Tor Metrics Data Collection, Aggregation, and Presentation

Iain R. Learmonth
The Tor Project
irl@torproject.org

Karsten Loesing
The Tor Project
karsten@torproject.org

Tor Tech Report 2019-03-001
March 25, 2019

Abstract

Tor Metrics is the central mechanism that The Tor Project uses to evaluate the functionality and ongoing relevance of its technologies as Internet freedom software. Tor Metrics consists of several services that work together to collect, aggregate, and present data from the Tor network and related services. The first section of this report gives a high level overview of the software behind these services. Sections 2 to 8 of this report give an overview of the codebases that make up the software. Section 9 then briefly compares our approach to collect, aggregate, and present data with OONI's data pipeline. Finally, this report makes recommendations for next steps to be taken for improving the existing code that makes up Tor Metrics which has now been evolving for more than ten years. Intended audiences of this report include current and prospective contributors to Tor Metrics codebases as well as other Internet freedom projects that face the same challenges while collecting metrics on their products.

1 Overview of the software behind Tor Metrics

As of August 2017, all user-facing Tor Metrics content has moved (back) to the Tor Metrics website. The main reason for gathering everything related to Tor Metrics on a single website is usability. In the background, however, there are several services distributed over a dozen hosts that together collect, aggregate, and present the data on the Tor Metrics website.

Almost all Tor Metrics codebases are written using Java, although there is also R, SQL and Python. In the future we expect to see more Python code, although Java is still popular with

This work was supported by Open Technology Fund under contract number 1002-2017-018, and by NSF grants CNS-1526306 and CNS-1619454. Support does not imply endorsement. With thanks to the OONI team for their assistance in completing a comparison between our two data pipelines, and thanks to Ana Custura for her contributions to the OnionPerf section of this report.

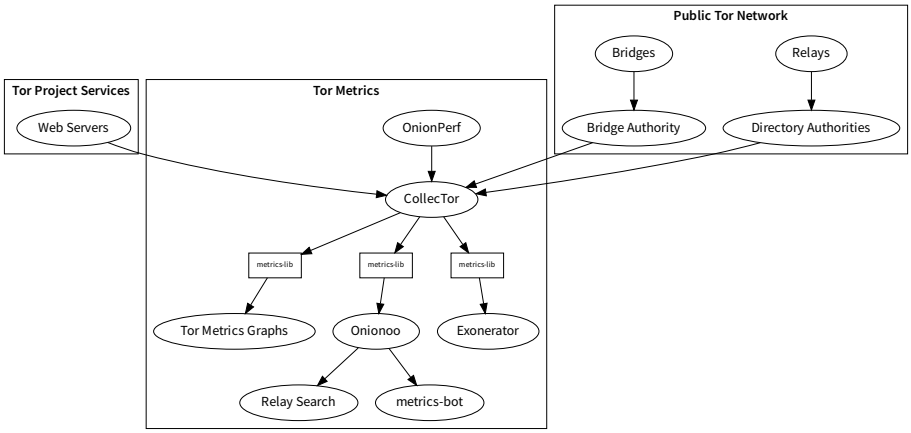

Figure 1: The Tor Metrics Data Collection, Analysis and Visualisation Pipeline

academics and we would like to continue supporting easy access to our data for those that want to use Java even if we are using it less ourselves.

Tor relays and bridges collect aggregated statistics about their usage including bandwidth and connecting clients per country. Source aggregation is used to protect the privacy of connecting users—discarding IP addresses and only reporting country information from a local database mapping IP address ranges to countries. These statistics are sent periodically to the directory authorities.

CollecTor downloads the latest server descriptors, extra info descriptors containing the aggregated statistics, and consensus documents from the directory authorities and archives them. This archive is public and the metrics-lib Java library can be used to parse the contents of the archive to perform analysis of the data.

In order to provide easy access to visualizations of the historical data archived, the Tor Metrics website contains a number of customizable plots to show user, traffic, relay, bridge, and application download statistics over a requested time period and filtered to a particular country.

In order to provide easy access to current information about the public Tor network, Onionoo implements a protocol to serve JSON documents over HTTP that can be consumed by applications that would like to display information about relays along with historical bandwidth, uptime, and consensus weight information.

An example of one such application is Relay Search which is used by relay operators, those monitoring the health of the network, and developers of software using the Tor network. Another example of such an application is metrics-bot which posts regular snapshots to Twitter and Mastodon including country statistics and a world map plotting known relays.

Figure 1 shows how data is collected, archived, analyzed, and presented to users through services operated by Tor Metrics. The majority of our services use metrics-lib to parse the descriptors that have been collected by CollecTor as their source of raw data about the public Tor network.

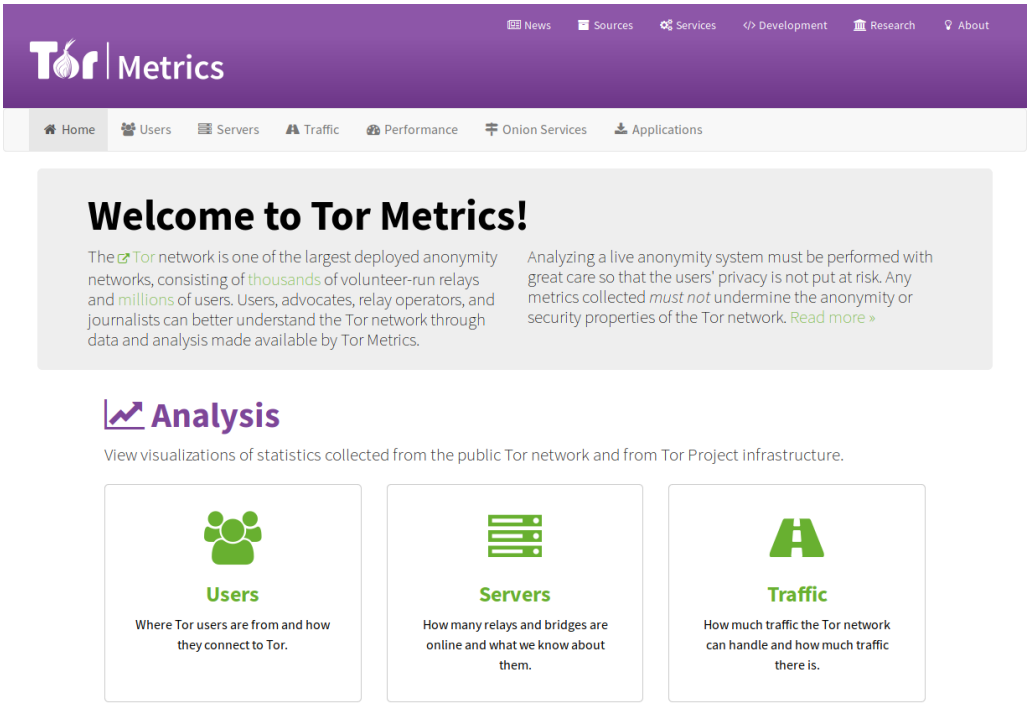

Figure 2: Screenshot of the homepage of the Tor Metrics website.

2 Tor Metrics Website

This is the primary point of contact for users that would like to learn more about the public Tor network. Figure 2 shows a screenshot of the homepage. The diagram in Figure 3 shows how information is arranged on the Tor Metrics website.

The Tor Metrics website itself is written in Java servlets and JavaServer Pages (JSP). The website is deployed as WAR file and contains an embedded Jetty server. The relevant code can be found in the metrics-web repository in the Java package `org.torproject.metrics.web`. Approximate lines of code count per programming language in the metrics-web source control repository for common parts can be found in table 1.

The "Analysis" section of the Tor Metrics website contains visualisations of recent and historical data about the Tor Network. These graphs are produced by R using the various tidyverse[1] packages like ggplot2, tidyr, dplyr, and somewhat more recently readr.

We have an Rserve[2] process running on the machine running the Tor Metrics website which accepts local connections with R function calls and as a result writes graphs and CSV files to disk. This Rserve process is started by cron on reboot, reads a file with all graphing code, and then listens for requests from the Java application. See `src/main/R/rserver/rserve-init.R` in the metrics-web repository for the relevant graphing code.

In Java, we have a fair amount of wrapping code for our Rserve process. `GraphImageServlet` accepts requests, `GraphParameterChecker` parses the parameters, and `RObjectGenerator` then

[1] https://www.tidyverse.org/
[2] https://www.rforge.net/Rserve/

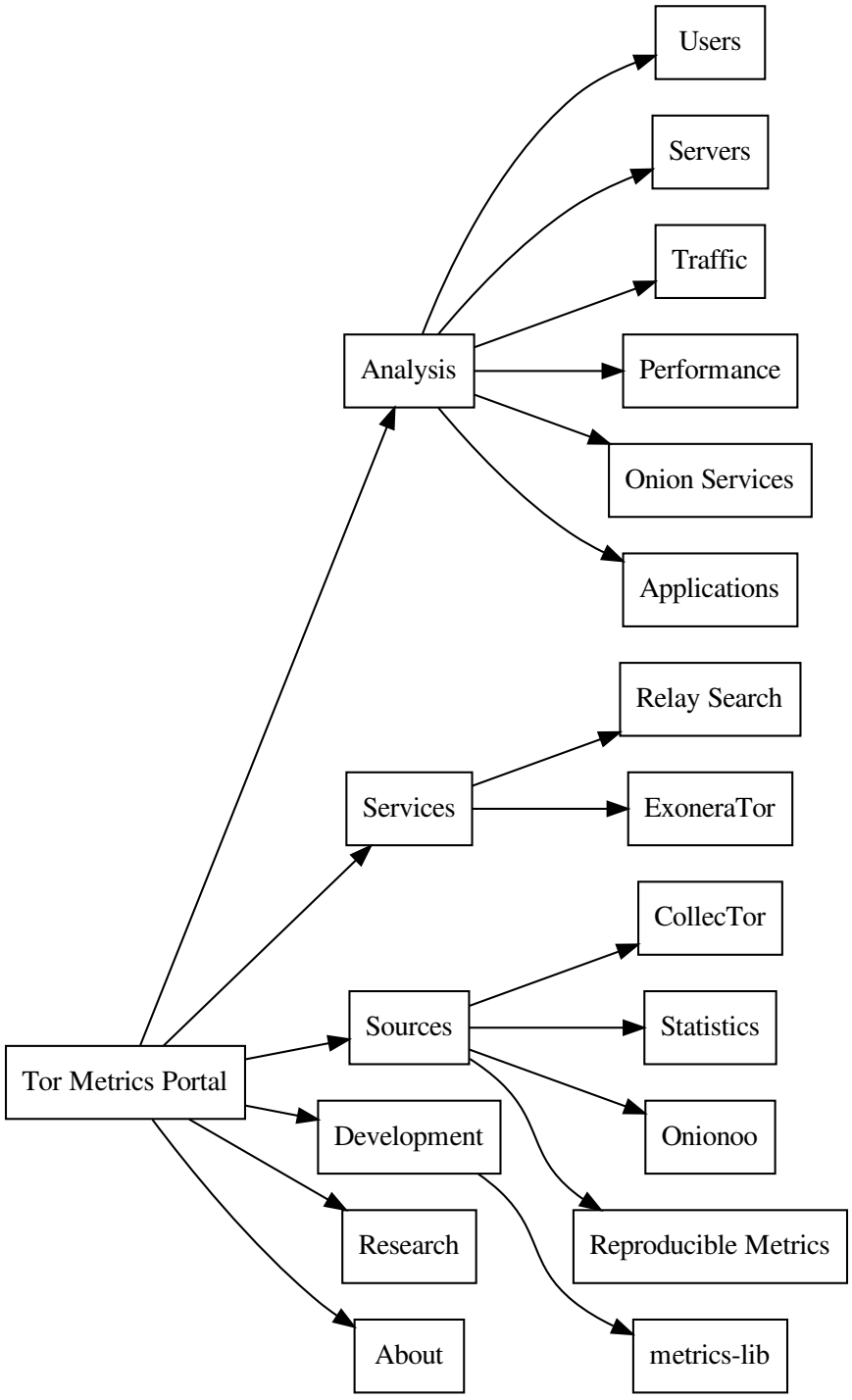

Figure 3: Information Architecture for the Tor Metrics Portal. External information is not shown.

Language	files	code
XSLT	1	9064
JSP	28	7036
JSON	3	4497
Java	39	2444
JavaScript	14	1513
R	3	1510
XML	6	1464
HTML	5	820
CSS	6	736
DTD	1	170
awk	1	33
Bourne Shell	4	9

Table 1: Approximate lines of code count per programming language in the metrics-web source control repository for common parts.

communicates with the Rserve process.

The data to be graphed is written to a shared directory which the Rserve process uses to make its graphs. That shared directory is updated at the end of the daily data-processing modules run. The Rserve process reads the CSV files contained in that shared directory for each new graph it plots, which works relatively fast with the readr package.

We implemented a simple caching mechanism in RObjectGenerator. If the graph or CSV file that we're asked to return already exists and is not too old then that will be returned rather than asking Rserve to produce a new file. Unfortunately, we never implemented a cache cleaner, which is why the directory of generated files grows forever. Fortunately, generated files are pretty small, so this has never caused issues so far.

The modules that produce the data for these visualisations are described in subsections 2.1—2.8. The approximate lines of code count per programming language in the metrics-web source control repository for each module is shown in table 3. Each of these modules has its own working storage and some also have databases. The current size of these is listed in table 2 and illustrated in figure 4.

The "Services" section of the Tor Metrics website embeds external code to provide query interfaces to Tor Metrics data. This includes Relay Search, described in section 6, and ExoneraTor, described in section 5.

In the "Sources" section there is an interface to browse the raw data available in the "CollecTor" service which is described in section 3. The remaining pages and sections are static content, included either directly or using light wrapping for theming purposes with JSPs.

Component	Storage Used
common filesystem	653 MB
connbidirect filesystem	11 MB
onionperf filesystem	3.7 MB
onionperf database	1332 MB
servers filesystem	30 MB
servers database	4299 MB
advbwdist filesystem	370 MB
hidserv filesystem	4.7 GB
clients filesystem	399 MB
clients database	101 GB
ipv6servers filesystem	1.9 MB
ipv6servers database	10 GB
webstats filesystem	14 MB
webstats database	6381 MB

Table 2: The storage space consumed by the component modules of metrics-web. This is illustrated in figure 4.

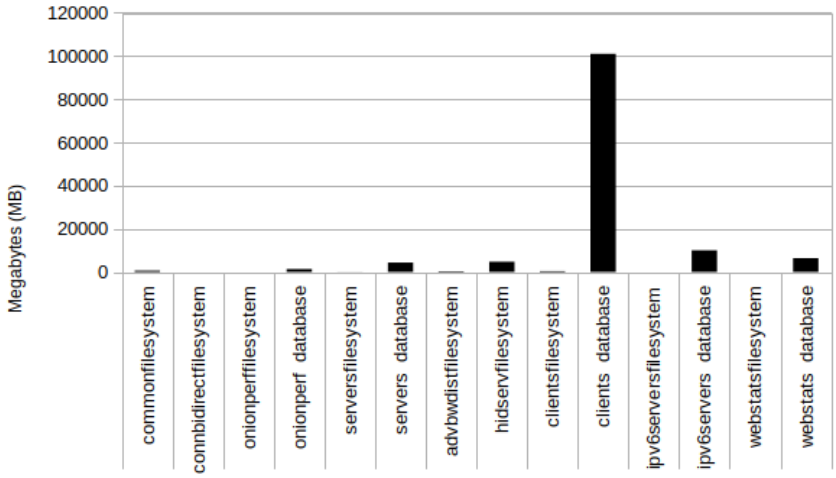

Figure 4: Bar chart showing the storage space consumed by the component modules of metrics-web. The raw values for this chart can be found in table 2.

2.1 metrics-web connbidirect module

This module aggregates data from extra-info descriptors, which are collected by the relaydescs module of CollecTor, to find the fraction of direct connections between a relay and other nodes in the network that are used uni- or bi-directionally.

Every 10 seconds, relays determine for every direct connection whether they read and wrote less than a threshold of 20 KiB. Connections below this threshold are excluded from the graph. For the remaining connections, relays determine whether they read/wrote at least 10 times as many bytes as they wrote/read. If so, they classify a connection as "Mostly reading" or "Mostly writing", respectively.

All other connections are classified as "Both reading and writing". After classifying connections, read and write counters are reset for the next 10-second interval. The aggregate data contains daily medians and inter-quartile ranges of reported fractions.

2.2 metrics-web onionperf module

This module aggregates data from OnionPerf instances, which are collected by the onionperf module of CollecTor, to find overall performance when downloading static files of different sizes over Tor, either from a server on the public Internet or from a version 2 onion server.

The data shows the range of measurements from first to third quartile, and highlights the median. The slowest and fastest quarter of measurements are omitted from the data.

Circuit build times and end-to-end latency are also calculated from the same OnionPerf measurements.

2.3 metrics-web legacy/servers module

This module is the oldest data-processing module in metrics-web. It imports various contents from relay and bridge descriptors into a database called tordir. Once the import is done, the module runs an SQL function to produce daily aggregates on the number of relays and bridges as well as advertised and consumed bandwidth.

The initial purpose of this database was to contain all relevant Tor network data for all kinds of Tor Metrics services, including an earlier version of Relay Search. However, this approach of using one database for everything did not scale. When the database grew too big, that Relay Search service has been stopped, and the import tables have been changed to only contain the last two weeks of data.

2.4 metrics-web advbwdist module

This module reads contents from network status consensuses and relay server descriptors to provide the data for two graphs: Advertised bandwidth distribution and Advertised bandwidth of n-th fastest relays, which display advertised bandwidth percentiles and ranks, respectively.

This module was created out of a one-off analysis of advertised bandwidth distribution. The main difficulty in providing this data is that advertised bandwidths can only be found in server descriptors, but consensuses are required to obtain the set of relays running at a given time. This module aims to provide this data without keeping a huge database by processing all server

descriptors published in the last 72 hours together with all newly published consensuses. This approach is different from all other modules which do not dictate an order in which descriptors are imported nor that recent server descriptors need to be re-imported over and over until they are older than 72 hours.

2.5 metrics-web hidserv module

This module aggregates data from extra-info descriptors, which are collected by the relaydescs module of CollecTor, to find the number of unique .onion addresses for version 2 onion services in the network per day. These numbers are extrapolated from aggregated statistics on unique version 2 .onion addresses reported by single relays acting as onion-service directories, if at least 1% of relays reported these statistics [2].

This module also uses the same extra-info descriptors to find the amount of onion-service traffic from version 2 and version 3 onion services in the network per day. This number is extrapolated from aggregated statistics on onion-service traffic reported by single relays acting as rendezvous points for version 2 and 3 onion services, if at least 1% of relays reported these statistics.

2.6 metrics-web clients module

This module aggregates data from extra-info descriptors, which are collected by the relaydescs module of CollecTor, to find the estimated number of directly-connecting clients; that is, it excludes clients connecting via bridges. These estimates are derived from the number of directory requests counted on directory authorities and mirrors.

Relays resolve client IP addresses to country codes, so that graphs are available for most countries. Furthermore, it is possible to display indications of censorship events as obtained from an anomaly-based censorship-detection system [1].

This module also aggregates data from extra-info descriptors from bridges, which are collected by the bridgedescs module of CollecTor, to find the estimated number of clients connecting via bridges. These numbers are derived from directory requests counted on bridges. Bridges resolve client IP addresses of incoming directory requests to country codes, so that graphs are also available for most countries.

2.7 metrics-web ipv6servers module

This module imports relevant parts from server descriptors and network statuses into a database and exports aggregate statistics on IPv6 support to a CSV file. Descriptors include relay and bridge descriptors.

The ipv6servers database contains import tables as well as tables with pre-aggregated data. The former contain all newly imported descriptor details, whereas the latter contain aggregates like the number of relays or bridges and total advertised bandwidth by every possible combination of relevant relay or bridge attributes. This approach reduces database size by a lot, but it also makes it difficult to add new relay or bridge attributes later on. In such a case, past data would have to be re-imported into the database.

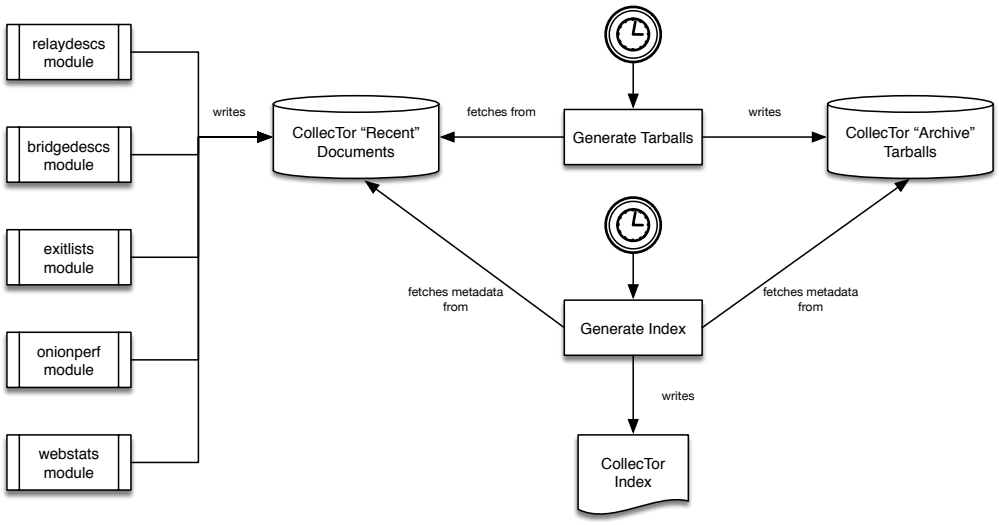

Figure 5: Data flow in the CollecTor application.

2.8 metrics-web webstats module

This module aggregates web server logs, collected by the webstats module of Collector. All data comes from logs which are provided in a stripped-down version of Apache's "combined" log format without IP addresses, log times, HTTP parameters, referers, and user agent strings.

This module finds absolute numbers of requests to Tor's web servers related to Tor Browser. Initial downloads and signature downloads are requests made by the user to download a Tor Browser executable or a corresponding signature file from the Tor website. Update pings and update requests are requests made by Tor Browser to check whether a newer version is available or to download a newer version. Data is also aggregated by platform and locale.

3 CollecTor

CollecTor fetches data from various nodes and services in the public Tor network and makes it available to the world. Descriptors are available in two different file formats: recent descriptors that were published in the last 72 hours are available as plain text, and archived descriptors covering over 10 years of Tor network history are available as compressed tarballs. Index files are also created that contain a machine-readable representation of all descriptor files available.

An illustration of data flow within the CollecTor application can be found in figure 5. The individual modules are described in their relevant sub-section below.

3.1 CollecTor common parts

CollecTor modules are all managed as part of a single Java application. This application uses `java.util.concurrent` to manage concurrency of the modules and schedule executions. A Java properties file is used to manage the configuration options for CollecTor instances.

connbidirect

Language	files	code
Java	1	412

onionperf

Language	files	code
Java	1	309
SQL	1	149

legacy/servers

Language	files	code
Java	5	1288
SQL	1	609

advbwdist

Language	files	code
Java	1	122
R	1	19

hidserv

Language	files	code
Java	11	1398

clients

Language	files	code
Java	1	427
Python	2	400
SQL	1	363
R	3	44

ipv6servers

Language	files	code
Java	8	415
SQL	1	89

webstats

Language	files	code
Java	1	260
SQL	1	156
R	1	13

Table 3: Approximate lines of code count per programming language in the metrics-web source control repository per module.

Language	files	code
Java	44	1597
Bourne Shell	3	151
XML	1	130
HTML	1	13

Table 4: Approximate lines of code count per programming language in the CollecTor source control repository for common parts.

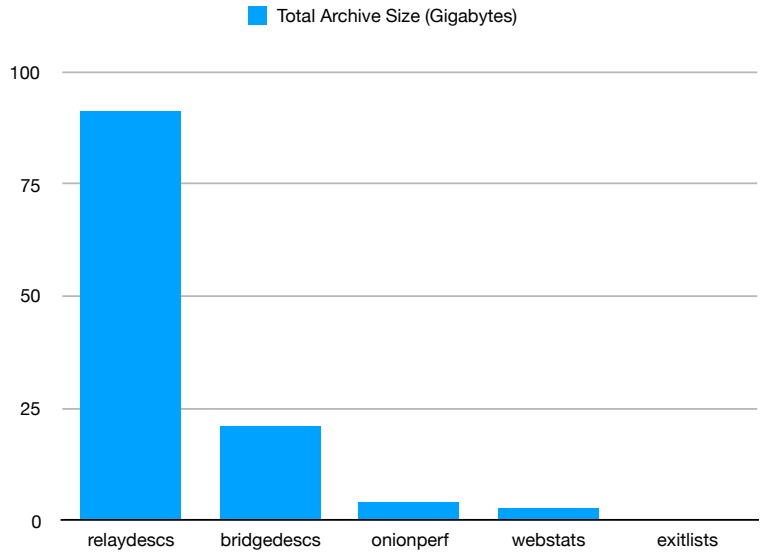

Figure 6: Total size of archived documents for each CollecTor module.

As part of the operation of CollecTor, the scheduler also checks the available space for the storage and logs a warning, if 200 MiB or less are available, and otherwise logs available space in TRACE level. These logs are scraped to raise alerts in the event that disk space is running low, by an external system.

Approximate lines of code count per programming language in the CollecTor source control repository for common parts can be found in table 4.

3.2 relaydescs module

Relays and directory authorities publish relay descriptors, so that clients can select relays for their paths through the Tor network. All these relay descriptors are specified in the Tor directory protocol specification [4]. This module is described in more extensive detail in a recent technical report [3].

The code specific to this module is found in the `org.torproject.metrics.collector.relaydescs` package.

New information is available for the relaydescs module to archive with each new consensus period, which is usually once per hour. If there is downtime, a consensus and the votes that contributed to it can be missed. Some descriptors are cached if they have appeared in any of the currently "valid" consensuses as clients do not necessarily need to update every hour but this does not apply to consensuses or votes.

3.3 bridgedescs module

Bridges and the bridge authority publish bridge descriptors that are used by censored clients to connect to the Tor network. We cannot, however, make bridge descriptors available as we do with relay descriptors, because that would defeat the purpose of making bridges hard to enumerate for censors. We therefore sanitize bridge descriptors by removing all potentially identifying information and publish sanitized versions here.

The requirement to handle this sensitive information is undesirable and in the future we may move the sanitizing process into the BridgeDB software that currently makes the descriptors available to us. If possible, an implementation of the Tor directory protocol may also be added to BridgeDB to allow us to reuse existing code from the relaydescs module.

3.4 onionperf module

The performance measurement services Torperf (now defunct) and OnionPerf publish performance data from making simple HTTP requests over the Tor network. Torperf/OnionPerf use a SOCKS client to download files of various sizes over the Tor network and notes how long substeps take.

The measurement results are published once a day via HTTPS for CollecTor to retrieve. Additionally, a JSON file containing more information is available along with the raw logs generated by the tor processes but these are not archived. A future version of this module may collect the JSON files in addition to the Torperf file.

3.5 webstats module

Tor's web servers, like most web servers, keep request logs for maintenance and informational purposes. However, unlike most other web servers, Tor's web servers use a privacy-aware log format that avoids logging too sensitive data about their users. Also unlike most other web server logs, Tor's logs are neither archived nor analyzed before performing a number of post-processing steps to further reduce any remaining privacy-sensitive parts.

Tor's Apache web servers are configured to write log files that extend Apache's Combined Log Format with a couple tweaks towards privacy. The main difference to Apache's Common Log Format is that request IP addresses are removed and the field is instead used to encode whether the request came in via http:// (0.0.0.0), via https:// (0.0.0.1), or via the site's onion service (0.0.0.2).

Tor's web servers are configured to use UTC as timezone, which is also highly recommended when rewriting request times to "00:00:00" in order for the subsequent sanitizing steps to work correctly. Alternatively, if the system timezone is not set to UTC, web servers should keep request times unchanged and let them be handled by the subsequent sanitizing steps.

Tor's web servers are configured to rotate logs at least once per day, which does not necessarily happen at 00:00:00 UTC. As a result, log files may contain requests from up to two UTC days and several log files may contain requests that have been started on the same UTC day.

The full steps taken for sanitizing the log files are documented on the Tor Metrics website[3].

Sanitized log files are typically compressed before publication. The sorting step also allows for highly efficient compression rates. We typically use XZ for compression, which is indicated by appending ".xz" to log file names, but this is subject to change.

3.6 exitlists module

The exit list service TorDNSEL publishes exit lists containing the IP addresses of relays that it found when exiting through them. The measurement results are made available via a web server a fetched regularly.

4 Onionoo

Onionoo is a web-based protocol to learn about currently running Tor relays and bridges. Onionoo itself was not designed as a service for human beings—at least not directly. Onionoo provides the data for other applications and websites which in turn present Tor network status information to humans. Relay Search, described in section 6, is one such application.

The Onionoo service is designed as a RESTful web service. Onionoo clients send HTTP GET requests to the Onionoo server which responds with JSON-formatted replies. The format of requests is described in the Onionoo protocol [5] which is a versioned protocol with change procedures described as part of its specification.

The Onionoo codebase is split into two parts: the hourly updater described in §4.1 and the web server described in §4.2. The reason for this split is to improve the operational security of the service. The web server has greatly reduced permissions compared to the hourly updater to reduce the impact of it being compromised.

Data flow through the Onionoo application is illustrated in figure 7. The hourly updater merges new information found in recent relay and bridge descriptors retrieved from the CollecTor service with its internal state files. These state files are accessible to the web server which uses these to respond to queries from user applications. As of early 2019, these state files are approximate 33 gigabytes in size.

In the future these flat files may be replaced with a relational database to improve performance and storage efficiency.

4.1 Onionoo common parts and hourly updater

The number of lines of code for each programming language contained in the source control repository for the common parts and hourly updater can be found in table 6.

[3] https://metrics.torproject.org/web-server-logs.html

relaydescs

Language	files	code
Java	6	2574

bridgedescs

Language	files	code
Java	4	1492

exitlists

Language	files	code
Java	1	181

onionperf

Language	files	code
Java	1	253

webstats

Language	files	code
Java	3	385

Table 5: Approximate lines of code per programming language in the CollecTor source control repository for the each module.

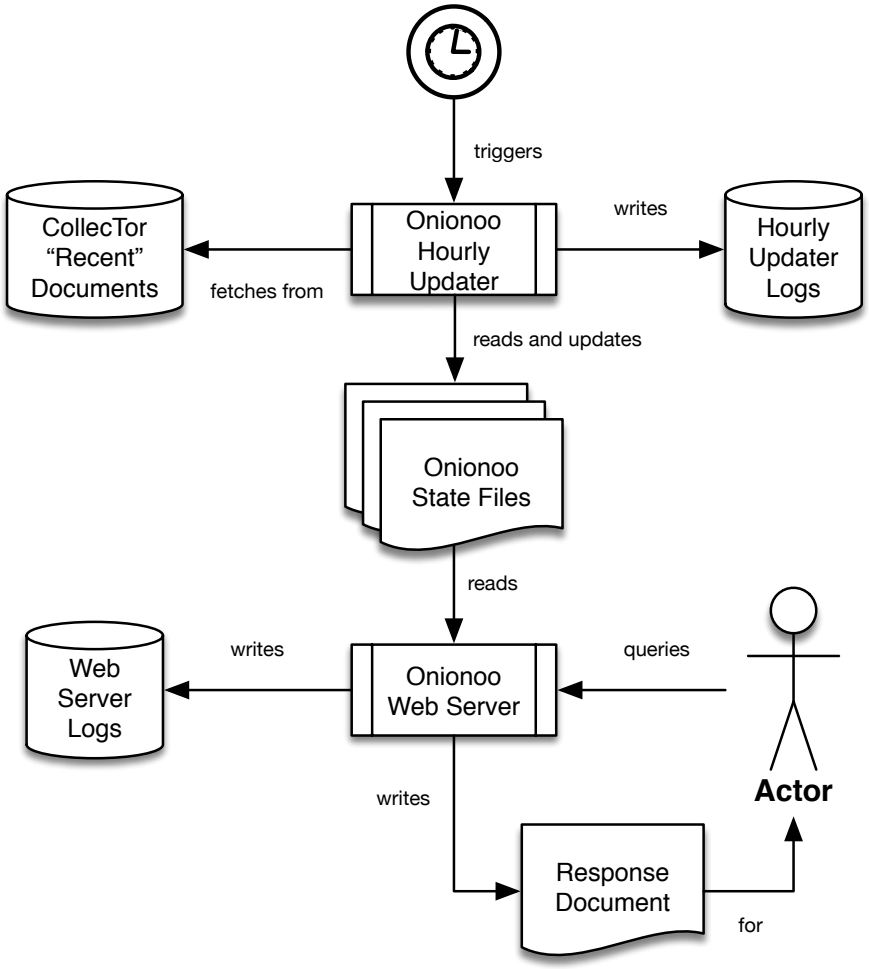

Figure 7: Data flow in the Onionoo application.

The hourly updater's main class is `org.torproject.metrics.onionoo.cron.Main` and uses the `java.util.concurrent` API to schedule the execution of the updater. When the updater runs it performs a total of 7 steps.

1. Initialize data structures and helper classes
2. Download latest descriptors from CollecTor using the "recent" documents
3. Update statuses to include new information from descriptors
4. Write documents to disk (to make available to web server)
5. Write parse histories[4] and flush document cache
6. Gather statistics about the update process
7. Clean up

Descriptor downloads and parsing from CollecTor are implemented using *metrics-lib*. In order to avoid missing the processing of a descriptor it is necessary to run the updater while that descriptor is available from the recent documents in CollecTor. Onionoo will not fetch from the CollecTor archives to process missed data.

The classes relating to Onionoo documents, e.g. summary or details documents, are shared between the hourly updater and the web server. These classes can be found in the package `org.torproject.metrics.onionoo.docs`. This package does not have any dependencies on other parts of the Onionoo codebase (but does depend on *metrics-lib*), and as part of releases we build a thin JAR file containing none of the Onionoo dependencies to allow this package to also be easily reused by client applications.

For the document classes we use the Jackson[5] library for JSON handling. This library reduces our code maintenance costs by making it easy to implement additions or modifications to the protocol. This library is also fast, which is the reason we switched to using this library from Gson[6]. Jackson is currently actively maintained by its developers.

4.2 Onionoo web server

The number of lines of code for each programming language contained in the source control repository for the web server can be found in table 7.

The web server specific classes are organised in the `org.torproject.metrics.onionoo.server` package and the main class is `org.torproject.metrics.onionoo.server.ServerMain`. The main class spawns a Jetty[7] web server and servlet engine.

When a request comes in to the web server application, responses are composed from the state files that have been written by the hourly updater described in the previous sub-section.

When a request is received, the following 5 steps are followed:

[4] Parse histories record which descriptors have already been processed to avoid their statistics being counted twice.

[5] https://github.com/FasterXML/jackson

[6] https://github.com/google/gson

[7] https://www.eclipse.org/jetty/

Language	files	code
Java	51	7359
XML	4	164
HTML	2	18
Bourne Shell	1	2

Table 6: Approximate lines of code count per programming language in the Onionoo source control repository for common parts and the hourly updater.

Language	files	code
Java	15	2517

Table 7: Approximate lines of code count per programming language in the Onionoo source control repository for the web server.

1. Known relays and bridges are filtered by search query
2. Ordering is applied
3. Offset is applied
4. Limit is applied
5. Final response is built including summary information

The combination of the offset and limit in these steps provides pagination to clients that can benefit from that. The final response is a JSON document that can be consumed by the client, either using Onionoo's document classes or an independent compliant implementation.

5 ExoneraTor

ExoneraTor is a small service. It has two parts: one part that fetches recent descriptors from CollecTor and imports them into a database, and another part that listens for incoming web requests and answers them by querying the database.

5.1 ExoneraTor common parts and database importer

The approximate lines of code count per programming language in the ExoneraTor source control repository for common parts and the database importer can be found in table 8.

The schema for the database can be found in `src/main/sql/exonerator.sql` and `exonerator2.sql` in the ExoneraTor source control repository. Both scripts must be used in order to set up an

Language	files	code
Java	1	351
SQL	2	501
XML	4	252
Bourne Shell	1	2

Table 8: Approximate lines of code count per programming language in the ExoneraTor source control repository for common parts and the database importer.

Language	files	code
Java	6	1083
JSP	3	39
CSS	2	33

Table 9: Approximate lines of code count per programming language in the ExoneraTor source control repository for common parts and the database importer.

ExoneraTor database. This database stores consensus entries and exit list entries. It is highly optimized towards the use case of looking up relays by IP address and date.

The database importer is contained in `ExoneraTorDatabaseImporter`. That code is supposed to run once per hour. It checks its configuration file, connects to the database, fetches descriptors from CollecTor, parses descriptors, imports them into the database, and closes the connection.

The database importer uses a simple lock file to avoid overlapping executions, and it doesn't start a new import until the lock file is at least six hours old. When that happens, the importer assumes that the lock file was left over from an aborted run, deletes it, and continues as usual.

ExoneraTor can handle not importing anything for 72 hours. After that time it will miss any descriptors that dropped out of CollecTor's recent folder.

ExoneraTor's working filesystem usage is currently 159MB and the database is currently 63GB in size.

5.2 ExoneraTor web server

The approximate lines of code count per programming language in the ExoneraTor source control repository for the web server can be found in table 9.

We're using an embedded Jetty that is started using ServerMain and that deploys a small number of servlets:

QueryServlet This servlet accepts an IP address and date and knows how to ask the database whether that IP address was a relay on the given date. It first parses its parameters, makes the database request, and puts together a QueryResponse object with the response. That response object is serialized to a JSON object.

Note that we're only making a single database request and returning a single response object, regardless of the output. For example, the response might be positive with details about the match, but it might also be negative with nearby addresses that we would have had a match for. We could have made several database requests, depending on what we find, but it seemed better to let the database do the heavy lifting and minimize interaction between web server and database, which is what we did.

ExoneraTorServlet This is the servlet that produces an actual web page for a given request. Internally it relies on QueryServlet to provide database results. But ExoneraTorServlet knows how to present a query response. This includes all kinds of error cases like having no database connection, not having relevant data, and so on.

However, ExoneraTorServlet is not deployed on https://exonerator.torproject.org/, because we moved all user-facing parts to the Tor Metrics website. Instead, Tor Metrics deploys ExoneraTorServlet and wraps it in its Tor Metrics specific website header and footer. See also ExoneraTorWrapperServlet in metrics-web for more details. In theory, it would be possible to change ExoneraTor's web.xml to deploy this servlet directly on the ExoneraTor host that also has the database.

ExoneraTorRedirectServlet This is deployed on the ExoneraTor host and which redirects all requests to the Tor Metrics website.

It's perhaps worth noting that ExoneraTor is the only page on Tor Metrics that comes with translations. They are contained in the ExoneraTor repository (for deployment on the ExoneraTor host which we're not doing) and in the metrics-web repository (which is what we have deployed on Tor Metrics).

6 Relay Search

Relay Search is maintained as part of the metrics-web codebase, and is tightly integrated into the metrics-web theming via a JSP. Approximate lines of code count per programming language in the metrics-web source control repository for Relay Search can be found in table 10.

Relay Search is a browser-based Onionoo client that allows users to search for Tor relays and bridges by using Onionoo search queries. Users can get a detailed view of how the relay is configured, what its exit policy is and all the data that you would normally find in the server descriptor. The historical data of a relay's bandwidth usage is available in graph form, also provided by Onionoo.

This application is built in a way that all the logic is delegated to the client. This means that the amount of requests made to the server can be minimized and that this application can potentially run by being loaded locally. The server is only interrogated for JSON objects that do not manipulate the DOM of the page.

Language	files	code
JavaScript	33	3846
HTML	8	1086
CSS	5	357
Python	1	17
JSON	1	1

Table 10: Approximate lines of code count per programming language in the metrics-web source control repository for Relay Search.

Language	files	code
Java	79	9385
HTML	1	62
Bourne Shell	1	2

Table 11: Approximate lines of code count per programming language in the metrics-lib source control repository.

Relay Search uses Backbone.js[8] as an MV* framework, with require.js[9] for AMD loading of dependencies. jQuery[10] and Underscore.js[11] as JavaScript utility libraries. Datatables[12] is used for visualizing data in tabular form with custom filtering and D3.js[13] is used for data visualisation.

7 metrics-lib

Tor Metrics Library API, which is provided and supported by Tor's Metrics Team, is a library to obtain and process descriptors containing Tor network data. It is the main Java tool for processing Tor descriptors and provides a standard API consisting of interfaces and a reference implementation for all of them.

Most Tor descriptors understood by this library are specified in the Tor directory protocol, version 3 or in the earlier version 2 or version 1 of that document. Other descriptors are specified on the CollecTor website.

The design and development of this library has been driven by two main goals originating

[8] https://backbonejs.org/
[9] https://requirejs.org/
[10] https://jquery.com/
[11] https://underscorejs.org/
[12] https://datatables.net/
[13] https://d3js.org

from the primary use case to make Tor network data accessible for statistical analysis:

- Complete coverage: This library is supposed to cover the complete range of Tor descriptors made available by the CollecTor service.
- Runtime and memory efficiency: Processing large amounts of descriptors in bulk is supposed to be efficient in terms of runtime and required memory.

At the same time the current design and implementation were done with a number of non-goals in mind, even though some of these might turn into goals in the future:

- Verification: The descriptor parser performs some basic verifications of descriptor formats, but no cryptographic verifications. It may not even be possible to write a cryptographic verification tool using parsed descriptor contents, though this has not been attempted yet.
- Potentially lossy conversion: Descriptor contents may be converted to a format that is easier to process, even if that conversion makes it harder or impossible to re-create the original descriptor contents from a parsed descriptor.
- Generating descriptors: This library does not contain any functionality to generate new descriptors for testing or related purposes, neither from previously set data nor randomly.
- Writing descriptors: This library does not support writing descriptors to the file system or a database, both of which are left to the application. Stated differently, there are no descriptor sinks that would correspond to the provided descriptor sources.

The `org.torproject.descriptor` package contains all relevant interfaces and classes that an application would need to use this library. Applications are strongly discouraged from accessing types from the implementation package (`org.torproject.descriptor.impl`) directly, because those may change without prior notice.

Interfaces and classes in this package can be grouped into general-purpose types to obtain and process any type of descriptor and descriptors produced by different components of the Tor network:

General-purpose types These comprise `DescriptorSourceFactory` which is the main entry point into using this library. This factory is used to create the descriptor sources for obtaining remote descriptor data (`DescriptorCollector`) and descriptor sources for processing local descriptor data (`DescriptorReader` and `DescriptorParser`). General-purpose types also include the superinterface for all provided descriptors (`Descriptor`).

Relays and Bridges The first group of descriptors is published by relays and servers in the Tor network. These interfaces include server descriptors (`ServerDescriptor` with subinterfaces `RelayServerDescriptor` and `BridgeServerDescriptor`), extra-info descriptors (`ExtraInfoDescriptor` with subinterfaces `RelayExtraInfoDescriptor` and `BridgeExtraInfoDescriptor`), microdescriptors which are derived from server descriptors by the directory authorities (`Microdescriptor`), and helper types for parts of the aforementioned descriptors (`BandwidthHistory`).

Network Statuses The second group of descriptors is generated by authoritative directory servers that form an opinion about relays and bridges in the Tor network. These include descriptors specified in version 3 of the directory protocol (`RelayNetworkStatusConsensus`, `RelayNetworkStatusVote`, `DirectoryKeyCertificate`, and helper types for descriptor parts `DirSourceEntry`, `NetworkStatusEntry`, and `DirectorySignature`), descriptors from earlier directory protocol version 2 (`RelayNetworkStatus`) and version 1 (`RelayDirectory` and `RouterStatusEntry`), as well as descriptors published by the bridge authority and sanitized by the CollecTor service (`BridgeNetworkStatus`).

Auxiliary The third group of descriptors is created by auxiliary services connected to the Tor network rather than by the Tor software. This group comprises descriptors by the bridge distribution service BridgeDB (`BridgePoolAssignment`), the exit list service TorDNSEL (`ExitList`), the performance measurement service Torperf (`TorperfResult`), and sanitized access logs of Tor's web servers (`WebServerAccessLog`).

8 OnionPerf

OnionPerf is a special codebase within the Tor Metrics ecosystem for two reasons. It performs active measurements of the Tor network, whereas other parts are either passive or analysing the existing data, and it is written in Python as opposed to Java. It is a tool which tracks data download performance though the publicly deployed Tor network.

It uses a traffic generator server TGen to serve and fetch random data on the running host. The data is transferred through Tor using Tor client processes and ephemeral Onion Services.

OnionPerf uses a "measure" component which controls the flow of all measurements done by the tool. The code for this is included in file `measurement.py`.

There are two measurement options:

- Files hosted on the local machine are downloaded via the Tor network using a Tor client only. This emulates accessing the Internet via Tor.
- Files hosted as an Onion Service using a Tor server are downloaded via the Tor network using a Tor client. This emulates accessing an Onion service via Tor.

OnionPerf has a traffic generation component and a Tor client component which underpin measurements and are included in the `measurement.py` file. The traffic generator OnionPerf uses is Tgen, a C application that models traffic behaviors using an action-dependency graph represented using the standard GraphML (XML) format. By default, OnionPerf's TGen client performs 50KB, 1MB and 5MB transfers in a probabilistic fashion using a weighted action graph. The TGen server listens for incoming transfers and serves data on a TCP port of the running host. The graph models that can be used for both TGen server and client are defined in a separate file, `model.py`.

OnionPerf uses the tor binary available on the running host for its Tor client component, which it calls as a managed subprocess. OnionPerf can also specify configuration strings for the

client or Onion Service. It also uses the Python stem[14] library for creating ephemeral Onion Services.

Tor control information and TGen performance statistics are logged and analyzed once per day to produce statistics in json and TorPerf formats. The code for extracting data from the logs on the disk is found in file `analysis.py`. The TorPerf files are archived by CollecTor's onionperf module.

9 Comparison to OONI's data pipeline

OONI is a free software, global observation network for detecting censorship, surveillance and traffic manipulation on the Internet. Using an active measurement tool, ooniprobe[15], measurements that can detect censorship or other network interference are collected. These are then fed in to OONI's data processing pipeline[16] for archive, analysis and visualisation.

All results from ooniprobe are submitted to one of a few instances of OONI Collector[17]. The OONI Collector is a web service that can receive measurements via HTTPS Internet and Onion addresses.

This is the first deviation from the Tor Metrics model. Tor Metrics' CollecTor uses a pull-model to fetch data from a number of services periodically, whereas the OONI Collector uses a push-model where measurement probes actively submit results to be processed at any time.

The OONI Collector is not fault-tolerant as the upload protocol currently binds the client to specific collector instance. Tor Metrics' CollecTor is fault-tolerant as if one CollecTor instance is down, the other instance will still archive data which can be sideloaded into the primary CollecTor instance once it recovers.

Apache Airflow[18] is used by the OONI pipeline for scheduling of batch processing tasks, and providing a UI and logging interface for those tasks. Daily, the airflow rsync's report files from "outgoing" spools of the collectors, merges those reports into a "reports-raw" bucket (a directory on the filesystem) and creates a flag file listing all the files.

From the raw reports, compressed archives are created for the day's worth of submitted results. These archives are for cold archive and known as "reports-tgz". There are also smaller archives created, known as "canned" archives, which are sorted by measurement type and lz4 compressed. Both the "canned" and "reports-tgz" files are uploaded to a private Amazon S3.

Tor Metrics' CollecTor does not initially create archives when data is collected but instead stores the raw documents directly in the filesystem. The last 72 hours of documents collected are concatenated at the time of the update and served from the "recent" directory to clients. Only after the 72 hour peiod is up are the documents archived in the tarballs for cold storage. Tor Metrics currently does not use any cloud storage or cloud data processing, but the CollecTor server does have backups managed by the Tor Sysadmin Team.

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

[15] https://github.com/ooni/probe

[16] https://github.com/ooni/pipeline/blob/master/docs/pipeline-16.10.md

[17] https://github.com/ooni/collector — Note that this software is not related to the Tor Metrics "CollecTor" application

[18] https://airflow.apache.org/

The data is sanitised (removing bridge IPs, for example) and compressed into data-aware *.tar.lz4 files where LZ4 framing is aligned with the framing of JSON measurements, those files are known as "autoclaved". "autoclaved" files are seekable (to get individual measurement) and streamable with ordinary CLI tools (`e.g, tar -to-stdout`) for those who want to get the raw data. "autoclaved" files are uploaded to a public Amazon S3 bucket.

Tor Metrics' CollecTor performs sanitisation before processing any data further to avoid sensitive data existing for any longer than it needs to.

Some metadata and features of those autoclaved files are stored in a PostgreSQL database to ease data mining and to handle API queries and presentation. Metadata that is used by API currently takes 32 GiB and corresponding indexes take 65 GiB. The OONI API[19] provides RESTful access to this data.

For visualisation on the Tor Metrics website, each visualisation is using its own Postgres database to maintain its working data, and access to these databases are only by their relevant modules. CSV files are produced from those databases that can, to a limited extent, be customised by users. Onionoo provides a more featureful query interface although is backed by flat-files and not by a database which may present scaling issues in the future.

OONI Explorer talks to the OONI API to present a global map which provides a location to explore and interact with all of the network measurements that have been collected through OONI tests from 2012 until today. This is a browser-based API client, very similar to Relay Search in its architecture although using some different components where trends in JavaScript development have changed.

The OONI pipeline setup allowed the team to grow 15 times from approximately 1.2 TiB of raw data to approximately 18 TiB. It is estimated that they could grow a futher 2 times from there without hitting major obstacles, which is approximately 2 years at their current growth rate.

10 Next Steps

From this report, we can draw some next steps for the Metrics Team to work on:

- The various metrics-web modules are basically applications on their own. We can likely save many LOCs by using common code for tasks like reading descriptors, importing into the database, and writing output files. We are tracking this task as ticket #28342[20]: Share more code between modules.

- We should make more use of databases. We should look into using a single database for metrics-web, possibly using schemas for tables belonging to a specific module. We should also look into using a database in Onionoo for storing prepared response parts. The goal is not just to improve service performance but also to increase robustness.

- We should try to make better use of existing libraries and frameworks. The CollecTor prototype was a good step in this direction, and we should continue investigating what we can do there.

[19] http://api.ooni.io/
[20] https://bugs.torproject.org/28342

- A combination of Tor Metrics data with OONI data could be very informative when it comes to Tor's censorship circumvention work. We should investigate how this could be possible.
- As our dataset sizes continue to grow, we will have to begin investigating new solutions for handling the data. We may also have to explore new ways of making the data publicly available, as OONI have done through their use of AWS.

References

[1] George Danezis. An anomaly-based censorship-detection system for Tor. Technical Report 2011-09-001, The Tor Project, September 2011.

[2] George Kadianakis and Karsten Loesing. Extrapolating network totals from hidden-service statistics. Technical Report 2015-01-001, The Tor Project, January 2015.

[3] Iain R. Learmonth and Karsten Loesing. Towards modernising data collection and archive for the Tor network. Technical Report 2018-12-001, The Tor Project, December 2018.

[4] Tor Project. Tor directory protocol, version 3. https://spec.torproject.org/dir-spec.

[5] Tor Project. Onionoo Protocol Specification. https://metrics.torproject.org/onionoo.html.