

A Tutorial to Planet-lab

Brian Sanderson - bts7@byu.edu - version 1.0 - 2007-09-17

I. INTRODUCTION

Planet-lab is a great tool for performing large-scale Internet studies. Its power lies in that it runs over the common routes of the Internet and spans nodes across the world, making it far more “realistic” than a simulation. But this reality comes at the price of route-failures, node outages, ssh-key sharing issues, and other network realities which make managing experiments difficult.

Fortunately, there are several tricks of the trade which I will outline in this paper. I have been working with Planet-lab for about 3 years, and have made plenty of mistakes in that time. I wish to save you some of that frustration, if possible.

A. Prerequisites

Before you begin this tutorial you should have a Planet-lab login as well as a slice assigned to that login. Dr. Zappala can set you up with both of these.

II. SSH KEYS AND SSH-AGENT

Computers on Planet-lab are called nodes. These nodes can only be managed via the OpenSSH protocol (i.e., secure shell or SSH). Even then, they can only be accessed using a setup public/private key pair which you create. To create a public/private key pair use the following command:

```
ssh-keygen -t rsa -f planetlab-key
```

You will be prompted for a password. This password is used to encrypt your private key and is all that protects others from using your slice. I highly suggest using a strong password for this key.

After the program runs, it will generate a planetlab-key file, which is your encrypted private key. `ssh` will not let you use the private key until you remove all group and other access to it. `planetlab-key.pub` is the public key and needs to be uploaded to your Planet-lab account on `planet-lab.org`. Planet-lab then distributes the key to all the nodes in your slice. Actually, each Planet-lab run a cron job which tells it to download any public keys that are assigned to it. This typically runs every 15 minutes.

We will get to adding nodes to the slice later, but for now make sure you have at least one node added to your slice.

To connect to a Planet-lab node you can now use the following syntax:

```
ssh -i planetlab-key slicename@host.edu
```

This lets you log into a node, typing in the private key password. If the connection timed out, it may be the node is down, but most likely it is because BYU has issues connecting to off-campus Planet-lab nodes. In general, trying the same command outside the domain of the BYU packet-eating firewall will yield more successful connections.

Because typing in the ssh-key password is cumbersome (yet secure) and that we will need to log in without entering in a password. To accomplish this, OpenSSH has a program called ssh-agent which allows private keys to be decrypted and stored in memory for password-less logins. To run an ssh-agent type:

```
eval `ssh-agent`  
ssh-add planetlab-key
```

You will need to type in the password once for the private key, but thereafter you should be able to ssh into nodes without the -i switch or entering in the password. The catch is that you need to run the ssh-agent agent every time (or recover the environment variables that were set with the last one). Some people add these lines to their `.bashrc` file, or use `expect` to automatically enter in the password. Insert your creativity here.

Before we leave the subject of SSH keys and agents, there is one more thing to change. You will notice that the first time you connect to a node via SSH, OpenSSH asks you whether or not you wish to accept the node's public key. You then have to type 'yes' and then you can authenticate. To eliminate the key check, you need to add a file called 'config' to your `$HOME/.ssh` folder. In this file, add the line:

```
StrictHostKeyChecking no
```

III. USING THE DOWNLOAD-NODES SCRIPT

Planet-lab comes with a full blown API to perform tasks such as adding nodes to slices, getting user permissions, getting lists of available nodes, etc. It is beyond the scope of this tutorial to give an introduction to the Planet-lab API. Refer to the Planet-lab.org API documentation for any questions.

`download-nodes.py` is a script I wrote to handle common functions with slices. It will add all nodes to a slice that are currently in boot status, meaning they're ready for production. It will also remove nodes that are no longer in boot status if you specify the `-r` argument.

In addition to those tasks, it produces three files which will be of use later. It produces `current-nodes.txt` (the current boot nodes), `all-nodes.txt` (all, even inactive nodes), and `new-nodes.txt` (new nodes that weren't part of the slice before). There is a lesser-used `-n` argument that tells `download-nodes.py` not to write out any of the above files.

IV. USING VXARGS

`vxargs` is a tool for running commands in parallel. It works by launching a certain number of threads, which is specified with the `-P` argument. Each thread runs the given command, substituting in the next value. These values come from a newline-delimited file (specified with the `-a` argument). Once a thread's job is finished, it pulls another item out of the list and runs it. The main use for `vxargs` is in executing commands on many remote servers, that would otherwise require serial execution.

There are many command line settings, most of which you end up using in a typical use. For example, to execute the `ls` command on every host in `current-nodes.txt`, use the command:

```
./vxargs.py -t 30 -p -P 15 -y -o results -a current-nodes.txt ssh username@{} ls
```

The command line argument descriptions can be seen by adding a `--help` parameter. I have added a few command line arguments to the out-of-the-box `vxargs`. The `{}` variable is substituted with the current line from the text file, much like the UNIX `find -exec` command operates.

`vxargs` saves results to a folder if you specify the `-o` option. I highly recommend this option because it gives the standard out, standard error for each process as well as the return code from the command. These are highly valuable since they let you determine which nodes failed, and perhaps why.

One word of warning with `vxargs` is that you can easily tank a machine with it. Specifying it to run too many concurrent threads or having it run bandwidth intensive programs, such as `rsync`, can cause threads to starve and timeout. Choose moderate values for `vxargs`, especially if you will be sharing the machine with other people.

Always use the `-t` timeout option. I have found that some nodes will respond with the SYN-ACK but will not send any following data. This causes that `vxargs` thread to hang, at least until the route fails or is reset, which can be a very long time. On the flip side, using a timeout value that is too small can

stop nodes from executing the command, because that node may be very slow.

The Curses output can be fun to watch (don't provide the `-p` flag) but it may not be compatible with some ttys. It doesn't add much value, however, since you can pull the complete information from the output directory.

V. DEVELOPING AND DISTRIBUTING YOUR SOLUTION

A. Development

I would recommend developing and testing your solution on a controlled network, such as the Schizo network. Once things get on Planet-lab it becomes more difficult to troubleshoot, if not impossible sometimes.

When developing your solution, consider how you will want to start planet-lab nodes running your solution. Will it be an always-running program or will it be a program that needs to start at a specified time? If the former, you can probably just use `vxargs` to run your program (perhaps with the `nohup` option) directly from the command line. If the latter, which was my case, and I needed to launch them with different workloads, then I recommend the following solution.

I decided on making a simple server that would run on each Planet-lab host and listen for a "start" message to come. I could then specify from one node the order in which to launch the Planet-lab nodes. With this approach I could get around the delays and terminal constraints associated with just running an ssh command at a given time. often the SSH-port just seems to be non-responsive on some Planet-lab hosts, and so using a custom-process seems to be more reliable.

I created a POSIX service that would listen for these start messages and run my program when it got one. This service can be started/stopped using the `sbinservice` command. To get it to work right, the service needs to first fork, and continue execution only if it was the forked process. Here is a basic template for a daemon:

```
pid = os.fork()
if pid < 0: sys.exit(1) # error forming, terminate
if pid > 0: sys.exit(0) # parent, terminate

# the rest of your server code here
```

Now, I wanted my server to always run, and to start running again if the machine was rebooted. To do this, you need to add a service file with the exact same name as your daemon to the `/etc/init.d`

folder. I have provided a sample service, called `btserver.service`. You also need to place your daemon in a place that is in the system path. I usually just place it in `/bin` because it had problems finding it in other places.

The format of this service file is beyond the scope of this paper and, apparently, beyond me because mine seems to have problems detecting other instances of the service running. If someone finds a fix to this issue, I would appreciate seeing the fix.

Now to configure your service use the following commands as root (assuming the service is called `btservice`):

```
root@localhost /etc/init.d # /sbin/chkconfig --add btserver
root@localhost /etc/init.d # /sbin/chkconfig btserver on
root@localhost /etc/init.d # /sbin/service btserver start
```

The first two lines add the service and enable it. The last line turns the server on.

B. Deploying

Now, to deploy your program to a fresh Planet-lab node, you will probably want to `tar` and/or `zip` it up including the service and all your code. This makes it convenient to transfer via `vxargs` in one pass.

In the files I provided, `push_distro.sh` does my day-to-day distribution. Be sure to fill in the correct values above and inside the script to get yours working right. The first step copies the latest distribution (I have a symlink to the latest distribution tarball) to each Planet-lab host. It runs `build_successlists.py` to build a list of the nodes that failed and tries the transfer again for these nodes. Step 3 does the install, which untars the file, moves the daemon to the `/bin` folder, adds the service, and runs the service. Be sure to understand completely what each line of this file is doing. I use a few helper python scripts to generate success/fail lists and to merge the good lists into a master list.

VI. USING THE WORKLOAD GENERATOR

Once the solution is deployed, we then can start running experiments. I wrote the `workload-generator` to launch my Planet-lab hosts with different distributions and inter-arrival times. The `workload-generator` I have provided is specific to my BitTorrent experiments, though the same basic concepts can be applied to any multi-host application which we have been talking about.

The `-epml` switches are specific to BitTorrent, and are parameters for running the experiment. You will want to change these to fit your specific needs. `-d` specifies the inter-arrival distribution to use. You will

notice I have several subclasses of the `WorkloadGenerator` which handle the different arrival times. Each has a `do_simulation` function which which calls `start_host` for each node followed by a determined sleep time. The parameters to these workloads are passed in as command line arguments. For example, to run my experiment with a constant workload delaying 1.5 seconds in between each host start I would use:

```
./workload-generator.py -d constant 1.5 current-nodes.txt
```

If you just want to get a feel for how it works, use the `-y` switch (dryrun) to see what order hosts would be launched in.

VII. DOWNLOADING RESULTS

Once you have run one or more experiments, you will want to pull the result files off the nodes and into some form on your computer. `vxargs` is the best answer for this. One question to ask is if you will have more than one log from a given host. If so, you'll probably want a separate folder for each host. To do this, use a syntax similar to the following:

```
./vxargs.py -p -P 8 -y -o dresults
-a $NODELIST bash -c
"ssh $SLICENAME@{} sudo gzip /*-bt-experiment.log;
mkdir -p $RESULTSDIR/{};
rsync -tz -e ssh $SLICENAME@{}:/*-bt-experiment.log.gz $RESULTSDIR/{}/"
```

This example makes some assumptions about the location and destination of the logs. It first gzips the target files, makes a local directory with the current planetlab hostname. It then `rsyncs` the logs over to the local folder. Thus, all log files end up in their own directory per host.

See `download_logs.sh` for more details.