Techreport ISI-TR-699
LegoTG: Composable Traffic Generation with a Custom Blueprint

Jelena Mirkovic*†, Genevieve Bartlett†
*University of Southern California, †Information Sciences Institute

ABSTRACT

Traffic generation is critically needed in networking research for experimentation, stress-testing and validation. Traffic generation is not just creation and sending of packets, but often includes routing of generated traffic, its modulation in network via drops and delays, and its consumption at the receiver. Current approaches to traffic generation all suffer from two major drawbacks: (1) A fixed traffic blueprint—a fixed set of traffic features and their models, which cannot satisfy a broad range of user needs. (2) A monolithic, large code base, which users find very hard to change.

In this paper we explore the problem "how to design a traffic generator that can meet a broad range of needs". We decompose traffic generation into modular parts, identify a basic set of modules which can satisfy a broad range of needs and realize all these functionalities in our LegoTG framework. We demonstrate the composability, versatility and ease of use of our framework. We further show how this narrow waist of traffic generation can be used to reproduce the functionalities of existing traffic generators, and go well beyond them.

1. INTRODUCTION

Network researchers often need to generate network packets in testbed experiments for problem modeling and system testing. Traffic generation needs are very diverse and closely depend on the details of the experiment and the researcher's goals. One researcher may want to generate many small packets at a high rate to test a router, another may want realistic and responsive TCP flows to test a middlebox, yet another may want a specific volume and content of HTTP GET messages to test a Web service. Traffic generation is not just the creation of packets, but also routing of generated traffic, traffic modulation via drops and delays, and proper consumption at the receiver. Even a simple replay of packets via tcpreplay [1] between just **two** machines requires many functionalities such as link-header rewriting, routing and IP aliasing (or address rewriting) for addresses in the replayed trace, and a sink to consume packets at the receiver. In this paper we explore the following research challenge: How do we design and implement traffic generation that (1) meets many diverse, existing needs, and (2) can be easily extended to meet new needs.

To date, the complex process of traffic generation has been explored by many (e.g. iPerf [2], Tcpreplay [1], Harpoon [3], Swing [4], D-ITG [5]). We see two main drawbacks with existing solutions:

Fixed traffic blueprints: We define a "traffic blueprint" as the (1) selection of traffic features that matter for a given experiment (e.g., packet rate, congestion responsiveness), and (2) how these features are modeled and initialized (e.g., Harpoon [3] models each TCP flow as a single file transfer, and can initialize models from Netflow traces). A traffic blueprint must be realized in the experiment, through generation and consumption of packets and emulation of network features such as delay, drop and jitter. Current approaches each define and implement a single traffic blueprint, limiting the set of traffic features a user can tune, fixing how these features are modeled, and tying the realization of traffic generation to this fixed model. Users must modify existing generator code to explore new blueprints, or write their own generators.

Monolithic code base: In real networks many entities interact to produce traffic. Traffic generators subsume this complexity into a single code base that realizes multiple functionalities, e.g., packet generation/consumption, IP virtualization, etc. This creates large, monolithic code and makes modifying or extending existing generators an arduous task.

We make the following **research contributions**:

1. We demonstrate the *need for custom traffic blueprints* based on a survey of current research practices. We decompose traffic generation into orthogonal tiers: traffic feature selection, feature models, and traffic realization. We then divide realization into separate functionalities, so that each functionality can be achieved by a stand-alone customizable component which we call a "TGblock". This modularization facilitates high code reuse, as researchers can combine TGblocks from different traffic generators, modify, replace and add TGblocks to fit their needs.

2. We propose a *narrow waist of traffic generation*—the basic set of functionalities that can satisfy many researcher needs, and that can combine to reproduce and go beyond the functionalities of existing traffic generators.

3. We *propose a framework called LegoTG*—for composable traffic generation with custom blueprints. Our framework facilitates easy reconfiguration, sharing and customization of traffic generation in testbed experiments, and easy adoption of new code by users.

We further address engineering challenges of implementing a framework which has: (1) self-contained traffic generation, (2) the ability to incorporate third-party traffic generators, and (3) portability to different experimentation platforms. Our **engineering contributions** lie in identification of these functionalities, their implementation in LegoTG, and their deployment and evaluation on the DeterLab testbed [6, 7]. LegoTG is free and publicly available under GPL [8] from our project Web site [9].

2. TRAFFIC GENERATION

Network traffic is the product of a complex set of interac-

tions between multiple entities at the physical, link, network, transport, application and user layers. We say that a traffic generator "operates at a given layer" if it exposes parameters to control traffic features at that layer. There are many ways a researcher may want to model and control these features—modeling some in detail, while ignoring or simplifying others. In this section we survey researchers' traffic generation needs, and discuss how we can meet these needs through decomposition of the traffic generation process. We focus on traffic generation in network testbeds and live networks.

2.1 Needs

To quantify researcher needs, we survey work published in Sigcomm 2013 and NSDI 2013, and note the usage of traffic generation tools that create network packets in testbeds or in live networks. Results are shown in Table 1. Around half of the papers use traffic generation and a majority of those (65–69%) use simple, custom generators written by the authors. The rest use existing, third-party, traffic generators. Out of these, near half use network-level traffic generators such as iPerf [2] and netperf [10], while the other half uses application-level traffic generators such as Selenium [11] and web-page-replay [12]. The majority of papers use traffic generators to stress-test their proposed system, and only a few use them to replay or generate realistic network traffic.

Need 1: Diversity and multiple layers. Our survey illustrates several points: (1) Researchers need a variety of traffic generation tools at different layers of the network stack. (2) Traffic generation in research papers is often simple. We believe this happens because simple tools like iPerf [2] are easy to understand and deploy. But if systems are not tested with realistic traffic, which is often very diverse and has unpredictable behaviors, this may lead to incorrect research outcomes [13]. (3) Custom tools are used by the majority of researchers (65–69%)! We believe this is because researchers need features that existing tools do not support, and changing existing tools is difficult, so researchers develop their own. Custom traffic generation hinders repeatability as tools are often not portable or released publicly. We conclude that researchers need *customizable* traffic generation, with generators that operate at *multiple layers of the network stack*.

Need 2: Modularity and customizability. Can diverse traffic generation be achieved with a single generator? One could design a generator that offers a feature superset of current generators, and allows users to control these features via a very long list of options. But, such a generator would be overly complex and require a large code base, making porting, installation and use challenging. We conclude that researchers need *modular* traffic generation, where functionalities are clearly separated, module code is small and modules are easily customized, added or replaced.

Need 3: Composability. Within even a single paper, the needs for traffic generation can be quite varied. For example, three papers from Sigcomm'13 and one from NSDI'13 use multiple traffic generators to meet their needs. Because traffic generator models and components differ widely (see Table 2), switching between generators is challenging and time consuming. Further, sometimes a researcher may need to combine multiple traffic generators in a single experiment. For example, she may need realistic and responsive TCP traffic (e.g., using Swing [4]) coupled with realistic UDP traffic (e.g., using tcpreplay [1]). Such a combination is very challenging today and requires a large amount of manual work to deploy and synchronize multiple traffic generators. We conclude that researchers need *composable* traffic generation, which allows for easy switching between and combining of traffic models or generators.

Traffic generator	SIGCOMM'13	NSDI'13
Some traf.gen.	17	16
Custom	11	11
iPerf, netperf	4	3
tpc-w, httperf, ClassBench	1	1
Selenium, web-page-replay	1	1
Commercial	1	0
Total	39	36

Table 1: Traffic generator use in Sigcomm'13 / NSDI'13

2.2 Novel Traffic Generation Paradigm

To address the needs we identified, we decompose the traffic generation process into multiple tiers and functionalities. First, we break traffic generation into three tiers, to decouple the goal of traffic generation from its realization:

Tier 1: Traffic features that matter in a particular network experiment, and consist of a subset of: (1) network, transport and application header fields, (2) traffic content, (3) and interactions of generator components between each other and with other facets of the environment.

Tier 2: Feature models specify how each traffic feature is parameterized and initialized, e.g., from a traffic trace (to achieve realism) or using user-specified values (to achieve controllability). Together Tier 1 and 2 define a traffic blueprint.

Tier 3: Realizers are pieces of software that interact together to realize the traffic blueprint. They create and consume the desired packets, modulate packets in the network (e.g. adding delays) and route packets on the experiment topology.

Decisions at higher tiers affect the lower tiers but not vice versa. For example, it is possible to change a feature model without changing the set of features that matter (e.g., replace a single-valued parameter with a distributional one), and it is possible to replace one realizer with another one without changing the feature model (e.g., use iPerf [2] to generate a given traffic load instead of netcat [14]).

There are many realizers. We group them into the following functionality classes:

Class 1: Feature extraction. If a researcher needs realistic traffic generation, parameter values for feature models need to be extracted from traffic traces. We sub-divide this functionality into **cleaning** (removing traffic from traces that

Decomposition	iPerf	D-ITG	Harpoon	Swing
Traffic features	one-way flow: dur., rate, proto. dst port tcp/OS options	one-way flow: dur., rate # pkts and # bytes DS flags, TTL, proto src/ dst IP/port pkt size and freq.	one-way flow: rate total: vol/interv. src/dst distrb/interv.	total: app. mix flow dynamics host delays congest. response
Feature models	single-value user spec.	single-value or distribution user spec.	src-dst pair: sessions session: flows, inter-flow delay flows: file transfer extr. from traces	app: sources source: sessions, inter-sess. delay session: RREs, inter-RRE delay RRE: conns, inter-conn delay conn: reqs, resps, inter-req delay extr. from traces
Realizers	Packet gener. Packet consum. iPerf(269K)	Packet gener. ITGSend(155K) Packet consum. ITGRcv(122K)	Feat. ext. harpoon_flowproc(48K) Packet gener. and consum. harpoon(153K)	Feat. ext, Packet gener. and consum. Swing (420K) IP virt. and Network emul. ModelNet (234K)

Table 2: Decomposition of several popular traffic generators

packet generators or testbed environments cannot realize, e.g., traffic to reserved IPs), **filtering** (removing traffic that cannot be modeled with chosen models) and **extraction** (extracting parameter values).

Class 2: Feature modulation. Some traffic features may need to be modulated before generation to meet experiment needs. For example, rewriting IPs or ports, increasing connection duration, dropping connections with low volume, etc.

Class 3: IP virtualization. Traffic generation often involves multiple senders and receivers. If the features-of-interest include source and destination IPs, a researcher will need to virtualize multiple IP addresses on a single physical machine during experimentation. We subdivide virtualization into **mapping** (deciding which physical machine virtualizes which IP), **aliasing** (setting up virtualized IPs) and **routing** of virtualized IPs.

Class 4: Packet generation. This functionality builds the actual packets, according to the model.

Class 5: Packet consumption. Some traffic generation tools produce one-way flows (e.g., tcpreplay [1], D-ITG [5], iperf [2]). These tools need packet consumption at traffic destinations, to avoid generating ICMP service unreachable or TCP RST messages.

Class 6: Traffic modulation. As traffic travels through the experiment network it can be modulated to introduce delays, drops and jitter, and to filter or modify packets. We subdivide traffic modulation into **network emulation**, **traffic filtering** and **traffic rewriting**.

Class 7: Synchronization. This functionality synchronizes certain actions among experiment nodes and/or different software modules, when needed.

Fig. 1 illustrates our decomposition into tiers and functionalities. Table 2 applies this decomposition to several popular traffic generators. We make the following observations from results in Table 2: (1) Current generators differ widely with regard to the traffic models they use, their parameterization and initialization, and their complexity, (2) Generator code is monolithic and large, offering multiple intertwined functionalities. We list these functionalities and the number of lines of code required for each in the last row of Table 2.

We propose a novel traffic generation paradigm where generation tiers and realizer functionalities are clearly separated, and implemented through stand-alone software modules. This enables composition and easy replacement of one module with another. Separation also keeps each module's code small, enabling easy customization. Modular software is not new, but our vision for traffic generation goes well beyond just modularization. Our traffic generation modules—*TGblocks*—are stand-alone and interchangeable, much like Lego™ blocks are, allowing for combination and exchange of blocks contributed by multiple authors. This leads to composable, customizable traffic generation.

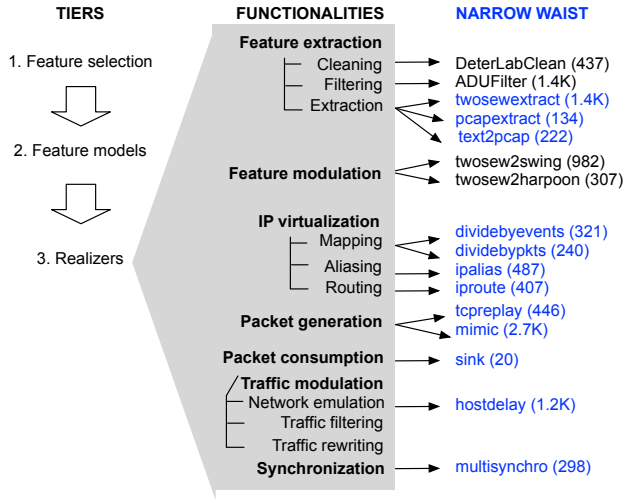


Figure 1: Decomposition of traffic generation into tiers and functionalities, and basic set, implemented by us.

3. NARROW WAIST

We use our decomposition into tiers to identify those elements at each tier that meet the needs of the majority of researchers. We further focus on how to realize these elements to provide a base for easy additions and customizations. This

leads us to a basic set of functionalities—the narrow waist of traffic generation—which we realize through TGblocks, described in Sec. 5.1.

3.1 Tiers and Functionalities

We make three observations which, in turn, define the scope and core of the narrow waist, and define how modular pieces communicate with each other to allow for easy customization and addition.

Scope: Network and Transport Layer. In research publications the most common traffic features of interest are at the network and transport layers, including TCP responsiveness to packet loss and delay and network emulation of drops and delays. Far less common is traffic generation at the application layer, where a multitude of tools exist for server benchmarking or traffic capture and replay. We thus focus in this paper at the network and transport layers only. This scope informs what functionalities we need at each tier.

Tier 1 (Traffic features): We need features in the IP and transport header fields, and a two-way TCP flow abstraction with responsiveness to packet drops and delays.

Tier 2 (Models): Most commonly used models in research literature for the features we selected in Tier 1 are: (1) packet-based models, with user-supplied fields (eg. [2, 5]), and (2) two-way, *application data unit* (ADU) based, flow models for congestion responsive traffic [4, 15]. We support these two models in our narrow waist. Other models can easily be added to extend the narrow waist (See Sec. 5.2).

Tier 3 (Realizers): We need only a handful of realizers to support the narrow waist. At the core is packet generation (class 4) at the packet and the TCP flow level. Packet consumption (class 5) is needed for the packet-based model only, and network emulation (class 6) is needed for the flow-based model. Feature extraction (class 1) is needed to support replay at the packet and flow level. IP virtualization (class 3) is needed to support multiple source and destination IPs on a single virtual machine. Lastly, tight synchronization (class 7) is needed to ensure all physical machines start traffic generation at the same time.

Core: Model-based Generation via Replay. Traffic generation tools typically perform replay-based or model-based generation—where the former replays deterministically from a traffic log, and the latter generates traffic based on a stochastic model. We observe that we can realize model-based generation by first letting the model produce a "script" of the desired behavior or traffic, and then feeding this output to a replay-based traffic generator. Thus we include only replay-based packet generators in the narrow-waist.

Universal Communication: Text files, packet/flow data. To support easy, human-readable communication between modeling tools and replay tools, we use textual and not binary "scripts" that are produced by modeling tools and consumed by replay tools. In this format, each packet or flow is described separately. This allows for easy debugging, extension and manipulation of a modeling tool's output by popular text-manipulation tools like grep, sed and awk. If disk space is tight, text files compress well when there is high regularity, e.g., if most packets/flows resemble each other, which is often the case in experiments.

3.2 Traffic Models

We support two traffic models in our narrow waist. Our **packet-based model** uses a subset of fields used by tcpreplay to describe each packet: time of packet's generation, all IP header fields (no options) and all transport header fields (TCP, UDP or ICMP, no options). Our **two-way, ADU-based flow model** describes each TCP flow as a series of ADU events. We start from models used in [4, 15], and modify and extend them to support a wider variety of flow dynamics. Each flow is equivalent to one TCP connection, initiated by a client with a server. Clients and servers send ADUs to each other. Each ADU is a chunk of data sent by the application in a single message to the transport layer, which may break the ADU into several packets. Either party may send an ADU based on internal application triggers (e.g., an FTP client has read a new chunk of data from disk and sends it to the server) or based on receipt and processing of an ADU sent by the other party (e.g., a Web server sends back a page after receiving and processing a user's request). We support both these dynamics with two types of events: SEND events, which record generation of data from one party to another, and WAIT events, which record waiting for a given amount of data to arrive from a party, before proceeding with events. We call our model *TwoSew*: Two-way SEND-WAIT model.

In TwoSew, each event has the following fields: (1) *actor*: the IP address of the party performing the event, (2) *eventtype*: SEND or WAIT, (3) *bytes*: the number of bytes to send or to wait for, (4) *twait*: the time to wait. For SEND events, *twait* is the time measured from the previous SEND by the same party, and it mimics waiting for application triggers. For WAIT events, *twait* is the time measured from when the party receives total of *bytes* from another party, and it mimics the processing time of ADUs and user think time.

TwoSew's event structure is versatile enough to support a variety of scenarios: (1) request-response—where one party sends a request and waits for a response from another party before proceeding, (2) parallel sends—where two machines are sending data to each other simultaneously and (3) a mixture thereof. TwoSew supports pauses between the connection start and the first ADU by using a non-zero $wtime$ value in the first SEND event. TwoSew can also support pauses between the connection's last ADU and its termination (via FIN or RST) by inserting a WAIT event for the client and the server at the end with non-zero $wtime$. All these scenarios are well-represented in real network traces.

TwoSew is more expressive than previous models. For example Swing's [4] model cannot support communications with parallel sends, and Tmix's [15] model cannot support the mixture of request-response and parallel sends on the same

connection. Further, neither model can support the ADU processing time at the server, and the delays at the start and the end of a connection.

4. THE LEGOTG FRAMEWORK

In this section we provide the details of our LegoTG framework and discuss how we meet our identified needs of modularity, customizability and composability. LegoTG consists of *TGblocks* that wrap various realizers, and an *Orchestrator* that combines and coordinates these blocks. TGblocks can combine in various ways to achieve differing traffic generation goals. Each block can be run independently, which allows for easy testing, modification and replacement.

The Orchestrator is a command-line tool responsible for deploying, driving and monitoring TGblocks. Using an *experiment configuration file*—called an *ExFile*, the Orchestrator identifies which blocks to use and how, where and when to deploy these blocks. The Orchestrator runs on a single control node, and uses SSH to coordinate with remote machines.

Self-contained traffic generation. The Orchestrator performs all the actions required to install, set up, run, stop and test each TGblock. It also keeps track of dependencies between blocks and propagates outputs of some blocks into inputs of other blocks (even across different physical nodes). Thus the Orchestrator has full control over the entire traffic generation process, allowing for self-contained generation.

Working with existing and new tools. Assimilating a tool into LegoTG is a simple process. One defines a wrapper—called a *block interface file* or *BIF*—that contains the details on how to install, set up, run, stop and test this tool. Together, a BIF and the tool that is wrapped comprise a TGblock. BIF functions use the Orchestrator's API to specify interactions with remote machines. A BIF is *tool specific*, not experiment specific, so a BIF need only be written once for each tool integrated into LegoTG. This allows for easy portability of new tools between experiments, and between users. BIFs are small and easy to write, as we discuss in Sec. 4.2. Excerpts from a sample BIF which wraps tcpreplay [1] are shown in Fig. 3. The entire tcpreplay BIF is 194 lines and includes rewriting link-layer headers and padding any truncated packets, to create valid packets that can be sent out.

Portability to different platforms. LegoTG and TGblocks are written in Python, which is supported on a wide range platforms. LegoTG uses SSH for orchestration of experiments which is supported in many testbeds, and live networks.

4.1 The Orchestrator

Fig. 2 depicts the Orchestrator running a simple replay experiment, called TcpReplay, in a typical set up—a testbed, with a researcher controlling the experiment from her laptop using the Orchestrator. The experiment uses three nodes: a node running tcpreplay [1] (n1) to replay traffic from a file, a sink node (n2) to consume this traffic, and a node running tcpdump [16] (r) to capture replayed traffic for analysis. If

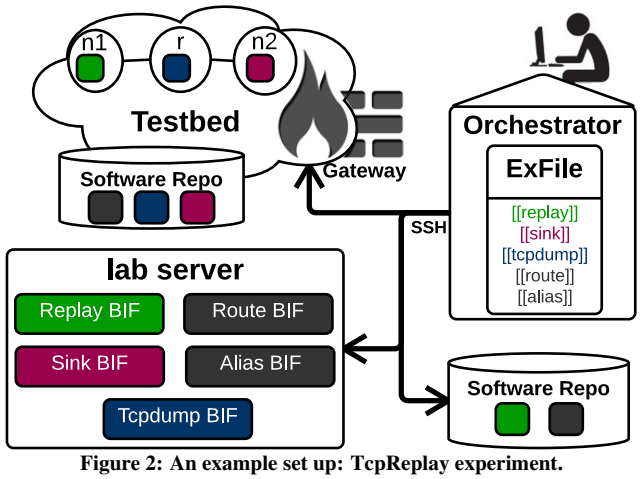


Figure 2: An example set up: TcpReplay experiment.

BIF Example: tcpreplay

```
class blockInterface():
 # Attributes filled in by Orchestrator at runtime:
 HOST_NAME = ''
 HOST_OS = ''
...
 # Attributes which can be configured from an ExFile
 replay_file = ''
 option_string = ''
...
 def install(self):
  o = OrchestratorFunctions.OrchestratorFunctions()
  if 'ubuntu' in self.HOST_OS:
   result = o.run("sudo apt-get install tcpreplay")
...
 def start(self):
...
  print("Starting replay on %s:%s"%(o.hostname(),iface))
  command ="sudo tcpreplay %s -i %s %s"
          %(self.option_string,iface,self.replay_file))
  result = o.run_long(command)
...
 def stop(self):
...
  result = o.run("sudo kill -INT %s" % pid)
```

Figure 3: An example BIF

a user wants to replay previously captured traffic without modifying its IP addresses, we also need virtualized IPs (IP aliasing) on traffic sources and destinations that are routable on all nodes.

The Orchestrator runs locally on the experimenter's laptop, and reads the local ExFile. The ExFile includes information on which blocks are needed, where to find the BIFs for these blocks, (e.g. in Fig. 2 from a remote machine), and where to find the software for these blocks (e.g. in Fig. 2 from two different software repositories—one local to the testbed, one not). The Orchestrator pulls the BIFs, and determines how to install the needed software (e.g. tcpreplay). Once installation is done, the Orchestrator calls setup, test and run for each of the required blocks, configuring software according to the options set in the ExFile. The Orchestrator works to run as many tasks as possible in parallel across the machines in an experiment, while ensuring that dependent blocks and their actions are run after all dependencies have completed. While the Orchestrator performs coordination between blocks, tight synchronization between TGblocks is achieved through a separate synchronization block (see Sec. 5.1).

4.2 BIF: Block Interface File

Each BIF contains a simple Python class. The functions of

this class are run by the Orchestrator at the appropriate times and on the appropriate remote machines during an experiment. The standard set of functions in the BIF are install, setup, test, start and stop. Depending on the tool being wrapped, some, none or all of these functions may be defined. Users can also add new functions to a BIF, and execute them by calling them from an ExFile, or from the Orchestrator's command-line interface. Attributes listed in a BIF are set at runtime and can be configured from the ExFile to customize how a block is set up and run on each machine.

Fig. 3 shows excerpts of a BIF which wraps tcpreplay [1] to replay raw packets. The BIF exposes several configurable attributes, which can be set in an experimenter's ExFile (eg. the replay file name "replay_file"). BIFs make use of a set of Orchestrator Functions which implement common tasks such as `run_long()`, a non-blocking call which runs a command in a detached remote shell, or `run()`, a blocking call to run the command remotely, and return results. These functions streamline creating new BIFs for tool integration.

4.3 ExFiles: Experiment Configuration

The ExFile captures all the details of an experiment: where software repositories are, where to install and how to configure these software tools, how tools are synchronized, and all the inputs and outputs. This single point of configuration enables easy prototyping, modifying and sharing of traffic generation processes. Fig. 4 shows an example ExFile for the TcpReplay experiment (Fig. 2) which uses three testbed nodes, located behind a gateway, and fetches BIFs from a remote server. Sections in an ExFile are denoted with single brackets ([section]), and nested subsections use an increasing number of brackets (eg. [[subsection]]).

4.3.1 Parts

The ExFile consists of five main parts: (1) **globals** (lines 1–4): definition of static variables. (2) **logging** (lines 5–7): optional log targets and settings. (3) **hosts** (starting line 8): list of hosts in the experiment that may run a TGblock, or serve content needed by a TGblock. This section contains any non-standard details about how a host should be accessed via SSH (such as via a gateway), and any preferred environment details for a host, e.g., which perl installation to use. (4) **groups** (lines 16–20): list of hosts in various functionality groups, e.g., traffic sources, traffic sinks, routers, etc. (5) **sections**: list of phases that comprise the experiment. Each section consists of subsections.

The example file contains three sections: "extraction" (line 21), "resource allocation" (line 23) and "experiment" (line 33). A section contains information on what blocks to run, and in which order. For example, line 34 instructs the Orchestrator to execute the blocks defined "alias", "route", "sink", "trace" and "replay" subsections, in that order, during the "experiment" section. Each block is described in a subsection (e.g. [[alias]]), which specifies how to run the block: its target (eg. line 37), the definition for the block (path to its BIF file e.g.

ExFile Example: TcpReplay Experiment

```
1  part_allocation=/home/msmith/allocation.txt
2  biflib=bifserver:/user/share/bif/
3  tracedir=/home/msmith/traces/
4  remote_repo = someserver:/user/share/software
5  [logging]
6     log_file = /home/msmith/logs/replay.log
7     log_level = 5
8  [hosts]
9     user = msmith
10    [[testbed.net]]
11    gateway = gate.testbed.net
12    nodes = n1, r, n2
13    [[somewhere.net]]
14    nodes = someserver.subdomain, bifserver
15    key_filename=~msmith/.ssh/bifkey
16 [groups]
17    replay_grp = n1
18    tcpdump_grp = r
19    sink_grp = n2
20    all_testbed = replay_grp, trace_grp, sink_grp
21 [extraction]
22 ...
23 [resource allocation]
24 ...
25    [[divide]]
26       target = local
27       def = $biflib/alias/BIF.py
28       [[[input]]]
29          trace_file = original_trace.dump
30       [[[output]]]
31         part_allocation = $part_allocation
32 ...
33 [experiment]
34 order = alias, route, sink, trace, replay
35 actions = install, setup, run
36    [[alias]]
37       target = all
38       def = $biflib/alias/BIF.py
39       [[[input]]]
40          config = $part_allocation
41       [[[args]]]
42          script = $remote_repo/alias/alias.sh
43    [[route]]
44       target = all
45 ...
46    [[sink]]
47       target = sink_grp
48 ...
49    [[replay]]
50       target = replay_grp
51       def = $biflib/tcpreplay/BIF.py
52       [[[args]]]
53       replay_file = $tracedir/replay.pcap
54 ...
55    [[trace]]
56       target = trace_grp
57       force_quit_during = start
58       def = $biflib/tcpdump/
59       [[[output]]]
60              dumpfile = %host%-%iface%.dump
61       [[[args]]]
62       auto_determine_iface = True
```

Figure 4: An example ExFile

line 38), and its inputs, outputs and arguments (details in Sec. 4.3.3). The subsection's name need not reflect the name of its associated TGblock.

4.3.2 Targets and Groups

A target of a TGblock is one or more *groups*—a collection of hosts the TGblock will run on. Groups are specified in the [groups] section, and provide an easy way for users to port ExFiles and change the roles of experiment nodes. Group definitions can be nested and a host can belong to multiple groups or to none.

4.3.3 TGBlock Attributes

A TGblock's attributes are specified in three optional subsections [[[input]]] , [[[output]]] and [[[args]]]. The Orchestrator will use attributes in the [[[input]]] and [[[output]]] sections to determine dependency between blocks, and attempt to move files between machines as needed. For example, if the section for a trace block lists "output1.pcap" under [[[output]]], the Orchestrator would look in sections for other blocks to see if any listed "output1.pcap" in their [[[input]]] section. If so, the Orchestrator would check that this file existed and was moved to the machine(s) that should run the dependent block, before it executes the block's run function. All other configurable options of a tool wrapped by a BIF, which do not represent inputs and outputs for a TGblock go into the [[[args]]] section.

4.3.4 Variable Expansion

The Orchestrator supports two types of variable expansion: static expansion of variables local to the ExFile and runtime expansion of variables that are calculated by the Orchestrator each time a TGblock is run on a host. Static expansion (variables starting with $) is useful in porting ExFiles between environments and making ExFiles more readable. For example, a user could specify `biflib=bifserver:/user/share/bif/` at the start of the ExFile, and use `$biflib` throughout the rest of the file. Runtime expansion (variables enclosed by `%`'s) is handy to specify variables, which change based on the target. For example, in Fig. 4, line 60, the output file for the trace block (`%host%-%iface%.dump`) will be replaced by the host name and network interface name for each host/interface pair this block is run on. This runtime expansion is handled by the Orchestrator based on a dictionary which matches a variable name with a function that returns its expansion. The Orchestrator includes a basic set of runtime expansion functions. Users can extend this set by providing a dictionary mapping variable names to custom expansion functions. Variable expansion and abstraction of hosts into functional groups facilitate easy porting of ExFiles.

4.3.5 An Orchestrator Run

At the command line, a user can specify the section (e.g. extraction) the Orchestrator starts in and optionally, a subset of functions to run (e.g. install) from that section. The "stop" function of each TGblock is called when a user cancels execution or if the `force_quit_during` variable (eg. line 57) is used. This variable specifies a function that once completed by all other TGblocks, triggers calling the stop function for the given TGblock. In Fig. 2, after all the TGblocks finish their "start" function, the Orchestrator will call the "stop" function for the trace TGblock.

5. IMPLEMENTED TGBLOCKS

Here we provide details about TGblocks we implemented: (1) to support the narrow waist of traffic generation (Sec. 3), and (2) to demonstrate the versatility and extensibility of our narrow waist functionalities. These blocks are summarized in Fig. 1. Some blocks contain only a BIF, wrapping an existing tool, (e.g. tcpreplay [1]) while others contain both a BIF and a tool built by us (e.g. mimic).

TGblocks in our narrow waist aim to support a large diversity of researcher needs with a small, modular code base. Each TGblock contains a small number of lines of code (a few hundred to a few thousand as shown in Fig. 1 in parentheses). Comparing the line count of our TGblocks and the count of state-of-the-art traffic generators (shown in Fig. 2), our code-base is two orders of magnitude smaller. This smaller size enables easy customization by users, when needed.

Though our code in our narrow waist is designed to be small, this small set of TGblocks may not suffice for some researchers. For example, a researcher may want to use a packet generation tool that is faster or more familiar to her than our tcpreplay block. Lego TG allows for easy addition of new blocks, and easy composition of blocks. Thus, a researcher can mix and match tools that we provide, with tools of her choice, to best meet her needs.

5.1 Blocks for the Narrow Waist

Blocks supporting the narrow waist of traffic generation are shown blue in Fig. 1. At the core of our narrow waist are two **packet generators**, tcpreplay for raw packet replay and mimic for TCP responsive replay, and one **packet consumer**, called sink.

Tcpreplay wraps the popular tcpreplay tool [1] that replays previously captured traffic. Tcpreplay reads packets from a libpcap file, and then builds and sends raw packets. While this works for experiments on a single machine, packets generated this way cannot be routed (because they have a wrong MAC address in the link header and may be truncated during capture). Further, these packets need to be consumed at the receiver, to avoid generation of TCP RST or ICMP service unreachable responses. Our tcpreplay block determines the appropriate MAC addresses for the experiment, then rewrites MAC addresses, pads truncated packets, and marks replayed packets (using the TOS field) so that the sink block at the receiver identifies and consumes only these packets.

Mimic wraps a tool we built of the same name, for congestion-responsive traffic replay. Mimic re-enacts TCP communications between many hosts, by reading the record of each connection from an input file, opening TCP sockets, con-

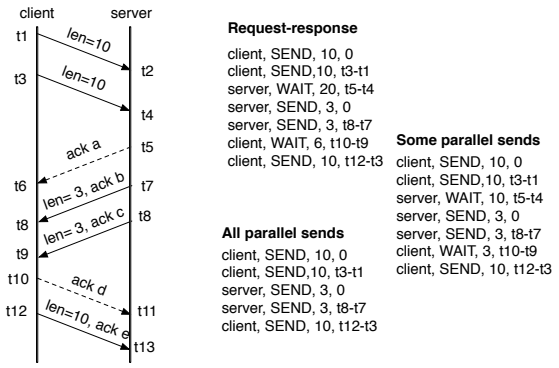


Figure 5: An example client-server communication

necting clients to servers, and handling the ADUs on each connection (performing SEND and WAIT actions). We run one mimic process per physical machine.

Extraction is supported through the twosew2mimic, pcapextract and text2pcap blocks. The twosew2mimic block wraps the same-named tool we designed, which extracts features used in our TwoSew model (Sec. 3.2) from libpcap traces and outputs files that are used by Mimic. Fig. 5 illustrates twosew2mimic's identification of events from traces. In this example, the client sends two 10-byte packets, the server sends an ACK, and then sends two 3-byte packets. Finally, the client sends 10 bytes to the server. Several scenarios are possible depending on the values of a–e in the acknowledgments:

Request-response. If $a = b = c = 20$ and $d = e = 6$ the client has sent a request to the server. The server waited for the full request (20B), processed it and sent a reply. The client waited for the full reply (6B), processed it and sent another request.

Some parallel sends. If $a < 20$ or $d < 6$ the corresponding SEND events occur *in parallel* with SEND events from the other party. We show an example where $a = 10, b = 20, c = 20$ and $d = 3, e = 6$ in Fig. 5. While the second SEND event from the server was acknowledging 20 bytes from the client, note that we do not generate a corresponding WAIT event. This is because we only view lone ACK packets as an indication of waiting for a transmission from the other party. ACKs appended to data packets are considered opportunistic—data has arrived from the other party and is being acknowledged, but such data was not crucial for the current packet emission.

All parallel sends. If there were no lone ACK packets in our example (shown in dashed lines) we would say that all SEND events from the client occur in parallel with SEND events from the server.

Twosew2mimic extracts TCP traffic for replay, and keeps the original IP addresses and ports. It ignores transport-level retransmissions and hardware duplicates. A connection in twosew2mimic is defined by the tuple of source and destination IPs and ports, as well as a start and end time. The first packet seen on a connection defines the start and the last packet or a final handshake is the end. Further, twosew2mimic gathers RTT samples between each original packet sent and its first acknowledgment, for each source IP. It exports the mean and the minimum of these into a file used by hostdelay for network emulation. A user can easily modify any of these design decisions in twosew2mimic by changing only a handful of lines.

Pcapextract wraps a tool we designed of the same name, which extracts traffic from a libpcap file to a textual format. The text2pcap block wraps the same-named tool, which translates a textual representation of packets to libpcap format. This allows users to very easily build custom packets with desired features, by writing tools that produce textual output, which is then converted into libpcap format and replayed using the tcpreplay block.

IP mapping is supported through the dividebyevents and dividebypkts blocks. Each block wraps the same-named tool, which maps IPs in an input file to physical hosts in the experiment, and balances either send/receive events (dividebyevents) or packets (dividebypkts) per host.

Aliasing & routing are supported by the ipailas and iproute blocks, which wrap scripts with standard UNIX commands to set up IP aliases and static routes for virtualization.

Network emulation is performed through the hostdelay block, which emulates propagation delay per source IP, and is implemented using the Click software router [17] [1].

Synchronization of TGblocks is performed through the multisynchro block. Such synchronization may be necessary, for example, when starting generators on different nodes at the same time. A tool may load large input from disk into memory *before* beginning generation, and a user will want to synchronize the generation *after* input loading on each node is complete. TGblocks, when configured by the ExFile to synchronize, make a blocking call to a specified synchronization tool before starting such a synchronized task. This call returns when all hosts in a group have been synchronized. A synchronization block is responsible for installing this tool and disseminating group information.

Our multisynchro block starts a master that waits for a ready signal from each node in a synchronization group. The master sends out announcements via IP multicast. When ready, nodes check for these announcements, and then send ready signals to the master via TCP. Once all nodes are confirmed as ready, the master sends a multicast count-down to a synchronized "go time". This synchronizing method scales easily on private network testbeds, e.g. Emulab [18] or DeterLab [6] (see Sec. 6.5). In environments where multicast is not available (e.g. PlanetLab [19]) a researcher could write and use a different synchronizing block based on an alternate method (e.g. NTP [20]).

5.2 Extending the Narrow Waist

We have developed two TGblocks to show how easily LegoTG's narrow waist can be used to implement different traffic models. These blocks, twosew2swing and twosew2harpoon

[1] Implementation of drops and jitter will be added soon

convert the output of twosew2mimic to encode traffic models used by the Swing [4] and the Harpoon [3] traffic generators, respectively. The output produced from these blocks causes mimic to generate the same traffic that Swing/Harpoon would generate. This illustrates the versatility of our narrow waist, since with just around 1,000 lines of code in these blocks, we can replicate functionalities of two third-party traffic generators with differing traffic models.

Swing [4] views the data exchange shown in Fig. 5, as a single two-way flow, and models the sizes of requests and responses on the flow, and inter-request delays. Swing's model assumes that the server has no significant processing delay. In our example, there would be 20B and 10B request samples, 6B and 0B for response size samples, and $t10 - t0$ for inter-request delay samples. Swing's model further groups connections into RREs, and RREs into sessions based on timing, as described in [4], and extracts several parameters for each identified connection, RRE and session, as summarized in Table 2. Swing's traffic generator then draws values from this empirical distribution to produce statistically similar traffic in experiments. Twosew2swing encodes Swing's traffic model using TwoSew's SEND and WAIT events by: (1) extracting the same parameters and values as Swing does from twosew2mimic's output, (2) drawing at random from these, as Swing does, to produce new connections and request-response exchanges, and then (3) for each request/response, if the request is non-zero Twosew2swing generates a SEND event with the $twait$ being the time of the last SEND event from the same party plus inter-request delay. For the non-zero response, twosew2swing generates a WAIT event for the request bytes, followed by a SEND event with zero $twait$.

Harpoon views the data exchange shown in Fig. 5 as two one-way flows, and only models the file sizes (amount of data sent on the flow) and inter-flow delays. twosew2harpoon also starts from twosew2mimic's output, compressing all SEND events from the client and the server. Using the example in Fig. 5, this model would arrive at two file sizes—30B and 6B—and use $t2$–$t1$ as the inter-flow delay. To generate Harpoon's models twosew2harpoon draws from distributions of total data sent on TCP connections and inter-connection delays, as suggested in [3]. It generates two events for each connection—SEND at the client, and WAIT at the server.

5.3 Additional TGBlocks

There are two additional utility blocks that we have developed: DeterLabClean which removes packets from libpcap traces that use IPs reserved on DeterLab [6, 7], which is our platform for LegoTG experiments, and ADUFilter, which removes all non-TCP traffic, and TCP traffic from connections that do not satisfy some user specified inclusion criteria.

6. EVALUATION

In this section we use a series of experiments to demonstrate LegoTG's capabilities: easy configuration and modification, combination of functionalities, and easy realization of new traffic models with the same code. All our experiments are performed on the DeterLab testbed [6]. For simplicity, we use a single topology with 13 physical nodes (Fig. 6) for all experiments, to highlight changes between traffic generation settings in the same environment. We use a trace from the MAWI traffic repository [21] collected on March 1st, 2012 at 2pm, in some of our experiments. This trace contains 15 minutes of traffic collected on a trans-Pacific link between Japan and US. We replay only 1 minute of traffic from this trace so we can highlight small-scale details. Other traces and longer durations can be as easily replayed. Table 4 shows some statistics about our chosen trace in the first row.

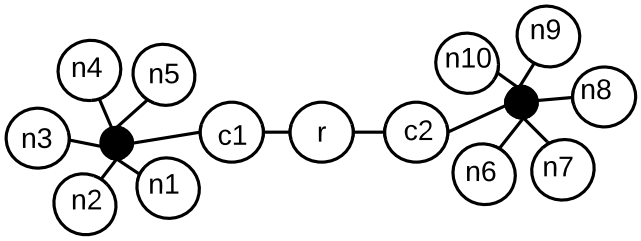


Figure 6: Experimental topology

NetworkReplay: TCPReplay's modifications

```
...
[hosts]
   ...
   nodes = n1,n2,n3,n4,n5,r,n6,n7,n8,n9,n10
...
[groups]
   replay_grp =  n1,n2,n3,n4,n5,n6,n7,n8,n9,n10
   sink_grp = replay_grp
   ...
order =alias,route,sink,trace,multisynchro,replay
...
   [[replay]]
...
   [[multisynchro]]
      .. target, def, args  cut for space ..
```

Figure 7: TcpReplay → NetworkReplay

6.1 Network Replay

We start with a simple experiment that replays traffic from the MAWI trace on our topology. Table 3 shows in the first column the TGblocks that need to be deployed and their

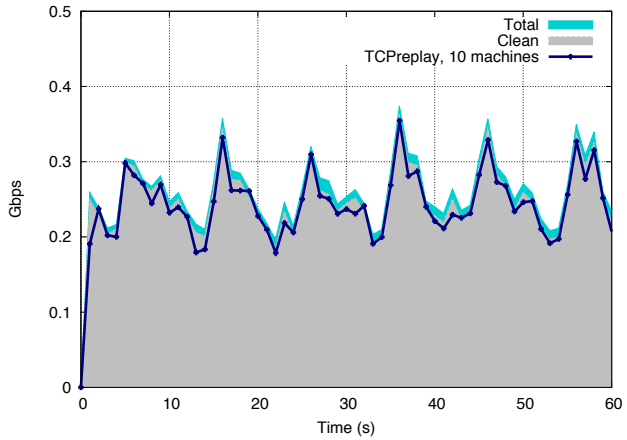


Figure 8: Replayed traffic vs. Original Trace

Experiment	Blocks and targets
Network replay	ipalias, tcpreplay, sink, multisynchro (n1–n10), route (all), DeterLabclean (any)
Transport replay	ipalias, mimic (n1–n10), route (all), DeterLabclean, ADUfilter, twosew2mimic (any), host delay (c1, c2)
Network+Transport replay	ipalias, sink, tcpreplay, mimic, multisyncrho (n1–n10), route (all), DeterLabclean, ADUfilter, twosew2mimic (any)
Swing	ipalias, mimic (n1–n10), route (all), DeterLabclean, ADUfilter, twosew2mimic, twosew2swing (any), hostdelay (c1, c2)
Harpoon	ipalias, mimic (n1–n10), route (all), DeterLabclean, ADUfilter, twosew2mimic, twosew2harpoon (any)
D-ITG	ipalias, tcpreplay, sink, multisynchro (n1–n10), route (all), text2pcap (any)

Table 3: TGblocks and their deployment

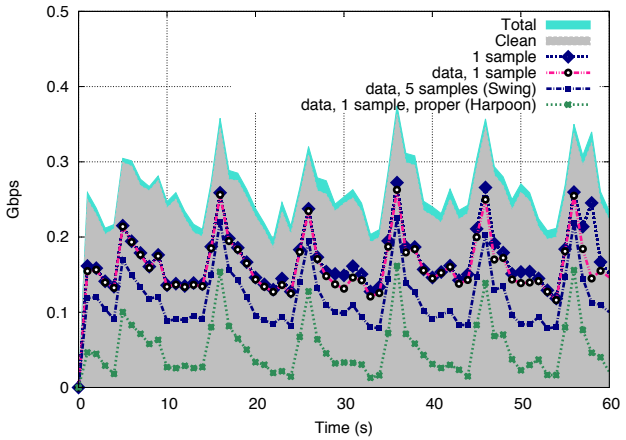


Figure 9: Bandwidth after applying a traffic filter

locations. Even this simple experiment requires complex functionalities to make generated traffic flow in the network. Packets need to be padded—since the original trace has truncated packets—and rewritten to adjust MAC addresses and fix checksums. Aliasing and routing must be set up to allow generated packets to reach their destinations. Sinks need to be set up to consume traffic. Lastly, tcpreplay processes need to be tightly synchronized so generation across machines starts at the same time. LegoTG does all this seamlessly. We evolve our sample ExFile from Fig. 4 into the ExFile for this experiment by only modifying the hosts and group sections (for the larger topology), and adding multisynchro block. These changes are shown in Fig. 7. Aggregated replayed traffic from ten separate machines is shown in Fig. 8: the replayed traffic (solid line) matches well the original traffic (gray area).

filter	pkts (M)	GB	IPs (K)	IP pairs (K)	conns (K)
none	3	2	137.6	145.2	-
clean	2.68	1.95	125.9	133	-
tcp	2.35	1.89	89	91	-
1 sample	1.54	1.31	5.3	5.5	24.8
data, 1 sample	1.46	1.26	3.9	4.4	9.2
data, 5 samples	1.1	0.9	1.9	2.4	6.9
data, proper	0.46	0.4	1.9	1.8	5.1

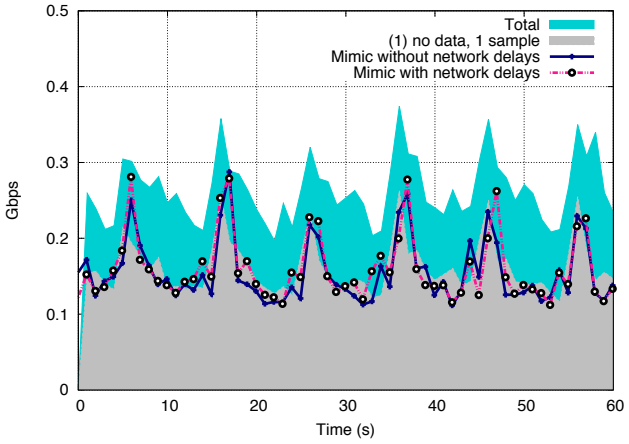
Table 4: Some statistics about our chosen MAWI trace

6.2 Transport Replay

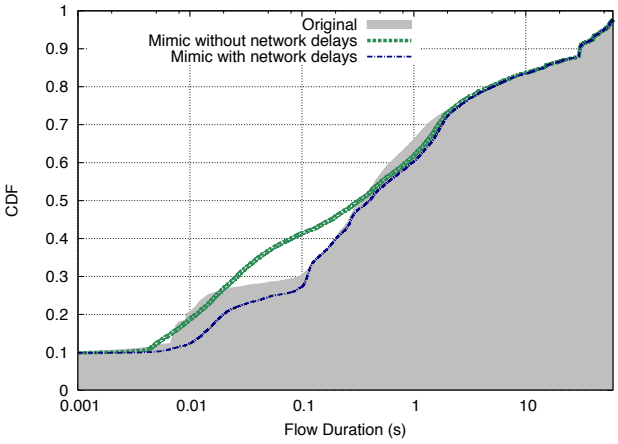
We now evolve our experiment to perform transport-level, congestion responsive replay with mimic. Existing tools for transport layer fidelity, such as Swing [4], Tmix [15] and Harpoon [3], extract model parameters only from TCP connections that meet certain requirements. To illustrate how little of a realistic trace meets these requirements we apply the ADUFilter block to our trace, to obtain four separate traces of TCP traffic, containing only the connections where: (1) both source and destination have at least one RTT sample (*1-sample*), (2) same as (1) and there is at least one data packet exchanged on the connection (*data, 1-sample*), (3) same as (2) but at least 5 RTT samples are required for each host (*data, 5-samples*), (4) there is at least one data packet exchanged and the TCP connections start and end with a proper 3-way handshake (*data, proper*). Swing's extraction requires conditions in filter #3 [4] and Harpoon's requires conditions from filter #4 [3]. Statistics for these filters are given in Table 4 and traffic per second in the resulting traces is shown in Fig. 9. All filters only leave a small fraction of the IPs and pairs communicating in the trace—less than 4% of the original pairs. Filters #3 and #4 additionally drop much of the bandwidth exchanged in the trace. For example, Swing's filter drops more than half of the packets and bytes, and Harpoon's filter drops more than 80%. We emphasize that this loss of fidelity may be acceptable for a wide range of experiments, but the fact that these filters have very different and large fidelity losses across different traffic features supports our argument for customizable (so researchers can try novel models) and composable (so researchers can combine models) traffic generation.

Fig. 11 shows required modifications to the ExFile to evolve the previous network replay into transport replay. We remove the sink group since the sink's functionality is provided by mimic, and introduce a hostdelay block and group for network delay emulation. We configure the hostdelay block with the location of the source code for Click and our custom Click element, and give this block twosew2mimic's output containing the delays inferred from the trace, and the assignment of virtual IPs to physical hosts.

In the experiment, we use "data, 1 sample" constraint for the ADUfilter, run this trace through twosew2mimic, and replay congestion-responsive traffic with and without network emulation. Replayed traffic, shown in Fig. 10(a) matches very well the traffic from the trace, regardless of the presence of network emulation. This is because durations of most high-volume connections in the trace are dominated by client/server delays and not by network delays. To illustrate this, we look at the connection durations with and without network delays. Fig. 10(b) shows the distribution of connection durations in replayed connections, with and without network emulation, compared to durations in the original trace. Connection durations above half a second are dominated by think times (client and server delays between and during data exchanges) and match well the original trace, with and without network emulation. Shorter connections require



(a) Traffic replayed



(b) Connection durations

Figure 10: Traffic replayed by mimic and connection durations

```
TransportReplay: NetworkReplay's modifications
[groups]
...
  sink_grp = replay_grp
  hd_grp = c1, c2
...
[extraction]
   [[twosew2mimic]]
        .. target, def, args cut for space ..
...
order = alias,route,sink,trace,hostdelay,
                multisynchro,mimic,replay
...
   [[hostdelay]]
       .. target, def, args  cut for space ..

   [[mimic]]
      .. target, def, args  cut for space ..
```

Figure 11: NetworkReplay → TransportReplay.

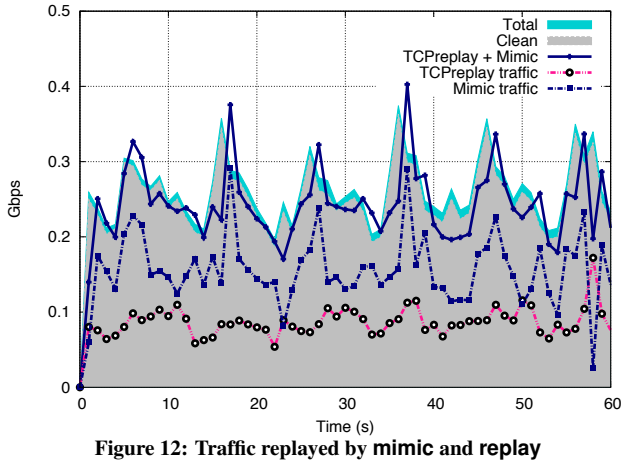


Figure 12: Traffic replayed by mimic and replay

network emulation for a good match. There are two areas of discrepancy—10–100 ms, and 0.5–5 s. Both of these occur due to a large number of events per second handled by each mimic process, and disappear when we use more machines.

6.3 Network + Transport Replay

Transport-level models impose restrictions on the TCP connections they model, as illustrated in the previous section. Compared to the original trace, these restrictions greatly reduce the volume of packets and bytes per second in the experiment, as well as the diversity of IP addresses in traffic. Some researchers may want to replicate both the responsiveness of traffic as well as the traffic volume and IP diversity in the original trace. This requires a combination of network and transport replay, and their synchronization. Without LegoTG this coordination requires great manual effort.

With LegoTG, we can run such a combination easily. To evolve TransportReplay into TransportPlusNetwork replay we return the sink and the replay blocks. Additionally, we preprocess the traffic trace into two traces: one containing traffic that meets "data, 1 sample", which serves as input to mimic, and the second containing the rest of the traffic, which serves as input to tcpreplay. The aggregate traffic replayed by mimic and replay blocks is shown in Fig. 12 (solid line); we also indicate the portions of traffic handled by each block. The aggregate replayed is nearly identical to the original traffic.

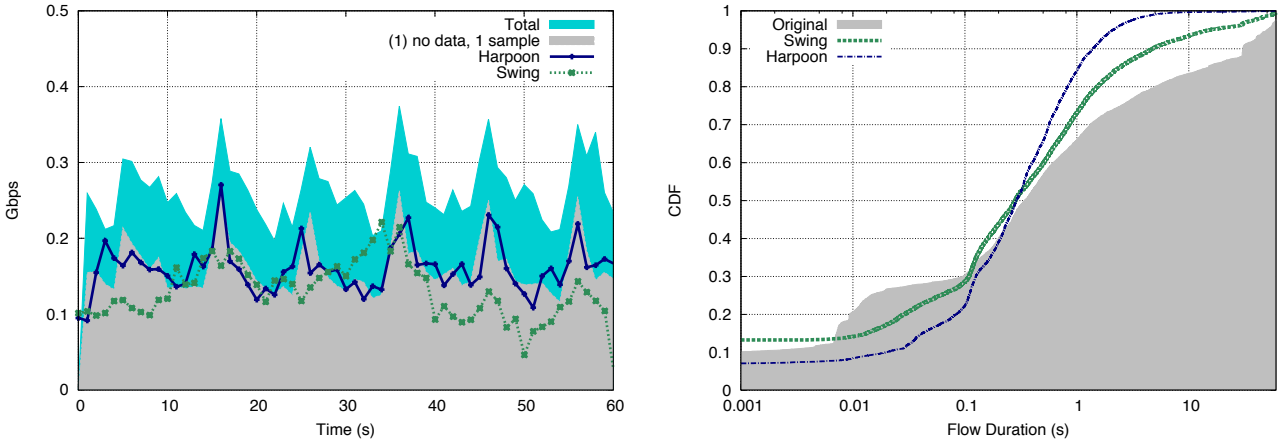
6.4 Model-Palooza

We now demonstrate LegoTG's narrow waist, which allows us to realize various traffic models with the same ADU structure, and with the same realizators.

For example, to run an experiment with the Swing's traffic model we only need to modify the extraction section to add the twosew2swing block. Similarly, for Harpoon, we modify the extraction to add twosew2harpoon. In the Harpoon experiment we also take out hostdelay, since the Harpoon traffic generator does not include network emulation.

Fig. 13(a) shows the traffic filtered by the "data, 1 sample" constraint, as modeled by Swing and Harpoon, and replayed by mimic. Swing is a stochastic traffic generator and does not aim to exactly match the original trace [4]. Harpoon, on the other hand, aims to match traffic in the input trace over coarse time intervals [3] and this is demonstrated in Fig. 13.

Fig. 13(b) shows the CDF of connection durations in experiments with Harpoon and Swing. Neither of these traffic



(a) Traffic replayed (b) Connection durations

Figure 13: Traffic replayed by Harpoon and Swing and connection durations

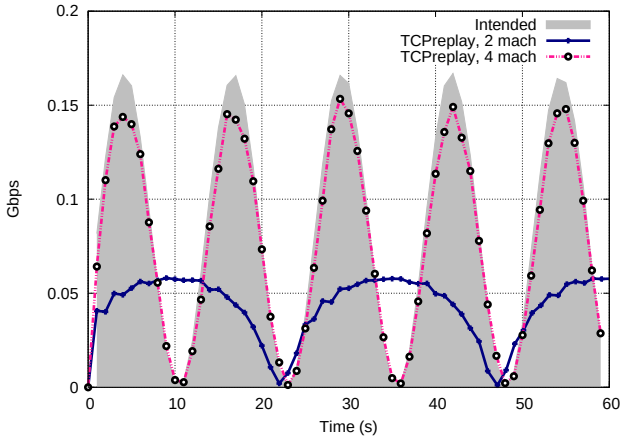


Figure 14: Intended and replayed traffic with "wavy" pattern: first attempt creates too high ops for 2-machine replay.

models matches well the original distribution of connection duration. Swing does not reproduce connections that last more than 1s with high fidelity, because Swing does not model server delays. Harpoon's connections are all very short (85% are shorter than 1s, versus 65% in the original trace) as Harpoon only includes one data exchange from client to server. We emphasize that these discrepancies may be acceptable in a large number of experiments, and that the choice of the model depends on the experimentation goal. LegoTG allows researchers to easily experiment with different models to identify the one that best meets their needs.

D-ITG [5] offers many options for traffic flow configurations, as listed in Table 2. We can achieve these same functionalities in LegoTG by generating a text file with packet descriptions, and using text2pcap to produce input to the NetworkReplay experiment.

But, we can go well beyond D-ITG's functionality! For example, the researcher can control the traffic rate to achieve any desired shape, and she can modify any network or transport level fields to any desired value. To demonstrate this we write a simple program that generates a text file describing communication between 10 sources and 10 destinations, using multiple flows. Each flow sends packets with lengths drawn at random from [10–1,000] byte range and send times drawn from exponential ($\lambda = 5$ ms) distribution. Flow durations follow Paretto distribution with $\alpha = 1$ and $x_m = 1$. We choose connection start and end times and the number of flows to achieve a "wavy" pattern of traffic over 60 seconds with wave interval of 12 seconds. The traffic from the created libpcap trace and the replayed traffic are shown in Fig. 14.

Separation of model-based generation from the trace-based packet generation allows for quick discovery and diagnostics of any causes for mismatch between intended and generated traffic. These could be due to model implementation errors or testbed limitations. For example, in Fig. 14, we show the wavy pattern from the created trace, and its replay on two and four machines. The intended traffic has too-high a packet rate to be replayed by two machines but four machines achieve the desired fidelity. Deploying four (or more) machines requires a single-line change in the ExFile.

6.5 Scalability

Scalability of traffic generation with LegoTG depends on two factors: (1) the scalability of tools used in TGblocks and (2) the scalability of LegoTG's Orchestrator.

Our narrow-waist TGblocks use many existing tools, such as tcpreplay and Click, and scale as well as these tools. Performance of our Mimic tool for congestion-responsive traffic replay is limited by the number of per-second send and receive operations that can be supported on a physical machine that deploys the tool, and by the number of simultaneous TCP sockets that can be opened by the operating system. On mid-line hardware—Dual Xeon with 2G of memory—Mimic can handle up to 500 (SEND/WAIT) events per second before fidelity degrades. But LegoTG's separation of models from realization, and our ExFiles, make diagnosis of scalability issues and scaling up of the traffic generation process to more

physical machines very easy.

We tested the ability of our synchronization tool used in our multisynchro block to synchronize 50, 100, and 500 processes. All processes start within 4–8 milliseconds of each other regardless of the number of processes being synced, but the time required to register all processes as ready increases linearly with process count (up to 30 seconds for 500 processes). We believe the vast majority of testbed experiments will need to synchronize far fewer than 500 processes. Researchers requiring higher-fidelity synchronization can replace our multisynchro block with a more capable block.

LegoTG relies on its Orchestrator to coordinate experiments. Orchestrator's scalability is most impacted by the SSH's configuration parameters that limit simultaneously open SSH connections by one client IP, at the testbed's SSH gateway. Orchestrator handles this limitation by cycling through SSH connections, opening and closing connections as needed. When tested with 500 virtual nodes, Orchestrator could deliver commands to all nodes within 10–20 seconds. Thus even with default settings, Orchestrator performs well in large-scale experiments.

7. RELATED WORK

To our knowledge, we are the first to propose a traffic generation solution, which fully decouples the many components of generation. The parts that comprise LegoTG are related to two bodies of work.

First, our Orchestrator is related to testbed execution management solutions such as SEER [22] and MAGI [23] on DeterLab [6], plush [24] on PlanetLab [19] and gush [25] on GENI [26]. Any of these solutions could have formed the basis for LegoTG's Orchestrator, but these solutions are tightly coupled to their respective testbed platforms and we wanted portability. The Orchestrator installs only on a single control machine, such as a researcher's laptop, sending commands via SSH to remote machines. SSH support is ubiquitous on many testbeds, such as Emulab [18], Planetlab, GENI, and DeterLab, thus Orchestrator is more portable and can support heterogeneous experimentation across multiple testbed platforms.

Second, TGblocks incorporate many existing tools and parts of other traffic generators, e.g., Click [17]. Throughout the paper we discussed several popular traffic generators [1, 3, 4, 5], and showed how these generators and their blueprints can be expressed through our framework. Thus functionalities of LegoTG surpass those of existing traffic generators, and achieve this with modular and much smaller code base. Our TwoSew model is further related to Tmix [15], a tool for NS-2, but it supports mixing of request-response dynamic with parallel sends on the same connection, while Tmix does not.

8. CONCLUSIONS

Traffic generation is essential in networking research, and needs for it vary greatly between experiments and users. In this paper we argued that a traffic generator's main goals should be modularity and flexibility, and not building a "one-size-fits-all" tool. To achieve these goals we decoupled feature extraction, modeling and generation and identified a narrow waist of basic functionalities which meet a wide range of needs. Our LegoTG framework realizes this modular traffic generation process. TGblocks that implement various functionalities can be easily interchanged, combined and modified. We demonstrated the power of LegoTG to emulate current traffic models, and build new ones through a series of experiments on the DeterLab testbed.

9. REFERENCES

[1] A. Turner, "tcpreplay." http://tcpreplay.synfin.net/.

[2] ESnet / Lawrence Berkleley National Laboratory, "iperf3: A TCP, UDP, and SCTP network bandwidth measurement tool." https://github.com/esnet/iperf.

[3] J. Sommers, H. Kim, and P. Barford, "Harpoon: A flow-level traffic generator for router and network tests," SIGMETRICS Perform. Eval. Rev., vol. 32, pp. 392–392, June 2004.

[4] K. V. Vishwanath and A. Vahdat, "Swing: realistic and responsive network traffic generation," IEEE/ACM Trans. Netw., vol. 17, pp. 712–725, June 2009.

[5] S. Avallone, S. Guadagno, D. Emma, A. Pescape, and G. Ventre, "D-itg distributed internet traffic generator," in Proceedings of the The Quantitative Evaluation of Systems, First International Conference, QEST '04, (Washington, DC, USA), pp. 316–317, IEEE Computer Society, 2004.

[6] T. Benzel, R. Braden, D. Kim, C. Neuman, A. Joseph, K. Sklower, R. Ostrenga, and S. Schwab, "Experiences with deter: A testbed for security research," in 2nd IEEE Conference on Testbeds and Research Infrastructure for the Development of Networks and Communities (TridentCom 2006), March 2006.

[7] The DETER Project, "DETERlab." http://www.deterlab.net/.

[8] Free Software Foundation, Inc., "GNU General Public License." http://www.gnu.org/copyleft/gpl.html.

[9] A. Author, "Legotg web page." Anonymized URL.

[10] Rick Jones, "Netperf." http://www.netperf.org/netperf/.

[11] "Selenium." http://seleniumhq.org/.

[12] "web-page-replay." http://github.com/chromium/web-page-replay.

[13] K. V. Vishwanath and A. Vahdat, "Evaluating distributed systems: Does background traffic matter?," in USENIX 2008 Annual Technical Conference on Annual Technical Conference, ATC'08, (Berkeley, CA, USA), pp. 227–240, USENIX Association, 2008.

[14] "GNU Netcat."

http://netcat.sourceforge.net/.
[15] M. C. Weigle, P. Adurthi, F. Hernández-Campos, K. Jeffay, and F. D. Smith, "Tmix: a tool for generating realistic tcp application workloads in ns-2," SIGCOMM Comput. Commun. Rev., vol. 36, pp. 65–76, July 2006.
[16] "tcpdump/libpcap." http://www.tcpdump.org/.
[17] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," ACM Trans. Comput. Syst., vol. 18, pp. 263–297, Aug. 2000.
[18] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An Integrated Experimental Environment for Distributed Systems and Networks," in Proceedings of OSDI, 2002.
[19] T. P. Consortium, "PlanetLab — An open platform for developing, deploying and accessing planetary-scale services." http://www.planet-lab.org/.
[20] Network Time Foundation, "NTP: The Network Time Protocol." http://www.ntp.org/.
[21] "MAWI Working Group Traffic Archive." http://tracer.csl.sony.co.jp/mawi/.
[22] S. Schwab, B. Wilson, C. Ko, and A. Hussain, "SEER: a Security Experimentation EnviRonment for DETER," in Proceedings of the DETER Community Workshop on Cyber Security Experimentation and Test on DETER Community Workshop on Cyber Security Experimentation and Test, 2007.
[23] "Montage AGent Infrastructure." http://montage.deterlab.net/montage/.
[24] J. Albrecht, C. Tuttle, A. C. Snoeren, and A. Vahdat, "Planetlab application management using plush," SIGOPS Oper. Syst. Rev., vol. 40, pp. 33–40, Jan. 2006.
[25] J. Albrecht and D. Huang, "Managing distributed applications using gush," in Testbeds and Research Infrastructures. Development of Networks and Communities (T. Magedanz, A. Gavras, N. Thanh, and J. Chase, eds.), vol. 46 of Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, pp. 401–411, Springer Berlin Heidelberg, 2011.
[26] M. Berman, J. Chase, L. Landweber, A. Nakao, M. Ott, D. Raychaudhuri, R. Ricci, and I. Seskar, "GENI: A federated testbed for innovative network experiments," Computer Networks, Special issue on Future Internet Testbeds, March 2014.