Expressing Different Traffic Models Using The LegoTG Framework

Genevieve Bartlett
University of Southern California/ISI
Email: bartlett@isi.edu

Jelena Mirkovic
University of Southern California/ISI
Email: mirkovic@isi.edu

***Abstract*—In this paper we demonstrate the ease of generating and modifying background traffic in testbed experiments through the traffic generation framework we developed, called LegoTG. LegoTG is a modular framework for composing custom traffic generation. It makes it easy to combine different traffic generators and traffic modulators (e.g., delay models), and to port the same background traffic to different topologies. In addition to the framework, we have developed several traffic generation/modulation tools that can be used in LegoTG to generate realistic and highly controllable network and transport-level traffic. We build our demonstration around a series of simple experiments which reinforce how much background traffic matters in experiments and how different traffic models can drastically affect experiment results and research conclusions.**

I. Introduction

In real networks, traffic is generated through interactions of multiple entities: humans that use applications to produce requests to servers, server applications that generate replies based on their configuration and content they are serving, network links that have limited bandwidth and a fixed propagation delay, routers that queue packets in limited-size queues, middle boxes that manipulate traffic according to their policies, etc. In testbed experiments, researchers aim to reproduce this complex process with traffic generators. Traffic generation needs can vary greatly from experiment to experiment, and are heavily dependent on the experiment details and goals. These diverse needs have led to the development of a large number of different traffic generators (e.g., Harpoon [1], Swing [2], D-ITG [3]).

Experiment design is an iterative process, and a researcher may not initially understand which background traffic features matter in her testbed experiment. This means designing and determining how to generate background traffic may become an experiment in itself, with the researcher trying out different traffic generators and traffic models, tuning them and customizing to her needs. Today's traffic generators do not support this process well. Existing generators adopt a single model for each of the traffic generation functionalities, e.g., packet generation, packet consumption, connection delay, and realize it through their code. For example, Harpoon models traffic as series of file transfers, Swing models link delay between two end-points as a single value and realizes it separately in each direction, D-ITG models two-way flows as two one-way flows and lets users control many transport and network parameters of the traffic. Fixed models make it more likely that a researcher will need to customize traffic generator's code to meet her needs. But, existing traffic generators subsume a great amount of complexity into a single code base that realizes multiple functionalities. A large code base makes customization very difficult. Further, a researcher may require functionalities from different traffic generators (e.g., packet generation from Harpoon and delay modeling from Swing). Combining pieces of different traffic generators in a single experiment today is a very complex and manual process.

In this paper, we demonstrate how the LegoTG traffic generation framework helps streamline generating background traffic with vastly different models and feature sets. It does so thanks to its modularity—each traffic generation functionality is a separate, independent module that can be customized, replaced or combined with other modules – and thanks to its extensive support for deployment, configuration and control of traffic generation code on distributed experiments. The ease of deploying and controlling traffic generation with LegoTG leads to rapid prototyping of testbed experiments which rely on background traffic. For our demonstration, we choose a simple experiment—a comparison of two bandwidth estimation tools, which was used in [4] to show the influence of background traffic on research conclusions.

II. LegoTG

In the LegoTG framework, each traffic generation functionality is realized through a separate piece of code, called a *TGblock*. The framework works like a child's building block set: TGblocks combine in different ways to achieve customizable and composable traffic generation. This combination and customization is achieved through LegoTG's *Orchestrator*, which sets up, configures, deploys, runs, synchronizes and stops TGblocks in distributed experiments. The entire specification of the traffic generation process for an experiment is in an *experiment configuration file*—called an *ExFile*, which is an input for the Orchestrator. The ExFile offers a convenient capture of all the details of an experiment's background traffic set up, which promotes sharing and reproducibility of experiments. LegoTG provides the following:

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.

Easy adoption of new tools. Assimilating a tool (e.g., a new packet generator or a new delay module) 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. A BIF need only be written once for each tool integrated into LegoTG. Together, a BIF and the tools that it wraps comprise a TGblock. BIF functions use the Orchestrator's API to specify interactions with remote machines. This allows for easy portability of new tools between experiments, and between users. BIFs are small and easy to write, as we discuss in Section II-B. Excerpts from a 90-line BIF which wraps tcpreplay [5] are shown in Figure 2.

Portability to different platforms. LegoTG and TGblocks are written in Python, which is supported on a wide range platforms. For orchestration of machines, LegoTG uses SSH—a commonly available tool on many testbeds, and live networks.

A. The Orchestrator

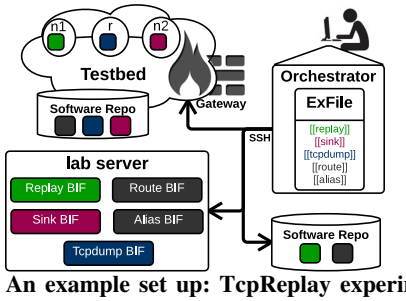

Fig. 1: An example set up: TcpReplay experiment.

Figure 1 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 [6] (n1) to replay traffic from a file, a sink node (n2) to consume this traffic, and a node running tcpdump [5] (r) to capture replayed traffic for analysis. If a user wants to keep the original IP addresses in the replayed traffic, we also need virtualized IPs (IP aliasing) on traffic sources and destinations and routes must be set up on all intermediate nodes for these IP addresses.

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 Figure 1 from a remote machine), and where to find the software for these blocks (e.g. in Figure 1 from two different software repositories). 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.

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)
```

Fig. 2: Excerpts from a BIF

B. 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 custom functions 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.

Figure 2 shows excerpts of a BIF which wraps tcpreplay [6] 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.

C. ExFiles: Experiment Configuration

The ExFile captures all the details of an experiment: where software repositories are for TGblocks, where to install and how to configure these TGblocks, how they are synchronized, and all the inputs and outputs. This single point of configuration enables easy prototyping and sharing of traffic generation processes. Figure 3 shows an example ExFile for the TcpReplay experiment (Figure 1) 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]]).

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    trace_grp = r
19    sink_grp = n2
20    all_testbed = replay_grp, trace_grp, sink_grp
21 [extraction]
22 ...
23 [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
```

Fig. 3: An example ExFile

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 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, etc. (5) **sections**: list of phases that comprise the experiment. Each section consists of subsections.

Our 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 in "alias", "route", "sink", "trace" and "replay" subsections, in that order, during the "experiment" section. Each block is described in a subsection, which specifies how to run the block: its target (eg. line 37), the definition for the block (path to its BIF file e.g. line 38), and its inputs, outputs and arguments (details in Sec. II-C3). The subsection's name need not reflect the name of its associated TGblock.

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.

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) 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 Figure 3, 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 matching 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 custom dictionary mapping of variable names to expansion functions. Variable expansion and abstraction of hosts into functional groups facilitate easy porting of ExFiles.

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 Figure 3, after all the TGblocks finish their "start" function, the Orchestrator will call the "stop" function for the trace TGblock, stopping the tcpdump [5] process.

D. Expressing Different Models in LegoTG

Traffic generation tools typically fit within one of two categories—replay-based or model-based generation. The former replays traffic deterministically as specified by a traffic log, the latter generates traffic based on a stochastic model. Typically, in model-based generation tools, the modeling and packet generation processes are intertwined and contained in a single monolithic code base [2], [1]. In LegoTG, any modeling process is separated from the generation process by having traffic models produce a *script*—a text file that can be translated and then read and realized by any number of generation tools. In this paper we will explore two models (Swing [2] and a model previously developed by us called TwoSew [7]). Both models view traffic as series of application data units (ADUs) that are sent between two communicating parties on a TCP connection. The realizers (TGblocks that implement the packet generator functionality) hand all data for a given ADU to the transport layer on an experimental machine, and the TCP stack handles its breaking into packets, and reliable delivery. This implementation via regular TCP stack makes the packet generation congestion responsive—if packets are dropped on a connection, the sending rate is reduced by the TCP's congestion response mechanism.

Swing. Swing views all communication on a single application port as a collection of *sessions*, where each session can have multiple RREs (request-response exchanges) and each RRE can have multiple TCP connections. TCP connections are grouped into RREs and sessions based on the pauses between them—those that are less than 30 seconds apart belong to the same RRE, and RREs that are less than 300 seconds apart belong to the same session. Within a single TCP connection, traffic is viewed as a series of requests and responses, each represented as a single ADU. To model traffic from a trace, Swing first groups all TCP traffic per application port. It then mines the number of sources for each application and for each source it mines: the number of sessions, the inter-session time, the number of RREs per session, the inter-RRE time, the number of connections per RRE, the inter-connection time, the sizes of requests and responses per connection and the inter-request time. All Swing's models are empirical distributions of the parameter values from a trace.

TwoSew. TwoSew [7] gets its name from being a two-way SEND/WAIT ADU model, and extends models used in [2], [8], to support a wider variety of flow dynamics that are present in realistic traffic traces.

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, just like inter-request time in Swing. 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. Swing does not have a corresponding model.

Another difference between TwoSew and Swing is that TwoSew supports communication with parallel sends (a client and server send each other data at the same time), while Swing only supports request-response exchanges (one party is silent while the other sends traffic). Parallel sends are present in realistic traffic traces, especially in media and P2P traffic.

Yet another difference between TwoSew and Swing is that Swing is a stochastic traffic generator and TwoSew performs deterministic, congestion-responsive replay. Swing mines value distributions for its parameters from traffic traces, and then draws parameter values at random from these distributions during traffic generation. TwoSew mines SEND/WAIT events from the traffic traces, exactly as they occurred.

E. TGblocks

We have developed over twenty TGblocks, some wrapping existing tools, and some wrapping tools we have written. Here we describe only the blocks and functions we will need for our experiments in Sec. III:

Aliasing & routing are supported by the ipalias 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 [9].

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 provides these functionalities.

Resource allocation 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) for ADU flow-based playback or packets (dividebypkts) for raw packet replay.

Tcpreplay. The libpcap tool tcpreplay [5] reads packets from a trace, and sends these packets out to the network. These are not congestive responsive flows, rather simply raw packets. The replay block wraps tcpreplay and takes an input file describing which hosts in the topology are assigned to replay each part of an original trace. This part assignment comes from the dividebypackets block, which specifies how many IPs are in each group, and information about which nodes in a topology are assigned by the researcher to be generators. During the "setup" function (run on each generating node by the Orchestra), the replay block extracts only the packets its host is assigned

to replay. The block then determines the appropriate mac addresses in the experiment topology to use when rewriting original trace packets to send out. This way, these packets can be routed on the experiment network.

Mimic. The mimic block wraps our same-named tool for deterministic ADU replay. Mimic reads TCP connection information (start time and IPs) and SEND/WAIT events from a text file. Mimic generates flows by creating TCP connections and sending data to the operating system on testbed nodes at the specified times. In our experiments we multiplex many IP addresses on a single experimental node. Only one Mimic process is run per machine, and it supports traffic generation on all these virtualized IPs. The dividebyevents block helps divvy up these virtualized IPs to balance the traffic generation load.

Twosewextract. This tool implements extraction of TCP connection information and SEND/WAIT events from traffic traces, that are needed by the Twosew traffic model, and creates scripts for Mimic to replay.

Twosew2Swing. Working with the data produced by twosewextract, this tool generates the information needed to encode the traffic model used by the Swing [2] traffic generator. The output from this block causes mimic to generate the same traffic that the full Swing implementation would generate.

Text2pcap. Text2pcap translates a textual representation of packets to libpcap format. This allows users to very easily build custom packets with desired features since this text input is human-readable and modifying text files is supported by a large number of tools (e.g. awk and sed).

III. Expressing Models and Rapid Prototyping

To demonstrate how LegoTG can be used to express different models we repeat and extend experiments previously done by Vishwanath et al. [4]. In this prior work, the authors demonstrate that background traffic characteristics can strongly impact results when evaluating distributed systems. In one of these demonstrations Vishwanath et al. compared the accuracy of two bandwidth measurement tools—pathchirp [10] and pathload [11]—which estimate the available end-to-end bandwidth between two hosts. In this demonstration, the authors examined three types of background traffic—constant bit rate UDP traffic, UDP Poisson traffic and congestion responsive TCP traffic replayed by the Swing traffic generator [2]. Among other results, Vishwanath et al. find that the accuracy of both tools degrades for more realistic and burstier traffic. Here we will reproduce some of Vishwanath et al.'s results when comparing constant bit rate traffic to traffic replayed according to Swing's *model*, though we use LegoTG, not the Swing generator, to produce all our background traffic. We also extend these prior experiments using a new traffic model (TwoSew, Section III-D) and a combination of generators (Section III-E). Through these small experiments we demonstrate: (1) the ease of setting up and customizing different background traffic using LegoTG, (2) how different background traffic influences research conclusions from an experiment.

A. Datasets and Experiment Setup

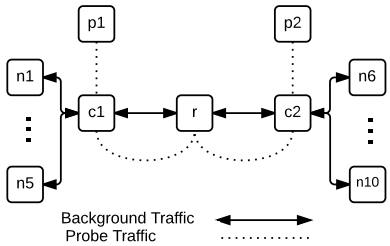

Fig. 4: Experiment topology for demonstrations. Bandwidth estimation tools are run on p1 and p2, while ten nodes (n1-10) generate background traffic. The two links between c1 and r and r and c2 are constrained to be a bottleneck, while other links are over-provisioned. Network emulation (using Click) is set up on c1 and c2.

For our illustration of using LegoTG, we use the DeterLab testbed [12], and a simple dumbbell topology (Figure 4), with 10 nodes to generate traffic (n1–n10). We have two nodes (c1, c2) to do network emulation via Click [9] but given LegoTG's modular nature, this emulation could easily be swapped for other implementations (eg. ModelNet [13]). In each run of our experiments we run one of the two bandwidth estimation tools on nodes p1 and p2, so that the probes from these tools go across the central link in the topology. As done by Vishwanath et al. in [4], we constrain the central links, and over-provision all other links, so that we are evaluating pathchirp and pathload based on the available bandwidth of the constrained link, congested by cross-traffic from nodes n1–10.

To examine each bandwidth estimator's accuracy in different congestion conditions, we modify the capacity of our central links via DeterLab. For each type of background traffic, we look at low (30% of bottleneck capacity), medium (50%) and high (70%) volumes of background traffic. While we could simply modify the amount of traffic generated to achieve this congestion effect, this modification to the number and volume of individual connections could impact the realism of the model in undetermined ways.

We test each tool separately, and run different combinations of the bottleneck link capacity, the tool being evaluated and the type of background traffic. We run each combination of these three factors multiple times, and evaluate each tool in each setting across five trials. Each trial, we gather the mean bandwidth reported by the tool over the entire trial and calculate the percent error for each measurement the tool reported. Pathload reports a low and high estimate of the available bandwidth (every 5–20 seconds depending on the interaction with background traffic). Pathchirp reports a single estimate measurement more frequently (0.5–2 seconds). To calculate the percent error we collect a trace of all background traffic at node r each run and use this to calculate the actual available bandwidth at the time interval of each reported bandwidth estimate. In the following graphs, we show the mean bandwidth as a percentage of link capacity as reported by each tool, and percentage error bars, including an error bar to denote variation in our background traffic between runs.

For background traffic based on features from a real-world trace (as used by Swing and our TwoSew model), we use a trace from the MAWI traffic repository [14] collected on March 1st, 2012 at 2pm. This trace contains 15 minutes of traffic collected on a trans-Pacific link between Japan and US. Though to keep our repeated trials short, we replay only 1 minute of traffic from this trace. Other traces and longer durations can be as easily replayed.

Through these experiments, we will illustrate how easy it is to change models and modify traffic with LegoTG. We start by modifying the ExFile example in Figure 3 to fit our experiment topology. The experiment described in Figure 3 has only three nodes (n1, r and n2). To work with our larger topology (Figure 4), we simply need to add the extra experiment nodes in the [hosts] section, and assign these nodes to groups in the [groups] section. We also write BIFs for pathchirp and pathload to create TGblocks which will deploy and run the sender and receiver for each of these tools and record output. [1] In the next sections, we add and remove TGBlocks from our experiment file to achieve different background generation. We run the [extraction] and [allocation] phases once for each set up, and then run the [experiment] phase multiple times to evaluate each tool in multiple (five) trials for each of three link capacities.

B. Constant Bit Rate (CBR)

We can use any number of tools to generate constant bitrate traffic (eg. iPerf [15]), but to illustrate how LegoTG enables easy customization of traffic, we instead use the text2pcap block and generate a 60 second pcap file to be replayed by replay. First, for text2pcap's input we write a small shell script to output a repeating list of text lines which describe a series of UDP packets sent by four separate IPs. All packets are the same size and timed such that we have a constant bitrate of 150Mbs. Figure 5 shows how we modify the ExFile in Figure 3 to set up LegoTG to replay this text specification. First, we add the text2pcap to our [extraction] phase and specify the location of our text file in the arguments. The pcap output of text2pcap will be passed on to the replay block. In the [allocation] phase, we will use the pcap file produced by text2pcap as input to dividebypkts, and specify we want two groups in our allocation.txt file. The dividebypkts block will divvy up the packets into two groups, which each have two IPs, and we can assign each group to a testbed node. For our generated UDP packets to be routed across our dumbbell topology, during the [experiment] phase, we still need the route block and we also need the alias block which will virtualize the four IPs we specified in our packet description text file. To avoid generating ICMP responses, we keep the sink block. Since we now have two testbed nodes generating packets, we need to synchronize the start of these generation processes. For this synchronization we use multisynchro and specify which set of hosts ("replay_grp") and which block (replay) to synchronize.

[1]Pathchirp also includes a master program to coordinate the sender and receiver.

ExFile Example: TcpReplay Experiment

```
part_allocation=/home/msmith/allocation.txt
...
[hosts]
...
   [[deterlab.isi.edu]]
   nodes = n1, ... n10, c1, c2, r, p1, p2
[groups]
   replay_grp = n1, n5
   trace_grp = r
   sink_grp = n1, n5
   all_testbed = replay_grp, trace_grp, sink_grp
[extraction]
...
   [[text2pcap]]
      target = local
      def = $biflib/text2pcap/BIF.py
      [[[input]]]
         pkt_desc = $home/udp_150Mbs_CBR.txt
      [[[output]]]
         trace_file = $home/udp_150Mbs_CBR.pcap
[allocation]
...
   [[divide]]
      target = local
      def = $biflib/dividebypkts/BIF.py
      [[[input]]]
         trace_file = $home/udp_150Mbs_CBR.pcap
      [[[output]]]
         part_allocation = $part_allocation
...
[experiment]
   order = alias, route, sink, trace, replay, synchro
...
   [[synchro]]
      target = replay_grp
      def = $topdir/multisynchro/legoTGinterface.py
      [[[args]]]
         block_list = replay
```

Fig. 5: Highlighted changes to Fig. 3 to generate CBR traffic.

From the ExFile in Figure 5, LegoTG's Orchestrator will generate 150Mbs, UDP traffic. To test pathchirp and pathload, we add the block for each in the [experiment] section, one at a time, by modifying the "order = " line to include either our pathchirp or pathload block. We use the Deter [12] web interface to set the bottleneck link capacity (to 500Mbs or 30% congestion for "low", 300Mbs or 50% for "medium" and 214Mbs or 70% for "high"). For each of these three settings we run five trials for each tool and calcuate mean and error across these iterations.

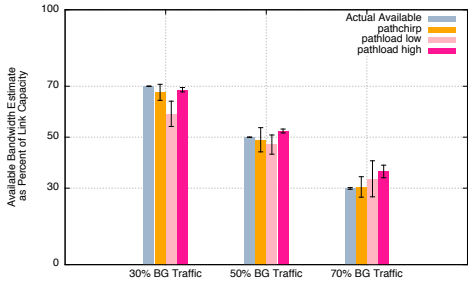

Fig. 6: Comparing pathload (high and low) vs. pathchirp bandwidth estimate accuracy with constant bit-rate UDP background traffic.

Figure 6 shows the comparison of pathchirp's and pathload's bandwidth estimation with CBR cross-traffic. From these, we can confirm earlier results [4], [10] that both tools are quite close to the actual values for low/medium congestion. Our results are more inline with [10] for the heavily utilized link, and we did not see some of the overestimation Vishwanath et al. observed [4]. These differences may be due to the longer duration of trials in [4], and differences in packet coalescence by network interface cards.

C. Using Swing's Model

We now look at using more realistic background traffic in evaluating pathchirp and pathload. We turn first to Swing's model for congestion-responsive traffic, since this was the model of choice for Vishwanath et. al when showing how more realistic traffic affects the performance of these bandwidth estimation tools. We start by modifying our ExFile for CBR traffic (Figure 5), and show relevant changes (in blue) in (Figure 7). As described in Section II-D, LegoTG separates the model from the realization of packet generation. To create flows as per Swing's model in LegoTG we first extract flow information from our Mawi trace using twosewextract, and translate this information into flow events that reproduce Swing's model using twosew2swing. We include these two blocks in our [extraction] phase, and remove text2pcap. The Swing traffic generator performs network emulation. To express this functionality in LegoTG we use the hostdelay block. In addition to extracting ADU information, twosewextract uses RTT samples to identify network delays on a per host basis. We feed these delays to the hostdelay block (not shown). For the [allocation] phase instead of using dividebypkts, we turn to dividebyevents which will group the IPs included in the output from twosew2swing such that the events per IP group are as balanced as possible. We then can assign these groups to generator nodes. Since creating and tracking multiple flows is more resource intensive than simple raw packet replay (as done in the previous section), we group IPs into ten groups (not just two) to enable distributing the generation task across more testbed nodes. We define a "generators" group of ten nodes (n1–n10). In the [experiment] phase, we drop replay for mimic to replay congestion responsive traffic according to Swing's model. Mimic is both a source and sink for packets so we no longer need the sink block. Mimic can synchronize across multiple nodes, so we drop multisynchro.

Just as we did with CBR background traffic, we use Deter [12] to modify the bottleneck link capacity. The bandwidth of the traffic replayed through Swing's model based on our MAWI trace has an average bandwidth of 165Mbs, so we vary the link congestion/capacities as follows: 30%/550Mbs, 50%/330Mbs, 70%/235Mbs. As before, we run the extraction and allocation phases once and then repeatedly run the experiment phase to run multiple trials with the pathchirp block and then pathload for each capacity. As Vishwanath et al. note when using Swing cross-traffic, we find that with more realistic background traffic both tools are less accurate than with steady CBR traffic. We observe this holds true for all link capacities. The first cluster of results in Figure 8 show the results for only the highest congested link setting (70%)—where both bandwidth estimation tools were the least accurate. Though pathload tended to over estimate the bandwidth for the congested link, its high estimates were more accurate than its lower bound estimates. As with the CBR traffic, we did not observe pathchirp overestimating, as Vishwanath et al. did, but again, this could be a difference in hardware.

ExFile Example: TcpReplay Experiment

```
part_allocation=/home/msmith/allocation.txt
...
[hosts]
...
   [[deterlab.isi.edu]]
   nodes = n1, ... n10, c1, c2, r, p1, p2
[groups]
   generators = n1, n2, n3, n4, n5, n6, n7, n8, n9, n10
   delay_grp = c1 c2
   trace_grp = r
   all_testbed = generators, trace_grp
[extraction]
...
   [[twosewextract]]
      target = local
      def = $biflib/twosewextract/BIF.py
      [[[input]]]
         trace_file = $home/mawi.pcap
      [[[output]]]
         output_dir = $home/mawi2twosew/
   [[twosew2swing]]
      target = local
      def = $biflib/twosew2swing/BIF.py
      [[[input]]]
         input_dir = $home/mawi2twosew/
      [[[output]]]
         output_dir = $home/mawi2swing/
[allocation]
...
   [[divide]]
      target = local
      def = $biflib/dividebyevents/BIF.py
      [[[input]]]]
         input_dir = $home/mawi2swing/
...
[experiment]
   order = alias, route, hostdelay, trace, mimic
...
   [[hostdelay]]
      target = delay_grp
...
```

Fig. 7: Highlighted changes to Fig. 5 to generate CBR traffic.

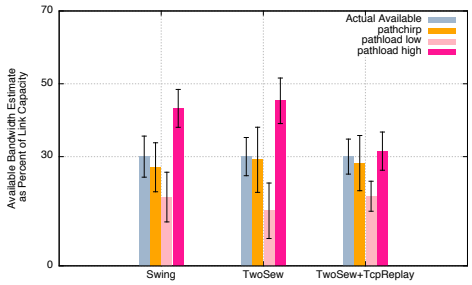

Fig. 8: Comparing pathload (high and low) vs. pathchirp bandwidth estimate accuracy with background traffic per different models: Swing [2] (congestion responsive), TwoSew (congestion responsive) and TwoSew + replay of UDP and other non-congestion responsive packets.

D. Using TwoSew's Model

LegoTG makes it easy for us to not only repeat past experiments, but extend these experiments by exploring other traffic models. In Section II-D, we introduced TwoSew—a two-way SEND/WAIT event based model for flows. Though TwoSew shares some similarities with Swing, the traffic TwoSew extracts and models has fairly different characteristics. For example, Figure 9 depicts the bandwidth over time for our original MAWI trace (shaded in gray), and LegoTG Swing-based and TwoSew-based replayed traffic. While the overall average bandwidth of the Swing-based (165Mbs) and TwoSew-based (191Mbs) traffic are similar, Swing's extraction and stochastic modeling changes the bursty characteristics of the original trace and smoothes some of the peaks.

To modify our previous ExFile (which replayed traffic based on Swing's model) to generate traffic based on TwoSew's

model, we only need to change our [extraction] section to only include extraction for TwoSew. After this modification, rerunning the [extraction] and [allocation] phases will produce the appropriate files needed by Mimic to replay traffic based on the TwoSew model with SEND/WAIT events balanced across generating hosts. We then repeat experiments, testing each bandwidth estimation tool at various link congestion points/capacities (30%/636Mbs, 50%/382Mbs, 70%/272Mbs). The second cluster of bars in Figure 8 show the results for only the highest congested link setting (70%). We note a slight decrease in accuracy for both tools, when compared to the accuracy with Swing-based background traffic. This result further supports Vishwanath et al. conclusions that burstier traffic degrades performance of both tools.

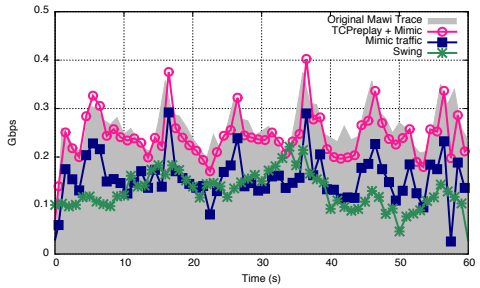

Fig. 9: Comparison of Swing-based and TwoSew-based (with and without additional replay of non-TCP traffic) replay of our MAWI trace.

E. Combining Generators: TwoSew+TcpReplay

In the previous sections, we demonstrated how LegoTG makes it easy to change how we model and replay traffic. LegoTG also facilitates combining generators, providing flexibility in how different traffic types are expressed. For example, TwoSew and Swing focus on TCP flows which exchange data. Real-world traffic contains much more than these flows, so a large portion of a real-world trace is not represented in traffic replayed by these models. If we wish to reproduce our original MAWI trace with higher fidelity, we need to replay these packets (largely UDP) in conjunction with the congestion-responsive flows—a task we can carry out with our replay block. Figure 9 shows how this synchronized combination of Mimic replaying a TwoSew extracted model with tcpreplay matches the original MAWI trace quite closely. To synchronize such a combination, we disable Mimic's internal synchronization and configure both mimic and replay to be synchronized by multisynchro. Additionally, we need to create a trace of all the packets not handled by mimic as input for the replay block.

We again run multiple trials for both bandwidth estimation tools with the combined background traffic from Mimic replaying TwoSew-based generation, and tcpreplay handling the remaining packets. The total average bandwidth for this experiment is 385Mbs, so to compare with our previous experiments we set the link capacity to 550Mbs to create a 70% congested link. In Figure 8, the third cluster of bars show the results for pathchirp and pathload at this high congestion level. We note that pathload appears to significantly reduce its overestimation seen in our previous experiments. This indicates that even with traffic models based on the same trace, results and conclusions can be drastically different. A traffic generator with a fixed traffic model does not offer enough customizability to reveal the results we can find through LegoTG.

IV. Conclusions

In this paper, we demonstrated the use of LegoTG—a framework for composable and customizable traffic generation. Through a series of small experiments evaluating two different bandwidth estimation tools, we demonstrated how the full generation process, from feature extraction, to resource allocation, to actual generation can be rapidly designed and realized through LegoTG. Extending LegoTG is as easy as writing a few lines of code—as we did for our evaluated bandwidth estimation tools. Changing which TGblocks we use, we can easily express different traffic models and combine different generation tools—all through small modifications to a single ExFile configuration. ExFiles capture the details of an experiment—enabling repeatability and easy sharing of experiment set ups on testbeds.

References

[1] 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.
[2] K. V. Vishwanath and A. Vahdat, "Swing: realistic and responsive network traffic generation," IEEE/ACM Trans. Netw., vol. 17, pp. 712–725, June 2009.
[3] 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.
[4] 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.
[5] "Tcpdump and libpcap." http://www.tcpdump.org/.
[6] A. Turner, "tcpreplay." http://tcpreplay.synfin.net/.
[7] "TwoSew." http://trac.deterlab.net/wiki/LegoTG/modules/twosew.
[8] 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.
[9] 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.
[10] V. Ribeiro and R. Riedi, "Pathchirp: A light-weight available bandwidth estimation tool for network-aware applications," in Proceedings of 2003 LACSI Symposium., pp. 1–12, 2003.
[11] M. Jain and C. Dovrolis, "Pathload: A measurement tool for end-to-end available bandwidth," in In Proceedings of Passive and Active Measurements (PAM) Workshop, pp. 14–25, 2002.
[12] The DETER Project, "DETERlab." http://www.deterlab.net/.
[13] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker, "Scalability and accuracy in a large-scale network emulator," SIGOPS Oper. Syst. Rev., vol. 36, pp. 271–284, Dec. 2002.
[14] "MAWI Working Group Traffic Archive." http://tracer.csl.sony.co.jp/mawi/.
[15] ESnet / Lawrence Berkleley National Laboratory, "iperf3: A TCP, UDP, and SCTP network bandwidth measurement tool." https://github.com/esnet/iperf.

This material is based upon work supported by the National Science Foundation under Grant No. 1127388.