DEGREE PROJECT IN TECHNOLOGY,
FIRST CYCLE, 15 CREDITS
STOCKHOLM, SWEDEN 2018

Simulation of chaos engineering for Internet-scale software with ns-3

TAI LUONG

ANTON ZUBAYER

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

DEGREE PROJECT IN TEKNIK,
FIRST CYCLE, 15 CREDITS
STOCKHOLM, SWEDEN 2018

Simulering av chaos engineering för Internet-skalig mjukvara med ns-3

TAI LUONG

ANTON ZUBAYER

KTH ROYAL INSTITUTE OF TECHNOLOGY
SKOLAN FÖR ELEKTROTEKNIK OCH DATAVETENSKAP

Abstract

Chaos engineering is the practice of volitionally fault injecting in to a built system with the main purpose of finding the eventual bugs existed in the system before the actual system error occurs. One of the pioneers in this practice is the Simian army, a part of Netflix, where they used random fault injection and linage driven fault injection in its own enormous system with the purposes of suggesting either a relevant approach to chaos engineering or to propose methods of how to uncover the hidden weaknesses. With this paper we seek to attain a simplified illustration about these submitted studies of chaos engineering applied on a simulated Netflix environment in ns-3 with the intention to provide some enlightenment for the principles of chaos engineering and in addition to this we will also at the end of this paper present a suggestion about how to infer an unknown system as a possible application, derived from the knowledges acquired from our chaos journey.

Sammanfattning

”Chaos engineering” är utövandet att själv injicera fel in i ett system för att i första hand hitta potentiella buggar som skulle kunna uppstå innan buggarna framträder i det faktiska systemet som grundades som principer av Simian Army. Slumpmässiga fel injektioner och LDFI (Lineage Driver Fault Injection) hade Netflix redan tidigare utfört i sitt stora system av anledningen att antingen föreslå ett tillvägagångssätt för Chaos Engineering eller att föreslå metoder att kunna upptäcka dolda svagheter. Med denna rapport är syftet att försöka, med en enkel förklaring av det vi redan vet om Chaos Engineering, att använda inom en simulerad Netflix miljö i ns-3 och intentionen att upplysa om principerna för Chaos Engineering. I slutet av rapporten kommer vi även att framföra ett förslag om hur man kan antyda ett okänt system som en potentiell tillämpning av användandet och lärdomarna av vår ”kaos resa”.

I. Introduction

A. Motivation and Problem Statement

Before we dive into the details, something must be spoken about the chief objectives of this paper. In this project we seek to indulge ourselves mainly with the idea of achieving a clear illustration of the concepts of chaos engineering, for instance LDFI and random fault injection in an environment simulated with the aid of the open-source software, ns-3. Other than that as a dedication to answer questions such as

- Is it possible to gain insights of the inner structure of an unknown system using chaos engineering ?
- Can informations about a system be recovered from its weaknesses ?
- Do weaknesses have a role to play or is it just something we must always exclude ?

an implementation, the so called Inferator, was also created by using the knowledge gained previously at the end of this paper .

B. Contributions

The main contributions of this paper would be sum up as below

- a simulator of chaos engineering in ns-3. the simulator consists of XXX lines of C++ code built on top of the NS3 library.
- an evaluation of the simulator on a scenario with X nodes and Y egdes, representing a Netflix-like architecture for displaying the idea of lineage driven fault injection, LDFI.
- a new algorithm to recover the architecture of a distributed system using fault injection.

Answers to the questions in the previous subsection:

Question 1: Is it possible to gain insights of the inner structure of an unknown system using chaos engineering?.

It is possible, since as demonstrated in the section IV the outline of the architecture can be found by implementing the algorithms described in the same section. Although it can be faulty about the ordering of the nodes, for example a weakness set {1,2},{3,4},{0},{5} and another permuted weakness set {3,4},{1,2},{0},{5}, both in figure 1, will give the same architecture of the system, but the ordering of nodes are not correct, therefore in order to achieve a better accuracy it is required that we must also have the knowledge about the ordering of weaknesses. Suggestions about how to achieve this is stated in the conclusion of this paper.

Question 2: Can information about a system be recovered from its weaknesses?

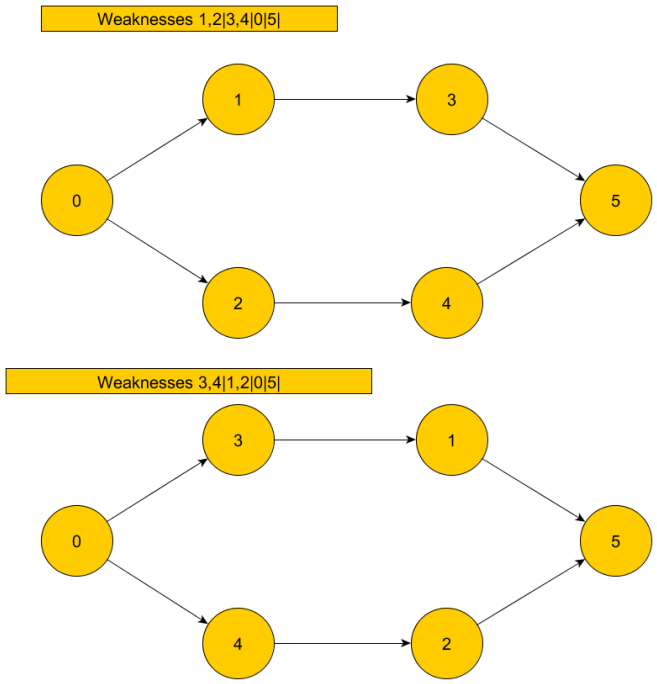

Fig. 1. 2 equivalent solutions about an unknown system with weaknesses {1,2},{3,4},{0},{5}.

Yes, as far as our achievements go with this study the best we were able to recover is the outline of the structure from the unknown system.

Question 3: Do weaknesses have a role to play or is it just something we must always exclude?

To the best of our knowledge acquired from our chaos journey, we strongly believe that weaknesses are important as it is one of the many factors which can characterize the uniqueness of the system, therefore it is reasonable that it should contain specific information about a system. This reasoning is also the base assumption for our section IV when making the Inferator.

C. Background of chaos engineering

Large companies all over the world such as Netflix, Google and Facebook use distributed systems. With such large, over the world and continuously working systems it is crucial to maintain a certain stability and be able to handle unexpected errors that can appear at any time and anyhow. Fault injection methods and the usage of chaos principles of some sort are widely used but not a very old and mature field yet. It's quite new and a lot of companies are starting to see the benefits of these principles and methods of chaos engineering and automated fault injections. With realization of the potential of this practice internet companies, that were pioneering large scale, employed the implementation of chaos upon their own distributed systems. These systems were so

complex and massive that they required a new approach to test for failure. Therefore in 2010 the Netflix Eng Tools team created Chaos Monkey[8] in response to Netflix's move from physical infrastructure to cloud infrastructure provided by Amazon Web Services, and the need to be sure that a loss of an Amazon instance wouldn't affect the Netflix streaming experience and as a result in 2011 the Simian Army[6] was born. The Simian Army added additional failure injection modes on top of Chaos Monkey that would allow testing of a more complete suite of failure states, and thus build resilience to those as well. They made a cloud architecture that when the testings made failures it didn't affect the entire system.

All in all the actual ideas about chaos engineering itself is already touched upon way back in 1975, mentioned as a recap of the childhood of chaos engineering in an article, written by Monperrus, [7] where you can read about Yau and Cheung. They proposed inserting fake ghost planes in an air traffic control system. All for the reason to verify a software's stability and reliability. From the article the reader was also introduced to Taleb's view, a key point of anti-fragility is that an anti-fragile system becomes better and stronger under continuous attacks and errors. Furthermore as a finer grain implementation regarding this aspect of a evolving system due to its own exposure for outer perturbations a recently written paper ,about the so called chaos machine [9] , displayed the capability of detecting and handling errors of the try-catch level.

D. Distributed systems

A distributed system is when the data is spread out across more than one computer, station, server, etc. Distribution networking applications and data operate more efficiently over several workstations. One very common usage is of client/server computing, where a client computer requests other computers that provides services for the clients. For example; a customer wants to buy a t-shirt through the internet from an online web-shop. He first has to log in into his account. The customer uses a web browser, a client, that sends a request to the web server, a server, that in turn sends a request, as a client, to the data server where the login information is stored. After the interpretation of the input, it then returns the output to the web server so that the server can display it to the web browser. So something can act as both a client and a server. This type of communication is effective and easy.

E. Principle of chaos engineering

By doing experiments on a distributed system in order to ensure a systems durability when conditions are changed is the core practice of chaos engineering. For example crashes, failures, outages and much more. When diving into a situation we want to observe the steady-state behavior. A systems behavior can for example be changed by user interactions, changes initiated by the engineers or a failure of a third party service that the system depends on. There are four accepted principles that can be used to design an experiment using chaos engineering[3]. First is to build a hypothesis around steady state behavior, second to vary real-world events, third to run the experiments in production and fourth to is to automate experiments to run continuously.

You build the hypothesis around how the injected conditions affect the steady state of the system[3]. It focuses on the steady-state behaviors that can be seen at the boundary of the system which visualizes the interactions between the users and the system.

To vary real-world events, you have to let your creativity and historic data assist you to find out all types of different events that could effect the system. For example outages, latency, failures, etc. These real-world events can be both hardware and software related. The creativity is important due to that engineers or programmers often uses "happy paths" meaning that they only inject events that do not effect the system, creating an illusion or convincing the self that the program runs smooth and does not fail. You have to often critique yourself and the injections so that they aren't just convenient. The historical data is used to see how earlier crashes, bugs, failures, etc. have affected the system. Sometimes you even have to simulate an event because injecting too large or critical errors could be devastating to the already up and running system. For example taking an entire Amazon region out is not a very good idea, so you simulate it instead.

It is vital to run experiments in production, in a real system, because of increasing micro services and high complexity systems[3]. To reproduce or simulate everything in a subsystem or system can be extremely inefficient, costly, slow and would not give the same real interactions. Therefore, running experiments in production is necessary for simulation of realistic outcomes.

Lastly, it is of grave importance that experiments run over and over again with newly generated hypotheses every time. With this mind set the confidence for the system is increased each time when the hypothesis is unable to cause a disruption in the steady state and a weakness is detected otherwise if a perturbation occurs. As time goes on the same fault injections should run continuously as the program develops to see that the earlier bugs that were caused by fault injections but later fixed still are fixed with a newer version of that same program. This creates a demand for automated fault injections. Regardless whether it is a failure or success in the chaos experiment, something positive can still be savaged as for the first case it is quite literally and for the second case

the detection of weakness create space for improvement to the system. Usually, there is a big amount of experiments when doing chaos engineering and this also leads to a demand for automated fault injection as it is quite tedious to test out all of the experiments manually, therefore we used this pattern to define our experiment; First you have to define a steady-state, then you make your hypothesis, then introduce your variables and lastly you observe the difference in steady-state.

F. ns-3 network simulator

ns-3[5] stands for network simulator 3 and is an open-source software meaning you can use, read, modify and redistribute the software as you want. ns-3 is written in C++ and is designed as a set of libraries, where libraries can be modified, new libraries can be imported etc. Different external visualization tools can be used for ns-3. In our case we used NetAnim.

In ns-3 there are numerous commands, syntax and other things one has to understand in order to make simulations. In particular the nodes, which are used in our studies, are either a csma, a p2p, a wifi Station or a wifi Access Point node. The communication between these are handled mostly by the ns-3 classes. A bit more detailed description about these node types presented in figure 2 and figure 3 .

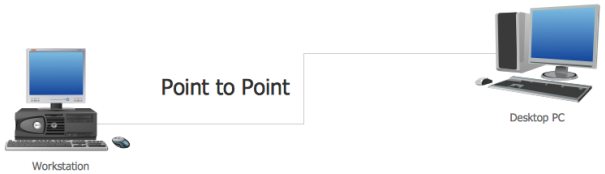

Fig. 2. A point-to-point-network-topology.

G. Background of lineage driven fault injection

LDFI stands for Lineage driven fault injection. Netflix has a LDFI system called ”Molly”[1] whose approach is either finding a violation of invariants and tracing the root of the bug or exhausts the execution without discovering a violation and then the program can be called ”certified”. LDFI has a specific approach which revolves around two key points. The first one is that a system can have enough alternative ways to recover from some injected fault conditions. The second one is to rather than to try different executions at the start instead start from the end of a successful execution and walk your way backwards in order to ask the question why the execution was successful. By asking these types of questions we can illustrate a lineage graph that shows all the factors that gave that certain outcome and ultimately it’s implicit

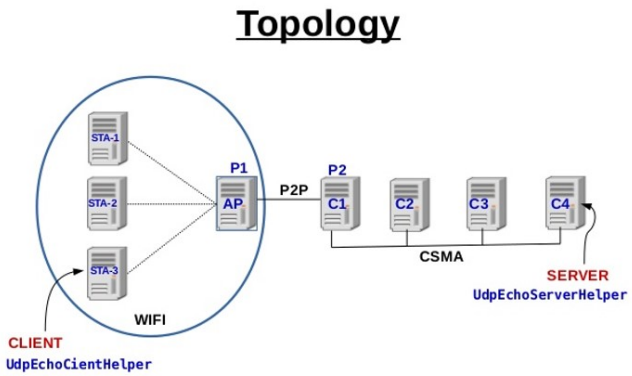

Fig. 3. A wifi topology example.

redundancy. Lineage graphs can be put together into a ”leaf”. Then it is important to find a combination that invalidates all the paths. In the end a fault injecting framework should be able to either fail and show the lineage graph or succeed to reach the positive outcome by alternative ways.

H. State of the Art

Several studies about fault propagations in the network of the distributed systems has been done in the recent decade with the main objective of finding the dependencies of components. By applying fault injection into one component the error is then expected to affect other components in the surrounding as a dispersion, which also defines as the fault propagation. Written by G. Candea, M. Delgado, M. Chen and A. Fox Automatic failure-path inference [4] is one of such studies relevant to the mentioned concepts about fault propagation, as in their research they proposed a technique for finding dependency by injecting Java exceptions. Another vital study is Dependency Analysis in Distributed Systems using Fault Injection, by S. Bagchi,G. Kar and J. Hellerstein , since they are one of the first to propose the idea of using fault injection for dependency [2]. Concepts and techniques like these are closely related to our implementation, the Inferator, due to the fact about the usage of weaknesses existed in a system for figuring out the outline of the structure of this system. In a sense this outline can also be seen as a kind of dependency graph, but with some differences to the studies proposed for fault propagation, because firstly the Inferator implements the chaos engineering concepts such as LDFI, which was used for rebuilding the network or dependencies and secondly it is the about the assumption we made regarding the fact that only the injection points,nodes , were known about the system as we regard it as a box where signal can be inject for an out signal and no knowledge at all about the interior of the box. Therefore as a conclusion the novelty of our study lies within

the very fact about the employment of chaos engineering for new possibilities of information retrieval from a systems weaknesses.

I. Relation to KTH Bachelor directory

Within the bachelor project in KTH we had to undergo and write about the sustainability and environmental aspects. Let us therefore before we head on to the design of the simulator mention the environmental benefits of Chaos engineering. There are some things we value a lot and one of those things is memory. They can be in forms of images, videos, texts, etc. They can be stored digitally. This has become an industry of its own. From USB-sticks to external hard-drives. You often times need portable memory in order to access a certain memory but nowadays carrying physical hardware, which has a small limited memory, is quite uncomfortable to bring everywhere and is not very common anymore. People are instead using cloud services such as Dropbox, Onedrive, Google Drive and Icloud. There you can safely store your memories and not be afraid of losing a physical memory. These cloud services are based upon distributed systems. Large companies also use their systems that run on distributed systems. Creating physical memory in the form of hardware wastes unnecessary natural resources such as minerals for example. The machines that mass produces these products needs energy. During and after the production there will be a lot of toxic waste. Therefore, a distributed system is key for a sustainable future. But these distributed systems are extremely complex. They easily bug out and the demand and pressure for cloud-services are high. So in order to maintain a reliable and fully functioning distributed system chaos engineering can be implemented. Continuous use of Chaos engineering will definitely ensure quick and positive outcomes to any distributed system. Ultimately, Chaos engineering will result in a sustainable distributed system which in turn results in a sustainable environment.

II. Design of the simulator

The fundamental of chaos engineering is to voluntarily inject faults into the system, so that the hidden weaknesses are uncovered during this process and as an application to this principle we will be introducing our designs for the two known methods for fault injecting in ns-3.

A. Random fault injection

To apply chaos engineering, the case studies is initialized in ns-3 by first designing our chaos injecting tool, so that it can automatically inject fault given a module. This module is the test subject where the experiment will be carried out upon. Mainly the whole structure of our design was intended to contain 4 parts working with each other. The first part is the module (the test subject) which produces an output according to the given input into the system. The output in our case is the Ubuntu terminal output, which we mostly rely on in our study. As for the second part it is the so called the chaos monkey[8], which is responsible for causing error in the module under the supervision of the third part, the controller. The controller controls the chaos monkey and make sure of its acquisition of knowledge about the current running experiment, such as destroying a random node, destroying a random device, change data-rate of a random device and so on. In addition the controller also plays an important role when analyzing the output of the module with the attempt to determine whether the chaos experiment was a success or a failure. The success here is when the system still manages to run as if nothing unusual happened despite the chaos was caused inside the module. As an interpretation of this occurrence we say that the system is resilient against the injected type of error and therefore the experiment is considered as a success. On the contrary if the system collapses or still manages to run, but as a degenerated version, then it is judged as a failure. After the trial was deemed as a success then controller will then be initiating a new chaos experiment on the module and the same procedure as elaborated above repeats itself. Finally as for the fourth part of the system, it is called the self-healing part. This part is responsible for repairing the module with the guideline to preserve the module's normal functions even when chaos was injected into it. In accordance to one of the principle of chaos engineering, the anti-fragility[7], this fourth part is not particularly of great importance in our study, but it is still included mainly for demonstration purpose when it comes to test out a resilient system. The self-healing part is what comes after the detection of a weakness of a system and after the discovery of a bug. The developer is expected to delve in to the healing part to insert a solution, which solves the problem when the weakness occurs, so that when the same type of error arise again next time, the system will remain unaffected. The pseudo code and design sketch is shown as below

Pseudo Code for random fault injection design:

while not out of hypotheses **do**
  if Controller starts **then**
    Controller initiate a chaos experiment and inject the hypothesis into the monkey as instructions
    MyMonkey receive instructions and modify parameters inside the module according to instructions.
    MyModule start running.
  end if
  if Error is detected in the module and Self-healing part is enable **then**
    Repair the module.
  end if
  if Module is done running and crashing is false **then**
    Inject another hypotheses into the monkey and rerun the module

end if
end while

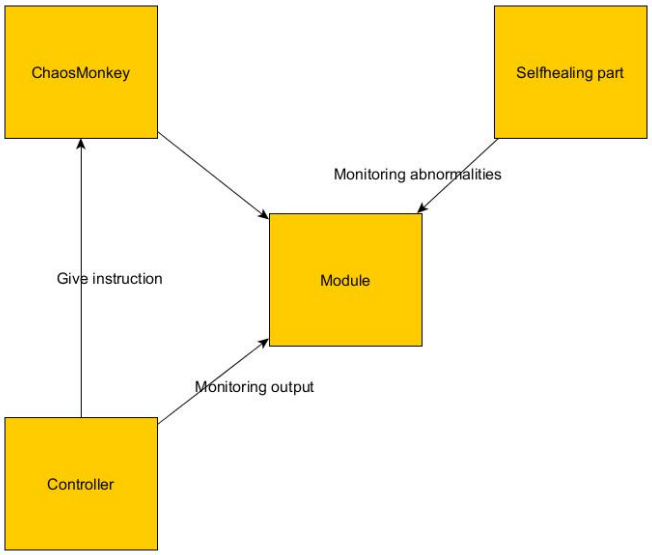

Fig. 4. Automatic faultinjecting tool design sketch.

B. Targeted fault injection with LDFI

As for linage-driven fault injection (LDFI) the main purpose is to uncover the weaknesses of the system derived from the successful cases. Unlike the random injection above this method is somewhat better in the appropriate context, such as a big network system, since it zooms in our scope of experiments. Regardless, this is design-wise about the same as figure 4 when it comes to automatic fault injection. We still need some form of controller to analyze the output of the module, but in this case we excluded the self-healing part as it is not very essential when demonstrating the kernel of LDFI and comparing the ability of detecting weaknesses between random and linage fault injection. Other than automatic fault injection it is also demanded for something to question the success of the module as an implementation of the principle of LDFI, therefore a mapping method in the module (test subject) is necessary. This mapping method is intended for deriving all of the possible links to the same outcome given an input from the controller. In order to achieve this we have made it convenient by saving a node as a neighbor of the other node every time a channel is being created to connect these two nodes. Therefore every node should have a list of neighbor nodes, which it is being in contact with. By using this fact we are able to map from one node to another, when the module is requested for mapping by the controller, by applying breadth-first search technique. The links found by this strategy is equivalent to the answers to the question posed by us previously when we asked about why is the outcome successful. The controller will then use these answers (links from the module) to calculate all of the possible injection points which cause failure of the system. If the links are given as

$$Link1 : \{A, B, E\}$$
$$Link2 : \{A, C, E\}$$
$$Link3 : \{A, D, E\}$$

Then the intersection of all these three links is

$$Intersect = Link1 \cap Link2 \cap Link3 = \{A, E\}$$

Also it is desirable to take difference of all these links with the previously found intersection and save them in to an array and while doing it we might as well calculate their union

$$Link1diff = Link1 \cap Intersect = \{B\}$$
$$Link2diff = Link2 \cap Intersect = \{C\}$$
$$Link3diff = Link3 \cap Intersect = \{D\}$$

$$unionlinkdiff = Link1diff \cup Link2diff \cup Link3diff = \{B, C, D\}$$

Before we begin the actual search of chaos solutions it is also convenient to first rebuild all of the elements in unionlinkdiff to nodes and for every node add every other nodes in the set as its children. The reader might already guess that breadth-first search technique will be used here once more. Now for each element in unionlinkdiff we build a test solution to run through the intersect array and check if whether it is successfully eliminating all of the linkdiff. An elimination happens if our test solution intersects with a linkdiff give a non empty set. So if the test solution is {B} then only Link1diff is eliminated. In this case we then build next generation of this test solution by appending each of the rest elements in unionlinkdiff. So now we have two new generated solutions {B,C} and {B,D} and yet again we are unable to eliminate all of the linksdiff. These two also result in two new test solutions {B,C,D} and {B,D,C}. These happen to be our chaos solution and therefore we stop the search regardless if there might be more or not, because we are only interested in the shortest possible links in this case. So as long as all solutions in the current test solutions generation (at the same tree level in figure 5) are found we will record them down and terminate the search. This procedure is repeated for each element in the unionlinkdiff set as a new start point to make sure that every solution is found. The reader should also notice that other than these solutions every element in the intersect set is by itself also a solution, since it is the set of the start and end node. Another remark is that eventual permutation and extension(for example set {A,B} is consisted

in {A,B,C}, meaning {A,B,C} is an extension), which arise at the final solution set will be eliminated. This is done by checking whether a set is a permutation of another set and this is the case if the difference of these 2 sets is an empty set and as a final remark for the current example the solution set is {(B,C,D),(A),(E)}. Our reasoning up until now can be summarized in figure 5, where the search as described start from node B, C and D in the unionlinkdiff set and ends at the green nodes. Another good thing to mention is that this discussion is in fact based on one of our real studies with LDFI in figure 6.

Pseudo Code for the design of LDFI:

if Controller running **then**
  Run the Module/System for a log file consisting of all package sending events, possible for applying LDFI.
  if events found in log **then**
    ask the Module/System about all of the possible nodal links from the start node to the end node.
  end if
  if nodal links information receive **then**
    calculate chaos paths as described above
  end if
  while the number of chaos paths are not 0 **do**
    inject back to the module for confirmation.
  end while
end if

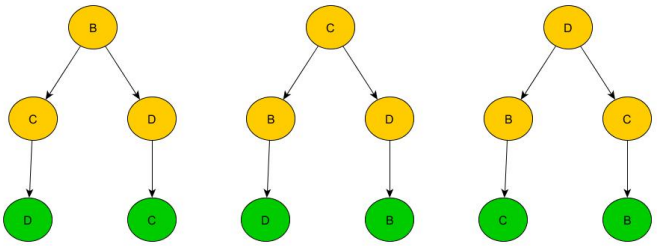

Fig. 5. Demonstration of how we find our solutions given links from the module.

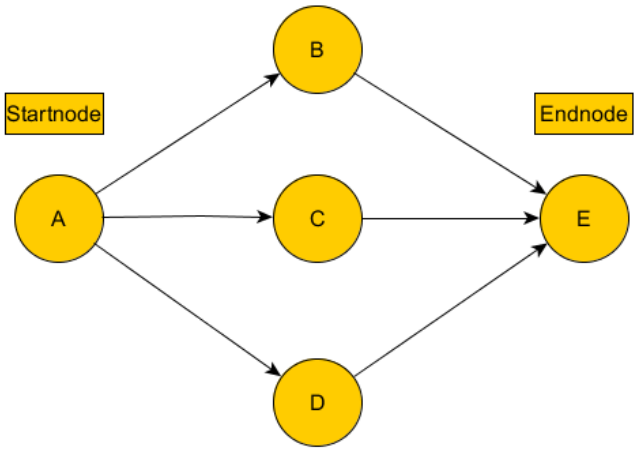

Fig. 6. LDFI example.

The current example is quite simple, but it is easy to demonstrate our idea with the chaos hypothesis. More complex problems are presented in the case study section and we will show that our algorithm will work even there. When the chaos hypotheses are found, they are used to inject back into the module to experiment the result under the supervision of the controller. The experiment is deemed as success if the signal is not echoed back from its destination and failed if it does.

III. Case study: NetFlix's simulated network

In this section we will be discussing the design of our own simulated Netflix module and the chaos engineering works which we applied on this module. Furthermore case studies for LDFI are also presented for demonstration.

A. Automatic random fault injection on Netflix module

It is quite known that the Netflix network is highly complex and big scaled, so we made an effort in simplifying this enormous system in ns-3 by abstracting several things. In this simplification we assume that the Netflix Network have services nodes which are interconnected to each other. An abstraction of this though are the csma nodes that can be seen as faculties with the responsibility for providing the client with the appropriated service, for example when the client request a data stream of his/her favorite movie, the services will be working together by sending signals/information to each other in order to produce the required result. After successfully producing this result this information will then be sending back through the main hub, which is mainly responsible for directing the in/out-coming signals from both the clients and the services. All of the nodes, connected to the Netflix main hub, are by point to point. Last but not least the contact between the central node and the client is maintained by a wifi access point node, which in this case is an abstraction of a modem in the client's vicinity. The clients themselves

are wifi station nodes, a representation of a device capable of connecting to the modem.

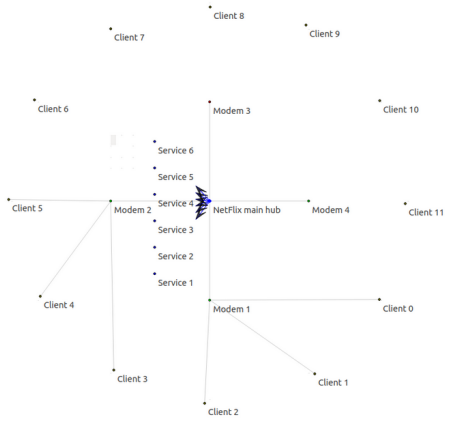

Fig. 7. a general outline of NetFlix's network.

As an application of chaos engineering automatic random fault injection was used on the simulated Netflix module. All the possible injection points in this case are the nodes (Node object in n3) as we have seen in figure 7 and by the node object it is possible to access all other aggregated objects, which was built into the node during compilation. The aggregated objects can for examples be a device (channel in network term), internet interface (Ipv4,Ipv6 ..) and so on. This also means that it is possible to change the attributes of all object as long as the node object is accessible. Our first chaos simulation started on this module by doing a classic case, the Simian army. As mentioned in the previous section there is a class, called MyMonkey in the script which can be used to inject fault in a node at either a predetermined or random time. For the sake of observatory clarity we have only made one monkey to inject fault in data-rate at time 1.004s at the Netflix mainhub node, figure 7. During runtime the system is expected to be delayed due to a decrease in data transferring. This is discovered by plotting the total amounts of events vs time. An event in this case is equivalent to an event message in the terminal output in the output document DataRatefaultinject in github, for example, "At time 1.004s monkey 2 caused chaos to datarate of device 0 is counted as an event" is an event a time 1.004s. To be more specific the only requirement is that the message must have a time when it occurs to be counted as an event in our case. If our auto-healing part is not on, we will have different time lapses at certain moments, when the non-chaos case and the chaos case without self-healing is compared. This is the case by taking a look at figure 8. We notice that for example during the time interval 1s and 1.05s the yellow step function took longer time to make a next step than the non-chaos case, blue step function. This also happens at other time intervals such as between 1.1s to 1.15s and 1.2s to 1.25s. The developer will then determine whether this result is a failure or a success by observing this data. If he believes that this is out of the fault tolerance, he will then dive into the system and find the cause to include a self-healing part, so that when the same error occur again the system will not be disturbed, as in chaos engineering term we say that the system is resilient. This case is demonstrated when comparing the blue step function of the non chaos case and the brown step function of the chaos case with self-healing. The brown step function is a bit higher in event counts than the blue one because it has 2 more events (monkey caused chaos and the system heals itself) written out in the terminal, but it can be confirmed that the time lapse is the same everywhere for both of them. In summary this example shows that previously our system was vulnerable to a data rate change caused by an outer influence. We then successfully uncover this weakness and made the system more resilient as a result.

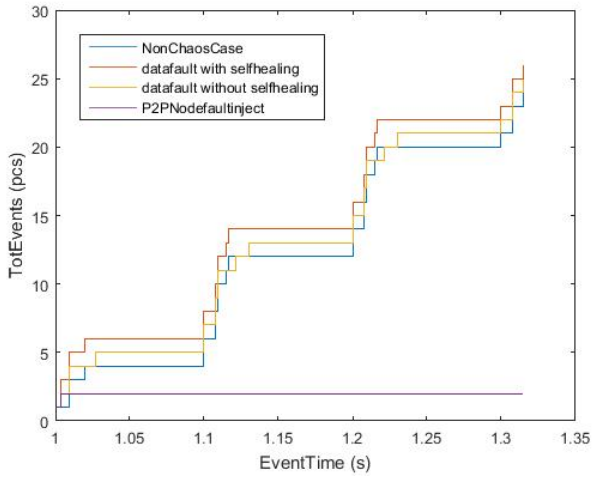

Fig. 8. Plot of total events vs time passed.Terminal output document: DataRatefaultinject

Other than fault injection in the data rate our tool is also capable of starting series of experiments through the controller. Each member of the series is a chaos hypothesis to be test out as a type of fault injection like the data rate shown in figure 8. A chaos hypothesis is an assumption, intuitively generated by the developer as he/she looks for all possible types of input into the system. Each hypothesis is to be tested and if the result is a success then the controller will move on to the next experiment otherwise it will report back an error message and stop the series of incoming hypotheses. This is shown in output document Multiplerandominject in github where the experiment 0 with data rate ran with a success outcome and the experiment with nodefaultinject was stopped due to a SIGSEGV since the destruction of a random node by some monkey resulted in a crash and our system is not resilient to that kind of error. The system crash can also be represented as the violet step function in figure 8.

B. Results of LDFI

In this section the focus of the discussion will be on the application and the advantages of LDFI. The principle of LDFI is to find weaknesses from the successful outcome and to demonstrate this we have done some studies on three modules and one of them was already demonstrated in the systematic design section as an example. In these modules only point to point nodes were used, because the idea was that we was currently in the internal system of Netflix and to be more precise they actually are the service nodes which we have seen in figure 7 in the previous section.

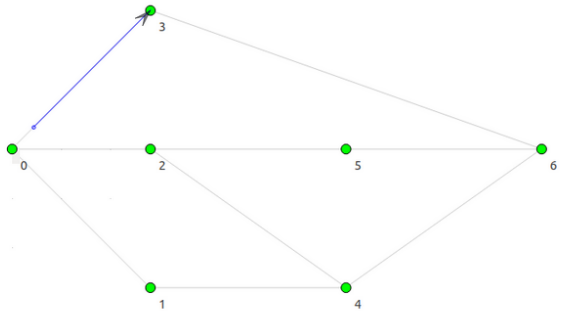

Fig. 9. LDFI first sample. LDFIsample1output.txt

In the example shown in figure 9 node 0 is making attempts to send a signal to node 6. The controller will run this module in non-chaos mode and analyze the output for potential chaos injection points. The controller catch events with label [PlanedSendEvent] and ask for the module to give back all of the possible links from node 0 to node 6. These links are {0,1,2,4,5,6},{0,1,4,6},{0,2,4,6},{0,2,5,6} and {0,3,6}. By using the algorithm described in the systematic design section we obtain solutions {1,2,3},{2,3,4},{3,4,5},{0},{6}. When testing out these hypotheses, no echo was seen in the output (in the output document LDFIsample1output.txt in github), which means the tested hypotheses are correct. Another example is presented in figure 10 and its output in the output document LDFIsample2output(solutions = {1,2},{3,4},{0},{5}). It is procedure-wise the same as example in figure 9. As in comparison to the random fault injection LDFI is obviously better in this case when finding weaknesses. For random injection we will have to test out all of about $\sum_{r=1}^{n} \binom{n}{r}$ solutions, n is the total number of nodes and r is size of the solution (if we assume that the solution, for example, only have r=2 elements), which is disastrously painstaking if we have a bigger system with for example 50 nodes. A more detailed comparison can be found in figure 11, where the first two stocks represent the time comparison between the LDFI case in figure 6 and its random version for 5 nodes, the second two stocks for figure 9 and its random version for 7 nodes and the last two stocks for figure 10 and its random version for 6 nodes.

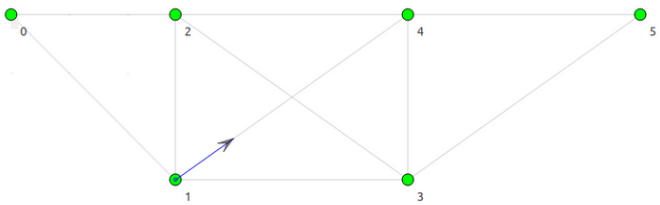

Fig. 10. LDFI second sample.Terminal output document: LDFIsample2output.txt

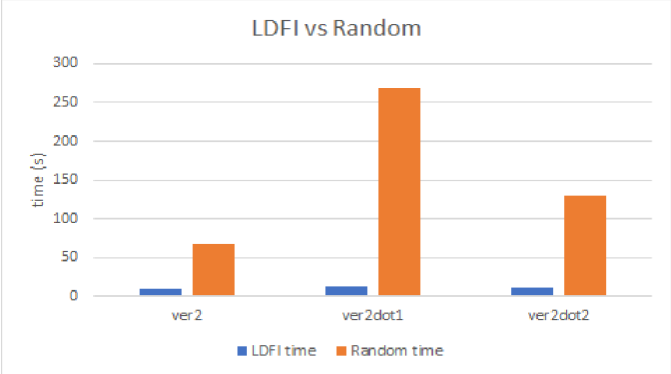

Fig. 11. LDFI vs random fault injection .

IV. Infer the structure of a system using LDFI and Random fault injection

As an application of the knowledge above, we will demonstrate the possibility of rebuilding the outline of the structure of an unknown system given its weaknesses. Therefore the discussion in this section will be rested upon the mechanisms and ideas of our constructed system.

A. Experiment protocol

Starting with the Unknown system in figure 12 there are not much to be mentioned about this component other than the fact that only the injection points of this system are known and in this case they are represented as nodes in figure 9 and 10. As for the random fault injector, its main tasks as described above are to test out all of the possible hypotheses on the unknown system and analyze the output for eventual occurrences of errors. The data acquired from these experiments is then sent to the Inferator as a text file, which content is separated in two types, the successful and failed hypotheses. When the data is obtained an effort will be made by the in sorting out the minimal sets of the successful cases and thereafter pass over as the weaknesses of the unknown system to the build generator, a set of functions responsible for building hypotheses. For each test build generated it is determined as a success if weaknesses of the test build completely matches or do contain the assumed weaknesses at the current build step. The weakness for the test build is imparted by the inferred controller, which apply LDFI onto

the unknown system by motoring the sent events of every created echo client. As a result each time a packet is sent the Inferator will notify the inferred controller as well as passing the information about the start node and destination node of the packet. After obtaining this information the controller will request all the possible link for its chaos calculation as described above in LDFI section and give the solutions back to the Inferator as the current test build weaknesses. One other thing worth mentioning is the criteria about that "weaknesses of the build completely matches or does contain the assumed weaknesses", which can be made to "weaknesses of the build must completely matches with the assumed weaknesses" in our program, if we are only interested in the exact solution. Last but not least everything discussed in this section can be summarized in figure 12 and about the algorithm used for calculating the minimal sets and the building process a more detailed discussion will take place in the next section.

Pseudo code for the protocol:

if Inferator starts **then**
  start the AutomaticFaultInjector for data gathering.
  if AutomaticFaultInjector start **then**
    while number of hypothesis are not 0 **do**
      Generate all possible combinations of the injection points, as hypotheses and inject it into the UnknownSystem
      if Hypotheses is success **then**
        save it into the successcases column.
      else
        save it into the failedcases column
      end if
      return the successcases and failedcases as a txt file to the Inferator.
    end while
  end if
  if txt file received from AutomaticFaultInjector **then**
    Minimize all the successful cases as described above and redefine them as weaknesses of the system
    Identify fatal points in the acquired weaknesses as start and end node, the start node is put in as a previous build point and begin the experiment for the rest of the weaknesses.
    for each weaknesses other than fatalpoints **do**
      Generate builds as described in the algorithm section and test out the build
      for each generated build **do**
        if the test build have all the assumed weaknesses **then**
          save it as a previous successful build
        else
          ignore it
        end if
      end for
      if number of previous successful build in the current buildlevel is not 0 **then**
        take one build from it and build the current test system
        save the newly included point for this chosen build and move on to the next weakness.
      else
        while number of previous successful build in previous buildlevel is 0 **do**
          Reverse the current test system by demolish the build in previous buildlevel.
          if the previous build level is not empty of success builds **then**
            take another successful build in the same build level and try it out again(call the function again recursively).
            break this while loop
          else
            if no previous build level left **then**
              discard this entire experiment until now and permute the weaknesses and start again.
              Impossible to infer the system if all the permutations has been tried out and still no solution found.
            end if
            continue demolishing.
          end if
        end while
      end if
    end for
  end if
end if

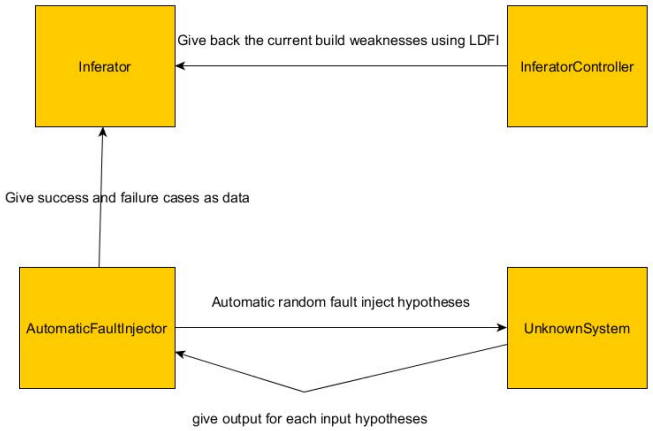

Fig. 12. The design sketch of the Inferator

B. Algorithms

To shed some light upon the algorithms inside the Inferator we will be describing the mechanisms by using sample 1 in figure 9 as an example. Beginning with the weaknesses of the unknown system calculated from the data from the automatic fault injector it is of the format

|weakness1|..weaknessN|fatalpoint1|..fatalpointN|and in the context of sample 1 |1,2,3|2,3,4|3,4,5|0|6|. Fatal point here is the points which cause a complete collapse of the system and in this case it is where the signal starts and ends, the start and end node. In the cases we have studied that it only contains 2 fatals points as we hypothesize that the case with multiple points are but a duplicative version of the most basic one, meaning if you have 3 fatals points A,B,C you can make combinations of start and ends nodes {A,B},{B,C} or {A,C} and run the Inferator with it. It will definitely be requiring some adjustments for our simple cases in order to infer for more fatal points, but principally it should be unchanged. Before going further into the calculation some definitions must be stated in order to achieve further clarity later on.

Definitions:

I A unique set of size 1 is a minimum size set.

II A set is not of minimum size if it is an extension of a minimum size set.

This means for instance that {A,B,C} is an extension of {A,B} or {A,C} , but not an extension of {A,D} nor {A,C,D} under the assumption that {A,B}, {A,C}, {A,D} nor {A,C,D} are of minimum size, while comparing {A,B,C} with respective cases. Using these we can therefore derive an algorithm for calculating the minimum size sets given the data from the random injector. It can also be seen that we also make it convenient for calculations by ordering the data in the text file RandomDatacaseNetFlixver2dot1 in github as smallest sets comes first, therefore the algorithm goes as we start by taking the first element 0 and check if any other sets contains it. If it does then we eliminate it as it is not of minimum size and also append the survivor in a new set with 0 for the next recursion. In the next recursion we take the set after 0 which is 6 in this case and do the same over again. The final result is the sets {1,2,3},{2,3,4},{3,4,5},{0},{6} similar to the chaos solutions of LDFI case for figure 10. As an important remark we would like to point out for those who might be mistaken this as LDFI, that these solutions similar to the LDFI is only possible for acquiring by this method under the condition that all of the injection points of the system are accessible and what most important is the accessibility of the success cases for the minimum size calculation are only possible after you have tested by using the random injector before that nothing can be said about the system other than the possible injection points.

Continuing to the building of the inferred system the fatals points(0 and 6) will first be set up as start and end node and 0 will be saved as a set in previous built point list and also by doing this the build will automatically fulfill the weaknesses 0 and 6. Next we will try to include {1,2,3} as a weakness and this starts by take the intersection of the will-be included weakness {1,2,3} and the previous built point sets

$$Intersect = \{1,2,3\} \cap \{0\} = \{\}$$

And then for taking the difference between the will-be included weakness and intersect, which we call the weakness points and also the same goes for the difference between the previous built point and intersect, called as injection points (this is a bit confusing for the previous real injection points so bear with us as we have a very good reason for it, also keep in mind that injection-/weaknesspoints are all nodes in this case) and uninterestingly the results are just empty sets as calculated below.

$$weaknesspoints = \{1,2,3\} \setminus \{\} = \{1,2,3\}$$
$$injectionpoints = \{0\} \setminus \{\} = \{0\}$$

Now if the injectionpoints are given as {A,B,C} and the weaknesspoints are given as {1,2,3}, then a new set will be constructed for each element in the injectionpoints with other elements in the weaknesspoints. Subsequently {A1,A2,A3}, {B1,B2,B3} and {C1,C2,C3} are received and as for the interpretation of these element A1 is understood as weakpoint 1 connected to injection point A and finally for each element in {A1,A2,A3} it will be combined with each in {B1,B2,B3} to acquire {A1B1,A1B2,A1B3,A2B1,A2B2,A2B3,A3B1,A3B2,A3B3} and this is then again used to combine with each in {C1,C2,C3} to give {A1B1C1,A1B1C2..} . This arduous procedure done recursively in our program until there are no more sets left and the program will then return all of the constructed sets as all of the possible build hypotheses for the current system. Getting back to sample one the first hypotheses are only the build 01,02,03 in accordance to our previous calculations. In attempt to find all of the successful test builds the controller will in succession try out all of the builds and save all of the successful ones for later use in case if a future test builds does not generate any hypotheses at all then it can reverse the build back to one of the other previously successful build and starting from there again as in figure 14. This is done recursively until it finds a solution or if it runs out of previous case to reverse to, then it will once again restart the experiment by permuting the would-to-be included initial weakness set and if it also failed to find a solution despite running through all of the possible permutations then it will be deemed as impossible. Finally to wrap up this section we would like end with an argument about the reason, where we first take the intersection and exclude those points existed in the intersection from the original sets of weakness, {1,2,3} and the previous built points set ,{0}. This argument is best displayed for the next weakness level {2,3,4} shown in figure 13 as the reader can see that to include {2,3,4} 2 or 3 is not an option since it already existed in some previous level, therefore 4 is the only

interesting choice in this context and at a closer look it can be found that a connection between 4 and 2 or 3 will never resulting in the weakness {2,3,4}, because an elimination of either node 2 will be unnecessary if 3,which exists in the same link as 2 illustrated in figure 13, is already eliminated as a reference back to how LDFI works.

Pseudo for Algorithm:

for each weakness in the acquired weaknesses about the system **do**
  Take the set intersection of the weaknesses and the previous build points in the system.
  Take the difference between the weaknesses and the found intersection. This defines as weaknesspoints.
  Take the difference between the previous build points and the found intersection. This defines as injectionpoints.
end for
create a list, call conpairslist.
for each element in injectionpoints **do**
  create a list, called conpairs.
  pair it with all the elements in the weakness points and save it in conpairs.
  when done, save conpairs in conpairslist.
end for
Produce build by initiate a recursion. Begin first by taking one element from the conpairslist and call it builds.
while conpairslist is not empty **do**
  take one element from the conpairlist.
  create a list called newbuilds
  for each element in conpairlist **do**
    pair it with each element in the elements in builds and save it to newbuild
  end for
  set builds equal newbuilds and start again.
end while

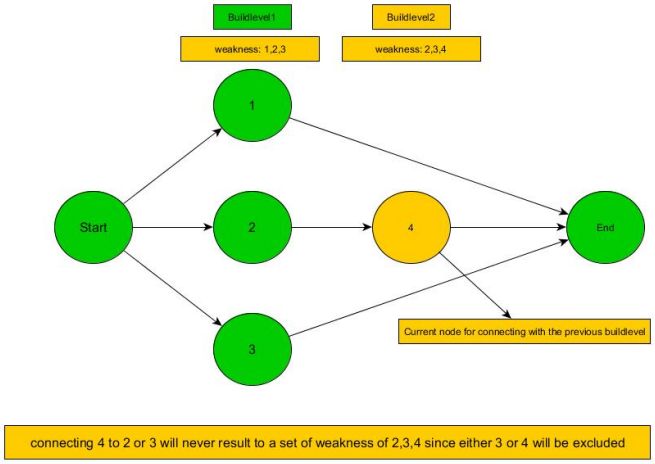

Fig. 13. Demonstration for build level 2 of the discussed sample 1 in this section. Here after including 1,2,3 we want to further include 2,3,4

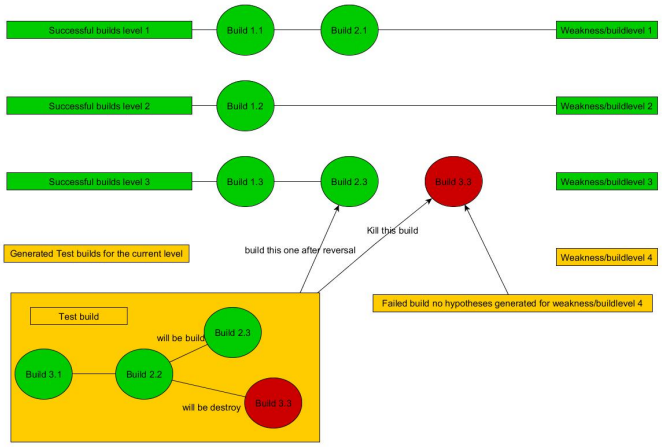

Fig. 14. The test build mechanism in the Inferator. Build i.j is interpreted as build nr i at buildlevel j. weakness/buildlevel 1 is for example the weakness 1,2,3 for sample 1

C. Results

Two case studies were done for sample 1(figure 9) and sample 2(figure 10). For sample 1 the weaknesses is {3,4,5},{2,3,4},{1,2,3},{6},{0} given from the random fault injector after minimizing the successful cases. This results to an inferred system in figure 15 with the approximate solution of weaknesses {1,2,3},{1,3,5},{2,3,4},{3,4,5},{0},{6}.

The same produce applies for sample 2 with weaknesses {3,4},{1,2},{5}{0} and the approximate solution in figure 16 has weaknesses {1,2},{1,4},{2,3},{3,4},{0},{5}.

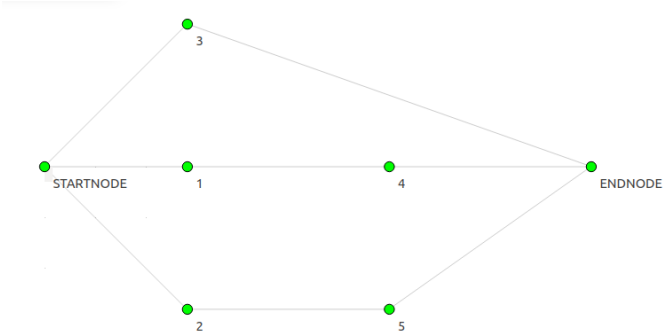

Fig. 15. The inferred version of sample 1 in figure 9. datafile: RandomDatacaseNetFlixver2dot1, XmlFile for NetAnim: Inferredver2dot1

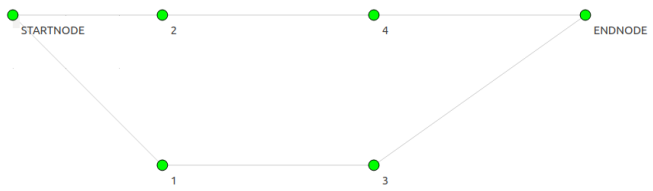

Fig. 16. The inferred version of sample 2 in figure 10. datafile: RandomDatacaseNetFlixver2dot2, XmlFile for NetAnim: Inferredver2dot2

It is likewise also the same for sample 3 figure 17 below here with the original weaknesses {1,2},{3,4},{3,7},{4,5,6},{5,6,7},{4,6,8},{6,7,8},{0}{9} and the approximate solution in figure 18 has weaknesses {1,2},{1,4},{1,7},{2,3},{3,4},{3,7},{2,5,6},{4,5,6},{5,6,7}, {2,6,8},{4,6,8},{6,7,8},{0}{9}.

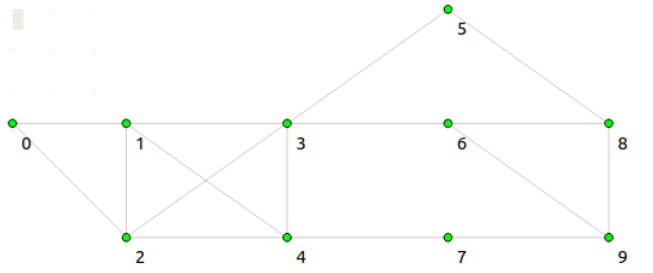

Fig. 17. LDFI third sample.Terminal output document: LDFIsample3output.txt

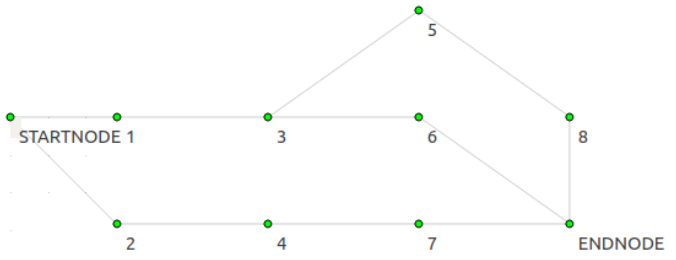

Fig. 18. The inferred version of sample 3 in figure 17, datafile: RandomDatacaseNetFlixver4, XmlFile for NetAnim: Inferredver4

V. Limitations

ns-3 does not really support multiple wifi access point nodes in the same channel (in the same animation run to be more precise), because it results in confusion for station nodes during information forwarding when multiple beacons interfere with each other. In figure 7 we have tried our best to make it as realistic as possible by connecting the client to the modem by a p2p connection other than only functioning as a wifi station node. This can be explained as the client write in his or her password when connected to wifi therefore establish a direct connection to the modem. Another problem with ns3 is the net animation. There still exists no service where we conveniently can print out our animation for exporting (Jpeg,pdf..) like Mathlab. This leaves us with the only option is to print screen.

VI. Acknowledgement

In this section we would like to thank Long Zhang and prof. Martin Monperrus for their valuable feedback on the project and their indispensable advice for our thesis as our workload has been significantly mitigated with their guidance.

VII. Conclusion

In summary to be able to automatically inject fault into the system, a controller and a good log system is required. ns-3 is an example for a good log system, since you can choose what kind of information level you want so that the controller save time to analyze the output and improve clarity when we read the output ourselves. Other than this we need an instance to cause trouble in the system and also another instance to fix the trouble, in which case we created the chaos monkey and the self healing part. These two are necessary for a complete demonstration of chaos engineering principles, therefore must also be included. All in all it can be witnessed that LDFI is a more realistic alternative than the random method when it comes to big-scale system such as Netflix, where time and resources are limited and therefore a random approach for detecting weaknesses is not quite suitable in this case. Meanwhile if the system is small, it is a bit excessive to go for a LDFI, since the implementation is quite arduous. In comparison to the random case where you normally only need to know what kind of signal you can put into the monkey LDFI required a complete knowledge of the structure of the system in order to make a correct calculation, which is why we need a map system for this. This poses a problem when it comes to an unknown system, which because of this we therefore can not implement LDFI without the knowledge of the structure.

Regarding the Inferator built by using LDFI and Random fault injection it is as demonstrated possible to some extend to plot the outline structure of the unknown system given the weaknesses found by randomly injecting into the system. However during our experiment we have found that if the ordering of the weaknesses were not given, then it will be more than just one solution of the system due to the fact that an inferred system with weaknesses such as 1,2|3,4|0|5|will be the same as 3,4|1,2|0|5|. This simply means 2 different inferred systems with the same weaknesses as in figure 1. Therefore, in order to acquire a more correct structure about the system another criteria must also be included, which is the knowledge about the ordering of the weaknesses. Regardless, this does not entirely dismiss the possibility of the main idea using

the weaknesses for retrieving information and the dependency between these nodes. As a remedy to this we would like to propose that in order to determine which weakness should be included first, it is desirable to for instance measure the time differences for different weaknesses before a failure do occur during the fault injection into the unknown system. Other than that to even further improve the accuracy we would also like to propose about, instead of only using the success cases gained from the AutomaticFaultInjector, it is wise to also make use of the failed cases from the datafile. Regardless, even if the ordering of nodes are wrong, it is still possible to recover the outline of the general structure of the system as shown in the examples.

Finally it is also important to point out some differences between this paper and the other papers, which also was written about chaos engineering. The main difference is that our implementations of these concepts are in ns-3 and also when doing LDFI as a simulation we used our own written algorithms in C++ for calculating chaos in accordance to our own mapping system, meanwhile Molly[1] used SAX library in order to achieve it and used their unique way of mapping by sending log messages. Other than about the original proposed methods like LDFI and random fault injection the demonstration of the Inferator have displayed a possible novel application by using a mix of both techniques as we have shown previously.

VIII. List of output documents

all the files below can be found in the master repo in https://github.com/kth-tcs/chaos-ns-3.

1) DataRatefaultinject
2) LDFIsample1output
3) LDFIsample2output
4) LDFIsample3output
5) Multiplerandominject
6) RandomDatacaseNetFlixver2dot1
7) RandomDatacaseNetFlixver2dot2
8) RandomDatacaseNetFlixver4
9) Inferredver2dot1
10) Inferredver2dot2
11) Inferredver4

References

[1] Peter Alvaro, Joshua Rosen, and Joseph M Hellerstein. "Lineage-driven fault injection". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM. 2015, pp. 331–346.

[2] Saurabh Bagchi, Gautam Kar, and Joe Hellerstein. "Dependency analysis in distributed systems using fault injection: Application to problem determination in an e-commerce environment". In: (2001).

[3] Ali Basiri et al. "Chaos engineering". In: *IEEE Software* 33.3 (2016), pp. 35–41.

[4] George Candea et al. "Automatic failure-path inference: A generic introspection technique for internet applications". In: *Internet Applications. WIAPP 2003. Proceedings. The Third IEEE Workshop on*. IEEE. 2003, pp. 132–141.

[5] Thomas R Henderson et al. "Network simulations with the ns-3 simulator". In: *SIGCOMM demonstration* 14.14 (2008), p. 527.

[6] Y. Izrailevsky and A. Tseitlin. "The Netflix simian army. http://techblog.netflix.com/ 2011/07/netflix-simian-army.html". In: (2011).

[7] Martin Monperrus. "Principles of antifragile software". In: *Companion to the first International Conference on the Art, Science and Engineering of Programming*. ACM. 2017, p. 32.

[8] NetFlix. "https://github.com/Netflix/SimianArmy/wiki/Chaos-Monkey". In:

[9] L. Zhang et al. "A Chaos Engineering System for Live Analysis and Falsification of Exception-handling in the JVM". In: *ArXiv e-prints* (May 2018). arXiv: 1805.05246 [cs.SE].