Requirements and Software Design for a Better Tor Performance Measurement Tool

Karsten Loesing, Sathyanarayanan Gunasekaran, and Kevin Butler
karsten@torproject.org,gsathya@torproject.org,kbutler03@qub.ac.uk

Tor Tech Report 2013-10-002
October 30, 2013

1 Introduction

Four years ago, we presented a simple tool to measure performance of the Tor network [1]. This tool, called Torperf, requests static files of three different sizes over the Tor network and logs timestamps of various request substeps. These data turned out to be quite useful to observe user-perceived network performance over time.[1] However, static file downloads are not the typical use case of a user browsing the web using Tor, so absolute numbers are not very meaningful. Also, Torperf consists of a bunch of shell scripts which makes it neither very user-friendly to set up and run, nor extensible to cover new use cases.

For reference, we made an earlier approach 1.5 years later that suggested redesigning the Python parts in Torperf, but that redesign never happened.[2]

In this report we outline requirements and a software design for a rewrite of Torperf. We loosely start with non-functional requirements, so aspects like user-friendliness or extensibility, because these requirements drive the rewrite more than the immediate need for new features. After that, we discuss actual functional requirements, so experiments that we want to perform to measure things in the Tor network on a regular and automated basis. Finally we suggest a software design that fulfills all these requirements.

2 Requirements

Before we talk about actual experiments the new Torperf is supposed to perform, we want to discuss more general requirements of a Torperf rewrite. For the time being, assume that the major functional requirement is to "make client requests over the Tor network and record timestamps and other meta data for later analysis." Whichever type of request this is, it's important that all or most of the non-functional requirements listed in the following are met. We start with configuration requirements, followed by requirements to results formats.

[1] https://metrics.torproject.org/performance.html
[2] https://trac.torproject.org/projects/tor/ticket/2565

2.1 Configuration requirements

2.1.1 Installation and upgrade

Whoever installs Torperf shouldn't be required to understand its codebase, nor rely on support by someone who does.[3] Ideally, the new Torperf comes as OS package, meaning that it only relies on third-party software that is also either available as OS package or that is shipped with the Torperf package. To put it simply, installing on Debian Wheezy should be as easy as `apt-get install`, and upgrading should simply be an `apt-get update && apt-get upgrade` away.

2.1.2 Run as service

The new Torperf should run as OS service, unrelated to a specific user. It should have a config file that is easy to understand, and it should come with scripts to start, restart, and stop the Torperf service. If the service operator wants to run multiple experiments in parallel, they'd simply configure all those experiments in the configuration file or directory, rather than starting multiple instances of Torperf.

2.1.3 Single configuration point

All requests that Torperf performs use a local Tor client, go over the Tor network, and are answered by some server. Some experiments may be designed to use remote servers not controlled by the person running Torperf, but others require setting up a custom server. In the latter case, configuring, starting, restarting, or stopping client and server should happen in a single place, that is, on a single machine as part of the Torperf service. Fortunately, running client and server on the same physical should not have any effect on measurement results, because all requests and responses traverse the Tor network.

Reasons for *not* separating client and server are:

- it's easier to deploy client and server together than deploying them separately and configuring the client to use the server;
- measurement results can easily contain client and server timestamps of a measurement;
- assuming the host has sufficient free bandwidth capacity, measurement results shouldn't be affected by client and server running on the same host; and
- it's not unusual to deploy client and server talking over Tor in a single tool: think of how a tor relay does bandwidth self-tests to itself.

What's not easily possible with this approach is to configure client and server to run on different hosts. It's unclear yet whether there is an experiment where this might be useful.

[3] This sounds obvious, but this was the case with the current, shell script based Torperf, and it's also the reason why its current known userbase is 2.

2.1.4 User-defined tor version or binary

A key part of measurements is the tor software version or binary used to make requests or even to handle requests in the hidden-service case. It should be easy to specify a tor version that is not the current tor version shipped with the OS. Similarly, it should be easy to point to a custom tor binary to use for measurements.

It should be possible to run different experiments with different tor versions or binaries in the same Torperf service instance.[4] An experiment should be able to be configured to run for multiple versions of tor, potentially simultaneously, to spot performance change across tor releases. Note that the tor version should be contained in the results.

It might be beneficial to provide a mechanism to download and verify the signature of new tor versions as they are released. The user could specify if they plan to test stable, beta, or alpha versions of tor with their Torperf instance.

This requirement should be implemented in three phases: 1) Torperf uses the default tor binary that comes with the operating system; 2) Torperf accepts the path to a tor binary that was previously built by the user; 3) Torperf accepts the path to a Git tag or tor source directory and downloads, verifies, and builds a tor binary itself.

2.1.5 User-defined third-party software version or binary

Similar to the previous requirement, it should be easy to specify custom versions or binaries of third-party software other than the version currently shipped with the OS. This applies, for example, to Firefox when attempting to make measurements more realistic.

2.2 Results requirements

2.2.1 Results format

The measurement results produced by Torperf contain no sensitive data by design, because all requests are made by Torperf itself, not by actual Tor users. We'll want to collect as much information as needed to perform useful analysis. But at the same time we want to keep it as easy as possible to process results.

Results may come from various sources, e.g., an HTTP/SOCKS client, Selenium[5]/Firefox, a tor client used to make the request or to answer the request as hidden service, an HTTP server, etc. Torperf should not store original logs, because that would only shift the issue of processing different log formats to the analysis step. Such an approach would also generate unnecessarily large results files. Torperf should rather process data from these sources and store them in a custom results format that can be easily processed using tools shipped with Torperf.

Results may include data which appear not immediately relevant to measuring Tor performance, but which may be useful for related purposes. For example, Torperf should include data about circuit failures in its results, even though these circuits may not have been used in actual requests.[6]

[4] Loosely specified tor versions, e.g. 'latest-stable, git:master, $\geq$ 0.2.4' (following a well-defined schema), would be a boon in reducing error prone configuration updates everytime there are new tor versions released.

[5] http://www.seleniumhq.org/

[6] https://trac.torproject.org/projects/tor/ticket/8662

Deciding which data to store should be the responsibility of whoever designs a Torperf experiment, though formats should be somewhat uniform between experiments. Torperf should store its results in a format that is widely used and already has libraries (like JSON), so that other applications can use the results and build on it.[7]

2.2.2 Results web interface

Torperf should provide all its results via a public web interface.[8] This includes measured timestamps as well as any meta data collected in an experiment. Results should be available for all measurements as well as for a subset specified by measurement time or filtered by other criteria. The interface should highlight experiment failures.

2.2.3 Results accumulator

A Torperf service instance should be able to accumulate results from its own experiments and remote Torperf service instances. Therefore, it should simply download new results from their web interfaces and incorporate them in its own database. It would be useful to have the accumulator warn if a remote service instance becomes stale and doesn't serve recent results anymore. Note how the service instance name or identifier should become part of Torperf results meta data. The accumulating Torperf instance must verify that all requested data contains the expected Torperf service identifier and otherwise discard the received data.

2.2.4 Truncate old results

A Torperf service instance should be able to limit space requirements for storing past results. This could be done by discarding results that are older than a configured number of days.

2.2.5 Results graph data

Ideally, Torperf provides aggregate statistics via its web interface which can then be visualized by others. Even more ideally, Torperf serves a few HTML pages containing the necessary JavaScript to visualize results. While this may sound like going overboard here, making Torperf the tool to run performance measurements *and* to present results has the advantage of not building and maintaining a separate tool for the latter.[9]

[7] In particular, we could use a de-facto standard like the HAR (HTTP Archive) format to store measurements results. Advantages are that such a format is widely used and that there are a lot of off-the-shelve tools that can process this format. Disadvantages are that such a format captures a lot of meta data that are not relevant to us, and that we'd have to add tor-specific data fields to the format which wouldn't be processed by existing tools. Note that we could use such a format only for request data, not for tor-specific data like building circuits and attaching streams.

[8] It is a requirement to also provide a machine readable representation to enable easier development of custom visualisations of the result data.

[9] For reference, the current Torperf produces measurement results which are re-formatted by metrics-db and visualized by metrics-web with help of metrics-lib. Any change to Torperf triggers subsequent changes to the other three codebases, which is suboptimal.

2.2.6 Results parsing library

The new Torperf should come with an easy-to-use library to process its results. The library should also make it simple to create Torperf compatible results or clearly highlight areas of non conformance with existing results. Alternatively, this library could be provided and maintained as part of stem[10] or another parsing library.

3 Experiments

3.1 High priority experiments

3.1.1 Alexa top-X websites using Selenium and the Tor Browser

One major reason for rewriting Torperf is to make its results more realistic.[11] It was suggested to track down Will Scott's torperf-like scripts, make them public if needed, and do a trial deployment somewhere.[12] An experiment making these more realistic measurements should use something like Selenium to control an actual Tor Browser to make requests.[13] As a variant, this experiment should be run with other Firefox versions, e.g. that one that uses optimistic data.[14]

3.1.2 Static file downloads

Another major reason for rewriting Torperf is to supersede the existing, bit-rotting codebase. The new Torperf should therefore provide an experiment that is identical to the current, single Torperf experiment: download static files of sizes 50 KiB, 1 MiB, and 5 MiB over the Tor network and record timestamps and relevant meta data.

There are quite a few details in getting these downloads right, which shall not all be specified here. For example, Torperf needs to enforce using a fresh circuit for each run, which is currently ensured by reducing the maximum circuit dirtiness to a value that is lower than the experiment period. An alternative may be to send the `NEWNYM` signal to the tor process after every stream.[15] The easiest path might be to leverage on the stream isolation features introduced in tor 0.2.3[16] and use a different SOCKS username/password for each requests.

The best way to extract these requirements is to read the source.[17] There also exists a proof-of-concept implementation of this experiment in Twisted which can and should serve as

[10] https://stem.torproject.org/

[11] https://trac.torproject.org/projects/tor/ticket/7168

[12] https://trac.torproject.org/projects/tor/ticket/7516

[13] Related to this, Zack Weinberg has written code that downloads the present Tor Browser alpha and Python Selenium client and monkey-patches them to play nice together; see https://github.com/zackw/tailwagger. This code is very specialized for his requirements right now, but might still be of use to us. We should look especially at what is done to TBB and python-selenium by the patches under pkg/tor-browser-3.0a3 and pkg/python-selenium-2.33.0, and the TbbDriver class in client/tbbselenium.py.

[14] https://trac.torproject.org/projects/tor/ticket/3875

[15] https://trac.torproject.org/projects/tor/ticket/2766

[16] https://gitweb.torproject.org/torspec.git/blob/HEAD:/proposals/171-separate-streams.txt#l93

[17] https://gitweb.torproject.org/torperf.git/tree

starting point for this rewrite.[18]

The results format of the current Torperf codebase shall serve as blueprint for designing a new results format.[19] As stated before, the new results format should make use of JSON or another data format, so that results can be processed more easily.

Results may even contain more information than the original Torperf results. For example, assuming that the new Torperf service controls both HTTP client and server, new timestamps could be added for serving the first and last byte. Matching timestamps between client and server can be achieved by serving new random content in each request and use this content (or a digest thereof) as request identifier.

3.2 Lower priority experiments

3.2.1 Canonical median web page

A lower-priority experiment would be to devise and deploy the canonical median web page.[20] Such an experiment would share a lot with static file downloads, but would serve an actual web page using Torperf's own webserver.

3.2.2 Hidden service performance

Another possible experiment would be to measure performance to a hidden service.[21] This hidden service could easily run on the same tor client instance, or on a separate tor client instance running on the same physical host. We'll want to add additional timestamps for hidden service events.[22] The hidden service could serve static files, a canonical median web page, or even an Alexa top-X website.

The latter has the minor disadvantage that server timestamps cannot be matched with client timestamps based on content, but only probabilistically via timing. This may become problematic when running lots and lots of experiments at the same time. Or maybe it's possible to match client and server observations using identifiers, like the rendezvous cookie, exchanged in the rendezvous process instead.

3.2.3 GET POST performance

The experiments so far all measured download speed, but it's also easy to measure upload speed.[23] Torperf's own web server could accept POST data and measure timestamp of first byte received, last byte received, etc. Similar to static file downloads, client and server timestamps can be matched based on the random data that is posted to the server, or a digest of that data.

Ideally such upload tests should also be done using a full web browser driven by Selenium.

[18] https://gitweb.torproject.org/karsten/torperf.git/tree/refs/heads/perfd
[19] https://metrics.torproject.org/formats.html#torperf
[20] https://trac.torproject.org/projects/tor/ticket/7517
[21] https://trac.torproject.org/projects/tor/ticket/1944
[22] https://trac.torproject.org/projects/tor/ticket/2554
[23] https://trac.torproject.org/projects/tor/ticket/7010

4 Software design

The previous sections outlined requirements to the Torperf rewrite and possible experiments, unrelated to an actual implementation. The purpose of this section is to suggest a possible software design matching these requirements. It's quite possible that there are better software designs. This section should serve as starting point for a discussion.

We have an initial proof-of-concept prototype that uses Twisted to implement a small subset of the new Torperf. Even though the new Torperf does quite a few things at once, like starting tor processes, making requests, answering requests, collecting results, etc., it can be implemented as a single Twisted application. This shifts some deployment problems to what people usually do when deploying Twisted applications, rather than forcing us to solve these problems yet another time. Of course, at the same time it forces us to follow the Twisted model of writing applications, which seems not too bad in this case. The current prototype also requires twisted-socks[24] as SOCKS client, txtorcon[25] for communicating with tor clients, and stem for event parsing.

The following list outlines tasks that the new Torperf needs to perform. These could be implemented as Python classes, though this list was not written with Python or Twisted specifics in mind:[26]

configuration handler Validate and parse configuration file or directory, provide configuration values to other application parts.

logger Log operation details for debugging purposes and for normal service operation, unrelated to storing measurement results.

tor process starter Configure, start, hup, and stop local tor client processes, using a previously configured tor version or binary.

tor controller Connect to tor's control port, force-set guards if required by the experiment, register for and handle asynchronous events, set up and tear down hidden services.

experiment scheduler Run experiments following a schedule.

experiment runner Handle the execution of a scheduled experiment. It should take care of the entire experiments lifecycle and reserve an exclusive tor instance for its lifetime. Finally, collect the results and post-process them (if applicable) before saving the data to the results data store. If the experiment fails it should be possible to track down the reason for failure from looking at the experiment results.

request scheduler Start new requests following a previously configured schedule.

request runner Handle a single request from creation over various possible sub states to timeout, failure, or completion.

[24] https://github.com/ln5/twisted-socks

[25] https://github.com/meejah/txtorcon

[26] An alternative to Python/Twisted might be Go, though it doesn't have tor controller libs like txtorcon or stem.

HTTP/SOCKS client Make HTTP GET requests using our own SOCKS client and connecting to a local tor client to gather as many timestamps of substeps as possible.

Selenium/Firefox wrapper Make web request using Selenium/Firefox, possibly using Xvfb and using a previously configured Firefox version or binary, and collect as many timestamps of substeps as possible.

SOCKS or HTTP proxy Capture even more timestamps by proxying requests made via Selenium/Firefox through our own SOCKS or HTTP proxy before handing them to the local tor client.

HTTP server Run on port 80, serve static files and the canonical median weg page, accept POST requests, provide measurement results via RESTful API, present measurement results on web page.

Alexa top-X web pages updater Periodically retrieve list of top-X web pages.

results data store Store request details, retrieve results, periodically delete old results if configured, possibly using a database.

results accumulator Periodically collect results from other Torperf instances, warn if they're out of date.

analysis scripts Command-line tools to process measurement results and produce graphs and other aggregate statistics. Could also become part of stem instead, together with a tutorial.

Acknowledgments

Lunar, Zack Weinberg, and Rob van der Hoeven provided valuable feedback for this report.

References

[1] Karsten Loesing. Performance of requests over the Tor network. Technical Report 2009-09-001, The Tor Project, September 2009.

A Data formats

In this section we give examples of the suggested data formats to be produced by Torperf.

A.1 File download

A file download document contains all data and meta data from making a static file download over Tor. It will only be used by the experiment specified in 3.1.2. In the following, we give a file download example document and define its fields:

```
{
  "type": "file_download",
```
Document type string; always `"file_download"` for this document type; required.

```
  "guid": "ec2_9001_1365473921.75",
```
Globally unique identifier for a file download document, consisting of service identifier, tor SOCKS port, and request start time; required.

```
  "source": "ec2",
```
Configured service identifier; required.

```
  "socks": 9001,
```
Configured tor SOCKS port; required.

```
  "filesize": 51200,
```
Configured file size in bytes; required.

```
  "tor_version": "0.2.2.35",
```
Tor software version; optional.

```
  "start": 1365473921.75,
```
Time when the connection process starts; required.

```
  "socket": 1365473921.75,
```
Time when the socket was created; required.

```
  "connect": 1365473921.75,
```
Time when the socket was connected; required.

```
  "negotiate": 1365473921.75,
```
Time when SOCKS 5 authentication methods have been negotiated; required.

```
  "request": 1365473921.75,
```
Time when the SOCKS request was sent; required.

```
  "response": 1365473922.07,
```
Time when the SOCKS response was received; required.

```
  "datarequest": 1365473922.07,
```
Time when the HTTP request was written; required.

```
  "dataresponse": 1365473922.40,
```
Time when the first response was received; required.

```
  "datacomplete": 1365473922.76,
```
Time when the payload was complete; required.

```
  "writebytes": 82,
```
Total number of bytes written; required.

```
  "readbytes": 51323,
```
Total number of bytes read; required.

```
  "didtimeout": False,
```
True if the request timed out, False otherwise; optional.

```
  "dataperc": {
```
Time when x% of expected bytes were read for $x = \{$ 10, 20, 30, 40, 50, 60, 70, 80, 90 $\}$; optional.

```
    "10": 1365473922.56,
    "20": 1365473922.64,
    "30": 1365473922.64,
    "40": 1365473922.68,
    "50": 1365473922.68,
    "60": 1365473922.72,
    "70": 1365473922.72,
```

```
    "80": 1365473922.75,
    "90": 1365473922.75 },
  "stream_guid":
    "ec2_9001_1365473922.76_884"
}
```

Global unique identifier of the stream used for this measurement; optional.

A.2 Stream

A stream document contains all data and meta data from opening a stream, attaching it to a circuit, maybe detaching and re-attaching it, to closing it. Any experiment can generate stream documents for the streams created by its tor process. In the following, we give a stream example document and define its fields:

```
{
  "type": "stream",
  "guid": "ec2_9001_1365473922.76_884",
  "source": "ec2",
  "socks": 9001,
  "create": 1365473922.76,
  "stream_id": 884,
  "circuits": [
    { "circuit_guid":
        "ec2_9001_1365473682.13_501",
      "reason": "END",
      "remote_reason": "DONE" } ]
}
```

Document type string; always `"stream"` for this document type; required.

Globally unique identifier for a stream document, consisting of service identifier, tor SOCKS port, and stream creation time, and locally unique stream identifier; required.

Configured service identifier; required.

Configured tor SOCKS port; required.

Time when the stream was created; required.

Locally unique stream identifier; required.

Sequence of circuits that the stream was attached to, each identified by its globally unique identifier and optionally containing reason and remote reason for detaching; optional.

A.3 Circuit

A circuit document contains all data and meta data about launching and creating a circuit. Any experiment can generate circuit documents for the circuits built by its tor process. In the following, we give a circuit example document and define its fields:

```
{
  "type": "circuit",
```
Document type string; always `"circuit"` for this document type; required.

```
  "guid": "ec2_9001_1365473682.13_501",
```
Globally unique identifier for a circuit document, consisting of service identifier, tor SOCKS port, circuit launch time, and locally unique circuit identifier; required.

```
  "source": "ec2",
```
Configured service identifier; required.

```
  "socks": 9001,
```
Configured tor SOCKS port; required.

```
  "launch": 1365473682.13,
```
Time when the circuit was launched; required.

```
  "circ_id": 501,
```
Locally unique circuit identifier; required.

```
  "path": [
```
Sequence of relays chosen for the circuit including build times since launching the circuit; optional.

```
    { "fingerprint":
        "$62680CF0743460E5F836F949E37A6DEC22622F9E",
      "buildtime": 0.53 },
    { "fingerprint":
        "$11064D066F892DC38AAEFDA5EDAE1A227D07D182",
      "buildtime": 1.10 },
    { "fingerprint":
        "$35F51DCE73B988CBAE06B06312B4D1271979FE3B",
      "buildtime": 1.32 } ],
  "buildtimeout": 4.83,
```
Circuit build timeout used when building this circuit; optional.

```
  "quantile": 0.800000
```
Circuit build time quantile used to determine the circuit build timeout; optional.

```
}
```