ChaosLemur: Injecting Failure into BGP

Davis Gossage
Duke University
dcg13@duke.edu

Emmanuel Shiferaw
Duke University
eas66@duke.edu

ABSTRACT

In this paper, we present ChaosLemur, a system built for applying Chaos, a recently developed and publicized principle in reliability engineering, to networks of virtual routers speaking BGP. The system is designed to take input from the user describing the desired number of routers, choice of topology, and distribution pattern of networks advertised in each router. Given this information, a set of linux containers are built, configured with software to act as routers, and connected in the described topology. The system then injects failures randomly into the network, collecting and saving the results/response of the system to those failures. We have implemented a prototype of ChaosLemur in Python, using Docker for containerization and Quagga as routing software and BGP implementation. As it currently stands, the system has been tested to configure and build networks of 5 - 20 containers in simple mesh and hub topologies speaking BGP, to take down specific and random nodes, and to report results in the form of logging routing tables.

1. INTRODUCTION

The first use of the word "Chaos" for the random, injected failure of elements in a distributed system was a Netflix technology blog post[4], which described a certain "Chaos Monkey", an actual unknown and inexplicable failure in Netflix's cloud infrastructure that drove engineers to focus more on fault-tolerance and reliability, both in the design and maintenance of the system. This phenomenon seems to be a simple result of the fact that failures that happen in production are hard to predict, and cannot be reliably recreated in traditional 'testing' environments. Thus, the injection of these failures frequently while in production acts as a sort of secondary testing phase, keeping engineers on their toes. This drove the team to build an automated version of this helpful bug that now runs in their production cloud on Amazon Web Services, taking down system elements at vastly different levels of scale, from single VM instances to entire AWS regions.[3]

Over time, Netflix has extended this idea to a more broad concept they call Chaos Engineering, and define as the practice of designing and building experiments to understand the response of distributed systems to failure, and to inform action on reducing the negative effects of that response.This has been applied over the past few years to improve the reliability of Netflix, an extremely large and distributed cloud/web service that experiences many unpredictable failures in production. It has also been used at Microsoft (also for cloud services), as well as within the networking realm (specifically for SDN)[7].

As the design of any specific network topology/configuration is, in essence, the design of a distributed system, a system that would allow rapid simulation of many different configurations of a network may be quite useful. BGP as a whole has been criticized for its occasional reduced performance due to hot-potato routing/oscillations and long convergence time for knowledge of available paths[6]. It is also increasingly being used in data centers for the same reasons: since it does not require any global state or knowledge of networks, it can be used in scenarios where the overhead necessary for this global state (memory, network, etc.) would be too expensive. Issues of oscillation and long convergence can also be mitigated through tuning and proper configuration of a network [2]. The goal of ChaosLemur is to provide the ability to automate this tuning process and test a desired topology's fault-tolerance, without having to build it in hardware.

2. RELATED WORKS

In an attempt to consolidate efforts and formally define this chaos-driven approach to fault tolerance, Netflix has compiled the core ideas of the approach into a single document[1], emphasizing the importance of the 'Experiment' as a singular unit of information-discovery in the application of these principles. This manifesto puts forth a four-step process which simply revisits the basic scientific method, with the role of "Nature" being played by the unwieldy and unreliable distributed systems that power our most important large web services [9]. The steps are as follows:

1. Start by defining 'steady state' as some measurable output of a system that indicates normal behavior.

2. Hypothesize that this steady state will continue in both the control group and the experimental group.

3. Introduce variables that reflect real world events like servers that crash, hard drives that malfunction, network connections that are severed, etc.

4. Try to disprove the hypothesis by looking for a difference in steady state between the control group and the experimental group.

This analogy works well in that the complexity of distributed systems and their unknown inputs in the real world/production cannot be easily built-up from first principles. The empirical approach attempts to build theories and models for systems out of small, testable hypotheses and experiments, which are often the only tools available to uswith such complex systems.

Step 3's "variables" are referred to by Netflix as failure modes, specific sorts of failure events in the system. In the case of cloud infrastructure, different examples of failure modes are single server failures, whole region failures, '80% decreased latency', and so on [9]. Note that these can be divided loosely into two categories: 1) Discrete, identifiable structural events in the system, and 2) Descriptions of decreased performance in some metric, not necessarily with any known cause. The guidelines outlined in Netflix's documents suggest using the second category more [3], in order to better be able to recreate real-world failures, which are often complex and difficult to diagnose. In a BGP network's 'distributed system', a single failure can come in the for of a node (router) going down, or a link between two nodes going down, as well as reductions in performance metrics under different loads.

Though developed for very general cloud VM infrastructure, the chaos approach has been shown to be applicable to a significant and diverse array of specific distributed systems, including Microsoft's search software/service on Azure[5]. Recasting chaos engineering as an entire mode of interpreting and approaching failure, the Azure team detailed a comprehensive ontology for injecting, classifying, and responding to failure conditions in the Search product. Failures, upon being discovered in production, were listed as "extreme chaos" - unknown failures with unknown consequences - the most urgent situations to deal with. The first step in tackling this unknown failure is to reproduce it, in order to gain a rudimentary understanding of its root cause. The next step is to automate this failure (with whatever accuracy or that is possible given the information gleaned from previous steps). Automation and variation of the "model" failure then allows engineers to see the system's response to the previously unknown failure.

3. ARCHITECTURE

ChaosLemur's architecture was structured primarily around the systems used to implement the components. We chose to use Docker containers instead of full virtual machines to implement each virtual router. Hardware-level virtualization was not necessary for our purposes; using a VMWare-like system would add unneeded overhead in the form of hypervisors and separate storage space for each virtual router [8]. Docker's interface for building and managing containers also lends itself well to scripting and automation.

Docker images, the base set of OS information/scripts used to build containers, can be generated from text configuration files called Dockerfiles. These can be easily maintained and distributed via popular version control software designed for tracking changes in text files. Dockerfiles describe the operating system, initial software to install, and files to load into the container from the host system.

Quagga is a software routing suite that implements many common protocols found in hardware routers, including BGP and OSPF. It has a simple architecture, with one daemon, *zebra*, acting as the central component of the system, interfacing directly with network hardware and presenting an API for communication. All protocol implementations are also individual daemons, each using the aforementioned zebra daemon's API to actually perform the routing. For the purposes of ChaosLemur, two of these additional daemons were necessary - bgpd and vtysh. bgpd is the quagga daemon that implements the BGP protocol. It maintains the routing table/forwarding information base, and issues/receives announcements and withdrawals. vtysh is the only Quagga daemon that does not implement a routing protocol; it presents a shell/command-line interface to zebra (and by proxy, the other daemons). Users interact with Quagga (issuing commands, retrieving information) using this CLI. zebra, bgpd and vtysh all need to be configured before running. Their configuration is loaded in text config files - one for each daemon (*zebra.conf*, *bgpd.conf*, *vtysh.conf*). Zebra and vtysh do not have a great deal of customizability in their configuration, but as would be expected, the protocol-implementation configuration files have to express quite a bit of information.

Topology information for the ChaosLemur experiment must be communicated to each router through *bgpd.conf*. Each router must also have its own Dockerfile (in order to be built and run as a separate container). Due to these two constraints, the simplest way to generate an entire ChaosLemur experiment from a simple set of inputs was to build an entirely separate Docker 'build context' for each router in the network, loading in a distinct *bgpd.conf* file on each node. Each docker build context is a linux directory containing the dockerfile and all files referenced/needed to run the container. The portion of *bgpd.conf* that is generated by ChaosLemur is shown below:

```
router bgp 7675
bgp router-id 172.17.0.3
network 159.12.0.0/16
neighbor 172.17.0.2 remote-as 7675
neighbor 172.17.0.4 remote-as 7675
neighbor 172.17.0.5 remote-as 7675
```

The ChaosLemur API call/constructor that generated this specific configuration file portion was

```
CL = ChaosLemur(4, "hub", "pareto", 2.3).
```

This router is the second in the collection, with address 172.17.0.3, configured as the 'hub'. (Note: this API call generates all the routers/build contexts, not just that of this hub). The single "network" line is a result of the 'pareto' distribution input, which says that the number of networks advertised by this router should follow the pareto distribution, with an α value of 2.3. The lines beginning with "neighbor" list the adderesses of the nodes directly connected to this node in the topology (as it is the hub, all 3 other routers are directly linked to it). The "remote-as" portion tells the autonomous-system number of the neighbors, although there is currently no option in the API to distribute routers amongst different autonomous systems.

Users are given the choice of a mesh or hub topology primarily because they are simple enough to be describable with a single string input, and they are different enough to provide meaningfully different responses to failures. The ability to design arbitrarily complex topologies would add a great deal to the meaning of results obtained from ChaosLemur.

The 'distribution' input is in the API in order to allow users of ChaosLemur to configure an arbitrarily large and complex network. Also, in order for results of failure-injections to have noticeable and meaningful effects, the amounts/overlap of subnets initially advertised by each router should be varied. For example, a situation in which a router that advertises a large majority of prefixes in the network is separated from another router by the "hub", it may be interesting to see the effects of the failure of that node. The current implementation has caps/limitations on the number of networks that can be advertised by each router (maximum 20), but within those constraints, can assign subnets to each virtual router with uniform, log-normal, and pareto distributions (with the number of networks as the random variable).

The Failure Injection/reversal component of ChaosLemur, though the central logical component, is actually quite simple given the design of the environment/context generator. It operates under the assumption that the (only) docker containers running on the system are ChaosLemur virtual routers configured using the previously mentioned API call. It contains methods for taking down individual addressed nodes, taking down *any* random node, reversing all failures, and dumping the BGP routing table to `stdout`. The node-takedown method relies on the `docker stop` command, which stops the Quagga process and the container entirely (preserving the storage space allocated for the container and its contents). The reversal method calls `docker start`, which simply restarts the container and re-runs the quagga initialization command (which reloads the configuration file and begins the daemons). We considered adding a checkpoint/restore system to this process in order to maintain the contents of the container's memory before restarting. However, in order to emulate a real hardware failure, having Quagga restart from persistent configuration files seemed more realistic than having the contents of memory preserved. The "Show route" command works by connecting to a specfic running container via *vtysh* (using `docker attach` command), and running the quagga command `show ip route`.

It also contains methods for taking down links between any two nodes (and any two random nodes), but those methods rely on a quagga *bgpd* command that is not accepted via vtysh. As such, link-takedowns have not been implemented in the current version of ChaosLemur.

4. RESULTS

We have tested ChaosLemur with scripts combining the different topology options, numbers of routers, and distribution patterns for numbers of networks advertised. For networks in both *mesh* and *hub* topologies, the process to build the entire environment, consisting of all containers and their contexts, takes, on average, 45 seconds per container, and scales linearly.

This is because, during the container build process, if Docker encounters an instruction (e.g. a file to be copied, command to be run) that it has already executed on the same base image, it will use a cached copy of an 'intermediate container' built during a previous container's build process. However, because a new *bgpd.conf* needs to be loaded into each router for every container, the whole process cannot be repeated from the cache (if it could, the build time would be on the order of seconds).

5. FUTURE WORK

ChaosLemur as it currently exists is a decidedly simple prototype. Its simplicity proves the ease with which any organization or network operator could implement a similar solution for any real network that needs to improve its reliability. Much of the future work on this project would/should go toward solidifying it as a proof-of-concept, addressing edge cases and performing more thorough tests. A proper implementation of the link-takedown methods would be our first step toward making a more presentable prototype.

The configuration-file centric design of Docker and Quagga make ChaosLemur considerably extensible and flexible. As the "Context Generator" portion of the current implementation is essentially a program that writes *bgpd.conf* files given simple input, it could quite simply be extended to exploit any of the capabilities that QuaggaâĂŹs bgpd has that can be configured via *bgpd.conf*. The next option to be provided to users as input to ChaosLemur should be the ability to configure networks with multiple autonomous systems as in figure 1. BGP is by definition an external gateway protocol, and much of the useful information that could be collected from this system and applied in the real world regarding failures would have to do with convergence time amongst many different autonomous systems that are spread out. Another option that Quagga offers, which could be a further input to the system, is configuration of route maps (which would allow operators to configure routers to filter which networks are advertised).

As mentioned previously, the choice of only two basic topologies is a limiting factor for the applicability of ChaosLemur's results. A system to allow users to input and describe a topology fully, individually describing nodes and links, would be ideal. This would be difficult to make work well, however, with a tradeoff between simplicity of input method and complexity of implementation. A fully graphical user interface would be ideal from the perspective of the user, but would require significantly more work to translate into the format of *bgpd.conf*.

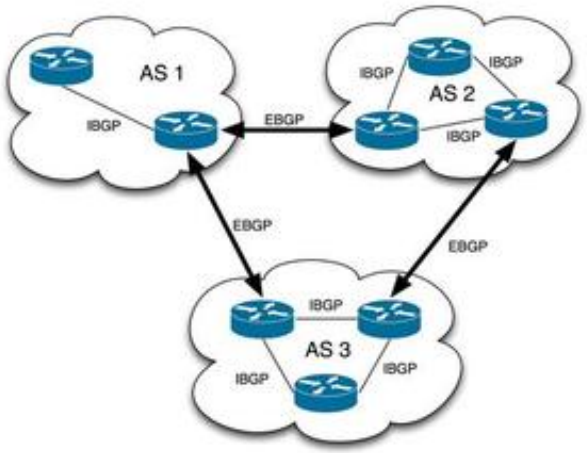

Figure 1: BGP topology with multiple Autonomous Systems.

Yet another, perhaps more interesting option would be to abstract ChaosLemur from *bgpd* and allow networks to be implemented that speak any protocol that Quagga implements. This, too, is made into a simple task by the fact that Quagga uses configuration files for all daemons with consistent structure and syntax. The exact same API methods that exist currently would work just as well for container-routers running *ospfd* or *olsrd*, the daemons for OSPF and OSLR respectively.

One way in which ChaosLemur's approach/application differs from most other "Chaos" implementations is the fact that it configures a 'test' environment itself, rather than causing actual failures in production. As BGP is a protocol implemented primarily in more expensive routers owned by ISPs and enterprises, it was not quite feasible to attempt to build a system that would work on production hardware and be easily developed using open source software-routing like Quagga. Quagga does, however, present the same abstractions as hardware routers, and a version of the ChaosLemur API could certainly be developed for a real ISP's BGP network. Other real-world Chaos applications have also taken the approach of configuring a separate, controlled sandbox separate from production to let failure-daemons run loose - the main benefit being that in automating the deployment of this environment, failures in the build/release/deploy process will be uncovered as well[5]. This is perhaps less applicable for networks outside of SDN, as they are more manually configured.

Finally, the reporting side of ChaosLemur currently exists as one method call to display current state, with no mechanism for timing response to the failures directly, or easily calculating convergence time (without developing a custom solution, calling the showIPRoute() method many times, which is certainly do-able). Dumping of full bgp logs is essentially implemented (it takes a vtysh command), but robust translation into readable, actionable information, in the time-frame of these failures being injected, will need to be implemented in addition to the other components of ChaosLemur.

6. REFERENCES

[1] Principles of Chaos Engineering. (English). *Principles of Chaos*.

[2] P. L. E. N. Adel Abouchaev, Tim LaBerge. Brain-Slug: a BGP-Only SDN for Large-Scale Data-Centers. (English). *Microsoft Global Foundation Services Presentation*.

[3] A. T. C. R. Ali Basiri, Lorin Hochstein. Chaos Engineering Upgraded. (English). *The Netflix Tech Blog*.

[4] J. Atwood. Working with the Chaos Monkey. (English). *Coding Horror*.

[5] H. Nakama. Inside Azure Search: Chaos Engineering. (English). *Microsoft Azure Blog*.

[6] J. R. Nick Feamster, Jay Borkenhagen. Guidelines for Interdomain Traffic Engineering. (English). *Computer Communication Review*.

[7] K.-T. F. M. C. T. B. L. V. Nick Shelly, Brendan Tschaen. Destroying networks for fun (and profit). (English). *HotNets*.

[8] R. R. J. R. Wes Felter, Alexandre Ferreira. An Updated Performance Comparison of Virtual Machines and Linux Containers. (English). *IBM Research Report*.

[9] B. Wong. Introducing Chaos Engineering. (English). *The Netflix Tech Blog*.