rushit Documentation

Release 0.2

rushit authors

Dec 19, 2018

Contents:

CHAPTER 1

rushit

`rushit` is a network micro-benchmark tool that is scriptable with Lua. It resembles well-known tools like iperf or netperf. It originates from the neper project.

The project aims for the sweet spot between a small C program that simulates a network client/server and a fully featured network micro-benchmark. It provides you with a basic multi-threaded epoll-based client/server program that is intended to be extended, as needed, with Lua scripts.

`rushit` can simulate the following network workloads:

- `tcp_rr`, a request/response over TCP workload; simulates HTTP or RPC,
- `tcp_stream`, a uni-/bi-directional bulk data transfer over TCP workload; simulates FTP or `scp`,
- `udp_stream`, a uni-directional bulk data transfer over UDP workload; simulates audio or video streaming.

1.1 How do I get started?

Best place to start is the project documentation at Read the Docs.

Introduction goes over basic usage and available command line options.

Script Examples demonstrate what Lua scripting can be used for. Be sure to check out the accompanying *Script API* documentation.

1.2 Don't know Lua?

Don't worry. See Learn Lua in Y minutes for a quick introduction.

Once you are hooked on Lua, take a look at these resources:

- Lua for Programmers blog series
- Programming in Lua book

- Lua 5.1 Reference Manual
- The Lua language (v5.1) cheat sheet

1.3 Packages

Fedora, CentOS: In pabeni's copr repository *(thanks Paolo!)*

1.4 Want to contribute?

Great! Just create a Pull Request. We follow these guidelines:

- C99, avoid compiler-specific extensions.
- Linux kernel coding style, with tabs expanded to 8 spaces.

CHAPTER 2

Introduction

2.1 Basic usage

`rushit` is intended to be used between two machines. On each machine, the `rushit` process (e.g. `tcp_rr` or `tcp_stream`) spawns T threads (workers), creates F flows (e.g. TCP connections), and multiplexes the F flows evenly over the T threads. Each thread has a epoll set to manage multiple flows. Each flow keeps its own state to make sure the expected workload is generated.

For ease of explanation, we refer to the two machines as the server and the client. The server is the process which binds to an endpoint, while the client is the process which connects to the server. The ordering of `bind()` and `connect()` invocations is insignificant, as neper ensures the two processes are synchronized properly.

When launching the client process, we can specify the number of seconds to generate network traffic. After that duration, the client will notify the server to stop and exit. To run the test again, we have to restart both the client and the server. This is different from netperf, and hopefully should make individual tests more independent from each other.

2.1.1 `tcp_rr`

Let's start the server process first:

```
server$ tcp_rr
percentiles=
all_samples=
port=12867
control_port=12866
host=
max_pacing_rate=0
interval=1.000000
pin_cpu=0
dry_run=0
client=0
buffer_size=65536
```

(continues on next page)

(continued from previous page)

```
response_size=1
request_size=1
test_length=10
num_threads=1
num_flows=1
min_rto=0
help=0
total_run_time=10
*(process waiting here)*
```

Immediately after the process starts, it prints several `key=value` pairs to stdout. They are the command-line option values perceived by `rushit`. In this case, they are all default values. We can use them to verify the options are parsed correctly, or to reproduce the test procedure from persisted outputs.

At this point, the server is waiting for a client to connect. We can continue by running

```
client$ tcp_rr -c -H server
*(options key-value pairs omitted)*
```

`-c` is short for `--client` which means "this is the client process". If `-c` is not specified, it will be started as a server and call `bind()`. When `-c` is specified, it will try to `connect()`, and `-H` (short for `--host`) specifies the server hostname to connect to. We can also use IP address directly, to avoid resolving hostnames (e.g. through DNS).

For both `bind()` and `connect()`, we actually need the port number as well. In the case of `rushit`, two ports are being used, one for control plane, the other one for data plane. Default ports are 12866 for control plane and 12867 for data plane. They can be overridden by `-C` (short for `--control-port`) and `-P` (short for `--port`), respectively. Default port numbers are chosen so that they don't collide with the port 12865 used by netperf.

Immediately after the client process prints the options, it will connect to the server and start sending packets. After a period of time (by default 10 seconds), both processes print statistics summary to stdout and then exit:

```
server$ tcp_rr
*(previous output omitted)*
invalid_secret_count=0
time_start=1306173.666348235
utime_start=0.062141
utime_end=0.348902
stime_start=0.003883
stime_end=5.798208
maxrss_start=7896
maxrss_end=7896
minflt_start=568
minflt_end=571
majflt_start=7
majflt_end=7
nvcsw_start=26
nvcsw_end=329455
nivcsw_start=46
nivcsw_end=1028
num_transactions=329605
start_index=0
end_index=9
num_samples=10
throughput=33009.84
correlation_coefficient=1.00
time_end=1306183.666374314
```

(continues on next page)

(continued from previous page)

```
client$ tcp_rr -c -H server
*(previous output omitted)*
*(new output lines are similar to the server)*
```

From the line `throughput=33009.84`, we know this test run finished 33009.84 request/response "ping-pong" transactions per second. A transaction for the client means sending a request and then receiving a response. A transaction for the server means receiving a request and then sending a response. The number in this example is very high because it was run on localhost.

To look closer, let's reexamine the test parameters (command-line options), most importantly:

```
response_size=1
request_size=1
test_length=10
num_threads=1
num_flows=1
```

That means we were using one thread (on each side) with one flow (TCP connection between server and client) to send one-byte requests and responses over 10 seconds.

To run the test with 10 flows and two threads, we can instead use

```
server$ tcp_rr -F 10 -T 2
client$ tcp_rr -c -H server -F 10 -T 2
```

where `-F` is short for `--num-flows` and `-T` is short for `--num-threads`.

That will be 10 flows multiplexed on top of two threads, so normally it's 5 flows per thread. `rushit` uses `SO_REUSEPORT` to load balance among the threads, so it might not be exactly 5 flows per thread (e.g. may be 4 + 6). This behavior might change in the future.

Server and client do not need to use the same number of threads. For example, we can create 2 threads on the server to serve requests from 4 threads from the client.

```
server$ tcp_rr -F 10 -T 2
client$ tcp_rr -c -H server -F 10 -T 4
```

In this case, the four client-side threads may handle 3 + 3 + 2 + 2 (= 10) flows respectively.

Also note that we have to specify the number of flows on the server side. This behavior might change in the future.

2.1.2 `tcp_stream`

`tcp_stream` shares most of the command-line options with `tcp_rr`. They differ in the output since for a bulk data transfer test like `tcp_stream`, we care about the throughput in Mbps rather than in number of transactions.

By default, it's the client sending data to the server. We can enable the other direction of data transfer (from server to client) by specifying command-line options `-r` (short for `--enable-read`) and `-w` (short for `--enable-write`).

```
server$ tcp_stream -w
client$ tcp_stream -c -H server -r
```

This is equivalent to

```
server$ tcp_stream -rw
client$ tcp_stream -c -H server -rw
```

since `-w` is auto-enabled for `-c`, and `-r` is auto-enabled when `-c` is missing.

In both cases, the flows have bidirectional bulk data transfer. Previously, netperf users may emulate this behavior with `TCP_STREAM` and `TCP_MAERTS`, at the cost of doubling the number of netperf processes.

Note that we don't have netperf `TCP_MAERTS` in `rushit`, as you can always choose where to specify the `-c` option. The usage model is basically different, as we don't have a daemon (like netserver) either.

2.2 Options

2.2.1 Connectivity options

```
client
host
local_host
control_port
port
```

2.2.2 Workload options

```
maxevents
num_flows
num_threads
test_length
pin_cpu
dry_run
logtostderr
nonblocking
```

2.2.3 Statistics options

```
all_samples
interval
```

2.2.4 TCP options

```
max_pacing_rate
min_rto
listen_backlog
```

2.2.5 `tcp_rr` options

```
request_size
response_size
buffer_size
percentiles
```

The output is only available in the detailed form (`samples.csv`) but not in the stdout summary.

```
server$ ./tcp_rr
client$ ./tcp_rr -c -H server -A --percentiles=25,50,90,95,99
client$ cat samples.csv
time,tid,flow_id,bytes_read,transactions,latency_min,latency_mean,latency_max,latency_
↪stddev,latency_p25,latency_p50,latency_p90,latency_p95,latency_p99,utime,stime,
↪maxrss,minflt,majflt,nvcsw,nivcsw
2766296.649115114,0,0,31726,31726,0.000019,0.000030,0.008010,0.000086,0.000023,0.
↪000026,0.000032,0.000033,0.000068,0.005268,0.479424,5288,71,0,28490,3360
2766297.649131797,0,0,62857,62857,0.000019,0.000031,0.007757,0.000078,0.000024,0.
↪000027,0.000032,0.000034,0.000080,0.022667,0.933914,5288,133,0,57761,5692
2766298.649119440,0,0,98525,98525,0.000015,0.000027,0.004187,0.000048,0.000023,0.
↪000025,0.000032,0.000033,0.000048,0.063623,1.481519,5288,204,0,91853,7383
2766299.649141269,0,0,138042,138042,0.000015,0.000024,0.009910,0.000091,0.000018,0.
↪000018,0.000027,0.000030,0.000041,0.084147,1.984098,5288,283,0,129072,9754
2766300.649148147,0,0,169698,169698,0.000019,0.000030,0.004938,0.000063,0.000024,0.
↪000027,0.000034,0.000036,0.000057,0.119381,2.493741,5288,346,0,160027,10551
2766301.649127545,0,0,202454,202454,0.000019,0.000029,0.006942,0.000060,0.000025,0.
↪000027,0.000032,0.000032,0.000060,0.165496,2.920798,5288,411,0,186603,16817
2766302.649152705,0,0,234954,234954,0.000018,0.000029,0.012611,0.000100,0.000025,0.
↪000026,0.000031,0.000032,0.000059,0.205488,3.349022,5288,475,0,212910,23195
2766303.649116145,0,0,269683,269683,0.000019,0.000027,0.004842,0.000038,0.000024,0.
↪000026,0.000031,0.000032,0.000048,0.242531,3.806882,5288,544,0,240914,30076
2766304.649131298,0,0,302011,302011,0.000019,0.000030,0.004476,0.000049,0.000025,0.
↪000029,0.000032,0.000033,0.000044,0.253141,4.294832,5288,608,0,270468,32944
2766305.649132278,0,0,340838,340838,0.000015,0.000025,0.000220,0.000006,0.000022,0.
↪000025,0.000031,0.000033,0.000035,0.284624,4.808422,5288,685,0,308307,34005
```

2.2.6 `tcp_stream` options

```
reuseaddr
enable_read
enable_write
epoll_trigger
delay
buffer_size
```

2.3 Output format

When consuming the key-value pairs in the output, the order of the keys should be insignificant. However, the keys are case sensitive.

2.3.1 Standard output keys

```
total_run_time # expected time to finish, useful when combined with --dry-run
invalid_secret_count
time_start
start_index
end_index
num_samples
time_end
rusage
```

(continues on next page)

(continued from previous page)

```
utime_start
utime_end
stime_start
stime_end
maxrss_start
maxrss_end
minflt_start
minflt_end
majflt_start
majflt_end
nvcsw_start
nvcsw_end
nivcsw_start
nivcsw_end
```

2.3.2 tcp_rr

```
num_transactions
throughput
correlation_coefficient # for throughput
```

2.3.3 tcp_stream

```
num_transactions
throughput_Mbps
correlation_coefficient # for throughput_Mbps
```

CHAPTER 3

Script API

This document describes the API available for use from within the Lua script that can be passed on the command line using the `--script` option. Such script will be executed by the main thread that controls the client/server network threads.

Typical script can be split into 4 parts:

1. initialize shared variables
2. register hook functions
3. trigger the test run
4. process collected data

3.1 Hooks

Hooks are user-provided functions that allow you to "hook up" into well-defined points of the client/server thread logic and execute a custom script.

There are currently two types of hooks: *Socket Hooks* and *Packet Hooks*.

3.1.1 Socket Hooks

Socket hooks are tied to socket events. Hooks are called once per each socket that gets opened or closed by the client/server threads.

They are intended for configuring the socket (e.g. with `setsockopt(2)`) or collecting the information about the socket (e.g. with `getsockopt(2)`).

Note: In TCP workloads (`tcp_stream` or `tcp_rr`), server-side socket hooks operate on the *listening* socket, not the *connection* socket. This might change in the future.

---

socket_hook_fn(sockfd, addr)
User provided function invoked right *after* the socket has been opened (i.e. after the `socket(2)` call), or just *before* the socket will be closed (i.e. before the `close(2)`).

Parameters

- **sockfd** (*int*) – Socket descriptor client/server thread uses to `read(2)` / `write(2)` data.
- **addr** (*struct addrinfo*) – Address used to `bind(2)` (for server threads) or `connect(2)` (for client threads) the socket.

client_socket(socket_hook)
server_socket(socket_hook)
Registers a hook function to be invoked after the client/server thread opens a connection/listening socket with a `socket(2)` call.

Parameters

- **socket_hook** (socket_hook_fn) – Hook function invoked after `socket(2)` call.

client_close(socket_hook)
server_close(socket_hook)
Registers a hook function to be invoked before the client/server thread closes a connection/listening socket with a `close(2)` call.

Parameters

- **socket_hook** (socket_hook_fn) – Hook function invoked before `close(2)` call.

3.1.2 Packet Hooks

Packet hooks are tied to the socket message queue and socket error queue events. They can be used to implement a custom way to read from or write to a socket within the client/server thread's main loop.

For TCP workloads (`tcp_stream` and `tcp_rr`), packet hooks always operate on connection sockets.

packet_hook_fn(sockfd, msg, flags)
User provided function invoked when the socket's message queue (or error queue) is ready to read/write. The packet hook function is called *instead* of a `read(2)` / `write(2)` call.

Packet hook function must return the number of bytes read/written or -1 in the event of an error. This is usually achieved by passing up the return value from either `read()` / `recv()` / `recvfrom()` / `recvmsg()`, or `write()` / `send()` / `sendto()` / `sendmsg()`.

Parameters

- **sockfd** (*int*) – Socket descriptor to read from or write to.
- **msg** (*struct msghdr*) – Message buffer to read data into or write data from. Buffer size is determined by command line option `--buffer-size` / `-B` (16 KiB or 16384 bytes by default). In case of reading from the error queue, `msg` also has a 512 byte control message buffer.
- **flags** (*int*) – `MSG_*` flags that should be passed to `recv*()` / `send*()` calls.

Returns Number of bytes read/written or -1 in the event of an error.

client_sendmsg(packet_hook)
server_sendmsg(packet_hook)
Registers a hook function to be invoked when a socket is ready for writting. i.e. on `EPOLLOUT` `epoll(7)` event.

Parameters

- **packet_hook** (packet_hook_fn) – Hook function to write data to the socket.

client_recvmsg(packet_hook)
server_recvmsg(packet_hook)
Registers a hook function to be invoked when a socket is ready for reading, i.e on EPOLLIN epoll(7) event.

Parameters

- **packet_hook** (packet_hook_fn) – Hook function to read data from the socket.

client_recverr(packet_hook)
server_recverr(packet_hook)
Registers a hook function to be invoked when socket's error queue is ready for reading, i.e. on EPOLLERR epoll(7) event.

Parameters

- **packet_hook** (packet_hook_fn) – Hook function to read data from the socket error queue.

3.2 Run Control

run()
Triggers the test run and waits for the client/server threads to finish.

It is used to separate the first part of the script that needs to be executed before the network threads start running from the second part of the script that can be executed only when the network threads have stopped running.

Before *run()* returns it collects values of local variables that have been marked for collection from client/server threads. See *collect()*.

3.3 Data Passing

collect(value)
Marks a value for collection from the client/server threads after the test run.

Returns the given value wrapped in a table with metadata that identifies it for collection. Returned table should be treated as an opaque object until after the test run.

The value will be automatically unwrapped (i.e. extracted from the table) when copied to the client/server thread.

After the test run (i.e. a call to *run()*), the wrapper table will be populated with corresponding values from each client/server thread for access from outside of the hook functions.

Returns Wrapped value that will be replaced by a table with values collected from client/server threads after the call to *run()*.

Todo: Add link to an example.

3.4 Syscall Wrappers

Lua syscall wrappers are provided by the ljsyscall library. We provide convenience aliases for symbols exported by ljsyscall so that the symbol names are more C-like. That is:

```
S = require("syscall")
-- Aliases for syscalls
recvmsg = S.recvmsg
-- Aliases for constants
AF_INET = S.c.AF.INET becomes
-- Aliases for data types
sockaddr_in = S.types.t.sockaddr_in
```

Warning: Only a small set of symbols have aliases at the moment (see `script_prelude.lua`). This will be resolved in the near future. In the meantime please access any symbol that is missing an alias via the `S` global variable.

CHAPTER 4

Script Examples

4.1 bind-to-device

- Script: examples/bind-to-device
- Demo: examples/bind-to-device_example

4.2 check-src-addr

- Script: examples/check-src-addr
- Demo: examples/check-src-addr_example

4.3 collect-values

- Script: examples/collect-values
- Demo: examples/collect-values_example

4.4 drop-count

- Script: examples/drop-count
- Demo: examples/drop-count_example

4.5 tcp-ack-interval

- Script: examples/tcp-ack-interval

- Demo: examples/tcp-ack-interval_example

4.6 time-script-run

- Script: examples/time-script-run
- Demo: examples/time-script-run_example

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Index