Introduction

These tools serve multiple purposes. At one level they provide some basic benchmarking capabilities, that is running swift object PUT, GET and DELETE tests of a fixed size and measuring their rates in terms of MB/sec ond and IOPS. For many users, all they want to do is measure the performance of an existing configuration and for them this is sufficient functionality.

At the other extreme are those users who want to characterize and stress swift, with the goals of observing scaling, tuning and/or exposing problems. These users will find a number of capabilities which allow them to measure individual operational latencies and even report the transaction IDs of any whose latency exceeds a specified threshold.

The kit currently contains three scripts: getput, gpmulti and gpsuite. The first is a single-node test tool and for many users this is all they will ever really need to use. The second, is really more of a helper script called by gpsuite. It facilitates running getput in parallel on multiple nodes and summarizing the results. However, due to the complexity of all gpmulti's switches, it's far simpler to use *gpsuite* to run parallel tests.

You should also note that this document is not intended to be a comprehensive description of best practices for benchmarking or trouble-shooting, nor will it cover all the various switches in detail, as there is sufficient information in the man pages. Rather, it will focus on the basics which should hopefully be sufficient to get one started as well as provide some guidelines to developing your own, more comprehensive tests.

One final thing to note and it's important – getput is based on the python-swiftclient bindings and so while running other tools should provide similar results, those other bindings may result in different behaviors. At the very least if/when differences are discovered they should be reported as they may indicate a bug in one of the bindings.

Credentials

The most important thing to do before even trying to use these scripts is to make sure you're using valid V1 or V2 credentials and for simplicity and it's assumed you've exported them into your environment before beginning. If you're planning on running parallel tests you need to create a file that contains them in a form like this for V1 credentials. A similar format would also be user do V2 noting these variable all start with OS_

```
export ST_AUTH=https://endoint:35357/auth/v1.0/
export ST_USER=projectid:projectname
export ST_KEY=password
```

This format makes it real simple to either *source* the file or reference it by name using the appropriate switch in a command. To be sure these credentials are valid, after exporting to your environment run

the command *swift stat* and you should see a summary of your account information. If this command fails, you do not have valid credentials and should not proceed until the problem is resolved.

Installation

First a little terminology: The *tester* is the Linux user who will be running the tests. A *test client* or simply a *client* is a machine that is used to generate traffic between itself and swift. When running single client tests, you simply unpack the installation tarball and run the command *sudo python setup.py install.* The 3 main scripts will be installed into /usr/bin and the manpages placed in their appropriate directories as well. This document along to an abbreviated *getting_started.txt* will be placed in /usr/share/doc/gptools.

When running parallel tests you need to identify a single node for the *control* node as well as any additional test clients you may wish to use. The *control* functionality can be provided by one of the clients as well rather than using an additional node. The one requirement is that there is passwordless ssh access between the control node and all the test clients for the tester.

All parallel tests are run out of the tester's home directory, so make sure there is a copy of getput in it, which you can do by simply copying /usr/bin/getput from the control node to all the other nodes, including the control node itself if it will be functioning as a client node as well.

Finally, you must copy the credentials file to the tester's home directory on all the test clients as well.

Single Node Operations

Running your first test

To get started, we're going to focus on running *getput* on a single testing machine and to do so you'll need to specify 5 basic switches, which intentionally do not have any defaults. They are:

Container Name
Object Name
Number of Operations
Object Size
Test(s) to run

Getput always appends an ascending object number, starting at 1, to the object name to assure uniqueness. It will also include a rank and process number, which will be explained later but for now they will both be 0.

The size is the object size in bytes, and you can append a K, M or G to any number as a convenience.

Finally the test can be any combination of p, g and/or d with 2 caveats: You can't GET an object if it hasn't already been written. After a delete test, getput will attempt to delete the container as a way of

cleaning up after itself. If you have more objects in a container than you're deleting, getput will generate an error when it tries to remove the container.

In the following example, we're writing 3 objects named *oname* to a container named *cname* and them immediately reading them back. You should note that to make the following output fit, 2 trailing fields that report the CPU load and type of compression have been left off:

```
$ getput -ccname -ooname -n3 -s1k -tp,g
Rank Test  Clts Proc  OSize  Start     End        MB/Sec   Ops   Ops/Sec Errs Latency  Median    LatRange
0    put      1    1     1k  12:51:23  12:51:23     0.02     3     17.97    0   0.056   0.050  0.03-00.09
0    get      1    1     1k  12:51:23  12:51:24     0.13     3    134.73    0   0.007   0.007  0.01-00.01
```

Now let's list the contents of that container to see what the names looks like:

```
 $ swift list cname
oname-0-0-1
oname-0-0-2
oname-0-0-3
```

Looking more closely at the output, the first field is the rank, which as previously mentioned is zero when running getput on a single node, unless you specify *–rank.* The Clts field is always 1 when running on a single client and only different when run as part of *gpsuite.* Proc names the number of threads being run, which is this case is also 1.

The next several fields should be self-explanatory, showing the object size and the start/end times for the test. Also reported is the rate, number of operations and IOPS. If there were any errors reported their number will also be reported, noting if there are more than 5 (and one hasn't changed the limit with *–errmaxI)*, execution will be terminated.

Getput is all about reporting latency information and the next three fields report the average and median latencies as well as their range. While these are generally enough information to get a good idea of the overall behavior, a tighter range is better than a wider one, one can also request a latency distribution histogram be displayed. More on this later.

Running sets of tests

Once you get comfortable running simple tests like these, you may be quickly tempted to run some scripts that run getput multiple times with different values. Don't do it! Getput already has this capability and you can easily rerun tests using different object sizes or even different numbers of threads (remember, the default number of threads is 1) by simply specifying them separated by commas.

Let's repeat the previous test using 2 different object sizes and 2 different process count, noting this time for the second set the Proc field becomes meaningful:

```
 $ getput -ccname -ooname -n3 -s1k,2k -tp,g --proc 1,2
Rank Test  Clts Proc  OSize  Start     End        MB/Sec   Ops   Ops/Sec Errs Latency  Median    LatRange
```

```
0    put      1    1     1k  13:18:01  13:18:01     0.01     3     14.43    0   0.069   0.051  0.04-00.12
0    get      1    1     1k  13:18:01  13:18:01     0.13     3    130.62    0   0.008   0.008  0.01-00.01
0    put      1    1     2k  13:18:01  13:18:01     0.06     3     30.58    0   0.033   0.035  0.03-00.04
0    get      1    1     2k  13:18:01  13:18:02     0.25     3    127.14    0   0.008   0.008  0.01-00.01
Rank Test  Clts Proc  OSize  Start     End        MB/Sec   Ops   Ops/Sec Errs Latency  Median    LatRange
0    put      1    2     1k  13:18:02  13:18:02     0.03     6     30.72    0   0.072   0.059  0.05-00.16
0    get      1    2     1k  13:18:02  13:18:02     0.24     6    243.79    0   0.008   0.008  0.01-00.01
0    put      1    2     2k  13:18:02  13:18:02     0.12     6     62.11    0   0.032   0.031  0.03-00.04
0    get      1    2     2k  13:18:02  13:18:03     0.51     6    258.68    0   0.008   0.008  0.01-00.01
```

Looking at the container contents we can now also see the process number in the second field, remembering the first number is the rank or client number:

$ swift list cname
oname-0-0-1
oname-0-0-2
oname-0-0-3
oname-0-1-1
oname-0-1-2
oname-0-1-3

But wait a minute. What if you're running a test with hundreds or even thousands of operations? What if you want to do this with 1 and then 50 processes, won't the 50 process test take a lot longer to run? Yes it will and the answer to this is to specify the test duration in seconds rather in the number of operations by using the *–runtime* switch. One should note that when running tests in this manner, each thread may actually write a different number of objects and when reading them back need to know how many were written so they don't get errors. Not to worry, getput tracks this for you.

Let's talk more about latencies

As has already been stated, having a wide latency range tells you some operations took a lot longer than others, but you have no way of knowing how many. It has also been observed that the latency timing tend to fall into 2 broad areas, small object from 0-.5 seconds and larger objects from 2-5 seconds. Under times of stress, smaller objects can take multiple seconds and larger object 10s of second.

And that's where the latency distribution histogram comes in and you can request this with the *–ldist* switch. By trying to come up with a set of bucket sizes that address these bi-modal time distributions, there are 11 predefined buckets and the argument of the switch specifies the fractional number of digits, so for *–ldist 1* we see 6 buckets from 0 to 0.5 seconds (.6 through .9 fall into the 0.5 bucket) and 5 more buckets from 1.0 to 5.0. If *–ldist 0* is specified one gets buckets from 0 to 5 and 10 to 50.

So now let's run a test that varies the number of processes enough to have a visible effect on the latencies, noting I'm leaving off a bunch of fields on the left-hand side to make everything fit.

```
Rank Test  Clts Proc    LatRange   0.0   0.1   0.2   0.3   0.4   0.5   1.0   2.0   3.0   4.0   5.0
0    put      1    1  0.04-00.13   181     3     0     0     0     0     0     0     0     0     0
0    put      1   16  0.02-00.41  2251   144    46    12     1     0     0     0     0     0     0
0    put      1   48  0.02-00.59  3922   915   165   142    52    12     0     0     0     0     0
```

As you can see, almost all of the latencies for single process test were <0.1 seconds. For 16 processes they started to drift out to 0.4 seconds and when running with 48 processes they spread even wider.

These numbers are very useful when stressing the environment and trying to identify when the behavior starts to go wrong.

When one wishes to dig deeper into what exactly is going on and whether this is a swift problem or perhaps a networking or other problem, it can be useful to look at the swift server/proxy logs for detailed timing information. However identifying those log entries that correspond to individual operations can be difficult.

Fortunately, swift assigned a unique transaction ID to every operation and getput provides the *–latexc* switch which allows you to specify latency threshold in fractional seconds, such that any operation taking longer than this will be reported on the terminal with the actually latency time as well as the corresponding transaction ID which can then be used to explore the swift logs. There are also options to tell getput to log the exceptions to a file as well as to exit upon encountering the first exception.

Container Naming

By default, when you tell getput the name of a container, that name is shared by all processes whether on the node the test is being run from or from all nodes when running parallel tests. By design, this puts the maximal strain on container services since each thread end up accessing the same container.

However, there may be times in which you would like a different container sharing scheme and that's where the *–ctype* switch comes into play and it can take one of three values:

- *shared* containers are the default and all processes write to the same one
- *bynode* containers share the same name for all processes running on the same node and so has no affect for single-node tests
- *byproc* containers have a unique name for every process on every node

When you run a test and want to make sure a brand new container is being accessed, you may want to make sure it is given a new name rather than reusing an existing name in which some data structures may have been cached. The *–utc* swift will append the UTC time at the start of the tests to the container name(s) in addition to honoring the *–ctype* switch. When running multiple tests in which you're changing the object size of number or processes, a new UTC value will be used each time.

As an aid to help prevent the number of containers becoming unwieldy after many tests, getput will attempt to delete the container after a DELETE test, unless you specify *–contnodelete.*

Random I/O

There is actually a fairly complete writeup on this topic in the manpage and rather than duplicate what it says, the reader is referred to that as a reference.

Parallel Operations

Concepts

Before running your first parallel tests, that is tests from multiple clients, it will be useful to understand how these tests are synchronized so they all begin at the same time. Rather than using any communications based synchronization in which tests are started and then wait for a *start the test* message to be received, a much simpler mechanism has been chosen.

When one runs a parallel test, a time in the future is chosen, typically 10-30 seconds from the time you run the test command. At that point, the corresponding future time is calculated and all clients simply told to wait until that time has occurred to begin test execution. If the time is too short, some tests may miss the window and begin late, generating an appropriate warning. If the time it too long, the test is simply started a little later. A time of about 30 seconds for getting many clients started together has been pretty successful. It also gives the environment some time to quiet down between tests.

The names of the test clients are stored in a file and the controlling test program, *gpsuite,* reads that file and simply executes *ssh client-name gpmulti switches…* to each machine, which is the reason why passwordless ssh access to each client from the control node is required.

Remember how getput can keep track of how many operations were performed by each process so GETs following PUTs will know how many objects were written by each thread? One of gpmulti's jobs is to track the number of operations across all client threads and to synchronize the tests between all of them. Gpmulti is also responsible for collating the output from all the instances of getput.

The config file

The tests themselves are defined in a config file, that is any file with the extension *.conf.* The very first thing done is to load /etc/gpsuite.d/gpsuite.conf if it exists. Then, any other config files in /etc/gpsuite.conf are loaded in their sort order. Finally any files in the current directory with a name in the format of gpsuite-*.conf are loaded. The reason that must begin with gpsuite is to avoid trying to read any other non-gpsuite conf files that may be present.

The config files consist of stanzas in the following format:

```
[testname]
parameter1 = value
parameter2 = value
parameter3 = value
…
```

When gpsuite is run, it loads all the parameters with the *testname* for that stanza. If it encounters a parameter with the name *include,* its value is used as a pointer to a different stanza and its parameters override and previously loaded parameters of the same name. This process continues until the requested test's stanza is loaded. This is why the order the stanzas are encountered/included are so important, since multiple conf files can contain stanza with the same names.

Gpsuite does include a *–list* swift which will list all the different stanzas along with their *comment* parameter's value as well as a *–dryrun* switch which will show the final values for all the parameters and their values associated with the selected test. You should also note that as part of this, gpsuite will also verify connectivity with all the nodes that are to be used in the test and also report the gpmutli command that will executed, making it possible to copy the gpmulti command and execute it manually.

The parameters themselves fall into two broad categories, the first is a subset of those used when running getput on a single node and include things like container/object names, runtime, object sizes, process counts and tests to run along with a few others.

The second set apply to running tests in parallel such as the time to delay before the tests start, the name of the file that contains the names of the test clients and even the name of the directory to write the test log into.

The names and description of all possible parameters are described in the beginning of /etc/gpsuite.d/gpsuite.conf

Running your first parallel test

To make this really easy, we're going to use the same node as you've used for the single-node tests as both the control and client node and only run a single clinet. The default gpsuite.conf file has a stanza for a test called firsttest which looks like this, noting that the type sets the credentials file to *first-creds,* the nodes file to *first-nodes* the logfile name to *firstest.log,* the tests to run to *p,g,d* and the object/container names to *first* as well.

```
[firsttest]
comment = first parallel test
type    = first
sizes   = 1k
runtime = 5
syntime = 5
```

so now all you need to is create a credentials file in your home directory along with a file named *firsttest-nodes* and make its only entry a single line consisting of *localhost* or your ip address or your hostname. It doesn't matter as long as you can passwordlessly ssh to whatever string you've placed in that file.

If all goes well, you should see the following:

```
$ gpsuite --suite firsttest
Writing test results to ./20140523-180902-firsttest.log
```

And the contents of the resultant file should so the results of a 5 second PUT and subsequent GET and DELETE tests. Furthermore, the container *first* should have been removed.

If you have not set up your credentials or passwordless ssh, you will get errors. In many cases you can figure out what the problem is but if not, appending –d2 to the gpsuite command will tell it to echo the gpmulti command it is about to execute and so you can cut/paste/execute the command and run it to see what happens. If it too fails and you don't know why, append –d2 to it and it will show you the ssh commands it uses to run getput which you can also run manually. In most cases this should be sufficient to figure out what is going wrong, most likely an authentication or configuration problem.

Running a multi-client suite

Assuming you've succeeded in getting your one client test to run successfully, you can now move on to multiple clients. To do this we're also going to help you get familiar with the way the config files work. The plan is to use the first node the tests were run on as the control node, so simply indentify one or more nodes you want to use for your new test and add them to the *first-nodes* file, making sure you can passwordlessly ssh to them. Now copy getput and first-creds to their home directories. The last thing you need to do is tell gpsuite to use multiple nodes when running the tests.

The way we're going to do this to help you become more comfortable with multiple conf files as well as inheriting setting, is in your home directory create a new config file and let's call it gpsuite-multi.conf. Simply add the following to it:

[multinode]
include firsttest
comment = Multinode test
maxnodes = 2

As you hopefully recall, the *include* will define al the same parameters as were used in your first test so all we need to do for this test is to tell it to run using 2 nodes (or however many you want to use) from the file named *first-nodes.* That's all these it to it.

To verify everything is correctly defined, first list the tests available to you like this:

```
$ gpsuite --list
example                Example Configuration
firsttest              first test
multinode              Multinode test
```

and next do a dry run like this:

```
ubuntu@gptest:~$ gpsuite --suite multinode --dryrun
cname        = first
comment      = Multinode test
creds        = first-creds
maxnodes     = 2
nodes        = first-nodes
oname        = first
procs        = 1
runtime      = 5
sizes        = 1k
synctime     = 5
tests        = p,g,d
type         = first

/usr/bin/gpmulti -cfirst -ofirst -s1k -tp,g,d --nodes first-nodes --
numnodes 1 --procs 1 --runtime 5 --creds first-creds --ldist 1  --sync
5 | tee -a ./20140523-192138-multinode.log
```

and you're ready to go. Simply remove the *–dryrun* switch and run your test and examine the log file when you're done. At the point you can change the number of nodes or any of the other parameters

available to you. Be sure to read the header of /etc/gsuite.d/gpsuite.conf to see all the options available to you to use.