper.

5.1.1 Network States. Let a network state s_i be defined as a 2-tuple (U, D) where U is a set of all links that are alive / up and D is a set of all links that are dead / down. (Consequently, $E_t = U \cup D$). We then define a set S that represents all possible network states in a given topology. The goal of functional verification is to identify the largest subset of S where each element of this set is a network state that can connect the source and destination node given a convergent behavior of the control plane. We will call this subset *reachability subset*. While NetDice posseses the ability to verify other properties, we only focuses on basic reachability, the most relevant property for latency verification.

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 probability events, one of which is the previously mentioned reachability subset. The way we assign probability values to a subset is to simply do a summation over the probability value of each network state, which in turn is the product of all the failure and success rate of each links in the network state.

The naive approach to compute the probability of the reachability subset is to exhaustively iterate over all the network

state and check the reachability property. While natural, this brute-force algorithm doesn't scale well since it will take $2^{|E_t|}$ iterations.

5.1.2 Equivalence Classes. NetDice [2] solves this issue by noticing that we could combine over some states that are guaranteed not to change the convergent paths (and thus the reachability property) of the current network state, by marginalizing over what is called *cold edges*.

Cold edges are defined to be links whose failure is guaranteed not to change the property in question. In the specific case of reachability, this essentially mean that the failure of such edge won't change the convergent paths of the network state in question. By marginalizing over its cold edges and combining the state into one equivalence class, instead of computing the probability of individual network states, we are left with computing the probability from the remaining edges, which is called *hot edges*.

Hot edges are the opposite of cold edges: their state (whether they are up or down) will change the convergent path of the network state in question. Mostly, they consists of the links that make up the convergent paths itself.

By identifying the cold edges of a given network state, NetDice reduces the search-space by merging multiple network states into fewer equivalence classes. However, this optimization also serves another purpose: providing an efficient exploration strategy.

5.1.3 Functional Exploration. To actually identify the equivalence classes that might emerge in a given network, NetDice employs a tree-based exploration method based on hot edges computation.

We start with a perfect network state (i.e. $s_i = (U, D)$ where $D = \emptyset$ and $U = E_t$). We then compute the convergent paths and determine the hot edges set of that network state. Since, by definition, shutting down the link in this set will change the convergent paths, instead of exhaustively exploring some other state at random, NetDice would continue the exploration by systematically failing one of these links instead.

NetDice will recursively continue this process for the new state and effectively form an exploration tree, where each level of the tree represents how many links we will explicitly put down. The root is the event where we have empty hot edges, and the leaves are where the hot edges are full.

5.1.4 Making Exploration Explicit. After the original exploration algorithm finishes, NetDice will output the final probability (and error range) of the reachability property in all the equivalence classes that has been explored. However, this information alone isn't useful for us to reason about latency. Thus, we modified the original exploration algorithm in order to make it more explicit, so that its output can be used effectively in the next stage.

We started by formalizing the notion of Equivalence Classes. An Equivalence Class $\mathcal{E}$ is defined as a 3-tuple (U_h, D_h, P_{conv}). U_h and D_h refers to a set of links in the hot edges that are up and down respectively. They are similar to U and D in network state, with a notable exception that $E_t \neq U_h \cup D_h$, since they only represents hot edges, and not including cold edges. P_{conv} refers to the convergent paths between src-dst pair when the links in D_h fails. We explicitly store P_{conv} in $\mathcal{E}$ (unlike the original algorithm) so that it could be used to for the temporal verification stage.

We then define the Exploration Tree $\mathcal{T}$ as a tree of Equivalence Classes. At the root, we have an Equivalence Class that corresponds to a perfect network, where P_{conv} is the path(s) that the control plane will produce given a perfect network condition and $U_h = D_h = \emptyset$. Each Equivalence Class in this tree will have children which have one of its parent's link in P_{conv} appended to its D_h, essentially failing one of the link in the path. If an Equivalence Class has an empty P_conv, then it would have no children.

To compute the functional probability of a given Equivalence Class, we compute the product of the 'up' probability of all the links in U_h and P_{conv} and the product of the 'down' probability of all the links in D_h.

After having the algorithm produced $\mathcal{T}$, we then continue to the temporal verification stage.

5.2 Temporal Verification

In the temporal verification stage, we will use the P_{conv} within each Equivalence Class in $\mathcal{T}$ and the latency distribution labels l_p and l_q to augment the functional probability we had in the previous stage to actually answer some questions about latency property.

5.2.1 Path Convolution and Weighted Average. Given P_{conv}, l_p, and l_q, we will do the following:

(1) Split P_{conv} into its individual path
(2) For each path, compute its latency distribution by convolving the latency distribution l_p and l_q of each link
(3) With the resulting path latency distribution, determine the probability of temporal property by computing the statistical property of said distribution
(4) Combine the temporal property of each path by computing the weighted average, based on the load-balancing scheme of the control plane

We do each of these steps to every Equivalence Class in $\mathcal{T}$, and thus adding additional overhead to the exploration algorithm (on top of functional exploration). In other words, for $\mathcal{T}$ of size n, we will need to do temporal verification in each of those n Equivalence Class. We aim to minimize the

overhead of temporal verification by introducing some optimization algorithms later.

5.2.2 Numerical Convolution. One problem with this algorithm, is that not every convolution between distributions have a closed-form solution and could be done analytically. Therefore, we need to leverage a numerical convolution method in order to convolve two arbitrary distributions.

Expand points on

- DIRECT
- orderings DIRECT

5.2.3 Equivalence Class Consolidation. The first is what we call **Equivalence Class Consolidation**. 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.

5.2.4 Path Memoization.

5.2.5 Convolution Grouping.

6 EVALUATION

6.1 Scalability

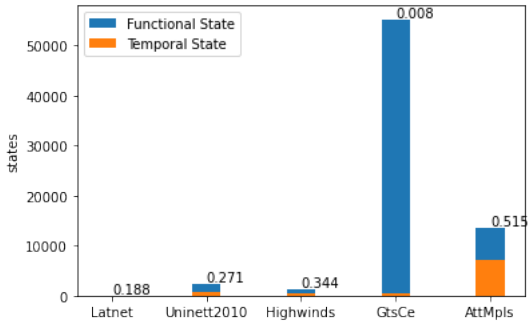

Figure 1: States explored for temporal verification and functional verification

6.1.1 States Explored.

6.1.2 Wall-clock Time.

6.2 Optimization Effectiveness

7 RELATED WORKS

8 FUTURE WORKS

9 CONCLUSION

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.

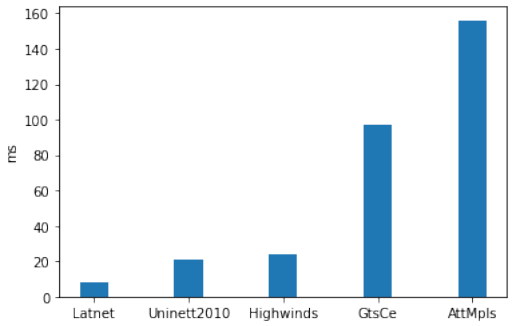

Figure 2: Time took for temporal verification

[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.