Manifold: $O(N^2)$ testing of network protocols

XXX
XXX
XXX,
XXX
Email: XXX

XXX
XXX
XXX
XXX

XXX
XXX
XXX,
XXX
Email: XXX

Abstract—**While developing a UDP-based peer-to-peer transport protocol, we faced the problem of testing the implementation, its state machine, and congestion control algorithms. The problem is known to be fundamentally hard. Discoveries of decades-old bugs in TCP/IP stacks give a good illustration to this. Not being satisfied with classic methods, we have created the Manifold framework for automated massively parallel testing. Our main challenge was the combinatorial amount of diverse network conditions, protocol states and code paths affecting the implementation's behavior. By running traffic flows between every pair of nodes, Manifold covers $O(N^2)$ combinations of simulated and/or real network conditions thus performing massive case coverage in limited time. Reports, graphs, and a full system-wide event log allows to trace code paths and investigate problems. Being integrated into the code/build/test loop, Manifold instantly reveals both progress and regressions and enables rapid iterations on the code.**

I. Introduction

The development and testing of network protocols is made difficult by the non-determinism of distributed systems. Congestion control is one of the most complicated topics as workings of the algorithms heavily depend on race conditions, packet losses, and other peculiarities of network behavior. Being put in somewhat different conditions, a "proven" code might turn "problematic", as it was the case with the famous LFN problem [1], [2] of TCP. Bugs in TCP implementations are found till this day [3], despite the excessive level of use and testing in past 30 years. New TCP congestion control algorithms are normally tested in the settings of dumb bell and/or parking lot [4] simulated network topologies, as well as in the wild. The dumb bell setting allows to test stream behavior while competing with other flows for a single bottleneck; the parking lot topology simulates a sequence of bottlenecks.

We have developed a multiparty (swarming) transport protocol [5] using the LEDBAT [6] least-than-best-effort ("scavenger") congestion control algorithm. We had to test the implementation's behavior in a swarm, an aspect not addressed by the classic methods. As well, we found out, that those methods do not allow to fully test a protocol and its implementation against various network behavior peculiarities.

As an illustrative anecdote, one of the authors was debugging the implementation on an ordinary DSL line, when its uplink losses suddenly went to 10% because of some technical problem on the ISP side. Regular web browsing was unaffected by the issue, as TCP is highly resilient to acknowledgement losses on the reverse (i.e. receiver to sender) path. Thus, the search for a non-existing regression lasted for a day, till the author decided to upload some photos thus uncovering the issue. Informally, a network might be "special" or simply "broken" in many different ways, and we needed a systematic way of checking our code against those peculiarities.

As any change in the code might cause unintended effects in particular network conditions, we needed a fast way of checking those effects to track our progress as well as regressions. While unit tests check for *correctness* of a deterministic result given certain inputs, we needed massive case coverage of network conditions to ensure the results are *acceptable* in every particular case. Ideally, a test run had to be integrated into the regular code-build-test loop, similarly to unit testing.

Thus, we came up with the idea of $O(N^2)$ testing where N real or emulated nodes represent different network conditions (high RTT, jitter, losses, NAT, asymmetry, etc). During one test run, we send traffic flows between every pair of nodes, ideally covering $O(N^2)$ combinations of network conditions, like "RTT and jitter", or "NAT and losses". The testing setup had to run tests and present the resulting statistics comprehensibly and quickly, to allow for repeated testing runs.

II. Manifold test suite

The building of the Manifold testing suite started with the realization that manually testing the code under various network conditions is an extremely cumbersome and error-prone process, as setting up network configurations involves multi-step technical operations, likely spanning several hosts.

Thus, the objective was to implement our $O(N^2)$ testing approach using simple, improvised means, allowing for maximum parallelism, supporting diverse real and emulated setups. The system had to be simple enough and flexible in adapting to conditions, as the testing setups necessarily included diverse existing servers as well as (uniform) clusters.

The resulting suite is a collection of shell scripts intended for use with Linux/Unix test machines. Scripts are launched from a single controlling machine using Secure Shell (`ssh`). Log parsing is done with `perl` scripts, graphs are created with `gnuplot`. Manifold scripts are included in the open source implementation of our protocol [7].

A. General workings

Manifold execution is centered around a "fan-out" shell script named `doall` that opens parallel `ssh` sessions to every server of the testing setup and runs all the necessary commands. Every run involves a sequence of operations, typically `build`, `netem`, `run`, `clean`, all ran by `doall`. For example, `./doall build` will check out a certain version of the code, check for dependencies, build it and do fast unit tests, on every server of the testing setup, in parallel. Individual server quirks are resolved through per-server plug-in extension scripts or environment variable profiles. Typically, in most cases it suffices to adjust environment variables in a profile script (named `env.hostname.sh`). Given a really special platform, an extension script (e.g. `build.hostname.sh`) might override the default process (i.e. `build.default.sh`).

In all scripts, servers are identified with their `ssh` handles (as opposed to hostnames or IP addresses). That extra level of indirection allows to run several testing nodes at the same physical server or to move a node from one server to another.

B. *Traffic manipulation*

Testing the code on real nodes in the wild has its advantages and drawbacks. The main advantage is that the code is tested in a *real* network. The main drawback is that live network conditions are transient and can't be fully reproduced, so different runs may not be comparable. As well, using real setups is expensive. Thus, we developed several test cluster setups using nodes with emulated network conditions.

We added scripts to control traffic conditions using two standard Linux kernel queuing disciplines (qdisc) for network devices, used in our previous work [8] as well. HTB (Hierarchy Token Bucket) [9] provides packet rate control capabilities. It also enables emulating different network conditions for several peers sharing the same physical server. Using Netem [10], we added different packet delay, jitter and loss rules to every HTB class. Egress and ingress packet flows can have different sets of qdisc parameters. Ingress qdiscs are attached to an IFB (Intermediate Functional Block) pseudo-device. Rules are applied based on the UDP port of a packet, as every node occupies a single port.

Thus, HTB/Netem scripts allow to emulate wide range of specific network conditions and to freely mix emulated and real network setups in a single test swarm.

C. *Test swarm setups*

Given N peers running on k servers, we may use different variants of a network topology to put an accent on different aspects of protocol behavior. We considered three types of topologies: swarm (mesh), chain (sequential data relay) and pairwise transfers. They test the code for swarming behavior, robustness, and single-stream performance, respectively.

1) Large swarms: This swarm topology mostly tests the code for general robustness, creating near-real-world swarming download scenarios. The main challenge with this topology was to run bigger swarms (and to process the resulting data). Limits of the swarm size are determined by the number of parallel ssh connections the control machine may start and the maximum number of peers each test server may run without exhausting its resources. (The former limit could be side-stepped by starting parts of a swarm from different control machines). So far, swarms of about thousand peers have been successfully run with one controlling machine (Lenovo T400 laptop with 2GB RAM) and 11-13 servers (Sun Fire X2100 servers with 8GB RAM).

2) Chain tests: Chain tests are mercilessly effective in finding state machine bugs. In a chained setup, each node is only connected to the previous (source) and to the next (sink) node. Thus, the data has to traverse the entire chain sequentially. That topology is the least forgiving with regard to state machine/ congestion control robustness, as a stall or a slowdown in one flow inevitably affects all the nodes further down the chain. That differs drastically from a swarm topology, that may run fairly well with 50% transfers failed, because of its high redundancy. Technically, our chained setup restricts node connections by starting local `iptables` firewalls at every node.

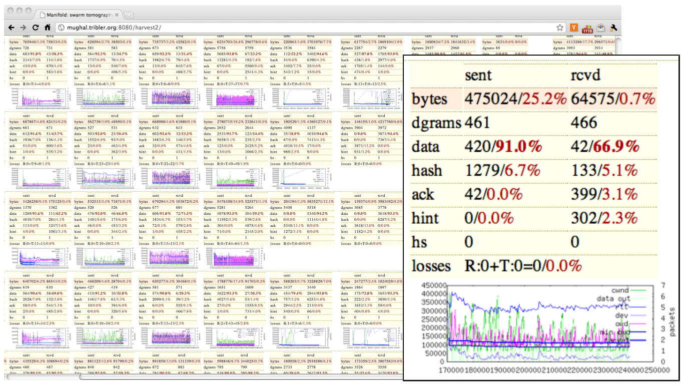
3) Pairwise tests: This setup aims to cleanly test protocol behavior in different network conditions, by eliminating third factors. Namely, with no swarming or data relay, precisely one transfer is done form every node to every other node. This topology puts flows on equal footing as opposed to the swarmed and chained setups, where one transfer typically depends on others. For larger N, it might pose a challenge to run N^2 streams without interference, using N servers. But, in this particular setup, we need just one node to represent one "peculiarity". So, we would not need larger N.

D. *Data harvesting*

Automatic harvesting and analysis of the data turned up to be a major challenge due to the sheer volume of it. While sending or receiving one datagram, a peer generates 10-20 events that are necessary to understand the inner workings of the state machine. A small 10MB transfer requires tens of thousands of datagrams. Given 20-30 peers in an average setup, that results in at least 10^7 log records per a single run, or around 1GB of logs. Not precisely the Google scale, but that data had to be digested and delivered to the user as soon as possible, in a form that allows rapid analysis.

The problem was solved the way it was created. Namely, log processing was implemented to run at the original servers, the controlling machine only left to do one-pass log merge and graph drawing. Thus, data harvesting and analysis was made to scale together with the cluster.

Although the bulk of parsing and statistics is done at the servers, it turned out, that with larger swarms (hundreds of nodes), even maintaining so many parallel `ssh` connections and merging the logs exhausted the control machine. In order to prevent this, we added an option to restrict the maximum number of parallel parsings. Thus, log processing may be done in a sequence of $\sim \frac{N}{k}$ batches, each batch no more than k logs. Since the number of sender-receiver pairs, and thus the number of traffic flows, might be on the order of N^2, the maximum number of running `gnuplot` instances can also be limited.



	sent	rcvd
bytes	475024/25.2%	64575/0.7%
dgrams	461	466
data	420/91.0%	42/66.9%
hash	1279/6.7%	133/5.1%
ack	42/0.0%	399/3.1%
hint	0/0.0%	302/2.3%
hs	0	0
losses	R:0+T:0=0/0.0%	

Fig. 2. The main N by N "harvest" spreadsheet (back) shows the big picture. Each cell (right) provides statistics on a flow.

E. Reports

The resulting reports must allow the user to rapidly examine the test run traces for performance and abnormalities. The top-level report must be simple enough to let the user grok the "big picture" of swarm/flow behavior. Once the user focuses on a particular location or event, it must be easy to switch fast to more detailed data, down to the full event log.

After harvesting and processing the data, Manifold produces an HTML spreadsheet N by N, showing summary stats for every flow, as well as small graphs showing dynamics of flows (Fig. 2). At this point, a user is able to estimate performance and stability of the streams. Closer inspection of every statistics bar reveals stats on message patterns. In case the summary raises some suspicions, the user may navigate to a large detailed version of the graph that gives a good overview of congestion control behavior and network conditions during the lifetime of the flow (see Fig. 1). The graph plots three groups of parameters: time-based (average round trip time, RTT deviation, one-way delay, minimum delay, delay target [6]), packet-based (congestion window, outstanding packets) and events (packet losses, detected by timeout or reordering). This data is sufficient to understand in great detail, how the transfer performed. Once the user is interested in finer details, then the event of interest, its causes and consequences, might be found in the full *all-swarm* event log. The log is primarily analyzed with `grep` and similar custom utilities. The process is helped by the uniform format of log records: (time, node, flow, event, parameters).

As a result, a Manifold user is able to start with a fast qualitative estimation of the swarm and flows, then delve deeper into details as necessary, down to quantitative examination of the log and event-by-event analysis.

III. Conclusion

The Manifold testing approach performs massively parallel $O(N^2)$ case coverage, showering your code with millions and millions of unpredictable state/event combinations. The results often lead to a realization, that your code's performance is never "perfect", but probably it is "good enough" for the current conditions. Despite the fact that Manifold invokes non-trivial computational resources, it still can be used in the routine code-build-test loop of software development. We consider Manifold a useful addition to the standard dumb-bell/parking-lot toolset of network protocol testbeds.

References

[1] V. Jacobson, "TCP extensions for long-delay paths," RFC 1072.

[2] S. Ha, I. Rhee, and L. Xu, "CUBIC: a new TCP-friendly high-speed TCP variant," *SIGOPS Oper. Syst. Rev.*, vol. 42, pp. 64–74, July 2008.

[3] S. Zehl, "The tale of a TCP bug," `http://blogmal.42.org/tidbits/tcp-bug.story`.

[4] A. L. et al, "Towards a common TCP evaluation suite," in *International Workshop on Protocols for Fast Long-Distance Networks*, 2008.

[5] [redacted]

[6] S. Shalunov, "Low extra delay background transport (LEDBAT)," `draft-ietf-ledbat-congestion-04.txt`.

[7] [redacted]

[8] [redacted]

[9] M. Devera, "Htb home," http://luxik.cdi.cz/~devik/qos/htb/.

[10] S. Hemminger, "Network emulation with netem," in *Proceedings of the 6th Australia's National Linux Conference*, April 2005.

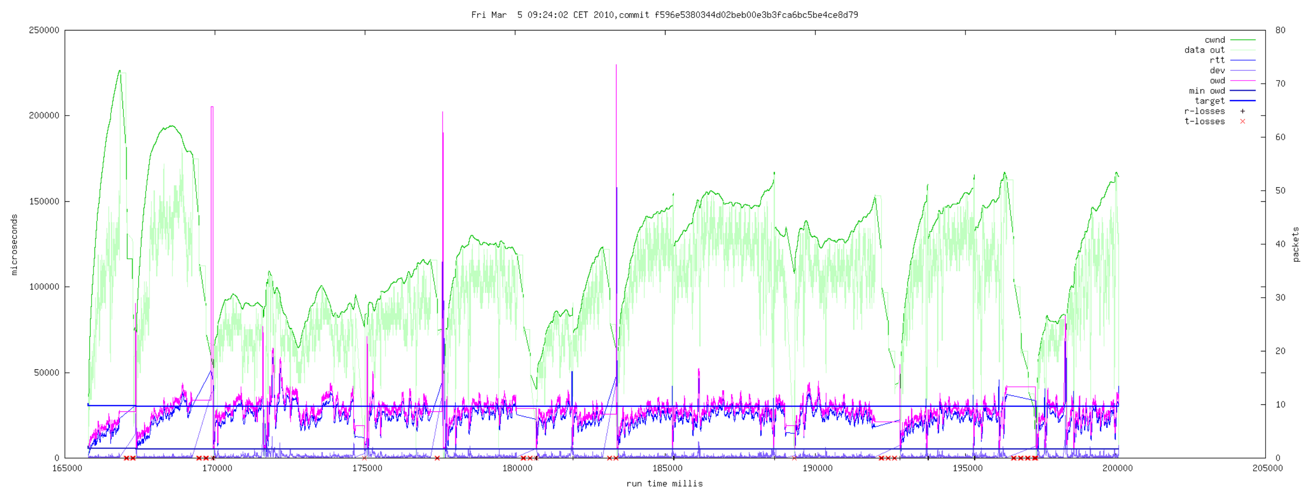


Fig. 1. A detailed graph exposes congestion control history of a flow.