

Analyzing OpenSTA Performance Results

StatGen

Proof Of Concept - Working With The StatGen Python API

Author: Corey Goldberg (corey@goldb.org)

Date: June 2006

goldb.org

OpenSTA
Open System Testing Architecture

Background

I have followed OpenSTA closely for several years, including conversations on the [opensta-devel](#) mailing list a few years back about the future development of OpenSTA. We discussed lots of issues and possible future directions. Since then, no major shifts have happened and developer involvement seems to be running very low.

The GUI used by OpenSTA is built using MFC and is showing its age. Its editing, test setup, and execution are adequate; but its monitoring, results analysis, and graphing abilities are far behind what proprietary rivals offer and are also very brittle. Furthermore, the fact that its development environment is complex and proprietary may drive prospective contributors away.

However, a strong point of OpenSTA is its modular architecture which includes a very fast and scalable C++ load generating component. It also has its own [scripting language \(SCL\)](#) and script compiler.

Many system components are accessible in a batch mode via the command line:

[Wiki \(portal.opensta.org\): OstaCommandLine](#)

Related utilities:

[Trickytools - Tools For OpenSTA](#)

For results reporting, OpenSTA logs to CSV files that are ideal for post processing with external tools. So basically OpenSTA's core can be driven without its existing GUI; for both execution and results analysis.

OpenSTA

OpenSTA is a popular open source web load testing toolset for HTTP stress & performance tests

from <http://www.opensta.org/>:

"OpenSTA is a distributed software testing architecture designed around CORBA... The current toolset has the capability of performing scripted HTTP and HTTPS heavy load tests with performance measurements from Win32 platforms. However, the architectural design means it could be capable of much more.

The applications that make up the current OpenSTA toolset were designed to be used by performance testing consultants or other technically proficient individuals. This means testing is performed using the record and replay metaphor common in most other similar commercially available toolsets. Recordings are made in the tester's own browser producing simple scripts that can be edited and controlled with a special high level scripting language. These scripted sessions can then be played back to simulate many users by a high performance load generation engine. Using this methodology a user can generate realistic heavy loads simulating the activity of hundreds to thousands of virtual users.

...

The OpenSTA toolset is Open Source software licensed under the GNU GPL (General Public License), this means it is free and will always remain free. If you wish to build your own customized version of OpenSTA or take part in its development then the complete toolset source code, buildable in Microsoft Visual Studio 6, and all related information is available from [OpenSTA.SourceForge.net](#)."

StatGen

StatGen is a toolset and API written in Python.

It is used for:

- parsing log files
- running statistical calculations on numeric sequences and time series
- generating performance graphs

Components

- Python
- Matplotlib (pure python library, has a dependency on python numeric)

Installation/Setup

1. install Python
2. install Python Numeric
3. install Matplotlib
4. unzip StatGen.zip

StatGen Module/Class Library

- [corestats.Stats](#)
- [statgen.LogParser](#)
- [timeseries.TimeSeries](#)

Where is the code?

I have just finished the first iteration of StatGen development and have a working code base. StatGen is designed as a versatile API that can be used to create a variety of tools on top of. Once I get the API more stable and get a few iterations down, I would like to open the code (GPL'ed) for others to contribute to and eventually use.

Future

This initial pass at development could be continued in several ways. In its simplest form, it could be used to create a standalone tool for data analysis and graphing. Matplotlib is the graphing backend used by StatGen. Matplotlib is an open source 2D plotting library. It supports several GUI toolkits used for graph output. It is easiest to use with Tk, so my initial iteration outputs graphs in Tk. However, if this was to be integrated into a larger UI framework, it can also use wxPython and pyGTK.

StatGen supports various input types. Before I had OpenSTA in mind, I created a version of StatGen that integrated with MS Log Parser.

I wrote a [testingReflections blog entry](#) about this.

However, StatGen does not depend on MS Log Parser (which is proprietary) or any platform specific code. I created some simple Python based parsing code that made analysis of OpenSTA log files pretty trivial.

As a next step, a wrapper could be built around the OpenSTA CLI and a new GUI for test execution and management could be developed. Eventually all UI modules could be converted over, while leaving the core of OpenSTA in tact.

Why Python?

The goal I have in mind is to create a flexible open source UI written in a dynamic object oriented language that has a lot of existing module/library support, can integrate well with a C++ execution core, and can be developed using all Free tools. Another reason is an eye towards future cross-platform compatibility.

Scripting With The StatGen API

As of today, a proper toolset for doing results analysis based on StatGen has not been built. However, I have developed enough of the framework to expose part of its API to show off some its functionality (with or without OpenSTA). I posted a page containing information on working with the initial API:

<http://www.goldb.org/statgen.html>

(You can ignore the MS Log Parser references as these are not relevant to OpenSTA parsing)

Below I have included a set of Python scripts that operate on Timer.txt, the log output from an OpenSTA Timer during a test run.

The API still needs a lot of work and eventually this would all be accessible from a nicely designed class library for scripting and a polished GUI for easy navigation and analysis.

Example: Request Throughput (HTTP requests/sec)

1 sec Time Series - from OpenSTA log

```
#!/usr/bin/env python

import timeseries

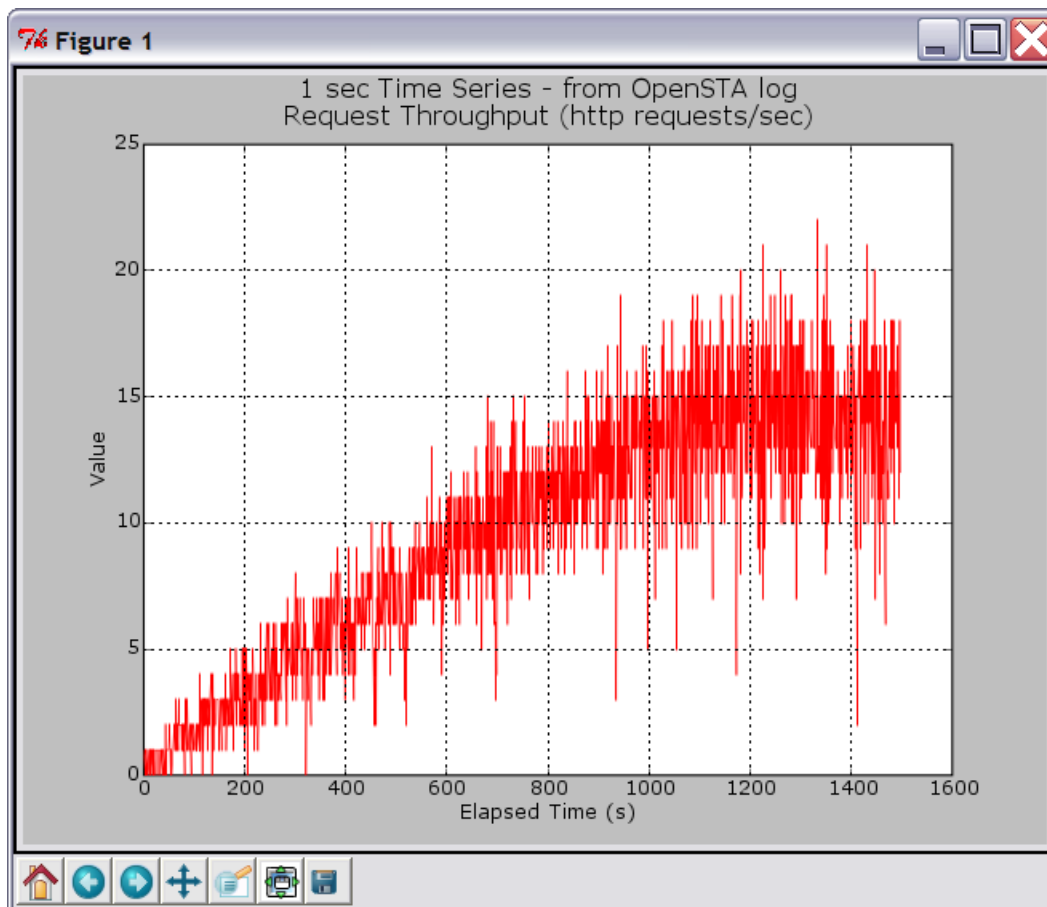
# open the OpenSTA output log
fh = open("./Timer.txt", "rb")
file = fh.readlines()
fh.close

# parse the log and create a data set usable by a timeseries object
data_set = [[int(line.split(',')[0]), float(line.split(',')[1]) / 100.0] for line in file]

# create the time series
ts = timeseries.TimeSeries(1, data_set, 'epoch')

ts.graph_title = '1 sec Time Series - from OpenSTA log\nRequest Throughput (http requests/sec)'
ts.calc_series('COUNT_PERSEC')
ts.graph_series_tk()
```

output:



Example: Request Throughput (http requests/sec)

30 sec Time Series - from OpenSTA log

```
#!/usr/bin/env python
import timeseries

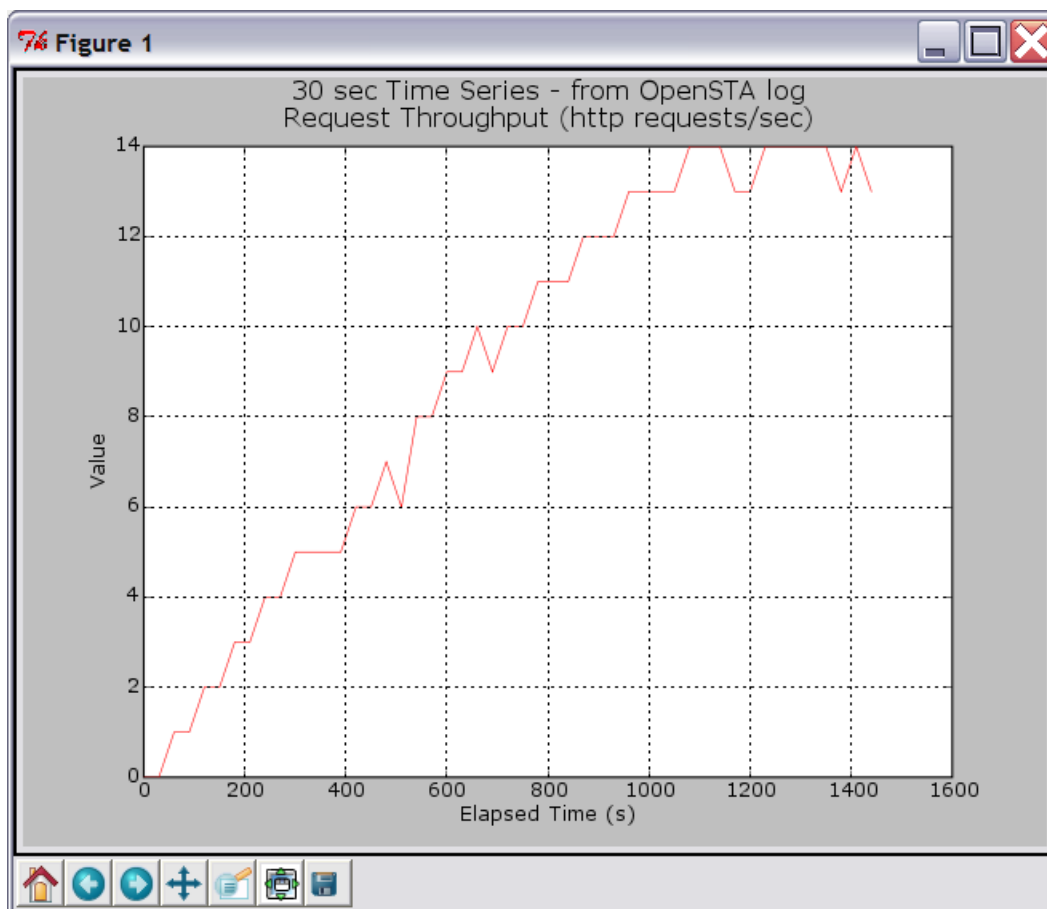
# open the OpenSTA output log
fh = open("../Timer.txt", "rb")
file = fh.readlines()
fh.close

# parse the log and create a data set usable by a timeseries object
data_set = [[int(line.split(',')[0]), float(line.split(',')[1]) / 100.0] for line in file]

# create the time series
ts = timeseries.TimeSeries(30, data_set, 'epoch')

ts.graph_title = '30 sec Time Series - from OpenSTA log\nRequest Throughput (http requests/sec)'
ts.calc_series('COUNT_PERSEC')
ts.graph_series_tk()
```

output:



Example: All Timer values from OpenSTA log

```
#!/usr/bin/env python

import timeseries

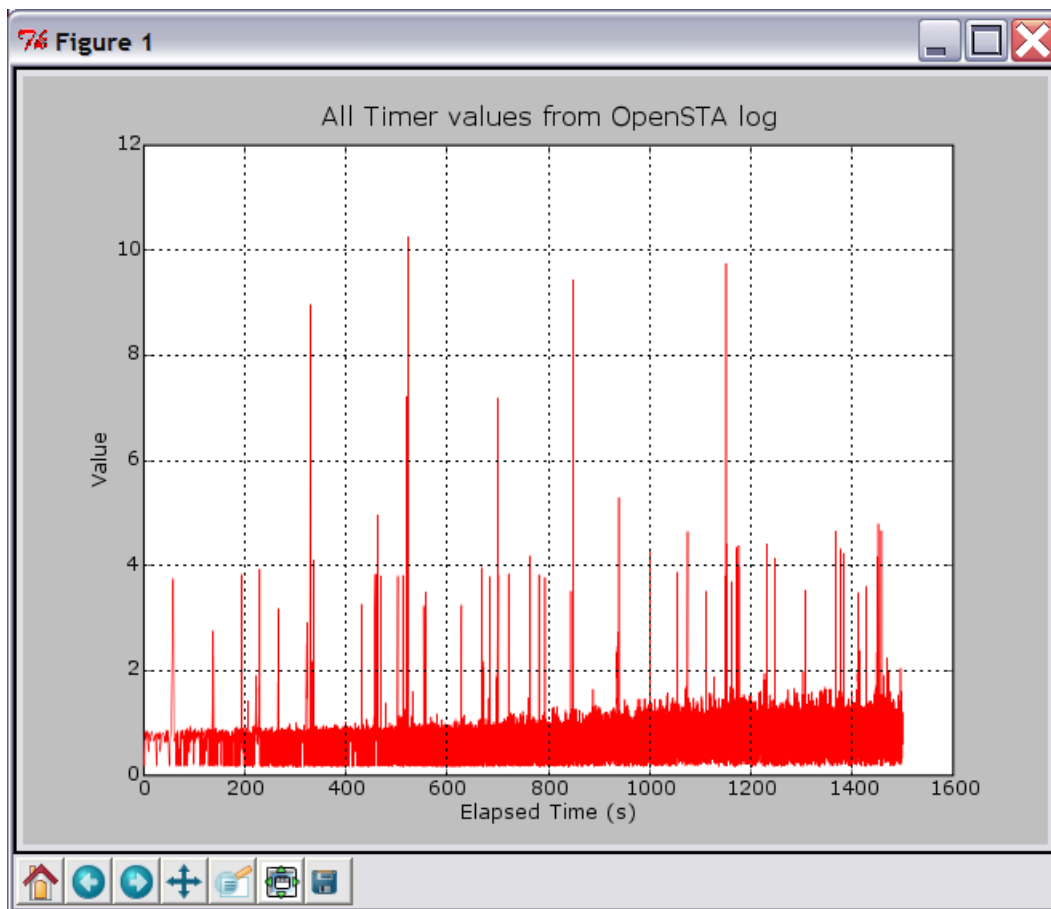
# open the OpenSTA output log
fh = open("./Timer.txt", "rb")
file = fh.readlines()
fh.close

# parse the log and create a data set usable by a timeseries object
data_set = [[int(line.split(',')[0]), float(line.split(',')[1]) / 100.0] for line in file]

# create the time series
ts = timeseries.TimeSeries(30, data_set, 'epoch')

ts.graph_title = 'All Timer values from OpenSTA log'
ts.graph_all_tk()
```

output:



Example: Average - Response Times

10 sec Time Series - Timer values from OpenSTA log

```
#!/usr/bin/env python

import timeseries

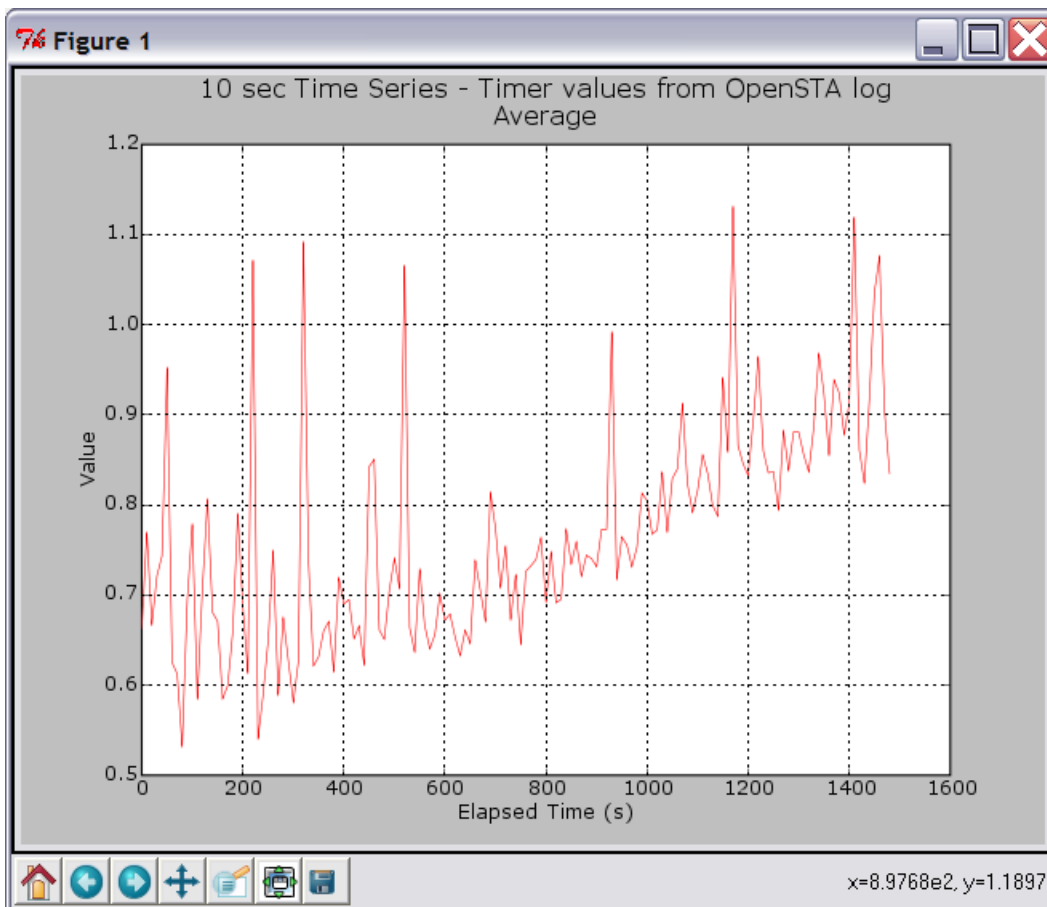
# open the OpenSTA output log
fh = open("./Timer.txt", "rb")
file = fh.readlines()
fh.close

# parse the log and create a data set usable by a timeseries object
data_set = [[int(line.split(',')[0]), float(line.split(',')[1]) / 100.0] for line in file]

# create the time series
ts = timeseries.TimeSeries(10, data_set, 'epoch')

ts.graph_title = '10 sec Time Series - Timer values from OpenSTA log\nAverage'
ts.calc_series('AVG')
ts.graph_series_tk()
```

output:



Example: 90th Percentile - Response Times

30 sec Time Series - Timer values from OpenSTA log

```
#!/usr/bin/env python

import timeseries

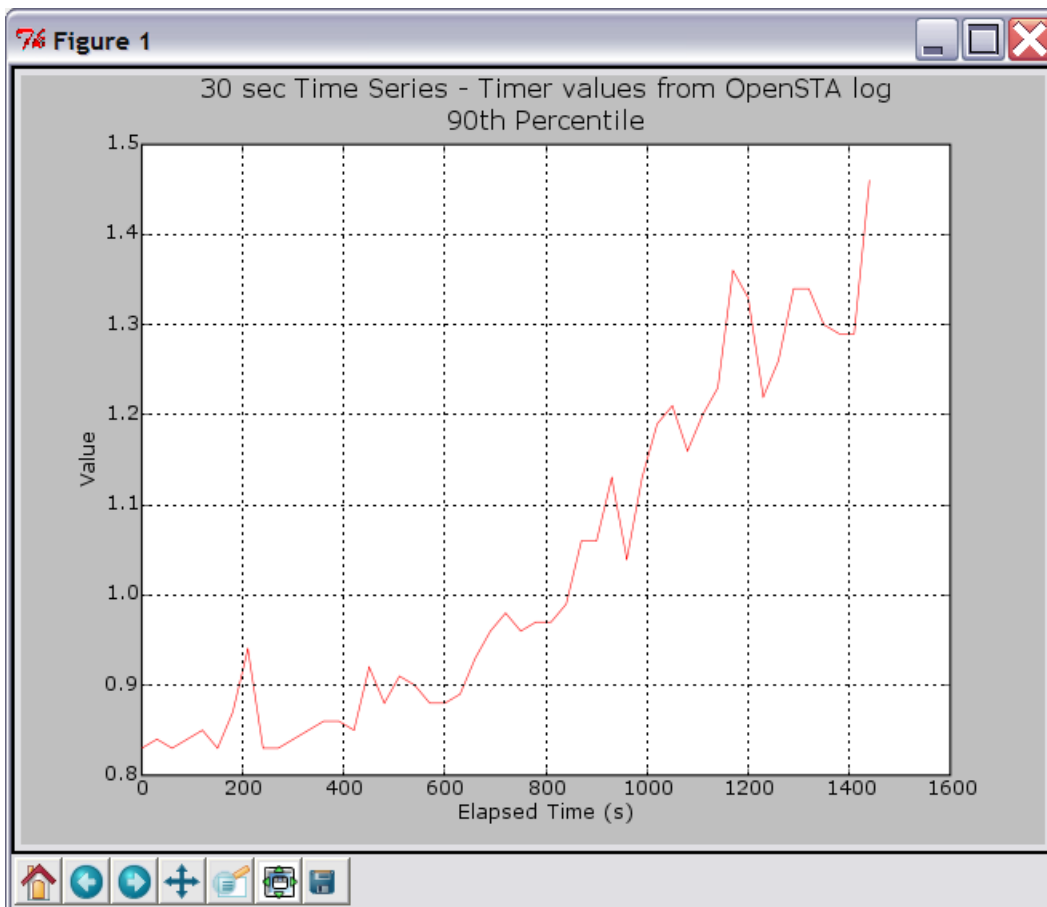
# open the OpenSTA output log
fh = open("./Timer.txt", "rb")
file = fh.readlines()
fh.close

# parse the log and create a data set usable by a timeseries object
data_set = [[int(line.split(',')[0]), float(line.split(',')[1]) / 100.0] for line in file]

# create the time series
ts = timeseries.TimeSeries(30, data_set, 'epoch')

ts.graph_title = '30 sec Time Series - Timer values from OpenSTA log\n90th Percentile'
ts.calc_series('PERCENT90')
ts.graph_series_tk()
```

output:



Example: Standard Deviation - Response Times

10 sec Time Series - Timer values from OpenSTA log

```
#!/usr/bin/env python

import timeseries

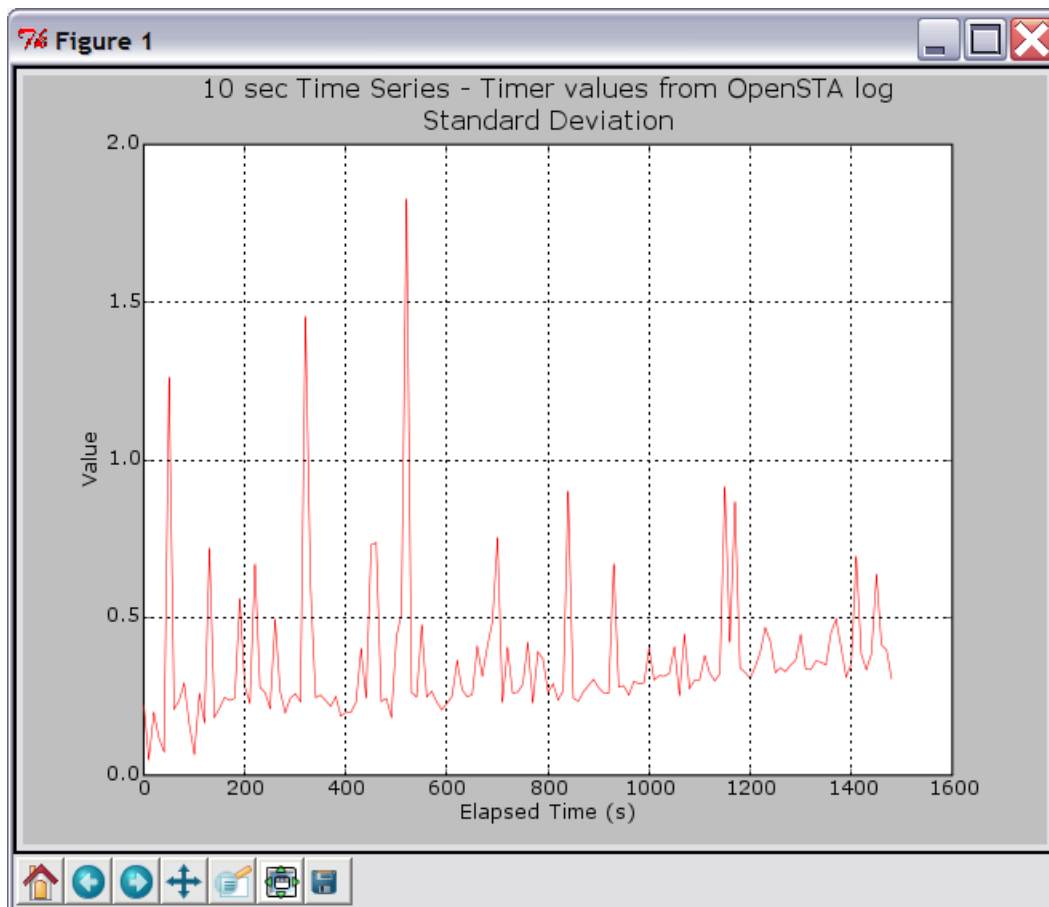
# open the OpenSTA output log
fh = open("./Timer.txt", "rb")
file = fh.readlines()
fh.close

# parse the log and create a data set usable by a timeseries object
data_set = [[int(line.split(',')[0]), float(line.split(',')[1]) / 100.0] for line in file]

# create the time series
ts = timeseries.TimeSeries(10, data_set, 'epoch')

ts.graph_title = '10 sec Time Series - Timer values from OpenSTA log\nStandard Deviation'
ts.calc_series('STDEV')
ts.graph_series_tk()
```

output:

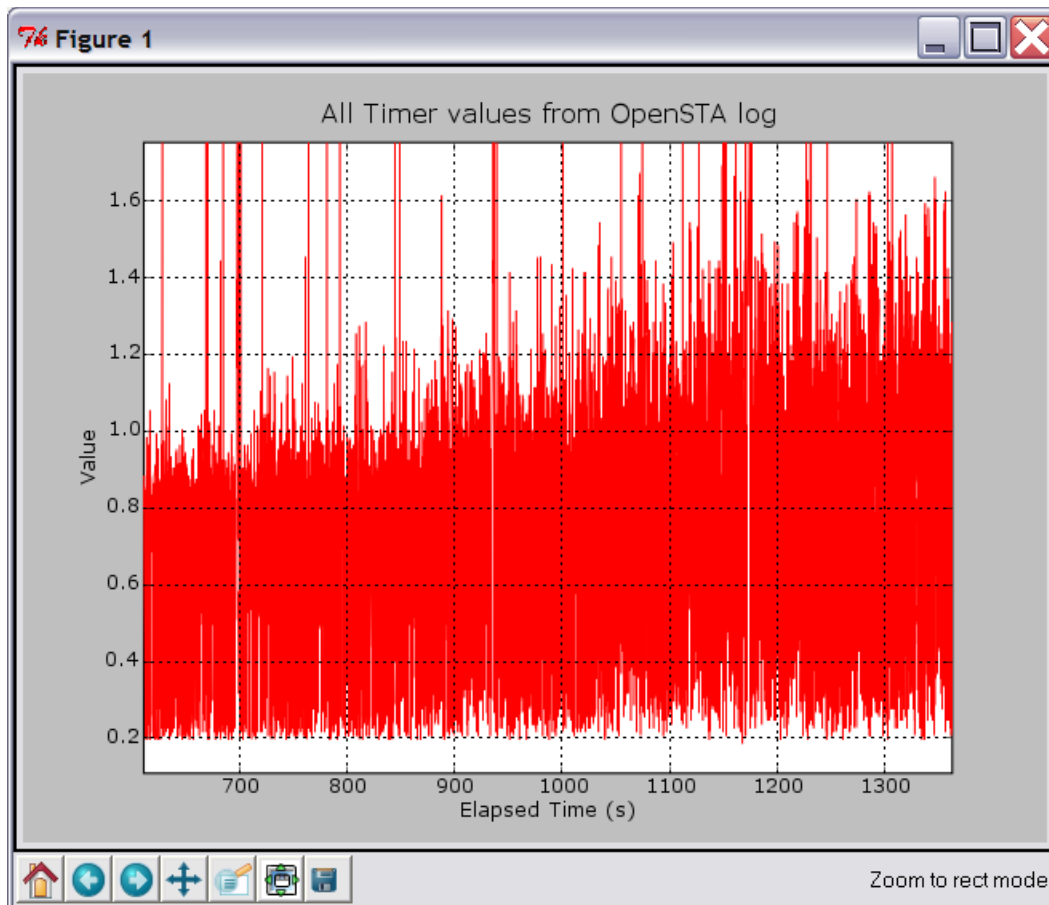


Viewing Features

The Tk panel that the graphs are rendered onto contains functionality for powerful viewing including zoom, resize, and save to image.

I haven't fully explored the capabilities of Matplotlib, but there seems to be lots of configuration available. The examples shown throughout are just a cursory exploration of its graphing features. It should be straightforward to further tune the graphing in terms of looks, style, and analysis capabilities.

Screenshot of zoom:



Appendix A: OpenSTA Configuration

Details of the test run used to generate the results that were analyzed.

OpenSTA SCL Script:

This is the script I created in SCL for the test run I analyzed. It is an extremely simple example where just send a single HTTP GET request to my homepage.

```
!Browser:IE5

Environment
  Description "TEST SCRIPT"
  Mode HTTP
  WAIT UNIT MILLISECONDS

Definitions
  Include      "RESPONSE_CODES.INC"
  Include      "GLOBAL_VARIABLES.INC"
  CONSTANT     SITE = "http://www.goldb.org"
  CONSTANT     DEFAULT_HEADERS = "Host: www.goldb.org^J" &
    "User-Agent: OpenSTAzilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)"
  Integer      USE_PAGE_TIMERS
  Timer         T_Response
  CHARACTER*32768 logStuff, Local
  CHARACTER*512  USER_AGENT
  CHARACTER*256  MESSAGE

Code

Entry[USER_AGENT,USE_PAGE_TIMERS]

Start Timer T_Response

PRIMARY GET URI "http://www.goldb.org/index.html HTTP/1.0" ON 1 &
  HEADER DEFAULT_HEADERS &
  ,WITH {"Accept: */*", &
    "Accept-Language: en-us", &
    "Connection: Keep-Alive"}

SYNCHRONIZE REQUESTS !wait until response is returned

End Timer T_Response

!Logging
  !Load Response_Info Header ON 1 INTO logStuff
  !Log logStuff
  !Load Response_Info Body ON 1 INTO logStuff
  !Log logStuff

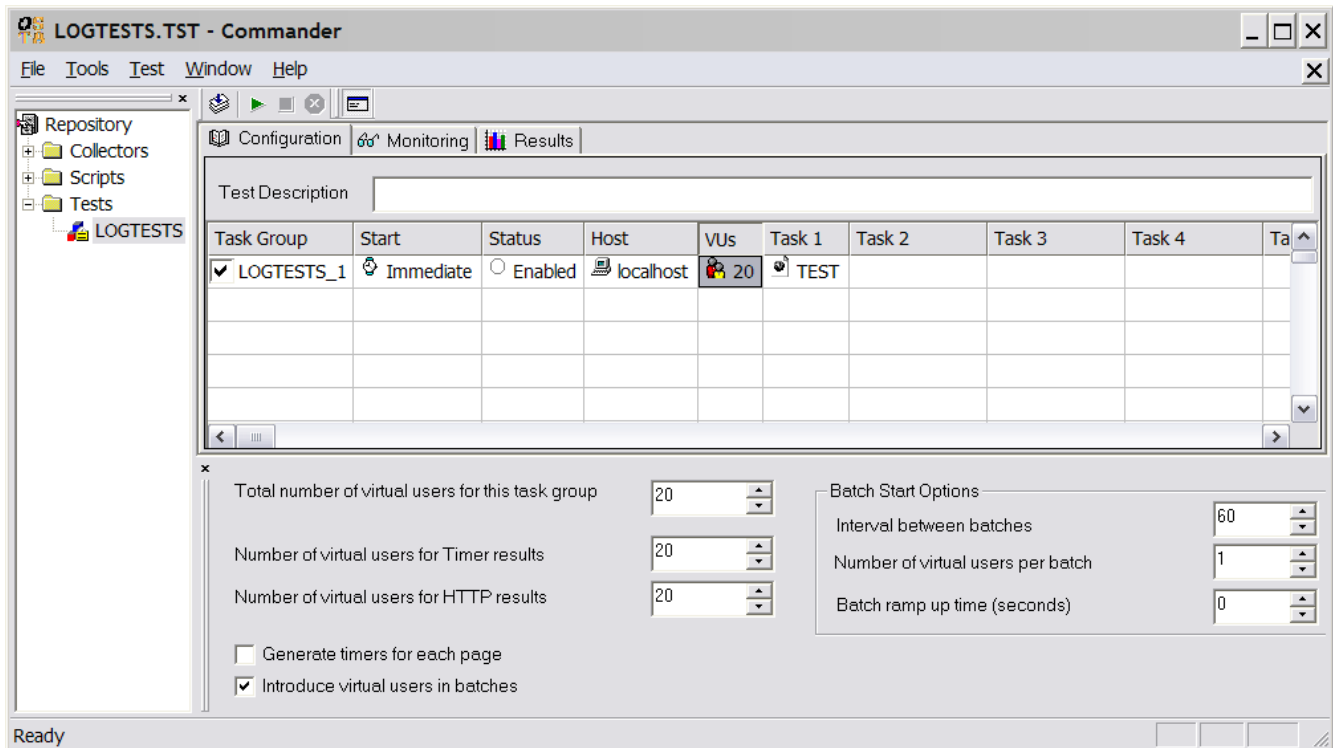
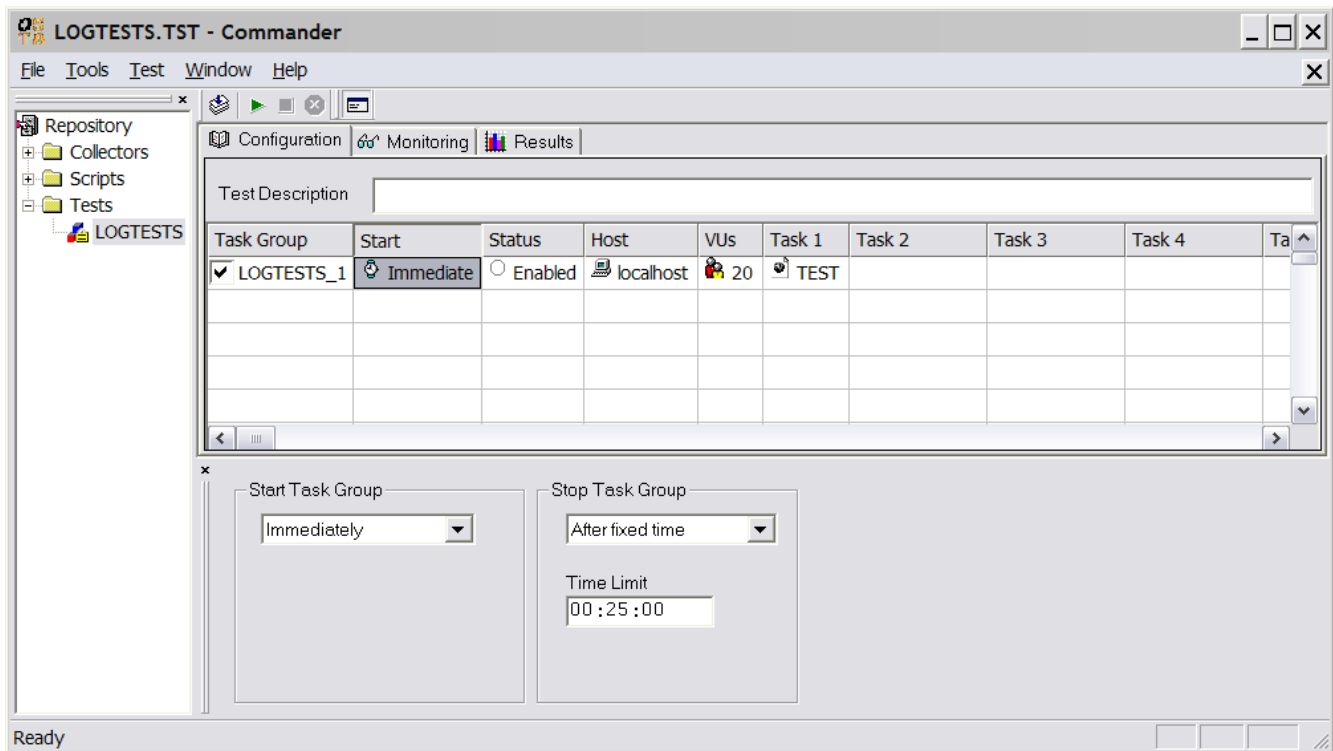
DISCONNECT FROM 1

Exit

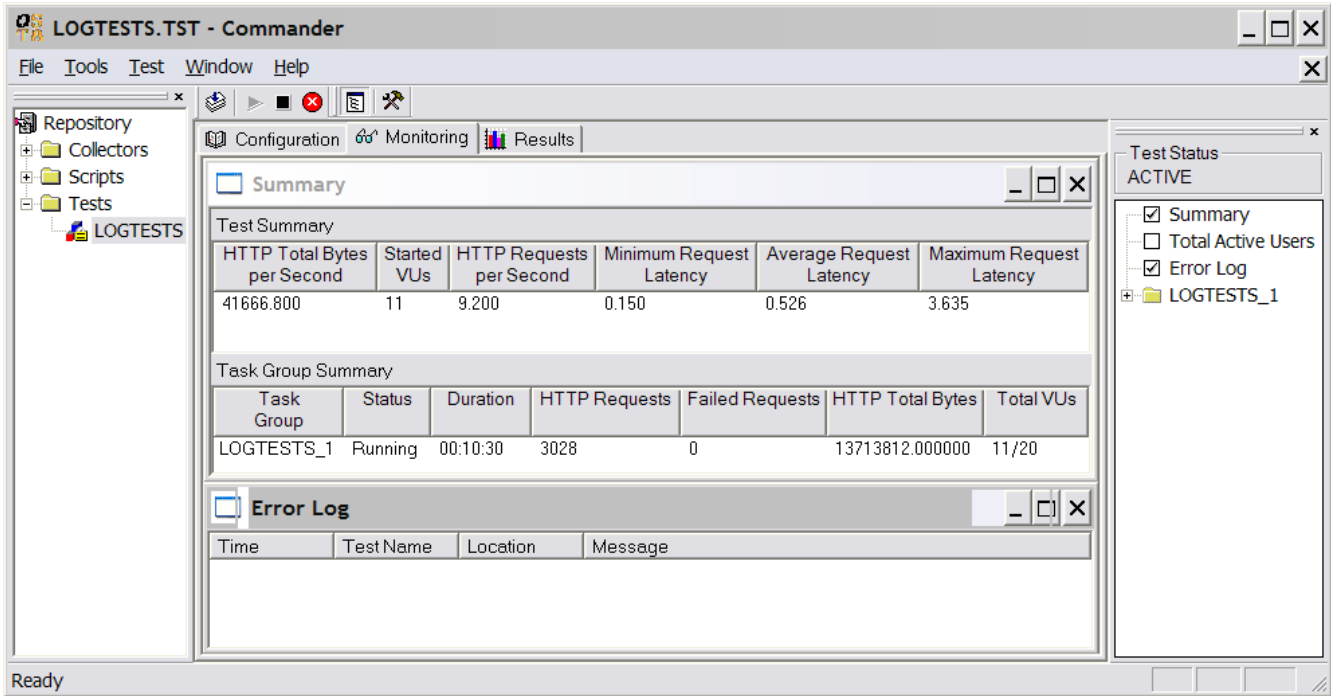
ERR_LABEL:
  If (MESSAGE <> "") Then
    Report MESSAGE
  Endif

Exit
```

Screenshots showing the test setup in OpenSTA Commander:



Screenshot of OpenSTA Commander's Monitoring Tab during the test run:



Screenshot of OpenSTA Commander's Results Tab after the test run (Test Configuration):

