Load Testing Using Libindy

Running Performance and Stability Tests

**Simple Version**

1. Setup
2. Installing Libindy
3. Executing the Scripts

# Files and Folders

I have been creating a "perf" directory on the machines running libindy, but you can create any directory you want.

Place all the python scripts in your directory (like perf).

perf/

- perf\_runner.py

- perf\_add\_requests.py

- perf\_get\_requests.py

- perf\_cleanup.py

- requests\_sender.py

- requests\_builder.py

- config.json (contains the pool\_genesis\_file path)

# Setup

Create a test pool.

Create one or more machines from which you will run libindy to simulate the client connections.

Install Libindy (see below)

Copy test scripts into a directory on the libindy machines.

Run scripts (see below)

# Installing Libindy

### First (Add repos)

sudo apt-key adv --keyserver [keyserver.ubuntu.com](http://keyserver.ubuntu.com/) --recv-keys BD33704C

sudo apt-key adv --keyserver [keyserver.ubuntu.com](http://keyserver.ubuntu.com/) --recv-keys 68DB5E88

sudo add-apt-repository "deb [https://repo.sovrin.org/](https://repo.evernym.com/libindy_crypto)deb xenial stable"

sudo add-apt-repository "deb<https://repo.sovrin.org/sdk/deb> xenial stable"

sudo add-apt-repository ppa:jonathonf/python-3.6

sudo apt update

### Second (Install packages)

##### Python 3.6

sudo apt install python3.6 -y

DEB Packages including Libindy

sudo DEBIAN\_FRONTEND=noninteractive apt-get install -y debsigs debsig-verify apt-transport-https python-pip python3-pip python3.5-dev python3.6 libsodium18 libsqlite0 libindy-crypto libindy

##### Libindy Python Wrappers

sudo pip3 install python3-indy

sudo pip3 install --upgrade python3-indy

**NOTE**

Make sure you have the indy files in the python3.6 directory (/usr/local/lib/python3.6/dist-packages/indy). If you do not you may need to copy or symlink the /indy directory in python3.5 to python3.6

/usr/local/lib/python3.5/dist-packages/indy to /usr/local/lib/python3.6/dist-packages/

# Understanding the scripts

The script "perf\_runner.py" is the script used to run all the automation. perf\_runner.py calls either perf\_add\_requests.py or perf\_get\_requests.py to execute either ADD or GET request (nym, schema, claim or attribute). The perf\_add\_requests.py and perf\_get\_requests.py call requests\_builder.py to build all requests into files and requests\_sender.py to read all requests from that files, then submit them to the ledger.

### config.json

This file contains some necessary configuration to run the script. Now, this just contain path to you pool genesis file location.

Template:

{

“pool\_genesis\_file”: [path\_to\_you\_genesis\_file]

}

### perf\_runner.py

There is several parameters possible with this script, the main parameters are:

“**-a**”: to show that you want to submit ADD request (runner will call perf\_add\_requests).

“**-g**”: to show that you want to submit GET request (runner will call perf\_get\_requests).

“**-t**”: to show that you want to simulate traffic.

“**-l**”: to show that you want to perform load testing.

“**-k**”: kind of request (schema, nym, claim, attribute).

“**-d**”: the directory you want to store request information in case you want to submit ADD request or the place you want to collect request information in case you want to submit GET request.

“**-c**”: the number of clients you want to simulate.

“**-s**”: number of thread of each client.

“**-n**”: number of ADD requests will be submitted in mode “**-a**” and “**-l**”. In case you use mode “**-t**”, this argument is the number of transactions will be submitted in a set.

“**-to**”: time limit. This argument is only visible when using two mode “**-t**” and “**-l**”.

“**--init**”: to build “GET” request, we need to send “ADD” request first. This argument is the number of “ADD” requests will be sent to ledger to initiate samples for “GET” requests (this argument is only visible in mode “**-t**”).

**How to run**

**-a** (for sending "ADD" request tests), **-n** (number of transactions), **-c** (number of clients (threads)), **-s** (number of threads per client), **-d** (path to save requests info), **-k** (kind of "ADD" request (nym, schema, attribute, claim)).

E.g: If you want to create **20 clients**, each client sends **200** "ADD" attribute requests with **5 threads**, the command should be like this:

python3.6 perf\_runner.py -a -k attribute -c 20 -s 5 -n 200

**-g** (for sending "GET" request tests), **-c** (number of clients (threads)), **-s** (number of threads per client), **-d** (path to requests info), **-k** (kind of "GET" request (nym, schema, attribute, claim)).

E.g: If you want to get requests info at "**/home/account/info**" to build "GET" attribute requests then sending built requests by **20 clients** with **5 threads** per client, the command should be like this:

python3.6 perf\_runner.py -g -k attribute -d /home/account/info -c 20 -s 5

**-t** (traffic tests), **-c** (number of clients (threads)), **-n** (number of transaction in a set), **--init** (number of samples for "GET" requests), **-to** (time limit).

E.g: If you want to simulate traffic in **200 seconds** with **100 clients** and **100 requests** per set and you want to initiate **100 samples** for "GET" requests, the command should be like this:

python3.6 perf\_runner.py -t -c 100 -n 100 -to 200 --init 100

**-l** (load test), **-n** (number of transactions), **-c** (number of clients (threads)), **-to** (time limit).

E.g: If you want to perform load testing in 200 seconds with 100 clients and 1000 requests, the command should be like this:

python3.6 perf\_runner.py -l -c 100 -n 1000 -to 200

### perf\_tester.py

Contain base class of all performance tester classes. This class contains some common variables and method that perform common step.

### perf\_add\_requests.py

There are several parameters possible with this script. The main parameters are:

“**-n**”: the number of transactions to run.

“**-s**”: the number of threads.

“**-d**”: the directory you want to store requests information when sending ADD request.

“**-k**”: kind of request (schema, NYM, claim, attribute).

The script will create the directory in the location it is running from. The directory will store text files with all the requests information (did, attributes, …) created in the test run. The text files are used by perf\_get\_requests.py to run GET request lookups.

### perf\_get\_requests.py

There are several parameters possible with this script. The main parameters are “**-s**” and “**-d**” for the number of threads to run and the location where the files containing requests information to lookup are located.

You will need to run perf\_add\_requests.py to generate the requests information to lookup before running perf\_get\_requests.py.

### perf\_cleanup.py

This script just removes the .indy/pool and .indy/wallet directories. There is an issue with libindy where it will not run if there is already a pool with the same name created. To work with this issue we clean up the pool and wallets so each run is clean.

### perf\_traffic.py

The script simulates the real time traffic. This script will send several set within specified length of time. In each set, a specified number of transactions will be submitted onto ledger and between two set, the system will be delayed in a random length of time (from 1 to 10 seconds).

There are several parameters possible with this script. The main parameters are:

“**-n**”: the number of transactions to submit in a set.

“**-c**”: the number of clients.

“**-to**”: the time limit.

“**--init**”: to build “GET” request, we need to send “ADD” request first. This argument is the number of “ADD” requests will be sent to ledger to initiate samples for “GET” requests.

### perf\_load.py

The script performs load testing which try to send a specified number of “ADD” requests to ledger in a length of time.

There are several parameters possible with this script. The main parameters are:

“**-n**”: the number of transactions to submit in a set.

“**-c**”: the number of clients.

“**-to**”: the time limit.

### requests\_builder.py

For each of clients, this script will build the requests based on the number of transactions. Next, the requests\_builder writes all requests into the files (the number of files based on the number of threads will be created by client).

### requests\_sender.py

This script will receive the list files of requests from “requests\_builder.py”, create threads based on number of file and submit those requests to ledger. Then, delete the requests file that completely submitted to ledger.

Beside of sending requests, sender will take the first and the last time it receives response from ledger and perf\_runner.py will make result base on those time of sender.

### utils.py

This file contains all utility function that need for all scripts above.

### Example for perf\_runner.py

To run ADD and GET nym request.

ADD nym: python3.6 perf\_runner.py -a -k **nym** -d /home/account/ -c 2 -s 2 -n 10

GET nym: python3.6 perf\_runner.py -g -k **nym** -d /home/account/**nym** -c 2 -s 2

To run ADD and GET schema request.

ADD schema: python3.6 perf\_runner.py -a -k **schema** -d /home/account/ -c 2 -s 2 -n 10

GET schema: python3.6 perf\_runner.py -g -k **schema** -d /home/account/**schema** -c 2 -s 2

To run ADD and GET attribute request.

ADD attribute: python3.6 perf\_runner.py -a -k **attribute** -d /home/account/ -c 2 -s 2 -n 10

GET attribute: python3.6 perf\_runner.py -g -k a**ttribute** -d /home/account/**attribute** -c 2 -s 2

To run ADD and GET claim request.

ADD claim: python3.6 perf\_runner.py -a -k **claim** -d /home/account/ -c 2 -s 2 -n 10

GET claim: python3.6 perf\_runner.py -g -k **claim** -d /home/account/**claim** -c 2 -s 2

# Script output

#### **Perf\_runner.py will output two types of logs.**

1. A set of log files which will roll over. Each log file uses the current date and time to make them unique.
   1. Example log from running GET\_NYM

2-perf-getnym\_12-02-2018\_17-32-33.log

======== Request: {"reqId":1518431553583690483,"identifier":"Th7MpTaRZVRYnPiabds81Y","operation":{"type":"105","dest":"UprsePQs15QggjW2pHj62b"}}

======== Status: Passed

======== Processed time: 0.0065724849700927734seconds

1. The second file is the test results file that gives a summary of the total test time and estimates on transactions per second. The estimate is based off the number of clients (threads), transactions, and number of seconds it took to run. **NOTE: The only thing I am using currently is the number of seconds. The minutes are correct, but the output for the seconds is the total number of seconds so it does not display correctly with HH:MM:SS.**
   1. result\_28-02-2018\_11-10-07.txt

----------- Total time to run the test: 0h:0m:0s -----------

Kind = send 'GET nym' requests

Clients = 1

Fastest transaction = 0.01311802864074707

Lowest transaction = 0.023035049438476562

Transaction per client = 100.0

Total requested transactions = 100

Total passed transactions = 100

Total failed transactions = 0

Average time of a transaction = 0.0018291640281677245

Estimated transactions per second = 546

**perf\_add\_requests.py**

The output from this script is written into ‘[path\_of\_your\_script]/request\_info’ if command line argument ‘-d’ is missing. The output is contains several lines, each line is a json and contains necessary information to build a GET request. **NOTE: The reason for this output is so when running perf\_get\_requests.py it can lookup actual which added by perf\_add\_requests.py (schema, claim, nym or attribute) on the ledger.**

1. nym\_requests\_info\_140719776364288\_13-02-2018\_07-57-46.txt

{"kind": "nym", "data": {"target\_did": "thHjHexbrQciEAuxk8YaM"}}

{"kind": "nym", "data": {"target\_did": "3YtAi4hkJFQGHDteq1JFh1"}}

{"kind": "nym", "data": {"target\_did": "AtWXTPDYPzyg3DGkCvEHxi"}}

{"kind": "nym", "data": {"target\_did": "9Y8XRryE6VLsDnwuVFQdgU"}}

{"kind": "nym", "data": {"target\_did": "5YALfPJALt1a61UJbdz9c6"}}

{"kind": "nym", "data": {"target\_did": "XoA1wbHLAe5y3SqyYmGuKJ"}}

{"kind": "nym", "data": {"target\_did": "nER88jgGBaivNRtLZKL4d"}}

{"kind": "nym", "data": {"target\_did": "C396mqGt7Uoe1RALvzCxGL"}}

{"kind": "nym", "data": {"target\_did": "Jv68YEqNb2nP3m6qUtZTvc"}}

{"kind": "nym", "data": {"target\_did": "CqsR6sLXzVUaF2hHU5Aidd"}}

1. schema\_requests\_info\_140544183269120\_13-02-2018\_08-28-12.txt

{"kind": "schema", "data": {"name": "testVZN29ZS6TM2A4W73", "version": "1.0", "dest": "V4SGRU86Z58d6TV7PBUe6f"}}

{"kind": "schema", "data": {"name": "testODT8IV2NZKZAYSZJ", "version": "1.0", "dest": "V4SGRU86Z58d6TV7PBUe6f"}}

{"kind": "schema", "data": {"name": "testGA9P33GQAJI59Q39", "version": "1.0", "dest": "V4SGRU86Z58d6TV7PBUe6f"}}

{"kind": "schema", "data": {"name": "testM5GAHE945CCJ61XD", "version": "1.0", "dest": "V4SGRU86Z58d6TV7PBUe6f"}}

{"kind": "schema", "data": {"name": "testAEV8V9KL2MQ7J7TV", "version": "1.0", "dest": "V4SGRU86Z58d6TV7PBUe6f"}}

{"kind": "schema", "data": {"name": "test1AXR0W28F5CERQH7", "version": "1.0", "dest": "V4SGRU86Z58d6TV7PBUe6f"}}

{"kind": "schema", "data": {"name": "test0Y8MPBHDD63TL1EA", "version": "1.0", "dest": "V4SGRU86Z58d6TV7PBUe6f"}}

{"kind": "schema", "data": {"name": "testA983P5ZHCO4499RT", "version": "1.0", "dest": "V4SGRU86Z58d6TV7PBUe6f"}}

{"kind": "schema", "data": {"name": "test5OLBLIYNJDGPW7CW", "version": "1.0", "dest": "V4SGRU86Z58d6TV7PBUe6f"}}

{"kind": "schema", "data": {"name": "test72991M54AG4MPU53", "version": "1.0", "dest": "V4SGRU86Z58d6TV7PBUe6f"}}

1. claim\_requests\_info\_140712484370176\_13-02-2018\_08-28-17.txt

{"kind": "claim", "data": {"issuer\_did": "FwdtK3tuqtzXRT9Q5wh8T", "seq\_no": 85862, "signature\_type": "CL"}}

{"kind": "claim", "data": {"issuer\_did": "DVvno4wPQyUjtZroQp3CPp", "seq\_no": 951629, "signature\_type": "CL"}}

{"kind": "claim", "data": {"issuer\_did": "UCbk4VsgXwkZKD9Lhh73Pt", "seq\_no": 552134, "signature\_type": "CL"}}

{"kind": "claim", "data": {"issuer\_did": "JogyYXjRMbx8fXUfKfDLrD", "seq\_no": 456222, "signature\_type": "CL"}}

{"kind": "claim", "data": {"issuer\_did": "EmmyRB2UAvEkaq2SPP5QZq", "seq\_no": 228539, "signature\_type": "CL"}}

{"kind": "claim", "data": {"issuer\_did": "LnUGEXBV5EqksXPsampRVi", "seq\_no": 470009, "signature\_type": "CL"}}

{"kind": "claim", "data": {"issuer\_did": "9aNkD1wm7yvTnFSMGKDphi", "seq\_no": 501530, "signature\_type": "CL"}}

{"kind": "claim", "data": {"issuer\_did": "CxnL6zzE3cY4spTCKhe4bB", "seq\_no": 197003, "signature\_type": "CL"}}

{"kind": "claim", "data": {"issuer\_did": "Njxata1jy18JtS3p7Yjgxf", "seq\_no": 80008, "signature\_type": "CL"}}

{"kind": "claim", "data": {"issuer\_did": "Q57eXwFhQQXjgEW9eoLSnu", "seq\_no": 591197, "signature\_type": "CL"}}

1. attribute\_requests\_info\_139781976147712\_13-02-2018\_08-28-26.txt

{"kind": "attribute", "data": {"target\_did": "DSQgTq3y5QPmABuBzwPgKJ", "raw\_name": "endpoint"}}

{"kind": "attribute", "data": {"target\_did": "3pdmpmytiFk8QLp9DFgijE", "raw\_name": "endpoint"}}

{"kind": "attribute", "data": {"target\_did": "5c6Lb9fk9DRQ24wtn3tMki", "raw\_name": "endpoint"}}

{"kind": "attribute", "data": {"target\_did": "KQLg3NLwAb3c6pVquLKKAr", "raw\_name": "endpoint"}}

{"kind": "attribute", "data": {"target\_did": "N5M6ysQqhyRtUfHjroevmj", "raw\_name": "endpoint"}}

{"kind": "attribute", "data": {"target\_did": "8RitEJxP7im1DhL5ad59ad", "raw\_name": "endpoint"}}

{"kind": "attribute", "data": {"target\_did": "U49ANKGQ1swFU3CjADZgcm", "raw\_name": "endpoint"}}

{"kind": "attribute", "data": {"target\_did": "QRXpYKLPWYszq4AHmWVXkW", "raw\_name": "endpoint"}}

{"kind": "attribute", "data": {"target\_did": "WTQuWDT4PcnREki1UYAxFs", "raw\_name": "endpoint"}}

{"kind": "attribute", "data": {"target\_did": "QRgNYAtkymXJJZ4zkk1UHG", "raw\_name": "endpoint"}}

# Measuring results

The measurement of transactions per second is done be getting the epoch time stamp from the first response that was received and subtracting it from the epoch time stamp of the last response that was received. The difference in the time stamps gives you the total number of seconds.

Dividing the total transactions by the total seconds of what was received from the ledger gives the number of transactions per second. And, average time to process a transaction is determined by dividing number of total transactions by total seconds.

Note: this is visible for both perf\_add\_requests.py and perf\_get\_requests.py.