# NTS: A Scalable Virtual Testbed Architecture with Dynamic Scheduling and Backpressure

Youbing Zhong[1,2,3], Zhou Zhou[1,2,3](✉), Da Li[4], Wenliang He[1,2,3], Chao Zheng[1,2,3], Qingyun Liu[1,2,3], and Li Guo[1,2,3]

[1] Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
zhouzhou@iie.ac.cn

[2] School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China

[3] National Engineering Laboratory of Information Security Technologies, Beijing, China

[4] Department of Electrical and Computer Engineering, University of Missouri, Columbia, USA

**Abstract.** Experimental platforms perform a key role in evaluating the proof-of-concept and innovations. Nowadays, researchers from academia and industries rely on expensive physical testbeds to evaluate their experiments, while there are very limited software testbeds in market, which usually not available or costly. In addition, the applications of existing traffic generators are restricted to their single function and performance in network area. It has come to a point that lack of validation and testing tools has tremendously jeopardized the innovation in this field. In this paper, we propose NTS, which is a scalable software-based virtual testbed architecture. The scheduling and management framework can dynamically schedule resource of services. The scheduling algorithm adopts the concept of cost proportional fairness scheduling, which takes the evaluated traffic proportion and packet arrival rate into account. By leveraging container technology, the resources of services are restrictedly managed and fully isolated without tampering the OS kernel's scheduling mechanisms. Another advantage of the proposed testbed architecture is that the software can generate most kinds of backbone network traffic and can also be extended easily for customized protocol or traffic patterns. Our experiments show that the virtual testbed is generic scalable and cost-efficient, which is suitable and affordable for researchers in the field of network.

**Keywords:** Resource scheduling · Testbed · Docker container · Virtualization

## 1 Introduction

Due to the increasingly huge number of applications on the Internet, the network is becoming more and more complex and congested. Many researchers are spending vast amount of efforts in this area to address with the challenges. With the help of new

This work is partially supported by National Key R&D Program 2016 (Grant No. 2016YFB0801300), Strategic Priority Research Program of the Chinese academy of Sciences (Grant No. XDC02030000) and Youth Innovation Promotion Association CAS.

network technologies, such as Software-Defined Networking (SDN) [1] and Network Function Virtualization (NFV) [2], Internet service providers (ISPs) and application service providers (ASPs) adopt virtualization solutions to reduce equipment investment. Institutions and enterprises also propose different kinds of novel network technologies. For instance, the Intel DPDK [3] provides high throughput I/O in high speed network to accelerate packet forwarding. Accordingly, there are also some traffic generating tools available for testing network, such as pktgen [4] and UDPGen.

However, access to real internet online platforms to capture backbone traffic are very limited to many researchers, especially for those from academia. On the other hand, traffic simulation tools usually provide limited functionalities, which are not suitable for innovative protocols and applications. In the meanwhile, the tools often fail to mimic the content based on user-defined mode and real-world traffic load. As the consequence, many new algorithms and architectures, which require network traffic with the above-mentioned patterns, can be hardly evaluated using existing tools. As the existing tools lack of the ability to simulate real-world traffic and evaluate the results, academia and industries have realized the demand and importance of redesigning new architecture of testbed to facilitate various research needs.

Fortunately, the emergence of physical testbeds [5], which is one of most popular options, is a significant improvement. The physical testbeds usually are able to simulate multiple types of protocols by generating specified traffic. With abundant hardware resources, the physical testbeds are also able to simulate very complex, large-scale, and hybrid backbone traffic. However, while presenting a wide variety of advantages, the physical testbeds also have following drawbacks: First, these physical testbeds are very expensive, which is not affordable for universities and institutions from academia. Second, the physical testbeds are usually developed for a few scenarios and they are very hard or impossible to adjust for many needs. Last but not least, NFs also have heterogeneous processing requirements, which results in evaluation using physical testbeds not being suitable completely.

With the advent of container technologies like Docker [6], individuals can easily build traffic generators to mimic real application [7]. Even though OS scheduler can efficiently manage system resources, it doesn't have the knowledge of packet arrival rate and traffic proportion cost, resulting in serious performance degradation during the evaluation of NF. By leveraging cgroups [8], the scheduling of generators process can be exposed to the OS without modifying OS scheduler, reducing onerous work to customize scheduler in OS, which might lead to unnecessary maintenance overhead and inaccurate results.

Based on above-mentioned facts, many researchers proposed to exploit virtualization to facilitate building a cost-efficient and flexible testbed. As a matter of fact, some networking hardware vendors have developed virtual testbed and include their commercial products in the market. Usually, the solutions are based on real-world application traffic emulation running on top of hypervisors and the network functions are provided by NFV. At the low-level, application simulations are divided into VMs. At the high-level, all application simulations are connected by virtual network function with the help of NFV or SDN. Several solutions even provide the abstraction network topology for the underlining test environment like Mininet [9]. With all these appealing features like portability, flexibility and hardware accelerated performance, these solutions are unaffordable due to high commodity prices.

After analyzing state-of-the-art solutions, we propose a novel architecture for testbeds to simulate network traffic load generated by real-world applications, namely Network Traffic Simulator(NTS). The architecture takes both task-based application service scheduling and packet arrival rate into consideration. The focuses of our proposed architecture are on the control and scheduling problem of application simulation and traffic generation. NTS has following features: (i) generating real traffic load through interaction between services and clients, (ii) leveraging open source software that can be deployed on commodity hardware, (iii) CPU shares adjustment for services based on packet arrival rate and computation cost, (iv) providing scheduling elasticity to achieve backpressure to avoid wasting work and outburst due to congestion, (v) a generic scheduling framework without modifying operation system or kernel.

We aim to define a lightweight, cost-efficient, and extensible traffic simulator with affordability for researchers in network area to easily verify and evaluate new ideas and innovations. The design also considers the possibility to extend with user behavior simulation to generate synthetic network behavior pattern in the future, which can be used as background traffic in network security.

The rest of this paper is organized as follows. In Sect. 2, we discuss the related work. We introduce our proposed testbed in detail in Sect. 3. Section 4 presents the evaluation result and discusses the implication. Finally, we give a conclusion to our work in Sect. 5.

## 2 Related Work

This section provides the preview of previous work related to our proposal. It consists of three parts: The first part gives the background of the use of testbeds to validate NF, especially for innovative algorithm and architecture in network scope; The second part describes the deficiency of existing OS scheduler for service scheduling; And the last part mainly encompasses works that have exploited the area of testbed.

### 2.1 Background

Evaluation serves as a very important and challenging part for any new proposed methods, frameworks or systems in many research fields. In the area of network, whether the evaluation input can reflect the real-world applications and the collected metrics are good and accurate enough for the evaluation influence the result. This means that the effectiveness of simulation depends on whether testbed can fairly simulate various network protocol features. In other words, it is dependent on the traffic load that testbed generates.

For ASPs and ISPs, QoS related metrics are critical for the evaluation and adoption of any innovative network solutions. Performance degradation of QoS damages the reputation of their services and operations, causes customer complains, eventually result in losing market share and business profits. Another challenge is that it is very risky to evaluate new solutions in production networks, given the concerns on service outage and interruption. Additionally, because of restrictions to access privileged resources and privacy protection of user data, NF validations that need massive online internet traffic are limited and infeasible for many researchers.

However, with the application of container technologies and NFV, there is a potential possibility to solve these problems.

To solve these challenges, our proposal provides the traffic load generation and exploits virtualization to reduce cost and minimize the impact on production during evaluation.

## 2.2 The Deficiency of OS Scheduler for Virtual Testbed

Network function evaluation includes: (1) function validation, and (2) performance evaluation. For instance, firewalls and content audit system commonly need to detect the correctness of protocol identification and blocking effectiveness during tests.

Linux provides completely fair scheduling (CFS) [10] in the default mode since kernel 2.6.23. It manages CPU resource allocation for all running processes and aims to maximize overall CPU utilization. Each task under CFS maintains a fair chance to get certain CPU shares and the time-slice is determined by the run-time of contending tasks. Thus, CFS presents a fair CPU proportion shares to all tasks.

However, the fairness scheduling is not suitable for service emulation. Intuitively, different types of services have different computation costs due to the characteristics of the service. Fairness scheduling won't take this into consideration. Furthermore, users' requirements are diverse during evaluation. For example, if two simulation services, which generate different kinds of traffic, have the similar computation cost with different sizes of traffic loads (one has twice traffic load than the other), then we hope the schedule can match the traffic loads between the two. Similarly, if the service has twice computation cost than the second, then we expect it has twice CPU run-time at the same traffic load. Obviously, we can introduce the weight or prioritization factor to supplement the deficiency of fairness.

Unfortunately, lack of enough information for the default CPU scheduler to allocate resources according to the computation cost and traffic load pose the hardness of scheduling resources on the basis of cost-proportional fairness [11]. As mentioned above, CFS scheduler usually provides a fair allocation of time-slice, but it cannot provide rate-cost fairness if services have diverse computation costs. To achieve this, the core of NTS should ingest more information to allocate CPU resources in priority or weight mode.

Cost-proportional fairness scheduling differs from Round-Robin [12] and Max C/I. It is a trade-off between throughput and fairness, which not only seeks to maximize throughput for the services at given traffic load but also ensures that all the contending services needed during a test to obtain a minimal CPU share keep running in the worst scenarios. What's more, we can adapt cost-proportional scheduling to meet diverse experiment demands. This scheduling method also ensures that the winner among application emulations will not impede others.

## 2.3 Exploration of Testbed

For network research, a testbed is often used to evaluate the function and performance of NFs. As a result, the major function of a testbed is to generate various network traffic load based on protocol types. Despite the adoption of NFV and SDN, it is still a huge challenge to provide traffic emulation solution, which can mimic real-world services for evaluation due to the following reasons: Firstly, the rapid evolvement of Internet protocols increases the hardness to simulate traffic load conforming to the features of services. Secondly, traffic emulation needs to be flexible enough to support diverse protocols. Lastly, the huge throughput and privileged access of backbone networks also hinder the simulation. Consequently, parts of existing works focus on designing and building high throughput packet generators. In the meanwhile, some of the works are aimed to build mathematical models to produce traffic loads similar to the statistical characteristics of real-world network traffic. As alternative solutions, others achieve the goal indirectly by designing capture and storage systems to store internet traffic and replay under test environment. We select the works [11,20], which are representative and closely related to our proposal.

In [13], the authors presented a flexible high-throughput packet generator, which uses only a single CPU core by running on top of packet processing framework called DPDK. In the experiments, the generator could saturate 10GBE links using packets with minimum sized and provide the highest possible flexibility by Lua scripts. By leveraging the high-performance hardware, [14] described an open-source traffic generator, which has highly accurate inter-packet delays. [15] proposed a virtual testbed solution using software agents to emulate the activity of users thus generating similar network activity automatically. The experiments are evaluated through the validation of a network-monitoring tool for Voice over IP (VoIP). [16] defined methods to generate connections to simulate the statistical patterns of real network.

For the related work focused on packets capture, storage and replay, [17] showed a system using a modified network driver with the help of Non-Volatile Memory express (NVMe) technology and Storage Performance Development Kit (SPDK) framework, which is capable of capturing, timestamping and storing 40Gbps network traffic. [18] presented a novel framework, namely "Record and Deterministic Replay" architecture to log the traffic and then replay during the test. [19] researched the feasibilities of using packet header fields to partition network traffic for efficiently enabling distributed packet capturing and processing. In [20], FloSIS was proposed, which was a highly scalable software-based flow storing and indexing system. [21] employed network simulator to build the infrastructure and capture bandwidth traces in the wild and replays the traces reproducibly. Although the traffic generators using this approach can provide high throughput, it just constructs the packet and send out without considering any interactive information. Most of time, researchers cannot modify the content according to demand. Network activity pattern emulation is usually simple application relatively. Capturing and storing traffic loads need significant investment in solid state disk (SSD) or hard disk and it also take times to get the data.

In [22], it introduced a global testbed, namely PlanetLab. As the next generation of federated testbed, it aimed to federate multiple testbeds that owned and operated by autonomous organizations. However, it was subjected to members of the organization.

In the field of wireless network, Nils Aschenbruck *et al.* presents a new software-based approach that essentially combines mobility modeling with link control to facilitate evaluation of routing protocols and node mobility in testbeds [23]. In [24], Matthias *et al.* proposes a security testbed for the evaluation of wireless sensor network.

For theses many reasons, we propose a scalable software-based virtual testbed called NTS, which can generate real-world interactive traffic with high throughput. Also, the proposed virtual testbed can simulate most types of existing well-known protocols on the internet. Furthermore, by leveraging backpressure, the management framework of NTS can reallocate CPU resources for services dynamically based on cost-proportional policy.

## 3 Virtual Testbed Architecture

In a common testing environment, the devices, such as DPI, proxies or network content audit system, are deployed in series or parallel. The traffic load is mirrored or through by one or a few switch ports. For the sake of validation of system functions, the traffic generator must output a specified type of traffic, which has features similar to protocol under test. What's more, the operation platform should be able to schedule different kinds of protocol emulators in such a way that users can specify the requirement for CPU resources. However, operating system's scheduler doesn't have enough information of emulation applications. As a result, the NTS should be able to convert the scheduling requirements of application emulators to a format understood by the OS.

Although each container is isolated respectively, all of them run on the top of same CPU. The control platform of NTS needs to know the resource requirements of each traffic generator to avoid exhausting CPU resources, as well as monitoring the average process time of each service emulator. Furthermore, it also needs to estimate how many CPU shares to allocate for each generator. As for the evaluation indexes of testbed, it focuses on the correctness in function and throughput in performance in the field of network. The services queues are modeled as M/M/1 queue based on the statistical analysis. With the help of queueing theory, we can approximately calculate the resource quota.

To be aware of workload of various traffic generators in NTS, the scheduler needs to include network specific parameters in the scheduling algorithms. For instance, the scheduling algorithm needs to change the priority of a generator based on its computation cost. One possible implementation is to modify OS scheduler directly. However, it is a troublesome task that may lead to unnecessary maintenance overhead or introduce bugs, which will affect the stability and efficiency of the system. Also, using this approach means that a change of the scheduling priority requires a system call, which consumes CPU resources heavily if changes are frequent. Our proposed NTS adopts cgroups, which is a standard userspace primitive provided by Linux OS to schedule process. For simplicity, NTS monitors packet arrival rate and process cost and allocate CPU shares accordingly.

## 3.1 Overview of NTS

As shown in Fig. 1, the architecture of NTS mainly consists of four modules: (1) NTS manager, (2) services simulation composed of a series of service containers and optional network containers (e.g. BGP container), (3) various analog clients corresponding to services, and (4) backpressure (not included in the figure). Commonly, each service container is responsible for only one protocol simulation. The network containers are used for stimulation of network protocol or generation of tunnel traffic. We can deploy network simulator (e.g. Mininet) and devices under test between analog clients and services emulation to set up a virtual network topology environment. The basic workflow of NTS is simple: According to the parameters in the configuration file, analog clients make requests to services. By this means, it generates interactive real-world Internet traffic of different protocol. This way also offers users the chance to adapt evaluation indexes, such as protocol type or traffic proportions for various protocols, to meet test requirement. By leveraging flexible configuration, NTS can simulate a variety of hybrid traffic to meet most kinds of evaluation scenarios.

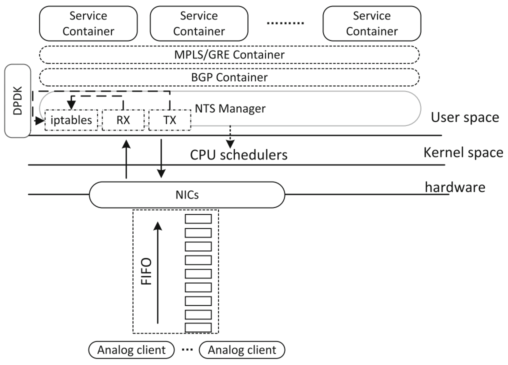


**Fig. 1.** The NTS architecture

To reduce context switches, NTS manager is allocated to a dedicated set of cores and is responsible for deciding which service container is to launch. NTS can also adopt DPDK optionally to accelerate packet forwarding with the help of user space protocol stack [25]. When the request packets from the analog clients arrive at the NIC, the RX does a look up in the iptables to transmit the packet to an appropriate application emulation in bridge mode, or directly send the packet to services in the host mode. Vice versa to the TX.

## 3.2 Traffic Simulation

As the core module of NTS, service simulation contains many types of application emulations. Each service based on a real-world application emulation is implemented in its own process and exposes ports from container without publishing them to host.

Moreover, most of service containers are constructed with open source software, reducing onerous task tremendously. Each analog client can perform basic operation. For the sake of reduction of deviation in the stage of resource evaluation, we have simplified clients. In other words, compared to real client, it only has one or two commonly used functions with single-mode. We can also specify the running time, execute count and so on in the configuration file for each one.

After establishing communication between services and analog clients, NTS can generate real interactive internet traffic. Through mount mode of volume provided by container technology, we can modify the content of traffic arbitrarily. This design brings great convenience to our experiment, especially for the function evaluation of network devices (e.g. content audit system). Besides, we can also decide which version of the protocol to mimic according to the test requirements. To the simulation of encrypted traffic, we can specify the certificate with different algorithms and length of secret keys. What's more, we have built diverse service applications for single specific protocol emulation to explore the difference of underlying servers. Every container is set up a threshold of CPU resources on the basis of calculated quota initially. The main supported protocols for emulation are listed in Table 1.

**Table 1.** Protocol Emulation

| | |
|---|---|
| Common emulation | NTS, DNS, Radius |
| Web emulation | HTTP, HTTPS, HTTP2, SPDY |
| Mail emulation | IMAP(s), POP3(s), SMTP(s) |
| Tunnel emulation | IPSEC, L2TP, PPTP, IKE |
| File transfer emulation | FTP(s) |
| Remote connection emulation | Telnet, SSH |
| Stream media emulation | HLS, RTSP, RTMP |
| Proxy emulation | Socks |
| Instant message emulation | XMPP, VoIP, H.323 |
| Network emulation | GRE, MPLS, BGP |

In addition, researchers can deploy their application emulation in container mode to generate backbone network traffic without interfering network devices in the system, which provides chances for the researchers to obtain privileged access traffic without impacting the infrastructure. NTS provides a flexible and cost-efficient traffic simulator with various protocols to evaluate novel algorithms, tools, and systems in network research.

## 3.3 Resources Estimation

Each time, NTS manger needs to set up the initial threshold, which is similar to slow start threshold (ssthresh) in TCP Congestion Control, for every application emulation in start-up phase. Although we can use the default mode, meaning to employ whole CPU shares for allocation, it is easily to overload the system because of resource exhaustion. For instance, if two services start at the same time in default mode but one requires more process time per packet and gains high priority, then the heavy one will hinder another because of contention. Besides, operators do not have information to generators when configuring parameters. If the parameters are set up so big that exceed peak, contending generators will occupy vast majority of resource, leading to boot failure or errors of others. Furthermore, presetting threshold for generators diminishes the number of resource reallocation times, reducing resource overhead.

By leveraging queueing theory, we can formalize the problem of estimation using a queue model. We assume that the set of application emulations can be represented as disjoint sets. In this paper, we only take CPU into account. As mentioned before, different application emulations have various computation costs and per-packet costs. In addition, the analog clients are normalized to perform simple single-mode operation to reduce impact. We treat these costs as variables. The application emulation services are modeled as M/M/1 queues. Using the standard formula for an M/M/1 process time, the average time spent in the service j by a traffic simulation is:

$$C_j = 1/(1 - \lambda_j/\mu_j) \tag{1}$$

$\lambda_j$ is the arrival rate of packets at $j^{th}$ service (the client packet delivery rate), and $\mu_j$ is the process rate. These two number can be easily obtained from directory /proc/pid/net (pid is service container process id). In this paper, the initial threshold is set at the base that total CPU resource account for not more than 50%.

$$\sum threshold_i \leq 0.5 * Total_C PU_s hares \tag{2}$$

$$threshold_i = \frac{C_i * s_i}{\sum C_i * s_i} \tag{3}$$

$s_i$ is the tuning parameter set in the configuration file. For simplicity, we select the minimum instead. In addition, only selected service containers to be run are taken into account.

## 3.4 Application Emulations Scheduling

Since multiple containers are likely to be available for scheduling to run at the same time, NTS must determine which service to schedule at any point in time. In our proposal, we leverage Linux's existing scheduling framework rather than designing an entirely new scheduler for application emulation. Furthermore, we tune the OS scheduler to provide cost-proportional fairness. Figure 2 shows how the NTS scheduler works. NTS manager governs OS scheduler via crgoups and assigns running simulated services to shared CPU cores ultimately.

If services are busy waiting for packets, the overall performance of the testbed will be very poor as it is a shared CPU environment. It is critical to design the management framework so that only services with packets available for them to process will be activated. NTS manager provides a relatively simple policy to trigger a service: once an operator configures parameters and specifies the types of protocols, manager will

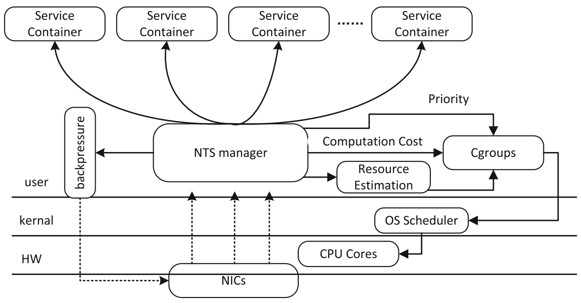


**Fig. 2.** Scheduling architecture of NTS

execute command (e.g. docker run) to start matched container with specified ports and other arguments, then service will be scheduled to run. This provides an efficient mechanism to trigger services.

For services with lots of packets to process, NTS supports preemption. The preemption decision and interaction with manager are arbitrated by the shared flag array set by backpressure. After processing a batch of packets, NTS manager will check the flag list to decide which state to keep. If the value in flag array is not set, the corresponding service will continue to run; if the value is not set and the parameter indexes (set in configuration file) has reached, the services will hold on (keep cpu ratio in balance); only if the value is set or there are no resources available, the service will be blocked until notified by the manager. This provides a flexible way for the NTS manager to indicate which service should be swapped out without the help from the kernel's CPU scheduler.

NTS manager provides mechanisms for application emulation to monitor arriving packets to estimate its CPU shares and adjust its scheduling weight accordingly. In this way, NTS manager can dynamically tune the scheduling weights for each service in order to meet operator evaluation demand.The packet arrival rate for a service can be easily measured. We measure the service time to process a packet inside each service using self-developed lib. NTS manager monitors all activated services to get a rate array. For simplicity, we maintain a histogram of timings and employ the median value to avoid outliers.

For service $i$ on a shared core, the load is:

$$load(i) = \lambda_i * S_i \tag{4}$$

$\lambda$ is the packet arrival rate and $S$ is service time. Then we can calculate the total load on the core $m$:

$$total\,load(i) = \sum_{i}^{n} load(i) \tag{5}$$

and assign CPU shares for service $i$ on core $m$ is:

$$share(i) = priority_i * \frac{load(i)}{total\,load(m)} \tag{6}$$

After figuring out the result, manager can modify the CPU shares directly without interrupting containers by writing the file located in /sys/fs/cgroup/cpu,cpuacct/docker-containerID (docker-containerID is created when service container launched). The allocation of CPU shares provides cost proportional fairness to each service. Besides, we can tune the priority to generate different proportional traffics indirectly for various evaluation scenarios. Comparing with adjusting the CPU priorities exposed by OS scheduler, this method provides a more intuitive control.

A key goal of NTS manager is to avoid blocking all generators. We can describe the situation as follows: when two or more services are triggered, NTS manager monitors new packets arrival rate of each service and reallocates CPU resources based on cost proportional fairness scheduling. However, if there is no idle CPU to utilize, all running analog clients will still send out excessive packets and services will be blocked, resulting in increased errors and performance reduction. We avoid this through backpressure, which ensures the NTS can detect bottlenecks quickly and minimizes the performance degradation due to blocking.

After allotting CPU shares to services, NTS manager communicates with other modules in the system. When the CPU surplus is not enough, NTS manager will drop extra packet and send a flag to analog clients, preventing request packet arrival rate from rising by increasing time interval. NTS manager maintains states of each running service, and in this case, it moves the service's state from uphill to steady. When the time expires, manager will stop all of running services, then the state moves to initialization. The state transition diagram is shown in Fig. 3.

## 4 Experimental Evaluation

To reducing hardware cost, our experiment uses four commercial computers and a 10GE switch device. We deployed two type of analog client, network simulator and NTS server, separately on each of the computers. All of them are connected by switch. In our

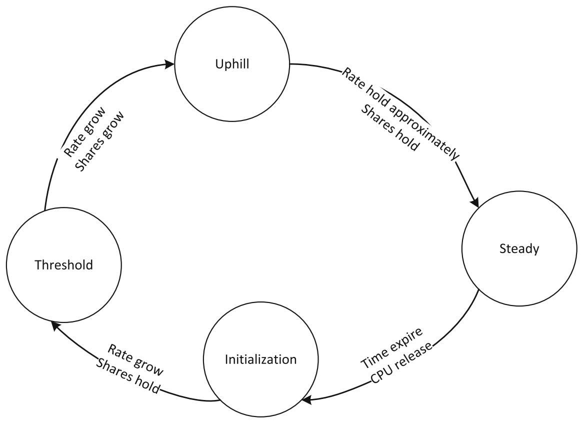


**Fig. 3.** NTS state diagram

experiments, we choose Mininet to simulate the network topology in all the tests. Each computer is equipped with a 2.4 GHz Intel Xeon E5-2680 processor and 128GB RAM. Each node is connected back-to-back with 10 Gbps dual-port DPDK compatible NICs. The OS is Centos 7.2 with kernel version 3.10.0-693.el7.x86_64.

## 4.1 Traffic Generation

Before proceeding with the NTS manager, we first validate whether the emulation can generate traffic that matches the protocol. In this stage, we select two kind of analog client each time. We start our emulators according to specified parameters and dump each kind of traffic to file in order respectively. In this paper, we adopt Wireshark to check the correctness of simulated traffic, protocol version and so on. Besides, to verify the function of content modification, we firstly examine the packet content before encryption. Secondly, we adopt available clients on the internet to validate cipher text. Part experimental results are show in Table 2.

**Table 2.** Protocol validation

| Protocol | HTTP(s) | HTTP2 | POP3(s) | IMAP(s) | SMTP(s) | FTP(s) |
|---|---|---|---|---|---|---|
| Traffic generation | ● | ● | ● | ● | ● | ● |
| Content modification | ● | ● | ● | ● | ● | ● |
| Algorithm type | ● | ● | ● | ● | ● | ● |
| Key length | ● | ● | ● | ● | ● | ● |
| Version | ● | ● | ● | ● | ● | ● |

Furthermore, we also employ NTS during the function test of audit content system for two month. Compared to the result of online Internet traffic, they are almost the same except that NTS can not support some latest protocol version (e.g. TLS1.3). In addition, after configuring Routing Table on switch, Mininet and network container, we can find route packet (e.g. BGP packet) between device and network container.

## 4.2 Evaluation of NTS Manager

Apart from traffic emulation evaluation, we also evaluate the effectiveness of NTS manager that influences the application scheduling decisions of the native Linux kernel scheduling policies. In this paper, we measure the throughput to evaluate NTS manager's overall performance every five seconds. We compare the default OS scheduler with our system.

To estimate the threshold of each service emulation, we start traffic generators one by one and count the packet arrival rate and process rate. Besides, we set tuning parameter the same. Then we can calculate the CPU shares and evaluate the threshold approximately. In this way, the threshold of services will be proportion to their actual computational cost.

Due to the high proportion in traffic and widespread use on internet, we select HTTP and HTTPS for evaluation. As for other kinds of traffic generators, we can evaluate them in the same way. To minimize the influence of other variables, the content and underlying servers of selected traffic emulation are same, as well as the configuration. In addition, we deploy the traffic generators in host mode, which can reduce the overhead of resource for looking up the iptables.

Apart from backpressure, one important optimization we apply to NTS is "Redundancy Estimate and Pre-allocation". In practice, we cannot evaluate resource requirement accurately, leading to the allocation of CPU shares inaccurately. We note that lightweight excessive assignment of the resource has a slight impact on the performance compared to inadequacy, which blocks generators and causes NTS manager to experience a marginal degradation in throughput and connections. Furthermore, frequent assignment increases the overhead of resource, leading to small amplitude degradation when involuntary switching.

To alleviate the unavoidable defect, we first adopt a buffering resources allocation, namely redundancy estimate, by referring to the existing optimization method in the scope. In this paper, we not only take the packet arrival rate and processing cost into account but also think about the throughput above. With the increasing magnitude of throughput gradually diminishing during the initial phase, we can estimate the computational cost based on queuing theory. We monitor the variation trends of throughput discontinuously and assign resources for activated emulation services with proximately ascensional range, resulting in redundant shares slightly. Once throughput do not arise anymore, we keep the CPU shares for a moment. This way could mitigate the impact of unbefitting allocation of CPU weights.

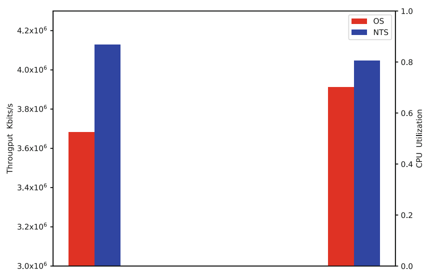
In addition, we also employ pre-allocation in NTS manager during the ascent stage. We could estimate the resources required to process packets for next time based on output and update the cgroup's weights of running services. That's to say we estimate the load at the present and compute the growth rate to predict the CPU shares approximately. Furthermore, once we detect that the output does not increase any more, we revert to the original method.

In this paper, we collect each service container state data every five seconds during the experiment with the help of docker technology (e.g. docker stats) to check their CPU shares. In all cases, the services specified in the configuration file are triggered by NTS manager.

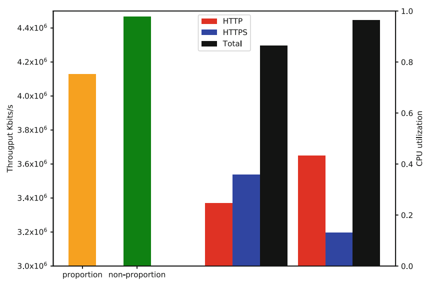
To evaluate the impact of threshold to NTS, we also set different initial tuning parameter and change the threshold proportion indirectly.

Figure 4 shows that NTS with backpressure can achieve improvement of throughput as much as 12.5% compared to operation system. It also shows that NTS can adjust services initial threshold by tuning the parameter directly according to evaluation requirement. In addition, it can also improve CPU utilization. By combining these, NTS with backpressure improves the overall throughput and can generate various proportional traffic.

For the default scheduler, the achieved throughput differs tremendously compared to NTS. What's more, we can observe that the value varies quickly without law. As mentioned above, OS scheduler just provides completely fairness scheduling.



(a)



(b)

**Fig. 4.** Throughput comparison

When contending CPU resources, services obtains CPU shares alternately. This leads to an inevitable context switch and amounts of overhead. As a result, the throughput goes up and down.

On the contrary, by leveraging backpressure, NTS will determine whether to process the additional packets depending on the residual resources. It also allocates CPU shares based on packet arrival rate and process cost. This way refrains from resource contention and avoid an outbreak of errors, maintaining throughput in a relative stationary status for a period. We can also observe that the value fluctuates in a small margin. This is because the variation in per-packet processing cost of services result in an inaccurate estimate of processing cost and thus an inappropriate CPU shares allocation. We could mitigate the impact of variable packet processing cost by profiling services precisely and frequently. We could also maintain a histogram of times to average the packet process cost. However, this can be costly because it consumes significant amount of CPU resources. This is why we use redundancy estimate and pre-allocation mentioned above to alleviate the penalty from the variability and get a relatively smooth and better throughput.

We consider two different initial threshold setting. Due to the high proportion in traffic and widespread use on Internet, we select two types from them, namely HTTP and HTTPS, for evaluation. The threshold proportion is set to be 0.22:0.28 at first and then set to be 0.35:0.15 for HTTP and HTTPS. The two analog clients and configurations are same except access port. There is no modification for the rest. As showed in Fig. 4(b), once allocating the initial resources according to the set threshold proportion rather than their actual computation cost, the aggregate throughput and CPU utilization has improvement with small margin. Besides, compared to the threshold of cost proportional fairness, HTTP obtains more CPU resources, which is opposite to the former situation. In addition, if we set the proportion unfairly, the lightweight and prior service container will occupy mostly CPU resources. Accordingly, it also obtain higher throughput performance.

On the one hand, if we take cost-proportion fairness into account and allot initial resources for service container, each container have the same probability to obtain more resources to process arrival packet. With the help of backpressure, all of them will keep balance finally and fluctuate slightly. On the other hand, if the service container start from high base, namely owning adequate initial resource, it has the priority during resource allocation, resulting in reduction of available shares of CPU. In addition, compared to HTTPS, HTTP is more lightweight and need fewer computation cost to process packet. Although HTTPS has limited shares to process packet, impacting the overall throughput, the throughput of HTTP service container could offset the part loss and improve the overall performance.

By this way, NTS can adjust the initial shares and priority to generate various proportion hybrid traffic. What's more, it can also preempt the shares for lightweight service emulator to improve throughput. The experiment result show that: NTS design with backpressure and cost-proportion fairness scheduling can support a number of different traffic emulators. It effectively supports heterogeneous emulation application and still provides superior performance.

## 5 Conclusion

In this paper, we have presented a proposal for the generation of network traffic load that behaves like most of the real interactive traffic of the Internet. Such a proposal can improve the development of experimental studies that require network traffic for practitioner in network scope. It enables the definition of virtual testbed by making use of container technologies. Besides, it can be extended easily according to evaluation scenarios and requirement. What's more, our proposal doesn't think about the problem of ethical concerns related to information disclosure and has no impact to online network.

As the key part of our proposal, we adopt the notion of cost-proportion fairness scheduling to improve the performance. It not only takes packet arrival rate into account but also considers processing cost. By carefully tuning the scheduler weight and employing backpressure to efficiently evict excessive load, NTS has substantial improvement in throughput and dramatically reduces overhead. Furthermore, it adopts the redundancy mechanism and pre-allocation to mitigate the fluctuation because of improper allocation of CPU. It also demonstrates how a management framework can efficiently tune the OS scheduler in a relatively simple way to meet our goal.

The technical viability of our proposal is rooted in the feasibility of virtualization. By leveraging existing standard user space primitive provided by OS, namely cgroups, and accompanying virtual file system, rather than modifying scheduler itself, NTS manager could compute the load and assigns the CPU shares with low overhead.

The experiment results shows that: NTS is a lightweight, cost-efficient and extensible network traffic stimulator with affordability for researchers in network area. Next work, we consider the possibility to extend with user behavior simulation to generate synthetic network behavior pattern, which can be used as background traffic in network security.

## References

1. Xia, W., Wen, Y., Foh, C.H., Niyato, D., Xie, H.: A survey on software-defined networking. IEEE Commun. Surv. Tutorials **17**(1), 27–51 (2015)

2. ETSI, N.F.V.: Network functions virtualisation (nfv). Management and Orchestration, vol. 1, V1 (2014)
3. Intel: Data plane development kit (2018)
4. Olsson, R.: Pktgen the Linux packet generator. In: Proceedings of the Linux Symposium, Ottawa, Canada, vol. 2, pp. 11–24 (2005)
5. Goel, U., Wittie, M.P., Claffy, K.C., Le, A.: Survey of end-to-end mobile network measurement testbeds, tools, and services. IEEE Commun. Surv. Tutorials **18**(1), 105–123 (2016)
6. Merkel, D.: Docker: lightweight Linux containers for consistent development and deployment. Linux J. **2014**(239), 2 (2014)
7. Olson, M., Christensen, K., Lee, S., Yun, J.: Hybrid web server: traffic analysis and prototype. In: 2011 IEEE 36th Conference on Local Computer Networks, pp. 131–134. IEEE (2011)
8. Menage, P.: Linux kernel documentation: Cgroups (2017)
9. Yan, J., Jin, D.: Vt-mininet: Virtual-time-enabled mininet for scalable and accurate software-define network emulation. In: Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research, vol. 27. ACM (2015)
10. Molnar, I.: Linux kernel documentation: CFS scheduler design (2017)
11. Kulkarni, S.G., et al.: NFVnice: dynamic backpressure and scheduling for NFV service chains. In: Proceedings of the Conference of the ACM Special Interest Group on Data Communication, pp. 71–84. ACM (2017)
12. Kanhere, S.S., Sethu, H., Parekh, A.B.: Fair and efficient packet scheduling using elastic round robin. IEEE Trans. Parallel Distrib. Syst. **13**(3), 324–336 (2002)
13. Emmerich, P., Gallenmüller, S., Raumer, D., Wohlfart, F., Carle, G.: MoonGen: a scriptable high-speed packet generator. In: Proceedings of the 2015 Internet Measurement Conference, pp. 275–287. ACM (2015)
14. Rotsos, C., Sarrar, N., Uhlig, S., Sherwood, R., Moore, A.W.: OFLOPS: an open framework for openflow switch evaluation. In: Taft, N., Ricciato, F. (eds.) PAM 2012. LNCS, vol. 7192, pp. 85–95. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28537-0_9
15. Muelas, D., Ramos, J., López de Vergara, J.E.: Software-driven definition of virtual testbeds to validate emergent network technologies. Information **9**(2), 45 (2018)
16. Weigle, M.C., Adurthi, P., Hernández-Campos, F., Jeffay, K., Smith, F.D.: Tmix: a tool for generating realistic TCP application workloads in ns-2. ACM SIGCOMM Comput. Commun. Rev. **36**(3), 65–76 (2006)
17. Julián-Moreno, G., Leira, R., de Vergara, J.E.L., Gómez-Arribas, F.J., González, I.: On the feasibility of 40 gbps network data capture and retention with general purpose hardware. In: Proceedings of the 33rd Annual ACM Symposium on Applied Computing, pp. 970–978. ACM (2018)
18. Shalabi, Y., Yan, M., Honarmand, N., Lee, R.B., Torrellas, J.: Record-replay architecture as a general security framework. In: 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 180–193. IEEE (2018)
19. Gad, R., Kappes, M., Mueller-Bady, R., Medina-Bulo, I.: Header field based partitioning of network traffic for distributed packet capturing and processing. In: 2014 IEEE 28th International Conference on Advanced Information Networking and Applications (AINA), pp. 866–874. IEEE (2014)
20. Lee, J., Lee, S., Lee, J., Yi, Y., Park, K.: Flosis: A highly scalable network flow capture system for fast retrieval and storage efficiency. In: USENIX Annual Technical Conference, pp. 445–457 (2015)
21. Frömmgen, A., Stohr, D., Fornoff, J., Effelsberg, W., Buchmann, A.: Capture and replay: reproducible network experiments in mininet. In: Proceedings of the 2016 ACM SIGCOMM Conference, pp. 621–622. ACM (2016)

22. Kim, W., Roopakalu, A., Li, K.Y., Pai, V.S.: Understanding and characterizing planetlab resource usage for federated network testbeds. In: Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference, pp. 515–532. ACM (2011)
23. Aschenbruck, N., Bauer, J., Bieling, J., Bothe, A., Schwamborn, M.: Let's move: adding arbitrary mobility to WSN testbeds. In: 2012 21st International Conference on Computer Communications and Networks (ICCCN), pp. 1–7. IEEE (2012)
24. Nils, A., Jan Bauer, J.B.A.B.M.S.: WSNLab - a security testbed and security architecture for WSNS. In: 2011 IEEE 36th Conference on Local Computer Networks, pp. 4–7. IEEE (2011)
25. Zheng, C., Tang, Q., Lu, Q., Li, J., Zhou, Z., Liu, Q.: Janus: a user-level TCP stack for processing 40 million concurrent TCP connections. In: 2018 IEEE International Conference on Communications (ICC), pp. 1–7. IEEE (2018)