Generating representative, live network traffic out of millions of code repositories

Tobias Bühler
ETH Zürich

Roland Schmid
ETH Zürich

Sandro Lutz
ETH Zürich

Laurent Vanbever
ETH Zürich

ABSTRACT

In theory, any network operator, developer, or vendor should have access to large amounts of live network traffic for testing their solutions. In practice, though, that is not the case. Network actors instead have to use packet traces or synthetic traffic, which is highly suboptimal: today's generated traffic is unrealistic. We propose a system for generating live application traffic leveraging massive codebases such as GitHub.

Our key observation is that many repositories have now become "orchestrable" thanks to the rise of container technologies. To showcase the practicality of the approach, we iterate through >293k GitHub repositories and manage to capture >74k traces containing meaningful and diverse network traffic. Based on this first success, we outline the design of a system, DYNAMO, which analyzes these traces to select and orchestrate open-source projects to automatically generate live application traffic matching a user's specification.

CCS CONCEPTS

• **Networks** → **Network simulations**; **Network experimentation**; *Logical / virtual topologies*;

KEYWORDS

network virtualization, traffic generation, traffic analysis

ACM Reference Format:
Tobias Bühler, Roland Schmid, Sandro Lutz, and Laurent Vanbever. 2022. Generating representative, live network traffic out of millions of code repositories. In *The 21st ACM Workshop on Hot Topics in Networks (HotNets '22), November 14–15, 2022, Austin, TX, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3563766.3564084

Acknowledgments: The research leading to these results was supported by an ERC Starting Grant (SyNET) 851809.

HotNets '22, November 14–15, 2022, Austin, TX, USA

ACM ISBN 978-1-4503-9899-2/22/11…$15.00
https://doi.org/10.1145/3563766.3564084

1 INTRODUCTION

Generating representative network traffic is a fundamental requirement for many network actors including researchers, operators, and vendors. Among others, network actors use traffic generators to test network devices, evaluate network algorithms (e.g., traffic engineering, congestion control, load-balancing, packet scheduling) or service-level agreements.

Problem Generating (large amounts of) representative application traffic is hard. We can distinguish two techniques: *(i)* replaying packet traces [2], possibly collected from a production environment; and *(ii)* generating traffic using a software [4, 8, 12, 19, 21] or hardware-based [20] traffic generator, possibly replicating some traffic features (e.g., packet inter-arrival time). Unfortunately, both techniques fall short when it comes to the representativity of the generated traffic.

While replaying packet traces (e.g., from CAIDA [6] or MAWI [7]) in real-time offers realistic traffic patterns, it does not consider the applications' behavior, meaning one cannot reason about how traffic would behave under different network conditions. Another problem is that publicly available traces exhibit low throughput (few Gbps, at best) preventing their use for, e.g., stress tests. And while replaying them at a faster speed is possible, it also comes at the price of representativity by violating the original traffic characteristics.

Similarly, while using stateful traffic generators allows to reason about end point behaviors under different conditions, they also fail to generate representative application traffic. Indeed, traffic generators either rely on simple "blasting" strategies (e.g., `iperf` [13]) or generate their traffic according to distributions (e.g., using the "web search" or "data mining" workloads from [1]). While doing so ensures that some of the traffic features are respected, it cannot capture complex applications' logic (e.g., client/server behaviors).

To sum up, our research question is : *Can we build a system that can generate large amounts of representative, live network traffic, following a wide variety of application logic?*

Opportunity While we lack access to application traffic data today, there is clearly no shortage of applications' code. In the last decade, the popularity of code sharing platforms like GitHub has skyrocketed. As of 2022, GitHub alone boasts a user base of over 83 million people and over 200 million repositories hosted on their platform [15]. (This massive, open dataset is sometimes referred to as "Big Code" [3].)

Intuitively, a significant amount of these 200 million repositories contain code that—when run and deployed—generates network traffic. We believe this traffic represents a huge and, to the best of our knowledge, completely untapped source of network-related data. "Tapping" into this source seems impossible though: how can we hope to run (let alone compile) arbitrary applications, written in arbitrary languages, and according to arbitrary code practices? Luckily, another recent trend provides an answer to this question: the always-wider adoption of container technologies and orchestration platforms such as Docker Compose [10] and Kubernetes [22]. Among others, these technologies provide a *standardized* way to package, run, and deploy arbitrary software.

Vision Our vision is to fundamentally change the way we generate network traffic by leveraging massive codebases. We envision to build a network traffic generator by orchestrating *real* applications selected from a large-scale dataset. Specifically, we envision building a system which, given a specification on the traffic to generate, automatically: selects relevant applications; deploys them in a network environment (virtual or physical); and orchestrates them so that the resulting traffic complies with the specification.

Challenges Realizing this vision is challenging and requires addressing the following questions:

- *How to find relevant applications in massive codebases?* The first key challenge involves efficiently sifting through massive codebases looking for any "orchestrable" application that generates (non-trivial) network traffic. (Simple exploration strategies are unlikely to work given the large number of repositories.)
- *How to model the traffic generated by each application?* Given an application generating traffic, the next key challenge is to model its network behavior. Unlike image or text data sets with a clear semantic, network behaviors are semantically richer. A challenge is therefore to develop a semantic representation to extract from each application. Finding the right level of abstraction is difficult: a representation which is too specific or fails to model network behavior will lead to inaccurate generated traffic later on.
- *How to select the applications to use to generate traffic?* Once we have a large collection of applications and characterized their network behavior, the next key challenge is how to automatically select the applications to orchestrate in a live environment so that the resulting traffic meets the specification. Doing so will require to find a way to interface the requirements with the semantic representation.
- *How to orchestrate the applications?* Finally, we need to develop a runtime system which can automatically deploy and orchestrate the selected applications in a (provided) live network environment. Doing so is again non-obvious.

First successes In this paper, we take a preliminary step towards building this vision. Besides proposing a design (and an accompanying research agenda), we collect a first large-scale dataset of "orchestrable", traffic-generating applications out of GitHub and showcase its potential for supporting traffic generation tasks. Thus far we found over 433k GitHub repositories containing one or more "containerized" applications and managed to explore around 67% of it as of October 2022. We found 74k orchestration files generating traffic following the default commands. We deployed each of these applications and captured the traffic they generate. Despite being still preliminary, we show that the network traffic we collect is diverse: it covers *many* applications (e.g., web, databases, cryptocurrencies, video streaming) and exhibits many types of workloads (with applications generating >100 Mbps of traffic). To the best of our knowledge, our dataset already constitutes the largest collection of traffic-generating applications to date, and we have released a list publicly [5].

Overall, we believe that leveraging massive codebases for solving networking problems opens up a new and exciting area of research in our community which goes beyond the problem of traffic generation.

2 LIVE TRAFFIC GENERATION

This section describes DYNAMO (*dynamic mass orchestration*), a system for live traffic generation. DYNAMO allows users to test their networks with real application traffic that reacts to the network's performance just as real traffic would. Users describe the traffic in a high-level specification language and DYNAMO automatically orchestrates open-source projects for the live traffic generation. It outputs a set of virtual network interfaces that can be connected to the user's network testbed and populated with the reactive live application traffic on-demand. To that end, DYNAMO operates in three phases:

(i) an *offline phase* that is performed only once for finding, executing and analyzing suitable open-source projects;
(ii) a *bootstrapping phase* for parsing the traffic specification, selecting open-source projects and preparing them as well as the virtual interfaces for traffic generation;
(iii) and a *traffic generation phase*, in which DYNAMO populates the interfaces with the live application traffic that a user can connect to and route through their testbed.

This allows users to dynamically test their network infrastructure, custom forwarding algorithms and the interactions of all network components with real application traffic. Existing solutions, such as statically replaying traffic traces that were recorded in other networks or using artificially generated application traffic, do not show the same benefits.

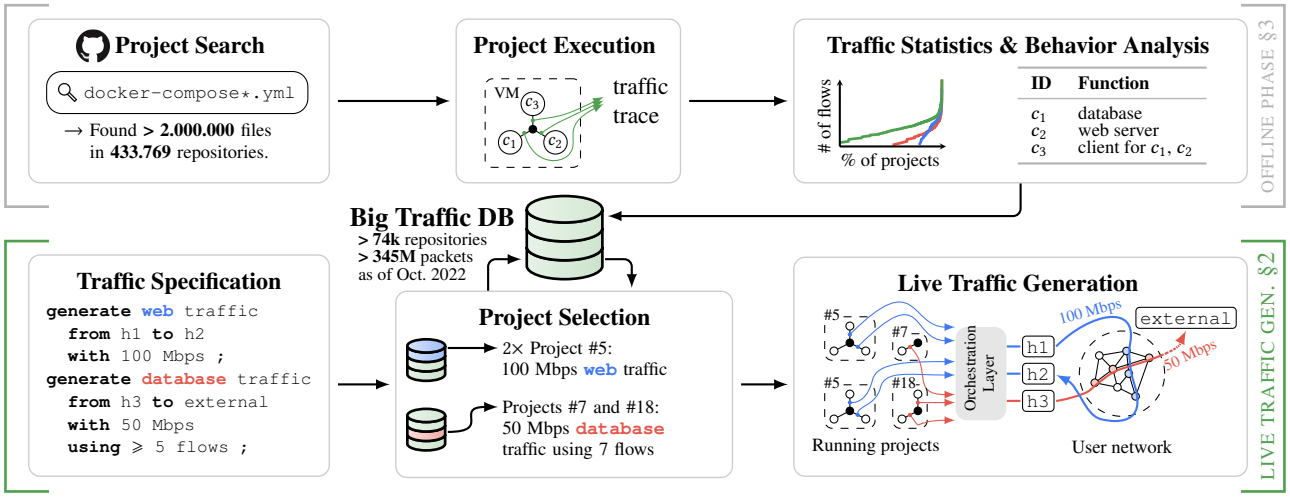

Figure 1: DYNAMO leverages the abundance of open-source projects to build the Big Traffic database. Based on a user's traffic specification, DYNAMO then finds and orchestrates adequate open-source projects for live traffic generation.

2.1 Overview

Figure 1 shows DYNAMO's main components divided into two parts. The upper block depicts the offline phase, which is executed only once. Its goal is to build the *Big Traffic Database*, containing traffic statistics and meta information on each evaluated open-source project's behavior. To obtain this database, we first find publicly available open-source projects which could potentially generate network traffic. We then execute each project individually in an isolated virtual machine and capture all generated traffic. We process the recorded traffic traces in two ways: (i) we extract several traffic features (e.g., application type, number of flows, etc.); and (ii) we analyze the behavior of the open-source project to identify which components send and receive which type of application traffic. See Section 3 for more details.

The lower block outlines DYNAMO's live traffic generation pipeline, which is executed on-demand. In a first step, the user specifies the desired traffic in a high-level traffic specification language. DYNAMO then parses the specification and queries the Big Traffic database to find matching projects. As illustrated, DYNAMO might have to combine multiple instances of the same or different projects to meet each of the user's traffic demands. Next, DYNAMO creates a set of virtual network interfaces that can be connected to the user's test network. Finally, DYNAMO orchestrates the execution of the selected projects to generate traffic as specified.

2.2 System Details

Bootstrapping phase The input to DYNAMO is a traffic specification, that is, a set of statements in the Declarative Traffic Specification Language (DTSL). See Figures 1 and 2 for the

```
stmt ::= generate <type> traffic
           from <host> to <dst>
           with <num> [k|M|G]bps
          [using ( ⩽ | ⩾ ) <num> flows] ;
type ::= <num>|TCP|UDP|database|web|...
host ::= h<num>
dst  ::= <host> | external
num  ::= (1-9)[(0-9)*]
```

Figure 2: A set of statements in Declarative Traffic Specification Language (DTSL) allows to easily, yet flexibly specify what traffic needs to be generated by DYNAMO.

syntax definition and an example. The DTSL parser infers the number of required hosts $h_1, ..., h_{max}$ and DYNAMO accordingly creates a set of virtual network interfaces. A `<type>` selects an application type as provided by the Big Traffic database, or explicitly specifies a desired destination port.

After parsing the input, DYNAMO selects a combination of open-source projects to match the user's traffic demands as closely as possible. To that end, DYNAMO converts the DTSL input into a constrained optimization problem and solves it using a generic solver, e.g., the Gurobi Optimizer [17]. The challenge herein constitutes in matching the high-level traffic specification to features captured in the Big Traffic database.

Finally, DYNAMO prepares the applications for traffic generation by downloading and bootstrapping all the selected open-source projects, as well as their dependencies. For this task, DYNAMO relies on the projects' build automation tools. DYNAMO patches the traffic sources with the virtual network interfaces via an *orchestration layer* and returns the list of virtual interfaces associated with hosts $h_1, ..., h_{max}$, which will

later emit the specified traffic. The user can now connect these interfaces to a physical or virtual network environment.

Orchestration layer DYNAMO's design relies on container-based virtualization for concurrently running the many diverse open-source projects on one or more machines. Each project may (and most likely does) consist of multiple containers that communicate with each other over an integrated network bridge. Using standard networking tools, DYNAMO can thus redirect any traffic between the project containers to an orchestration layer, which connects the containers via the user's network. To that end, the orchestration layer requires access to the projects' behavior (found in DYNAMO's Big Traffic database); e.g., which container initiates (→ **`from`** `<host>`) or merely replies to (→ **`to`** `<dst>`) the traffic. The orchestration layer mainly serves three purposes:

(i) it connects the containers via the user's network on the virtual interfaces following the DTSL specification;
(ii) it implements a bidirectional NAT to mitigate any IP or port conflicts on the source and destination side; and
(iii) it dynamically performs rate- and flow-limiting on the generated traffic to match the DTSL specification.

By default, DYNAMO can interconnect all hosts using a simple bridge, which ensures that the traffic is correctly forwarded between the different components. While this default setup can already be used to generate user-specified static traffic traces, we argue that this may become obsolete with the availability of tools like DYNAMO. As DYNAMO supports live traffic generation using real applications, a user can simply connect the virtual interfaces to an arbitrary virtual or physical network and, for instance, introduce network events such as specific packet loss or congestion.

Traffic generation phase Once the user issues the *'run command'*, DYNAMO populates the generated virtual network interfaces with the desired traffic. To that end, DYNAMO reruns the selected open-source projects and combines their traffic using the orchestration layer (i.e., transparent to the user's network). The main challenge of this phase is to precisely control the open-source projects in order to generate the specified traffic. DYNAMO achieves this by overprovisioning the generated throughput while regulating it using rate-limiting or partial tunneling. For example, the orchestration diverges specific traffic away from the user's network by routing it directly to its intended host. As the open-source projects were not originally designed to generate traffic at a constant rate, DYNAMO uses a sliding window to monitor the generated throughput and to adapt its control mechanism.

Advanced live traffic generation While DYNAMO still presents several technical challenges that are yet to be solved, we want to outline further extensions making DYNAMO even more flexible. For the input, a user could specify more complex traffic behavior patterns, for instance, expected inter-packet arrival times, burstiness of traffic, or even matching custom traffic distributions. In turn, DYNAMO would query the Big Traffic database to identify the best matching set of projects. Another extension would be for DYNAMO to identify separate client/server projects that belong together. We expect that many open-source projects would not simply generate traffic when run in their default configuration, but require a specific counterpart. To find these, we suggest DYNAMO could analyze the repositories' meta information, such as other repositories hosted by the same organization or links to other repositories in the README files.

3 PRELIMINARY EVALUATION

This section describes how we find suitable open-source projects and execute them to generate traces (§3.1). Then we analyze the currently collected traces in detail (§3.2).

3.1 Project Search and Trace Generation

Project search As a first step, we need to find open-source projects which should: (i) be easy to build and run automatically; (ii) have a high chance of generating network traffic that we can record; and (iii) relate to diverse applications such that we end up with an interesting dataset.

We *currently* focus on Docker-based projects to fulfill the first two points. Docker [9] is a lightweight virtualization solution that runs different software applications in "containers". In addition, we look for so-called docker-compose files [10], which are easy to execute and define how multiple containers are connected. Communication between these containers is typically based on network traffic that DYNAMO can capture. Finally, we search through the massive number of GitHub projects to find projects generating diverse traffic.

To do so, we use GitHub's REST API v3 [14] and query for `docker-compose*.yml` to search for projects containing one or more docker-compose files. The API returns all matching projects with some activity in the last year. There are two main challenges. First, GitHub performs rate limiting on the number of queries per minute, which we solve by authenticating the requests with an access token. Second, the API only returns the first 1000 matches. Hence, we partition the search space by requesting specific file sizes. DYNAMO incrementally queries all 100-byte ranges and dynamically decreases (increases) the range if the query results in more (fewer) than 1000 matches. We ignored additional files in case a query for a single file size returned more than 1000 results.

We found more than 433k open-source projects which contain at least one docker-compose file. Our current focus on GitHub and docker-compose files is only one of many possibilities to leverage "Big Code" for network traffic generation.

Table 1: DYNAMO successfully executes docker-compose files of more than 38k GitHub projects. A single project can contain successful and failed docker-compose files.

status	# docker-compose files & projects	
Totally tried	767.622	(293.131 GitHub projects)
Successfully executed	74.448	(38.871 GitHub projects)
Failed to execute	693.174	(266.348 GitHub projects)

Table 2: The ten most-observed destination ports belong to diverse applications such as web, database, or Bitcoin.

port	description	port	description
5001	iPerf and IPFS	8333	Bitcoin traffic
443	HTTPS traffic	55606	Dynamic/private port
5672	RabbitMQ traffic	27017	MongoDB traffic
4001	IPFS (file system)	6099	Selenium (browser tool)
80	HTTP traffic	1433	MySQL traffic

In the future, we plan to extend the search to other sources (e.g., GitLab [16]) and other build systems (e.g., the platforms' continuous integration pipelines, automatic test frameworks, or VM automation tools such as Vagrant [18]).

Project execution In a second step, we try to execute all docker-compose files in the previously found projects. For safety reasons, we execute all docker-compose files in VMs and restrict the direct access to our local network. However, an Internet connection is required to provision and run most projects. Running multiple projects in parallel on the same VM results in undesirable outcomes (e.g., due to resource sharing). Therefore, we execute each docker-compose file sequentially and deploy multiple VMs in parallel.

We face two main problems: (i) rate limiting on Docker Hub [11]; and (ii) how to execute the docker-compose files. We circumvent the first problem by creating multiple paid Docker Hub accounts (i.e., higher rate limits) and deploying a local container cache, which helps if containers are used by multiple projects. Even so, our requests were occasionally rate-limited. For the second problem, we run each compose file with the default command: `docker-compose up`. At first, we provide the `--no-start` option, thus creating and provisioning all containers without starting them. This allows DYNAMO to identify all interfaces to capture traffic on.

We distribute the 433k found projects over all VMs after shuffling them to prevent artifacts due to their ordering according to docker-compose file sizes. At the point of writing, we had six VMs (four cores & 16 GB memory each) nearly constantly running for around nine months. We already tested more than 767k docker-compose files, which belong to 293k projects, i.e., we processed around 67% of all found projects.

Table 1 shows that we successfully executed 74k (ca. 10%) of the 767k docker-compose files. While that might sound low at first, DYNAMO is merely executing the default run command and relies on the abundance of available open-source projects: There are always more candidates, especially if we extend DYNAMO to other project sources.

The reasons for failed executions range from bugs in the project's source code to no longer available containers and insufficient VM resources. However, one main problem is that the default command is not sufficient to execute all compose files. We plan to improve that by parsing project-related documentation, e.g., often available READMEs, for custom docker-compose run commands. A preliminary analysis of 1k random projects shows potential. By simply "grepping" for `docker-compose` in the main README, we found run commands adding flags, custom build targets or input files. We envision that DYNAMO builds a set of run commands for each compose file and iterates through all of them.

Traffic collection In a third step, DYNAMO captures traffic on all identified interfaces. This includes all traffic *between* containers and traffic towards/from external destinations. As most containers run for a long time, it is challenging to identify *when to stop* the capturing process. We also observe that traffic often appears shortly after startup and during teardown, which leads to two termination conditions: (i) if we capture no new packets in two consecutive ten-second slices; and (ii) if we reach a maximum capturing time of ten minutes. In the future, we will adapt these conditions to the project behavior and, e.g., extend the ten-minute limit if a project continues to send "interesting" traffic (not just keep-alive messages).

To perform the capturing process, we use `tshark` [23] and only store the first 96 bytes per packet. This covers the Ethernet, IP, and transport headers, even with some optional fields. Note that users see the entire packets as DYNAMO reruns the projects during the live traffic generation (see §2.2).

3.2 Dataset Analysis

DYNAMO automatically generates metadata for each captured trace, such as packet counts or application types (based on port numbers). In addition, we compute the two main metrics used by DYNAMO's DTSL, namely the average traffic throughput and the number of flows per trace.

Diversity of Traces We first analyze the different application types for which we observe packets in DYNAMO's Big Traffic database. Table 2 shows the ten most-observed destination ports over *all* our traces. The table shows traffic for "common" web applications (HTTP/HTTPS) and a lot of database, Bitcoin, or message/file exchange traffic. These ten applications cover roughly 33% of all collected packets. Other interesting

Table 3: The top 5 distinct applications generate 100s of Mbps of throughput in the first ten minutes (cut-off time).

order	#pkts/flows	throughput	short description
top 5 pkts	25.343.285	45 Mbps	.NET HTTP stress tests
	16.228.707	356 Mbps	network speed test
	15.952.659	19 Mbps	message scheduling platform
	13.011.157	417 Mbps	multi-paxos school project
	11.897.994	31 Mbps	integration tests
top 5 flows	367.559	4 Mbps	Telegram messenger proxy
	141.351	80 Mbps	pcap generator: "IHULK" DoS
	139.836	270 kbps	load testing suite
	112.964	6 Mbps	Internet commerce database
	103.992	4 Mbps	distributed key-value store tests

applications observed in the remaining 67% are DNS (often short flows), other blockchain technologies (e.g., Ethereum), remote server access (ssh, telnet, FTP), or BitTorrent.

Selected trace examples Table 3 shows the five distinct docker-compose files which generated the most packets and flows, respectively. These traces contain multiple millions of packets or more than 100k flows. In the short descriptions (manually labeled), we identify network-related projects such as a HTTP stress test, but also arbitrary applications such as integration tests for which it is not immediately apparent that we would observe network traffic. This manifests the application diversity in the Big Traffic database allowing DYNAMO to find matching projects for various user queries.

We also want to highlight the speed test in the second row of Table 3. It connects to an external server via a shared 1 Gbps link, potentially limiting its throughput. In addition, DYNAMO stopped the capturing process for nine of the ten projects, including the speed test, preemptively (after ten minutes). By dynamically adapting the termination conditions as outlined before, we expect to obtain even bigger traces.

Per-trace statistics The left plot in Figure 3 shows the number of TCP and UDP flows per compose file which generated at least one flow identified by its five-tuple (src/dst IP and port & protocol number). Most of these traces (around 89%) contain at least one TCP flow, a smaller number (around 16%) at least one UDP flow. In the median case, there are around ten flows. Although this number is relatively small, DYNAMO leverages the massive amount of available compose files. We can always combine projects or duplicate them to fulfill the user requirements during the live traffic generation (see §2.2).

The observed IPs reveal around 38% of compose files with at least one external flow, meaning that the source or destination IP was public. This indicates communication with destinations outside of the VM. However, some docker-compose files also build networks that contain public IPs *locally*.

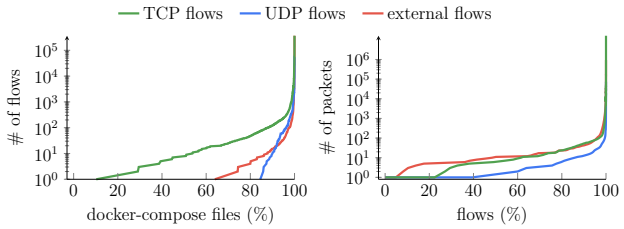

Figure 3: Half of the applications generate more than ten flows, while the median flow size is around ten packets.

Flow sizes The right plot in Figure 3 shows the flow sizes, i.e., number of packets per flow. First, note that we observe flows that only contain a single packet. This might hint that some of the docker containers could not reach all of their destinations, or that the containers generated attack traffic (e.g., TCP SYN flood attack). In the median case, we have around ten packets per flow. This is expected as a lot of our traffic relates to databases or similar applications, which often produce short request/reply style flows. However, we also observe very long flows which contain millions of packets. Overall, UDP flows contain the least amount of packets as they often perform short DNS queries in our dataset.

Behavior analysis During the last part of DYNAMO's offline phase, we analyze the behavior for each container in a project. To obtain fine-grained control of the containers for DYNAMO's live traffic generation phase, we must know which type of traffic uses which interface. We envision that DYNAMO could learn such a mapping either during the capturing process or later via the collected MAC addresses. Next, DYNAMO parses the docker-compose file to determine which interfaces belong to which container, as there is not always only a single interface per container. Combining these insights with other behavioral patterns, e.g., which container initiated most connections, allows DYNAMO to identify "client" and "server" containers. This opens completely new scaling options to reach users' traffic demands. Instead of combining or duplicating projects, DYNAMO could also spin up additional client containers to communicate with existing server(s).

4 CONCLUSION

In this paper, we presented our vision of DYNAMO, a live application traffic generator that leverages the abundance of available open-source projects and learns their traffic characteristics. Given a user's traffic specification, DYNAMO queries its Big Traffic database to select the best-matching projects and orchestrates them to generate live traffic. Our preliminary analysis of >293k repositories shows that our vision is feasible: many repositories generate network traffic.

REFERENCES

[1] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. pFabric: Minimal near-Optimal Datacenter Transport. *ACM SIGCOMM Computer Communication Review* (2013).

[2] AppNeta. 2022. Tcpreplay - Pcap editing and replaying utilities. (Aug. 2022). Retrieved 2022-10-11 from https://tcpreplay.appneta.com/

[3] Pavol Bielik, Veselin Raychev, and Martin Vechev. 2015. Programming with "Big Code": Lessons, techniques and applications. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*.

[4] Alessio Botta, Alberto Dainotti, and Antonio Pescapè. 2012. A tool for the generation of realistic network workload for emerging networking scenarios. *Computer Networks* (2012).

[5] Tobias Bühler, Roland Schmid, Sandro Lutz, and Laurent Vanbever. 2022. Dataset of Automatically Orchestrable GitHub Projects. (Oct. 2022). https://doi.org/10.5281/zenodo.7194189

[6] CAIDA. 2019. The CAIDA UCSD Anonymized Internet Traces. (Dec. 2019). Retrieved 2022-10-11 from https://www.caida.org/catalog/datasets/passive_dataset

[7] Kenjiro Cho. 2022. MAWI Working Group Traffic Archive. (Oct. 2022). Retrieved 2022-10-11 from https://mawi.wide.ad.jp/mawi/

[8] Cisco Systems. 2022. TRex - Realistic Traffic Generator. (Oct. 2022). Retrieved 2022-10-11 from https://trex-tgn.cisco.com/

[9] Docker Inc. 2022. Docker. (Oct. 2022). Retrieved 2022-10-11 from https://www.docker.com/

[10] Docker Inc. 2022. docker-compose documentation. (Oct. 2022). Retrieved 2022-10-11 from https://docs.docker.com/compose/

[11] Docker Inc. 2022. Docker Hub Container Library. (Oct. 2022). Retrieved 2022-10-11 from https://hub.docker.com/

[12] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. 2015. MoonGen: A Scriptable High-Speed Packet Generator. In *Internet Measurement Conference 2015 (IMC'15)*. ACM.

[13] ESnet. 2022. iperf3: A TCP, UDP, and SCTP network bandwidth measurement tool. (Oct. 2022). Retrieved 2022-10-11 from https://github.com/esnet/iperf

[14] GitHub Inc. 2022. GitHub REST API. (Oct. 2022). Retrieved 2022-10-11 from https://docs.github.com/en/rest

[15] GitHub inc. 2022. Where the world builds software. (Oct. 2022). Retrieved 2022-10-11 from https://github.com/

[16] GitLab B.V. 2022. The One DevOps Platform. (Oct. 2022). Retrieved 2022-10-11 from https://about.gitlab.com/

[17] Gurobi Optimization, LLC. 2022. Gurobi Optimizer Reference Manual. (Oct. 2022). Retrieved 2022-10-11 from https://www.gurobi.com

[18] HashiCorp. 2022. Vagrant. (Oct. 2022). Retrieved 2022-10-11 from https://www.vagrantup.com/

[19] Juniper Networks. 2020. WARP17, The Stateful Traffic Generator. (Sept. 2020). Retrieved 2022-10-11 from https://github.com/Juniper/warp17

[20] Spirent Communications. 2022. Spirent TestCenter Hardware. (Oct. 2022). Retrieved 2022-10-11 from https://www.spirent.com/products/testcenter-hardware

[21] Srivats P. 2022. Ostinato. (Aug. 2022). Retrieved 2022-10-11 from https://ostinato.org/

[22] The Kubernetes Authors. 2022. Kubernetes Documentation. (July 2022). Retrieved 2022-10-11 from https://kubernetes.io/docs/concepts/overview/

[23] Wireshark group. 2022. tshark manual page. (Oct. 2022). Retrieved 2022-10-11 from https://www.wireshark.org/docs/man-pages/tshark.html