CCU-LANL-Batsim User Guide

Craig Walker

July 19, 2021

Contents

1 Intro

Batsim is built on Simgrid. It is able to simulate at a good speed and has the advantage that one can use many different languages to run the scheduler, as the scheduler is decoupled from the simulator. Also, it is used inside what is called a nix-shell. Nix-shells provide reproducibility. I plan on explaining things further:

1. Simgrid
2. Batsim vs The Scheduler
3. Nix

1.1 Simgrid

Simgrid is the backbone to Batsim. Batsim uses an API called S4U with Simgrid in order to interface with it. You've mainly got Actors, which can be thought of loosely as processes, and these communicate via Mailboxes. There are many more parts to the S4U API. Simgrid S4U documentation can be read from `https://simgrid.org/doc/latest/app_s4u.html`. General Simgrid documentation can be read from `https://simgrid.org/doc/latest/index.html`.

Basically, just know that Batsim's simulation is mostly controlled by Simgrid in the background. You use the "batsim" command (actually you will use robin, but that comes later), but you can send options directly to simgrid using

```
--sg-cfg <opt_name:opt_value>
or
--sg-log <log_option>
```

Simgrid uses a concept called profiles to define a computer system. This is an xml file. You can edit things like cluster size, latency between operations and between nodes in a cluster, energy usage, multiple clusters, and many more things. So if you want to simulate something grizzly sized at 1490 nodes you would need to make a profile with a 1490 node cluster. This is fairly trivial to do. Just edit the "radical" property of a "cluster" element.

Usually you will interface mainly with Batsim and the scheduler, and that brings us to our next topic.

1.2 Batsim vs The Scheduler

Batsim uses Simgrid under the hood, but fortunately we interface with Batsim, mostly. This is because Simgrid can get fairly complex. The code for Batsim, on the other hand, is much more straight forward. It also has the great property that it controls the time of the simulation and things running, while the scheduler performs the submitting and taking care of events, etc... . Batsim is written entirely in c++, and while that language has its benefits, some people are much more comfortable in python, java, go, c#, or some other language. The scheduler aspect of Batsim is such that messages are written from Batsim to the scheduler and from the scheduler to Batsim. If this scheduler can handle this message passing, then it is fairgame for being used as a scheduler. This scheduler can be written in most any language one desires.

There are two languages used for Batsim schedulers out of the box: python and c++. Python scheduler is called pybatsim, and c++ scheduler is called Batsched. Many tests were done by the CCU-LANL team using both, however pybatsim was unpredictable in its current state. At first, pybatsim would not run with more than 13,000 - 20,000 jobs. There was an error with it keeping a connection to Batsim.

Soon, our team's requests to the Batsim team for info seemed to stir some updates to all the code and their documentation. This proved invaluable. Their updated pybatsim ran better and faster, but on some people's machines it would still not run large workloads, and on my machine it was running what would take Batsched about 2 hours to run, in 6.5 minutes. That could sound like a feature, but it made pybatsim seem unreliable. I used Batsched and that's what I've been using since.

For anyone that comes after that wants to experiment with pybatsim: I used a grizzly 138,000+ workload with Batsched using easy_bf for a scheduler algorithm. It finished in about 1hr 50min on an i5-8600k machine. I tried it on a pybatsim easyBackfill algorithm and it took about 6.5 minutes with very different results. Taking into account that others were not able to run pybatsim with our workload led me to ditch pybatsim in favor of Batsched. I also have more respect for c++. More research could go into this potentially.

Batsim is called first, then the scheduler, in this case Batsched. We use a program provided from Batsim developers called Robin. Robin is like a coupler that runs Batsim then runs Batsched with the appropriate commands, but you only need to run one command to achieve this. Basically you run "robin generate [opts]" to make a .yaml file, then you run "robin <file>.yaml"

Typically we run these commands in a nix-shell, and this brings us to our next topic.

1.3 Nix

UPDATE: 07/04/2021
We have moved on from using Nix. Nix still gives us the most stable run when using the full 140k grizzly jobs and batsched's "easy_bf". However, with the headache of not being able to "pin" the versions correctly and Nix wanting to update itself, as well as not always being able to use a docker all the time (to use Nix with user permissions), we decided to ditch Nix. Nix will still be documented here under How To Use Batsim With Nix. There is a lot of good material in there, so feel free to read through it, even if not using Nix. If you want to see how to use Batsim without Nix, see our section How To Use Batsim Without Nix

Pre 07/04/2021
Nix is a full blown OS...called Nix-OS. However, it is also used as a sort of container, or maybe an "environment" is a better term. You can create an environment with certain versions of software, and so, a program will run the same on any linux system. A Batsim Nix environment can be made system-wide such that all commands that are available to Nix will be available at the regular command-line. This is not advised, however. It is much better to make a nix-shell.

A Nix-shell is made using a functional language that follows nix syntax. Basically you tell nix what programs will be included in your shell and what versions, and it either downloads and/or builds the programs necessary to use those programs, or once it has already done that, dumps you into an environment where all those programs are available.

Sounds good, however functional languages aren't everybody's favorite, and figuring out how to incorporate one's own code into the nix-shell took some time. But, the only way the CCU-LANL team could get Batsim running reliably (for large workloads) was using Nix. There are scripts I developed to make using nix fairly easy to use, however.

Much more can be said about Nix, so I've included a subsection Nix Internals in the section Internals. But, next we should dive into actually using Batsim and Batsched.

2 How To Use Batsim Without Nix

This is our updated Batsim info, without Nix. The first thing you are going to need to do is install batsim/batsched. This is covered in the section:

- Installation

I will be covering how to use Batsim in 4 different ways:

1. Normal Single Use
2. Running In Some Kind Of Loop
3. Parallelization Of Batsim, AC-Cluster,Sweeps
4. Batsim Docker

2.1 Installation

2.1.1 Get The Code

The first step to getting things installed is getting the code. You can use our deploy token to get our repo. Run the following:

```
git clone https://gitlab+deploy-token-7:nYnSAc2hWtuVWi1jDumx@gitlab.newmexicoconsortium.org/
    lanl-ccu/simulator.git
```

What you want, in order to install batsim, is the monte_carlo folder.

```
cd ./simulator/monte_carlo
```

2.1.2 Prepare

You will want to read simulator/monte_carlo/README_FILES.txt. This tells you what each file in this folder is used for. Pay special attention to deploy.sh and generate_config.py

You will want to edit deploy.sh quite a bit, for your uses. There is not a lot of documentation to this deploy script, but mainly, it downloads dependencies, compiles them, and installs them. Since everything will not work right away, you may find deploy.sh more useful to just copy and paste certain lines. I am leaving this fully up to the reader. I installed Batsim/Batsched/robin on our "ac-cluster" dubbed "acordar". We had 13 nodes from this cluster and each node had 30 cores (jobs), effectively 390 jobs at once.

Our versions were as follows:

- gcc: 10.2.0
- Kernel: 3.10.0-1160.6.1.el7.x86_64
- python: 3.6.8

The idea is that you can edit deploy.sh to your system's needs.

2.1.3 Install

From there it is just a matter of running deploy.sh or running the lines manually. There really isn't much else to say about it. In deploy.sh I used the `/home/$USER` directory for running most things and `/home/$USER/Downloads` for the location of all downloading of source code. I used `/home/$USER/Install` for the installation of most things.

These commands may help you see what was run on our ac-cluster

Change Directory to user directory
`cd /home/$USER`
Copy deploy.sh into user directory
`cp ./monte_carlo/deploy.sh /home/$USER/deploy.sh`

Copy patches of batsim and batsched to Downloads
`cp ./monte_carlo/patch_batsim.patch ~/Downloads/patch_batsim.patch`
`cp ./monte_carlo/patch_batsched.patch ~/Downloads/patch_batsched.patch`
Run deploy.sh . Again, this will need to be edited first
`cd /home/$USER && ./deploy.sh`

2.2 Normal Single Use

2.2.1 Create workload

There are 3 main ways to create a workload.

1. Workload Synthetic
2. Grizzly Synthetic
3. Pure Grizzly

2.2.1.1 Workload Synthetic

A purely synthetic workload can be generated using

```
python3 ./python-scripts/generate_synthetic_workload.py --help
```

2.2.1.2 Grizzly Synthetic

A synthetic workload based on grizzly can be achieved using:

```
python3 ./python-scripts/generate_synthetic_wokload.py
```

and using the "–number-of-resources" and "–duration-time" options with the csv option:

```
'--number-of-resources "simulator/monte_carlo/wl1.csv:0:csv"'
'--duration-time "simulator/monte_carlo/wl1.csv:1:h:csv"'
```

This says that wl1.csv with the given path is used to determine the number of resources and the duration times. Position 0 is used for number of resources and position 1 is used for duration times. The duration time represents hours (h).

- **wl1.csv**:
  - All jobs are one node wide and 24 hours long. This represents the analog of wl6, where all jobs span the entire cluster.
- **wl2.csv**:
  - Jobs are based on 11 months of data from a capacity cluster with 1490 nodes. This workload contained very few large jobs, with roughly 48% of the jobs requiring one node, and less than one hour of wall time, with the remaining jobs distributed in duration and width.
- **wl3.csv**:
  - Job widths are uniformly distributed from one to size of cluster, with durations also uniformly distributed from one to 24 hours.
- **wl4.csv**:
  - Job widths are uniformly distributed from 32 to size of cluster, with durations from 6 to 24 hours.
- **wl5.csv**:

  - Job widths are divided into two bins: 512 or the entire cluster, 1490, with durations ranging from one to 24 hours, according to the distributions of durations present in the original wl2 workload.

- **wl6.csv**:
  - Jobs are all 1490 nodes wide with durations of 24 hours. This represents a "worst-case" scenario from a reliability point of view, as any node failure will result in a job failure that spans the entire cluster.

It is reasonable to think of workloads 1-6 increasing in "size": small, regular, medium, medium-high, large, full.

2.2.1.3 Pure Grizzly

A pure grizzly workload can be generated using

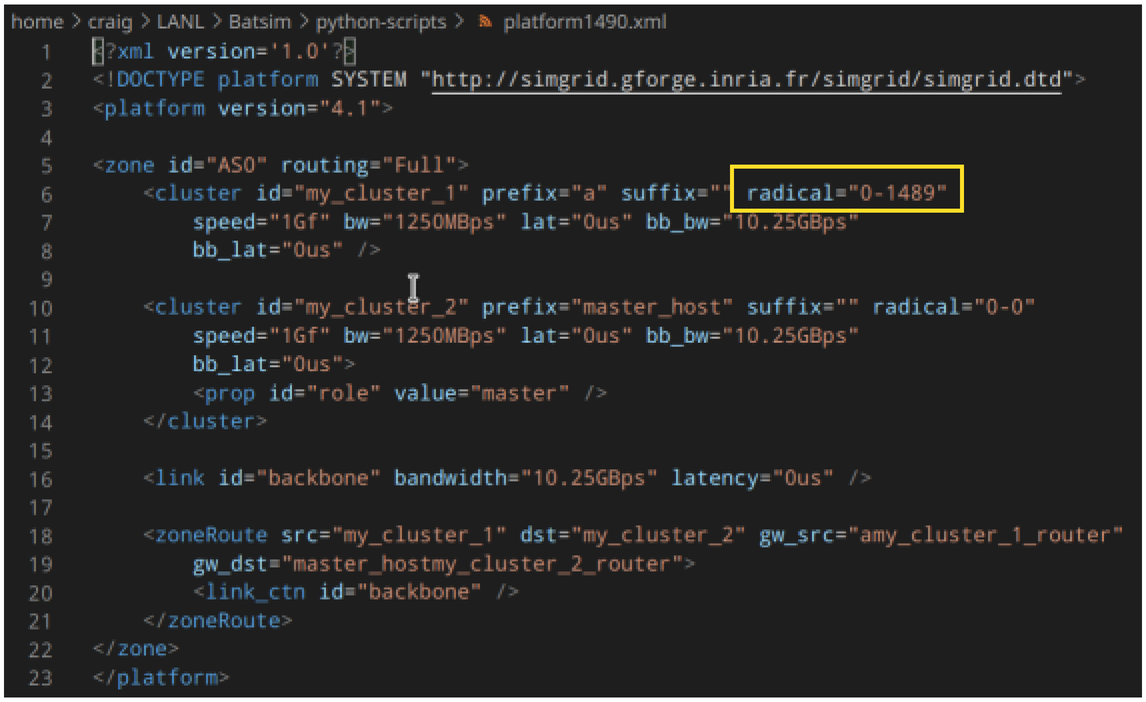
```
python3 ./python-scripts/generate_grizzly_workload.py --help
```

2.2.1.4 What To Do With Generated Workload

I typically put the workloads in the base folder I am dealing with. This is the folder where all the scripts are, and I put the workload in the "workloads" folder in this base folder. You will use this location in the robin command to tell batsim where the workload resides.

2.2.2 Create Platform

In `simulator/monte_carlo/platforms` there will be a platform file `platform_1490.xml` There are many options to change in a profile, but basically all we are changing are the number of nodes in a cluster. You change this in the radical property of a cluster element.


home > craig > LANL > Batsim > python-scripts > platform1490.xml

```
<?xml version='1.0'?>
<!DOCTYPE platform SYSTEM "http://simgrid.gforge.inria.fr/simgrid/simgrid.dtd">
<platform version="4.1">

<zone id="AS0" routing="Full">
    <cluster id="my_cluster_1" prefix="a" suffix="" radical="0-1489"
        speed="1Gf" bw="1250MBps" lat="0us" bb_bw="10.25GBps"
        bb_lat="0us" />

    <cluster id="my_cluster_2" prefix="master_host" suffix="" radical="0-0"
        speed="1Gf" bw="1250MBps" lat="0us" bb_bw="10.25GBps"
        bb_lat="0us">
        <prop id="role" value="master" />
    </cluster>

    <link id="backbone" bandwidth="10.25GBps" latency="0us" />

    <zoneRoute src="my_cluster_1" dst="my_cluster_2" gw_src="amy_cluster_1_router"
        gw_dst="master_hostmy_cluster_2_router">
        <link_ctn id="backbone" />
    </zoneRoute>
</zone>
</platform>
```

One could also use:

```
python3 change_platform.py -i ./platforms/platform_1490.xml -o ./platforms/platform_1500.xml
    --nodes 1500
```

This makes a new platform with 1500 nodes.

2.2.3 Run simulation

The main things you need to run a simulation are the workload and platform, as well as a properly working batsim/batsched/robin. You also need to know the options you want to send Batsim. With this information you generate a .yaml file with robin that saves all this information. You then run this .yaml file.

In the stage we are in now, an events file is not needed, however I refer you to “Making An Event.pdf” for more info on events. Events are basically used for when faults happen in our simulation or other events. They include the type of event and when, in simulation time, they occur. Events files have basically been replaced with MTBF and SMTBF options to Batsim that set how often failures happen in our simulation.

2.2.3.1 Robin

When you run a simulation, you are actually running two programs: batsim and your scheduler. In our case we are running batsim and batsched. To easily run these two programs from a single command line, robin comes in very handy. It will start both once you generate a .yaml file and run the file. Another benefit to robin is that in making a .yaml file you can run the same simulation over again without typing out the options again.

2.2.3.1.1 generate a .yaml file

To make a .yaml file to run use robin:

```
#issue the robin command
robin generate experiment.yaml --output-dir=/home/$USER/experiments/TestSingle/expe-out \
--batcmd="batsim -p {platformPath} -w {workloadPath} -e {outputPath}/expe-out/out {
    batsim_options}"\
--schedcmd="batsched -v fcfs_fast2 --verbosity quiet"\
--failure-timeout=31536000\
--ready-timeout=31536000\
--simulation-timeout=31536000\
--success-timeout=31536000
```

You will need to replace `'{platformPath}'` with the path to your platform , `'{workloadPath}'` with the path to your workload, `'{outputPath}'` where you want your output, and `'{batsim_options}'` will need to be replaced with your needed options. If you want to know what options you can use then run:

```
batsim --help
```

2.2.3.2 Run a .yaml file

Now you can run your .yaml file:

```
robin experiment.yaml
```

You can edit a .yaml file, but just be cautious, it is very sensitive to white space. Robin definitely doesn’t want any newline characters in the batsim command. So if your simulation is not running, try the tried and true method of generating a new yaml file first, as opposed to editing your yaml file.

2.2.4 While the Simulation Runs

There is no information that is passed to STDOUT during a simulation. The only way to get the progress of your simulation is looking in the log files. The main log file you want is ./expe-out/log/batsim.log . so run

```
cd ./expe-out/log/
tail -n 30 batsim.log | grep ACTUALLY
```

This will tell you how many jobs have finished.

You can also look at sched.err.log to see what the scheduler’s log says.

You may be wondering why you “grep ACTUALLY”. What you are really searching for is “ACTUALLY COMPLETED”, as opposed to “JOBS COMPLETED”. The difference? If you are using failures, such as with MTBF or SMTBF options, failed jobs will complete, and add to the total “JOBS COMPLETED”. Suppose you have a workload of 1 job. It restarts 10 times because there was a failure 10 times. It will look like 10 jobs have completed if you search for “JOBS COMPLETED”. “ACTUALLY COMPLETED” is my addition to the code that tells you how many original jobs have completed. So with 1 job, when it completes “ACTUALLY COMPLETED” will list 1 job, not 10. Unfortunately I had to use the logging category of “Error” so that it would show up even if we set the logging to “quiet”.

2.2.5 Process Results

Once the simulation ends you should have a `./expe-out/out_jobs.csv` file. But this file is not the best file when you have re-submitted jobs. You can post-process the jobs using `simulator/monte_carlo/post-processing.py` One can use

```
python3 post-processing.py --help
```

This python script can output makespan, average application efficiency, utilization, and a number of more informative fields than `out_jobs.csv` .

2.3 Running In Some Kind Of Loop

In `simulator/monte_carlo/real_start.py` you will find the code for, essentially, a single run through a loop. It uses a json config file to eventually produce the .yaml file. It expects the config file to be called `./input/config.ini`.
Here is an example config file:

```
{
   "nodes": 1490,
   "SMTBF": 1159731.543624161,
   "checkpoint-interval": "optimal",
   "performance-factor": 1.0,
   "checkpointing-on": true,
   "seed-failures": true,
   "platformFile": "/home/cwalker/basefiles/platforms/platform_1490.xml",
   "synthetic-workload": {
      "workloadFile": "/home/cwalker/basefiles/workloads/wsl1.csv_CLN_0_CLN_csvres_wsl1.
          csv_CLN_1_CLN_h_CLN_csvdur_False_scl_1490_sclTW_0_CLN_fixeds_-1lim_3%D_2%
          R_medium_1_2_4_8_16_32x_e_synth.json"
   }
}
```

While this script is usable by itself and with a config file in the appropriate places, I mention it mainly so you can read through it and adapt it to your purposes, and so that you can see what options are in the config file.

Keep in mind, later we will talk about config files that do sweeps that we feed into a script called `generate_config.py`. This config file is a bit different than what you see here. `generate_config.py` is expecting a .config file, while `real_start.py` is expecting a config.ini file. The `generate_config.py` script will generate the workloads,platforms,folder structure and all the individual config.ini files you will need. Since I don't provide the best documentation for a single loop, it is advisable to look into sweeps and to look at the resultant config.ini files if you get stuck.

2.4 Parallelization Of Batsim, AC-Cluster, Sweeps

The main problem with running Batsim on your own computer is the amount of memory it takes. One could also make an argument that it may render your computer useless while running, as it may take too much cpu if one has a slower computer. Once you do Monte Carlo averages with 1500 runs, you are going to need a cluster to do meaningful experiments. We've been using “proton.usrc”, “gamma.usrc” and finally the acordar cluster “ac-fe1”. If SLURM is setup, we can also take advantage of some easy parallelization with this approach.

2.4.1 Connect to remote

The first thing you want to do is connect to “ac-fe1” by first connecting to “comely” through ssh.

```
#connect to "comely"
ssh <user>@comely.newmexicoconsortium.org

#next connect to..say.."ac-fe1"
ssh ac-fe1
```

2.4.2 Run A Simulation

~~~
`file1=./configs/configFileName.config` <br/>
Make sure the folder name is different each time <br/>
`folder1=~/experiments/configFileName` <br/>
``` base=`pwd` ```<br/>
```
sbatch -p usrc-nd02 -N1 -n1 -c1 --output=./myBatch.log \
--export=folder1=$folder1,file1=$file1,basefiles=$base \
 ./myBatch
~~~

2.5 Batsim Docker

3 How To Use Batsim With Nix

First you need to install Nix and download the appropriate files from our gitlab repo. This is covered in the section:

- Installation

I will be covering how to use Batsim in 3 different ways:

1. Normal Single Use
2. Running In Some Kind Of Loop
3. Proton, Docker, and Sweeps

3.1 Installation

3.1.1 Install Nix

First one must install Nix. Nix programs install to

```
/nix/store
```

Notice that you must have permissions to the root directory. If you don't, the easiest way to use Batsim is through a docker. The section Proton, Docker, and Sweeps will show you how to achieve this.

If you have permissions then installation of Nix is as simple as running:

```
curl -L https://nixos.org/nix/install | sh
```

Note: once you install Nix you MUST execute

```
source ~/.nix-profile/etc/profile.d/nix.sh
```

for any nix commands to work. For convenience you can add the following to your /.bashrc file:

```
alias source_nix="source ~/.nix-profile/etc/profile.d/nix.sh"
```

and then simply run

```
source_nix
```

Note about .bashrc

Note: .bashrc does not get ran until a new session of bash is run. If you just added this alias then you must start a new bash session or source your .bashrc file :

```
#either do this
bash
source_nix

#or this. replace "source" with a dot "." if using /bin/sh on ubuntu
source ~/.bashrc
source_nix
```

3.2 normal single use

So you have Nix installed. Now you need the CCU-LANL edited Batsim and Batsched, and while you're at it, helper files.

3.2.1 Getting Batsim Code

The first thing you want to do is create a folder to contain all things Batsim:

```
mkdir /home/USER/Batsim
```

Where USER is your username. next:

```
cd /home/USER/Batsim/
```

Now you are in the correct directory to clone some git repos. As far as git repos go there are 4 options:

1. Provide a username and password each time you upload or download to/from the repo
2. Provide an ssh key to the repo and use this key when uploading and downloading so you don't need to supply a username and password
3. Use a token, assigned from the repo, and tied to the cloned repo so you don't need to supply a username and password
4. Use a deploy token so you don't need to use a username and password. This is a read-only token, so no uploads to forks. You will need to copy the git command exactly.(recommended)

I have been using the 3rd option. The 2nd option is the most complex, and though many tutorials exist, is not covered here.
The 1st option is simple, just provide your username and password when prompted for them.
The simplest of all is the 4th option and is the recommended way if you are not contributing to the code.

3.2.1.1 Option 1: Enter username and password

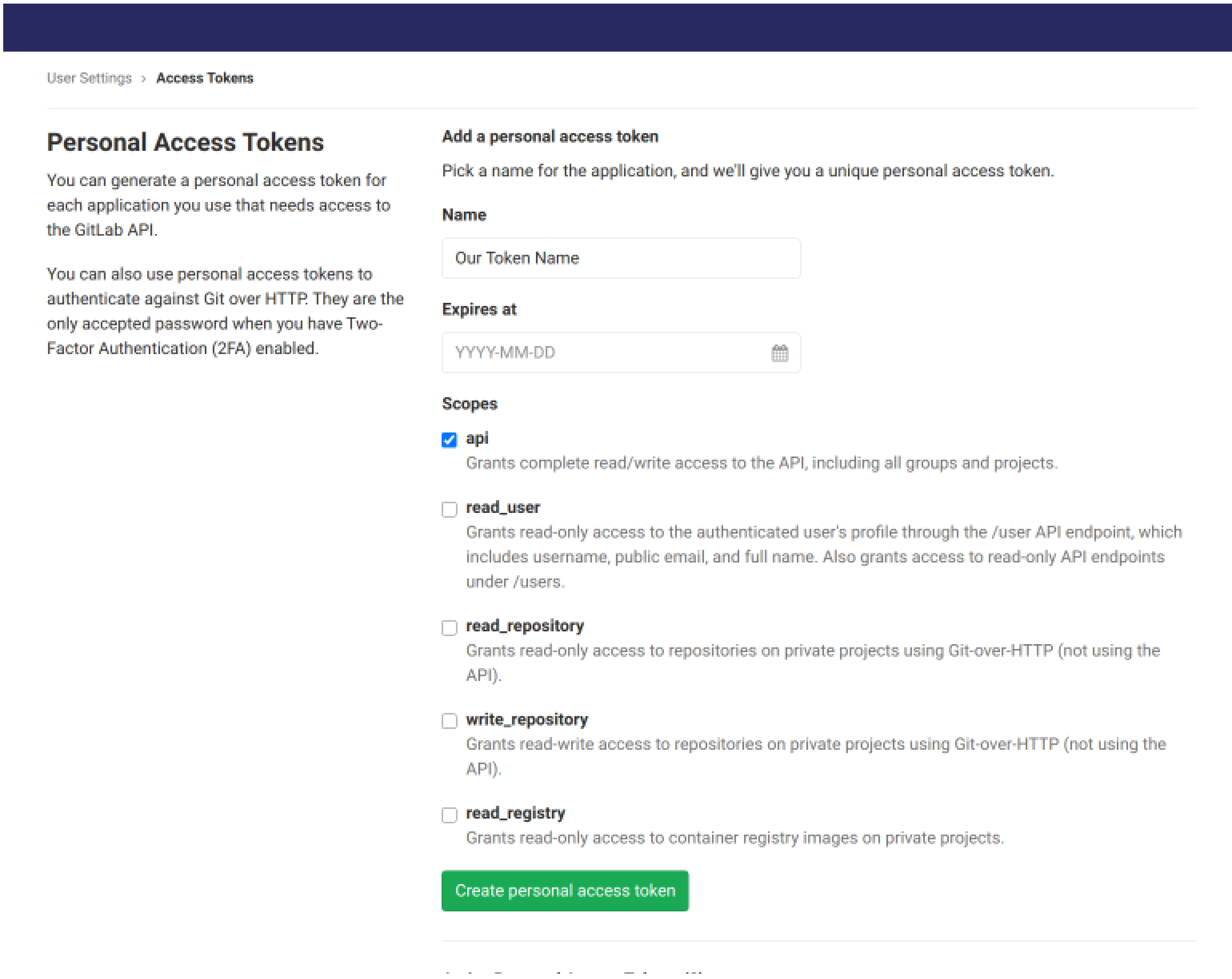
If you don't want the hassle of creating a token just use:

```
git clone https://gitlab.newmexicoconsortium.org/lanl-ccu/batsim.git
git clone https://gitlab.newmexicoconsortium.org/lanl-ccu/batsched.git
git clone https://gitlab.newmexicoconsortium.org/lanl-ccu/helper-scripts.git
git clone https://gitlab.newmexicoconsortium.org/lanl-ccu/python-scripts.git
```

you may now continue.

3.2.1.2 Option 3: Make a token

The third option involves going to the site: `https://gitlab.newmexicoconsortium.org/profile/personal_access_tokens` and selecting the following:


User Settings > **Access Tokens**

Personal Access Tokens

You can generate a personal access token for each application you use that needs access to the GitLab API.

You can also use personal access tokens to authenticate against Git over HTTP. They are the only accepted password when you have Two-Factor Authentication (2FA) enabled.

Add a personal access token

Pick a name for the application, and we'll give you a unique personal access token.

Name

Our Token Name

Expires at

YYYY-MM-DD

Scopes

[✓] **api**
Grants complete read/write access to the API, including all groups and projects.

[ ] **read_user**
Grants read-only access to the authenticated user's profile through the /user API endpoint, which includes username, public email, and full name. Also grants access to read-only API endpoints under /users.

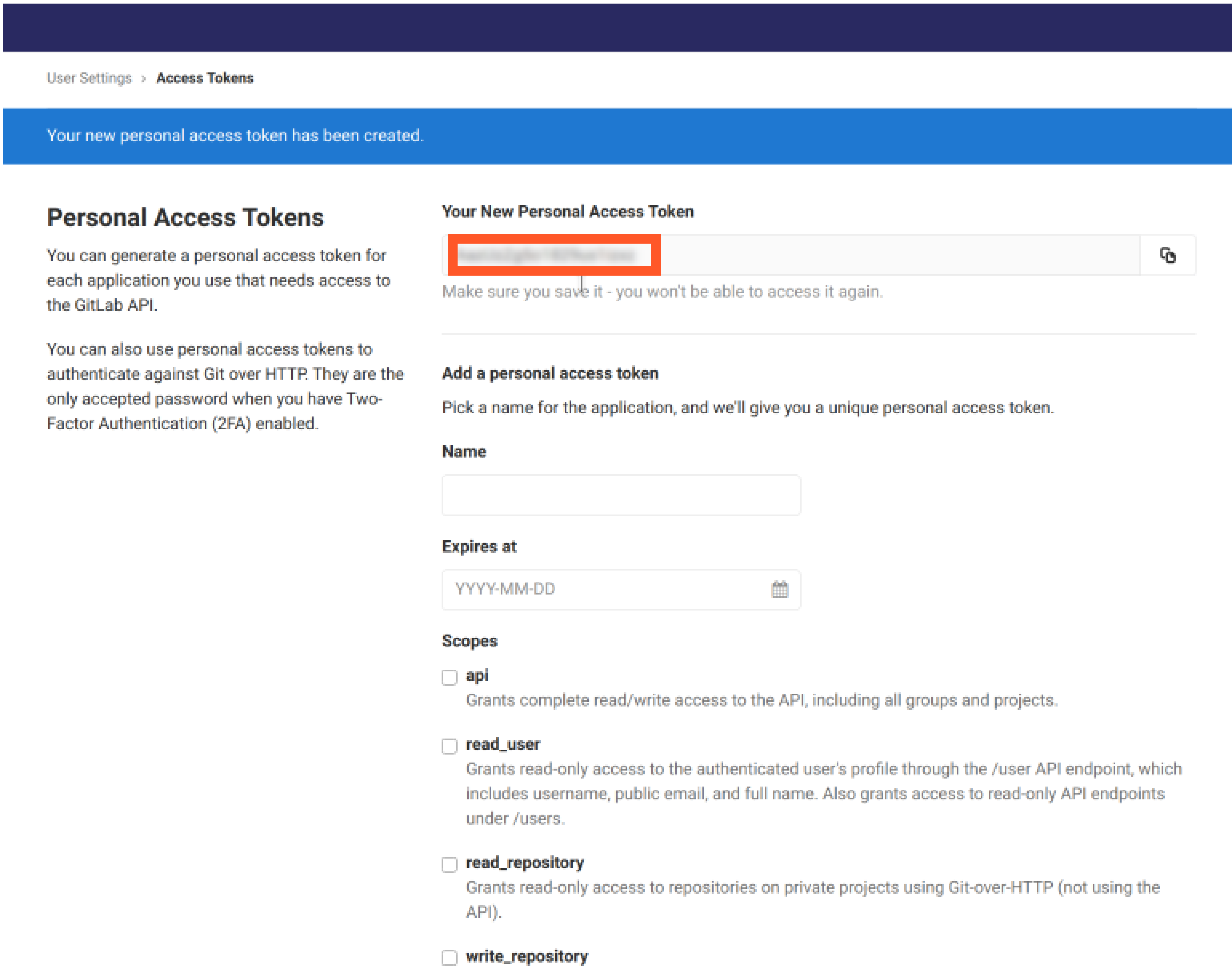
[ ] **read_repository**
Grants read-only access to repositories on private projects using Git-over-HTTP (not using the API).

[ ] **write_repository**
Grants read-write access to repositories on private projects using Git-over-HTTP (not using the API).

[ ] **read_registry**
Grants read-only access to container registry images on private projects.

Create personal access token

Once you select "Create Personal Access Token" you will get:


User Settings › **Access Tokens**

Your new personal access token has been created.

Personal Access Tokens

You can generate a personal access token for each application you use that needs access to the GitLab API.

You can also use personal access tokens to authenticate against Git over HTTP. They are the only accepted password when you have Two-Factor Authentication (2FA) enabled.

Your New Personal Access Token

[redacted]

Make sure you save it - you won't be able to access it again.

Add a personal access token

Pick a name for the application, and we'll give you a unique personal access token.

Name

Expires at

YYYY-MM-DD

Scopes

- [ ] **api**
  Grants complete read/write access to the API, including all groups and projects.
- [ ] **read_user**
  Grants read-only access to the authenticated user's profile through the /user API endpoint, which includes username, public email, and full name. Also grants access to read-only API endpoints under /users.
- [ ] **read_repository**
  Grants read-only access to repositories on private projects using Git-over-HTTP (not using the API).
- [ ] **write_repository**

Copy this code that is returned in the red rectangular block. This is your access token, don't lose it.

Now you can clone using this token and you will never need to use your username or password with this cloned repo again, just replace <TOKEN>.

```
git clone https://oauth2:<TOKEN>@gitlab.newmexicoconsortium.org/lanl-ccu/batsim.git
git clone https://oauth2:<TOKEN>@gitlab.newmexicoconsortium.org/lanl-ccu/batsched.git
git clone https://oauth2:<TOKEN>@gitlab.newmexicoconsortium.org/lanl-ccu/helper-scripts.git
git clone https://oauth2:<TOKEN>@gitlab.newmexicoconsortium.org/lanl-ccu/python-scripts.git
```

you may now continue.

3.2.1.3 Option 4: Use deploy token

By far the easiest method to get the git repos, though you will not be able to contribute to the code. Run the following

```
git clone https://gitlab+deploy-token-1:GPX2qt_gDCbx5RfxnYws@gitlab.newmexicoconsortium.org/
    lanl-ccu/batsim.git
git clone https://gitlab+deploy-token-2:MsUCHwQNhrsQakyHy_GC@gitlab.newmexicoconsortium.org/
    lanl-ccu/batsched.git
git clone https://gitlab+deploy-token-3:gJsK8skQR84RtTxHXXMC@gitlab.newmexicoconsortium.org/
    lanl-ccu/python-scripts.git
git clone https://gitlab+deploy-token-4:_YwczPxeh75jYH9b1f9z@gitlab.newmexicoconsortium.org/
    lanl-ccu/helper-scripts.git
```

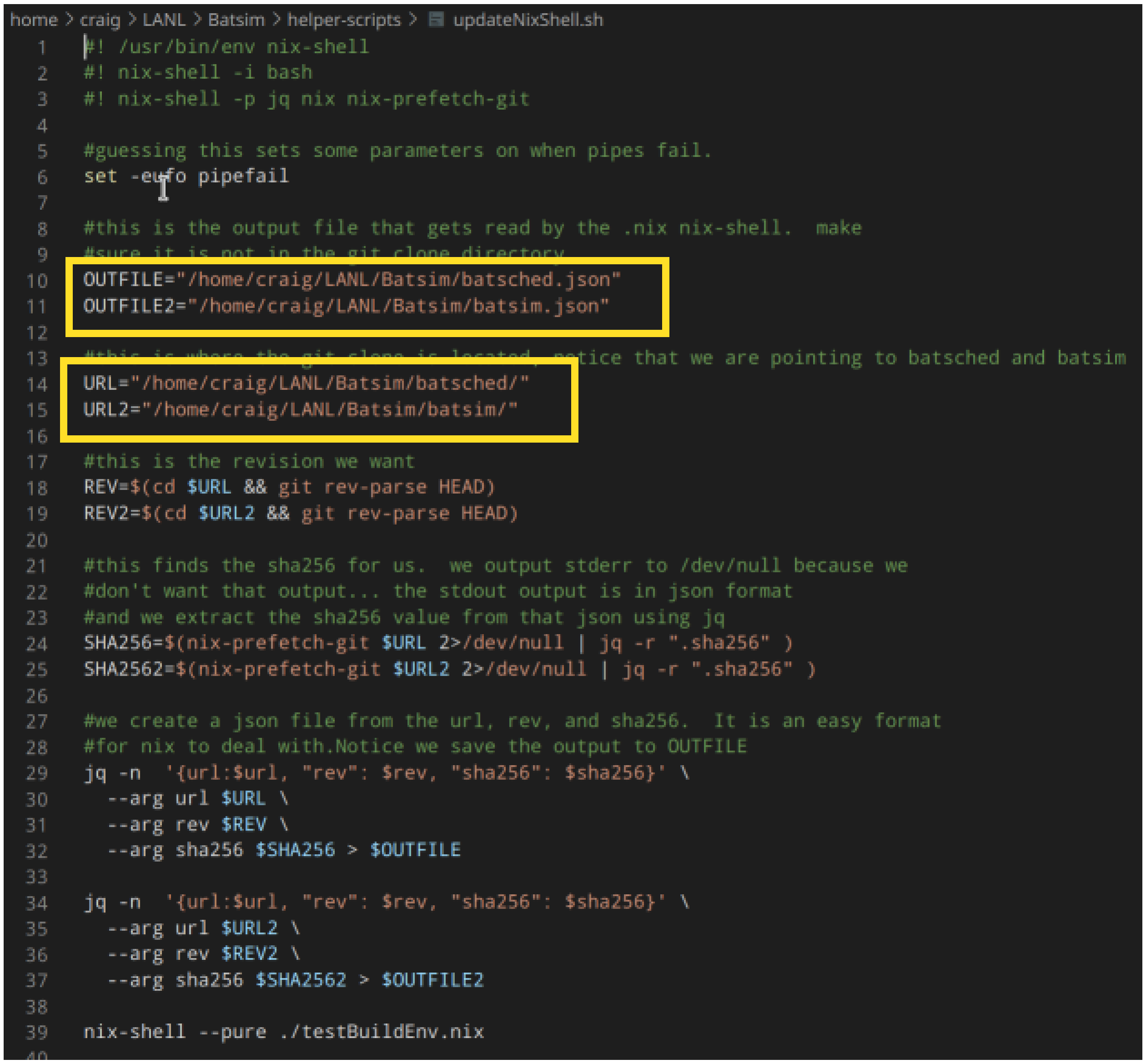
you may now continue.

3.2.2 Setup Folders and Files

Now copy the helper scripts to your base folder:

```
cp ./helper-scripts/*.* ./
```

Now edit updateNixShell.sh:



home > craig > LANL > Batsim > helper-scripts > updateNixShell.sh

```
#! /usr/bin/env nix-shell
#! nix-shell -i bash
#! nix-shell -p jq nix nix-prefetch-git

#guessing this sets some parameters on when pipes fail.
set -eufo pipefail

#this is the output file that gets read by the .nix nix-shell.  make
#sure it is not in the git clone directory
OUTFILE="/home/craig/LANL/Batsim/batsched.json"
OUTFILE2="/home/craig/LANL/Batsim/batsim.json"

#this is where the git clone is located. notice that we are pointing to batsched and batsim
URL="/home/craig/LANL/Batsim/batsched/"
URL2="/home/craig/LANL/Batsim/batsim/"

#this is the revision we want
REV=$(cd $URL && git rev-parse HEAD)
REV2=$(cd $URL2 && git rev-parse HEAD)

#this finds the sha256 for us.  we output stderr to /dev/null because we
#don't want that output... the stdout output is in json format
#and we extract the sha256 value from that json using jq
SHA256=$(nix-prefetch-git $URL 2>/dev/null | jq -r ".sha256" )
SHA2562=$(nix-prefetch-git $URL2 2>/dev/null | jq -r ".sha256" )

#we create a json file from the url, rev, and sha256.  It is an easy format
#for nix to deal with.Notice we save the output to OUTFILE
jq -n  '{url:$url, "rev": $rev, "sha256": $sha256}' \
  --arg url $URL \
  --arg rev $REV \
  --arg sha256 $SHA256 > $OUTFILE

jq -n  '{url:$url, "rev": $rev, "sha256": $sha256}' \
  --arg url $URL2 \
  --arg rev $REV2 \
  --arg sha256 $SHA2562 > $OUTFILE2

nix-shell --pure ./testBuildEnv.nix
```

You must change the OUTFILE/2 to the root folder and URL/2 to point to where batsim and batsched were cloned.

Now run the script that will initiate the nix-shell you will use to run the simulator. It may take a while to run as it will need to download all files it needs to build simgrid, batsim, and batsched. This includes such basic files as gcc, the compiler that will build all these programs. So this process takes a while right from the start, and then will take a while to build simgrid and compile the newest batsim and batsched.

So run

```
source_nix
#or
source ~/.nix-profile/etc/profile.d/nix.sh
#then
./updateNixShell.sh
```

You are now in a nix-shell based on the newest versions of batsim and batsched. You now need to create a workload which is explained next.

3.2.3 create workload

There are 3 main ways to create a workload.

1. Workload Synthetic
2. Grizzly Synthetic
3. Pure Grizzly

3.2.3.1 Workload Synthetic

A purely synthetic workload can be generated using

```
python3 ./python-scripts/generate_synthetic_workload.py --help
```

3.2.3.2 Grizzly Synthetic

A synthetic workload based on grizzly can be achieved using:

```
python3 ./python-scripts/generate_synthetic_wokload.py
```

and using the "–number-of-resources" and "–duration-time" options with the csv option:

```
'--number-of-resources "~/500000.csv:0:csv"'
'--duration-time "~/500000.csv:1:h:csv"'
```

This says that 500000.csv is used to determine the number of resources and the duration times. Position 0 is used for number of resources and position 1 is used for duration times. The duration time represents hours (h).

3.2.3.3 Pure Grizzly

A pure grizzly workload can be generated using

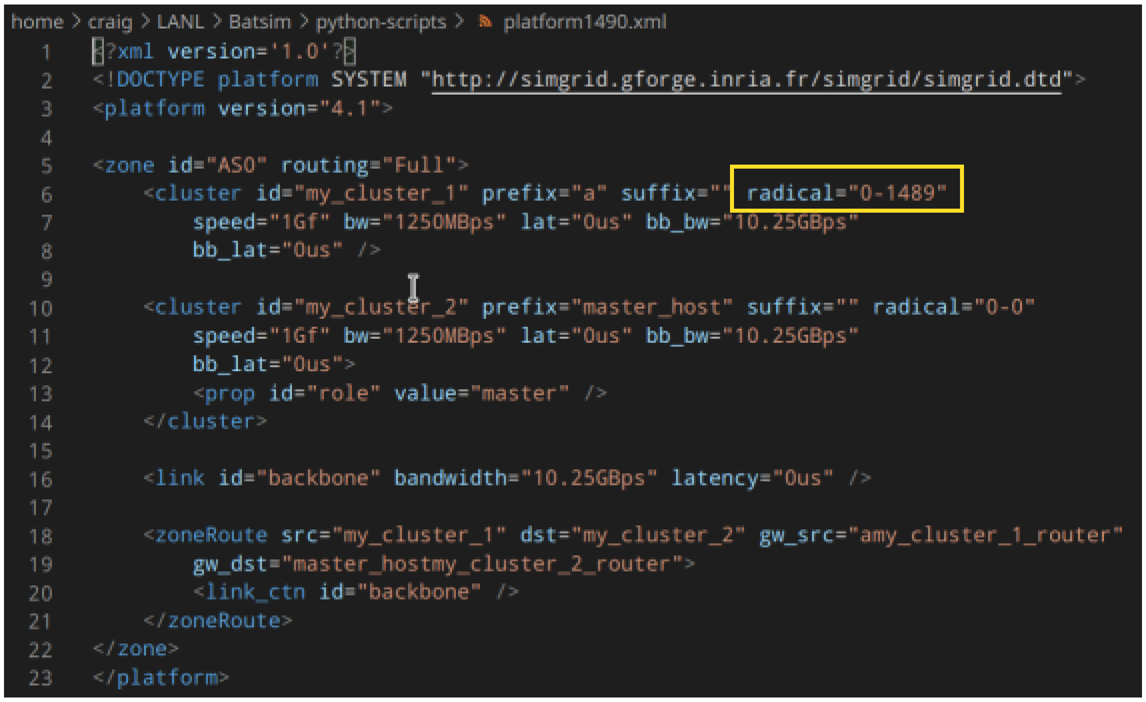
```
python3 ./python-scripts/generate_grizzly_workload.py --help
```

3.2.3.4 What To Do With Generated Workload

I typically put the workloads in the root Batsim folder ∼/LANL/Batsim/workloads
You will use this location in the robin command to tell batsim where the workload resides.

3.2.4 create platform

If you cloned "python-scripts" you have "platform1490.xml". There are many options to change in a profile, but basically all we are changing are the number of nodes in a cluster. You change this in the radical property of a cluster element.


home > craig > LANL > Batsim > python-scripts > platform1490.xml

```
<?xml version='1.0'?>
<!DOCTYPE platform SYSTEM "http://simgrid.gforge.inria.fr/simgrid/simgrid.dtd">
<platform version="4.1">

<zone id="AS0" routing="Full">
    <cluster id="my_cluster_1" prefix="a" suffix="" radical="0-1489"
        speed="1Gf" bw="1250MBps" lat="0us" bb_bw="10.25GBps"
        bb_lat="0us" />

    <cluster id="my_cluster_2" prefix="master_host" suffix="" radical="0-0"
        speed="1Gf" bw="1250MBps" lat="0us" bb_bw="10.25GBps"
        bb_lat="0us">
        <prop id="role" value="master" />
    </cluster>

    <link id="backbone" bandwidth="10.25GBps" latency="0us" />

    <zoneRoute src="my_cluster_1" dst="my_cluster_2" gw_src="amy_cluster_1_router"
        gw_dst="master_hostmy_cluster_2_router">
        <link_ctn id="backbone" />
    </zoneRoute>
</zone>
</platform>
```

3.2.5 Run simulation

The main things you need to run a simulation are the workload and platform, as well as a properly working nix-shell. You also need to know the options you want to send Batsim. With this information you generate a .yaml file that saves all this information. You then run this .yaml file.

In the stage we are in now, an events file is not needed, however I refer you to "Making An Event.pdf" for more info on events. Events basically are used for when faults happen in our simulation or other events. They include the type of event and when, in simulation time, they occur. Events files have basically been replaced with MTBF and SMTBF options to Batsim that set how often failures happen in our simulation.

3.2.5.1 Robin

When you run a simulation, you are actually running two programs: batsim and your scheduler. In our case we are running batsim and batsched. To easily run these two programs from a single command line, robin comes in very handy. It will start both once you generate a .yaml file and run the file. Another benefit to robin is that in making a .yaml file you can run the same simulation over again without typing out the options again.

3.2.5.1.1 generate a .yaml file

To make a .yaml file to run use robin:

```
#enter your nix-shell
./updateNixShell

#issue the robin command
robin generate experiment.yaml --output-dir=./expe-out \
--batcmd="batsim -p {platformPath} -w {workloadPath} -e {outputPath}/expe-out/out {
    batsim_options}"\
--schedcmd="batsched -v fcfs_fast2 --verbosity quiet"\
--failure-timeout=31536000\
--ready-timeout=31536000\
--simulation-timeout=31536000\
--success-timeout=31536000
```

You will need to replace platformPath with the path to your platform , workloadPath with the path to your workload, outputPath where you want your output, and batsim_options will need to be replaced with your needed options. If you want to know what options you can use run:

```
batsim --help
```

inside the updateNixShell.sh shell

3.2.5.2 Run a .yaml file

Now you can run your .yaml file:

```
robin experiment.yaml
```

You can edit a .yaml file, but just be cautious, it is very sensitive to white space. Robin definitely doesn't want any newline characters in the batsim command. So if your simulation is not running, try the tried and true method of generating a new yaml file first, as opposed to editing your yaml file.

3.2.6 While the Simulation Runs

There is no information that is passed to STDOUT during a simulation. The only way to get the progress of your simulation is looking in the log files. The main log file you want is ./expe-out/log/batsim.log . so run

```
cd ./expe-out/log/
tail -n 30 batsim.log | grep ACTUALLY
```

This will tell you how many jobs have been finished.

You can also look at sched.err.log to see what the scheduler's log says.

You may be wondering why you "grep ACTUALLY". What you are really searching for is "ACTUALLY COMPLETED", as opposed to "JOBS COMPLETED". The difference? If you are using failures, such as with MTBF or SMTBF options, failed jobs will complete, and add to the total "JOBS COMPLETED". Suppose you have a workload of 1 job. It restarts 10 times because there was a failure 10 times. It will look like 10 jobs have completed if you search for "JOBS COMPLETED". "ACTUALLY COMPLETED" is my addition to the code that tells you how many original jobs have completed. So with 1 job, when it completes "ACTUALLY COMPLETED" will list 1 job, not 10.

3.2.7 process results

Once the simulation ends you should have a ./expe-out/out_jobs.csv file. But this file is not the best file when you have resubmitted jobs. You can post-process the jobs using ./python-scripts/post-processing.py One can use

```
python3 post-processing.py --help
```

This python script can output makespan, average application efficiency, and a number of more informative out_jobs.csv files.

3.3 running in some kind of loop

In ./python-scripts/ you can find a file called loop_start.py . While this script is usable by itself and with a config file in the appropriate places, it is mainly there for reading through how to run a loop for your purposes. The highlights of this file show how to run a command inside of a nix shell without entering your terminal shell into a nix-shell. This allows your scripts to access everything it needs to run...things that may not be available in your nix-shell.

Keep in mind that there are many places in this file that use the shell command “source”. This can be changed from “source” to “.” and achieve the same thing. “source” is not universal, while “.” appears to be.

3.4 Proton: Remote parallelization

The main problem with running Batsim on your own computer is the amount of memory it takes. One could also make an argument that it may render your computer useless while running, as it may take too much cpu if one has a slower computer. Because of this, to do meaningful experiments it may be necessary to run them on a bigger machine. We’ve been using “proton.usrc” and “gamma.usrc”. If SLURM is setup, we can also take advantage of some easy parallelization with this approach. Since we often do not have the ability to install nix on a remote computer, we use a docker.

3.4.1 Connect to remote, and acquire the docker image

3.4.1.1 Connect

The first thing you want to do is connect to “proton” or “gamma” by first connecting to “comely” through ssh.

```
#connect to "comely"
ssh <user>@comely.newmexicoconsortium.org

#next connect to..say.."gamma"
ssh gamma.usrc
```

Next, we need to setup ”gamma” for our uses.

Now, the first thing you need to do is acquire the docker image. You can build the docker image yourself or use the image I have already built.

3.4.1.2 Build the image

So you’ve decided to do it the “docker way”. Keep in mind this build will take a while. First it needs to build the docker image, but it also, inside this image, needs to build the nix-shell.

You will need to run the following:

```
#clone the docker-batsim repo
git clone https://gitlab+deploy-token-5:qLo_fzBaKh4DGQfvTY4i@gitlab.newmexicoconsortium.org/
    lanl-ccu/docker-batsim.git

#change directory
cd ./docker-batsim

#build the docker image, notice the period at the end...include that
docker build -t batsim:w_performance_pinned .
```

You’re all set. Now you can check that the image is loaded

3.4.1.3 Use the image already built

I have made the image available to anyone in the ”ccu” group. Do the following:

```
#first copy the file to your home directory. It is 2.6GB so it may take a while to copy.
cp /usrc-nfs/cwalker/batsim_docker.tar.gz ~/batsim_docker.tar.gz

#now unzip the file, you will be left with a batsim_docker.tar file. Again, it may take a few
    minutes.
gunzip ~/batsim_docker.tar.gz
```

You now have the .tar file. You might as well check if you are part of the "docker" group, as this is critical to go further with docker.

```
#run this command to see if you are part of the docker group
groups

#on my system the following appears:
[cwalker@gamma cwalker] groups
cwalker usrc ccu docker
```

Good, I am part of the "docker" group, as it appears in the list. Notice I am also part of the "ccu" group.

Next, you will want to load the image into docker.

```
#load the image into docker. This can take a while. About 10 minutes.
docker load --input ~/batsim_docker.tar
```

3.4.1.4 Check that the image is loaded

Now check that the image is loaded.

```

#check that it is indeed loaded. The image will be called batsim:w_performance_pinned
docker images

#on my system the following appears:
[cwalker@gamma cwalker] docker images
REPOSITORY     TAG                         IMAGE ID       CREATED         SIZE
batsim         w_performance_pinned        0b5fbdf12ca6   40 hours ago    9.53GB
```

3.4.2 Setup Proton

3.4.2.1 Get some scripts

now you must get some scripts:

```
#make a directory called Batsim and cd into it
cd ~/
mkdir Batsim
cd Batsim

#clone our python-scripts, you will be asked for username and password
git clone https://gitlab.newmexicoconsortium.org/lanl-ccu/python-scripts.git

#you can optionally use this syntax to clone if you have a token:
git clone https://oauth2:<TOKEN>@gitlab.newmexicoconsortium.org/lanl-ccu/python-scripts.git

#or use the deploy token, though this is read-only
git clone https://gitlab+deploy-token-3:gJsK8skQR84RtTxHXXMC@gitlab.newmexicoconsortium.org/
    lanl-ccu/python-scripts.git
```

3.4.2.2 Create your input and output folders

Now create your input and output folders

```
mkdir ~/Batsim/input
mkdir -p ~/Batsim/output/expe-out/
```

3.4.2.3 Make a base folder

While you are at it, you might as well make a folder for all things your experiments will need:

```
mkdir -p ~/Batsim/base/workloads/
mkdir -p ~/Batsim/base/platforms/
cp ~/Batsim/python-scripts/post-processing.py ~/Batsim/base/post-processing.py
cp ~/Batsim/python-scripts/docopt.py ~/Batsim/base/docopt.py
cp ~/Batsim/python-scripts/docker_real_start.py ~/Batsim/base/real_start.py
cp ~/Batsim/python-scripts/generate_synthetic_workload.py ~/Batsim/base/
    generate_synthetic_workload.py
```

3.4.2.4 Your input config file

Now create an input config file: Copy the following text:

```
{
   "nodes": 46000,
   "SMTBF": 18782.608695652172,
   "checkpoint-interval": 14400,
   "performance-factor": 0.7,
   "checkpointing-on": true,
   "seed-failures": true,
   "platformFile": "/home/sim/base/platforms/platform_46000.xml",
   "synthetic-workload": {
      "workloadFile": "/home/sim/base/workloads/46000n_500j_Grizzley-synthetic500000_512_1490.
          csv_CLN_0_CLN_csvres_Grizzley-synthetic500000_512_1490.
          csv_CLN_1_CLN_h_CLN_csvdur_0_CLN_fixeds_-1lim_3%D_2%
          R_Performance_vs_Memory_failures_e_synth.json"
   }
}
```

and put it in a file:

```
nano ~/Batsim/input/config.ini

#paste the clipboard contents
Ctrl+Shift+V

#edit text (workload, platform)

#save
Ctrl+o
Enter

#exit
Ctrl+x
```

3.4.2.5 Your output config file

Do the same for your output config file: Copy the following text:

```
{
   "AAE": true,
   "raw": 1,
}
```

and put it in a file:

```
nano ~/Batsim/output/config.ini

#paste the clipboard contents
Ctrl+Shift+V

#edit text (shouldn't need to)

#save
Ctrl+o
Enter

#exit
Ctrl+x
```

3.4.2.6 Setup python environment

You may want to setup your python environment now, since we are about to be using some python scripts. Make sure you "source" your python environment.

3.4.2.7 Make a platform file

If you cloned the python-scripts.git, you will have a platform file named "platform1490.xml". You can edit this according to these directions on creating a platform.

3.4.2.8 Make your workload

Make your workload according to these directions.

You are now ready to use docker on proton or gamma.

3.4.3 Docker

3.4.3.1 A note about permissions

An Important Note About Permissions

```
tl;dr
quick and dirty:
```

```
        chmod -R 777 /home/$USER/Batsim/
```

```
tl;dr
more refined:
        make sure running getent group ccu   returns:   ccu:*:4503...some other stuff.
        The 4503 is critical. Then run:
```

```
        chown -R $USER:ccu /home/$USER/Batsim/
        chmod -R 770 /home/$USER/Batsim
```

```
Explanation:
         Almost all linux distros will have a file called /etc/group.
        The format is
```

```
        group_name:passwd:group_id:users,users2...
```

```
        Where passwd is usually "x"(blank), group id is a unique id number for the group,
        and the last colon has a list of users that are part of that group
        getent is a good command for getting group information from all sources when computers
             are networked (there is no "ccu" group entry in /etc/group on proton, probably
            the same with gamma)
        so if I type in
        getent group ccu
        I get
        ccu:*:4503:cwalker,ndebard,gbailey,bslade,wjones,bparker
        so as you can see the group id is 4503
        In /etc/group inside the docker container I put the following entry in /etc/group  :
        ccu:x:4503:sim
        and the reason I do that is so we can write into a folder that has the group id 4503
            as long as the folder has write permission for groups.
        Permissions are like this in linux:  owner,group,other , so the owner of a file or
            folder will have (r)ead,(w)rite,e(x)ecute permissions and so will people of the
            same group and also others.
        So if permissions are    rwxrwxr-x    then the owner can read,write,execute ; the same
             group can read,write,execute, and others can read and execute, but not write
        btw, if it is a folder, then to enter the folder you must have execute permission.  if
             you want to change the permissions of a file or folder you use chmod
        chmod +x somefile.txt  will give execute permission to a file for the owner,group, and
             other.  If you want to be more fine grained you can use an octal number.
        Notice rwx can be thought of as 3 bits. 7 would be rwx, 6 would be rw-, 5 would be r-x
        so you can fine-grain who gets execute permissions like so:
        chmod 740 somefile.txt   The permissions here are owner can rwx, group can only read,
            and other can't do anything
        So how do you change owner and group of a file/folder?
        chown owner:group file.txt  you can also use the -R flag to change all of the files
            and subfolders as well (R is for recursive)
        So that is why the following commands will work for you:
chown -R $USER:ccu /home/$USER/Batsim/    set all files and folders under and including ..../
    Batsim/  to have the group ``ccu"
chmod -R 770 /home/$USER/Batsim   change permissions to allow all permissions for the owner
    and group
```

```
why chmod -R 777  /home/$USER/Batsim   works:
It allows anybody to read write and execute.  That can be problematic, but I don't think it is
     a big deal here.
```

3.4.3.2 Another note on removing files

Another note on removing files

```
Another note:  The owner of any files that the docker container creates will be owned by the
    user "sim", which is user id 1000.
When your container completes and you are back to the host, you will have files owned by
    "1000".  you don't have permission to remove these files.
you must run the following:
docker exec CONTAINER_NAME /bin/bash -c 'chmod -R 777 /home/sim/shared/'
or
docker exec CONTAINER_NAME /bin/bash -c 'chown -R sim:ccu /home/sim/shared/'
now you can remove the files from your usual host environment.
```

You have setup proton or gamma, now we must make sure we have a docker container made that is based on our batsim:w_performance_pinned image. We'll call the container "batsim_container1".

3.4.3.3 Create Container

```
docker create --name "batsim_container1" -v "~/Batsim/base/":/home/sim/base/ -v "~/Batsim/":/
    home/sim/shared/ -t batsim:w_performance_pinned
```

This creates a container named "batsim_container1" and sets up a volume (-v) in docker that can output stuff back to our normal filesystem and get input from there as well. We setup two volumes, one at the base (so we can get files like real_start.py and post-processing.py as well as our workloads and platforms) and the other at Batsim folder itself because that is where our "input" and "output" folders are. The -t option sets which image we will be using to setup this container.

3.4.3.4 Start Container

Now we start our docker container in non-interactive mode (without the -i flag)

```
docker start batsim_container1
```

3.4.3.5 Run Container

Now we can run "real_start.py" through our docker:

```
docker exec --env HOME=/home/sim --env USER=sim batsim_container1 /bin/bash -c 'python3 /home/
    sim/base/real_start.py --path /home/sim/shared/'
```

So this executes non-interactively (no -i flag) whatever you tell it to through a docker container. In this example we first set the environment variables HOME and USER because those get stripped somehow from the docker container and nix needs them set. The next argument is the name of the docker container we are executing in. And finally, the command we want to be ran inside the container. We execute our shell "/bin/bash" and send "bash" a single command with the -c option. That command is the python script "real_start.py" and it will find that within the docker at /home/sim/base/real_start.py, but we know that is mapped to our filesystem at " /Batsim/base/real_start.py".

Might as well reassure you, the docker is setup with a python environment already, so anything that Batsim needs will have been included in the docker already. But anything that you need done outside of Batsim and thus the docker, you need to setup your python environment on gamma/proton accordingly. And don't forget to source the correct file to activate your python environment.

3.4.4 File Access and Jupyter Notebook Access

You can access files from gamma and proton and use a jupyter notebook using an ssh tunnel. For file-access, you will also be using the sftp protocol. These file-access directions are for any Linux distro with KDE or Gnome, but hopefully they are adaptable to any setup you are using.

The first thing you want to do for File Access or Jupyter Notebook is setup your ssh.

3.4.4.1 Setup .ssh/config

Open up a terminal and type:

```
nano /home/$USER/.ssh/config
```

you'll want to edit this file. This is my file:

```
Host *
      ServerAliveInterval 60
      ServerAliveCountMax 10

Host comely
      User cwalker
      HostName comely.newmexicoconsortium.org

Host LANL_proton
      User cwalker
      HostName proton.usrc
      ProxyJump comely
      LocalForward 9999 localhost:8888
      LocalForward 7777 localhost:22
```

What this file says

```
Host *                                  # this says all hosts, do this
       ServerAliveInterval  60          # this sends a null packet to the server every 60 seconds to keep connection
                                        # alive
       ServerAliveCountMax 10           # this is the amount of retries to do if one gets no response from server
                                        # after sending a null packet.

Host comely                             #this is the name of our new host item, an alias for our connection.
       User cwalker                     #this is my username
       HostName comely.newmexicoconsortium.org      #this is the connection we want

Host LANL_proton                        # this is the name of our other host item
       User cwalker                     #same user
       HostName proton.usrc             #this is the second connection we want.
                                        #We are going to jump hosts commonly referred to as a Jump Host.
                                        # first we connect to comely then to proton.
       ProxyJump comely                 # this is the jump we make. Notice it uses the host item, comely, that we
                                        # created above.
       LocalForward 9999 localhost:8888             # Now we forward two ports to our computer.
                                                    # So anything on the server that is port 8888 will come in as
                                                    # localhost:9999, 8888 is usually what port jupyter notebook
                                                    # is on
       LocalForward 7777 localhost:22   # and anything on the server that is port 22 will come in as localhost:7777
                                        # 22 is the usual port for ssh
```

3.4.4.2 Connect to hosts

Now you want to connect using these hosts. Type the following in a terminal:

```
ssh LANL_proton
```

you will need to enter your password twice.
You are now ready for File Access or Jupyter Notebook Access.

3.4.4.3 File Access

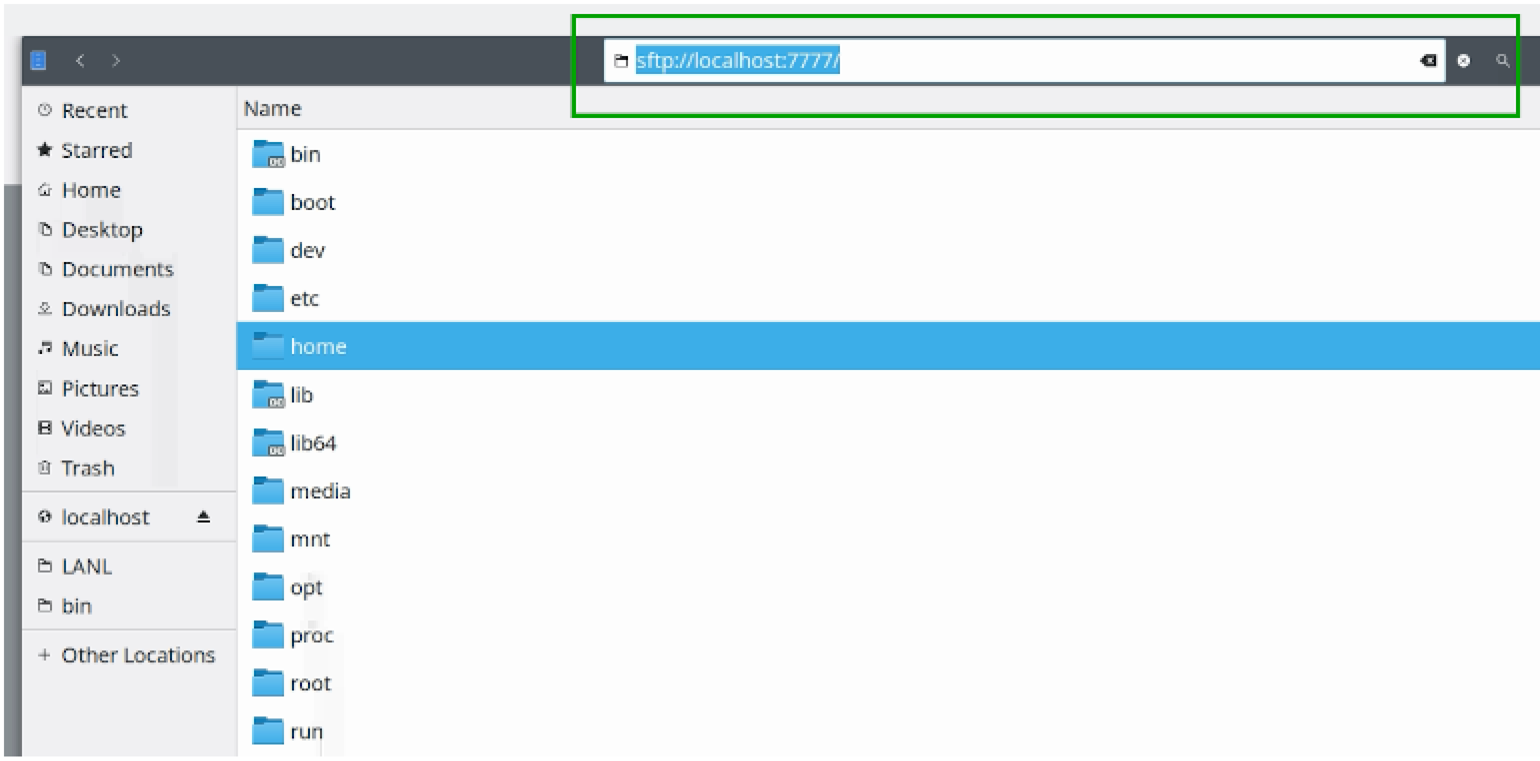
Now fire up Nautilus(Gnome) or Dolphin(KDE).

Nautilus Renamed

I always call the file manager by Gnome ``Nautilus'', but in many distros it has been renamed to just `` Files ''.

3.4.4.3.1 Nautilus(Files) Directions

press Ctrl+L to bring up a box where you can enter a path and enter sftp://localhost:7777



you will be asked for your username and password again.

In the future, to get back to this, simply use the command:

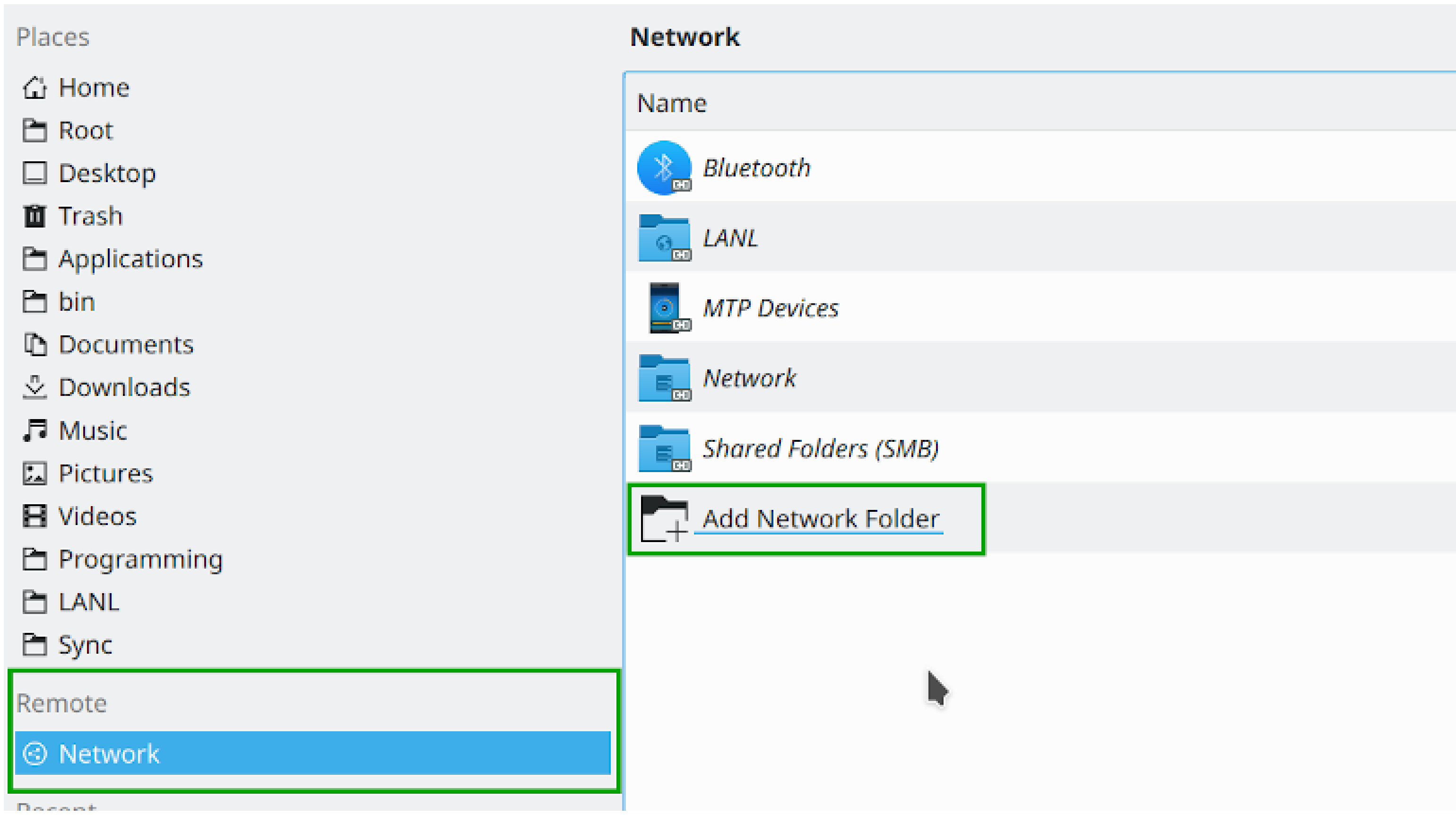
```
ssh LANL_proton
```

you will be asked to enter your password twice. Fire up Nautilus, hit Ctrl+L and enter the path again to sftp://localhost:7777/ . You will be asked again for username and password, and you're done.

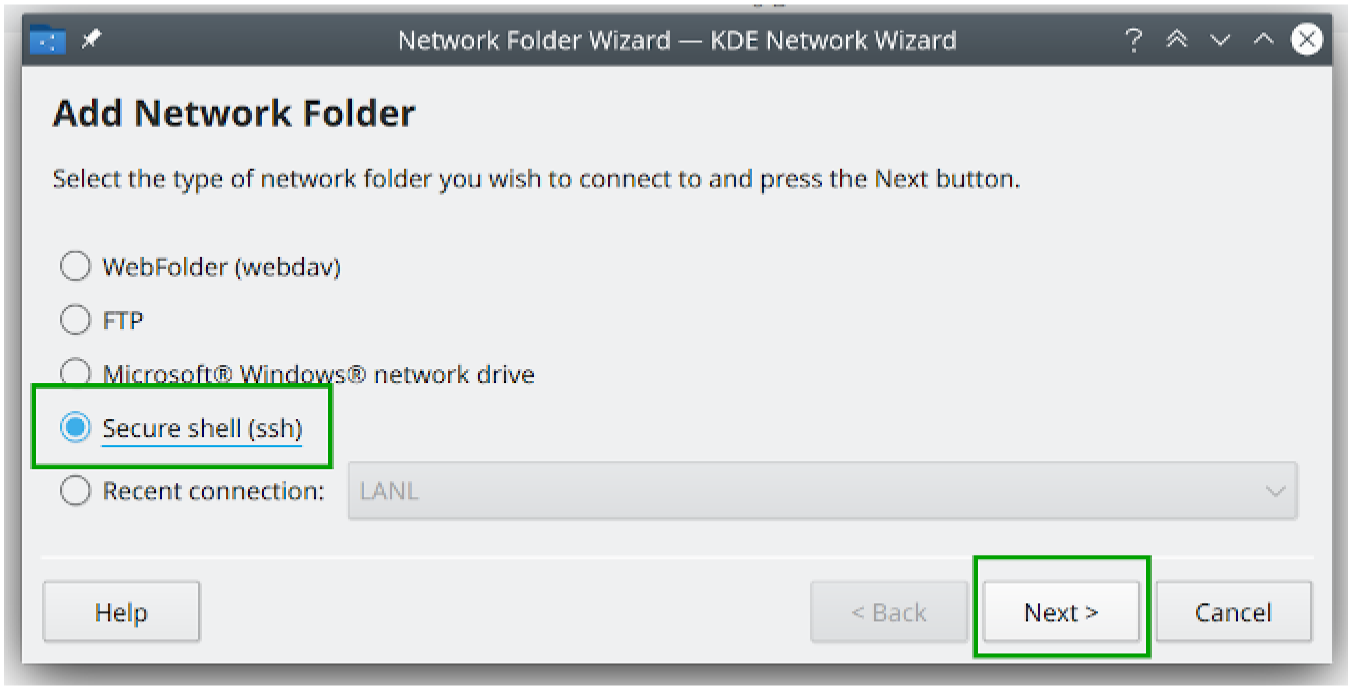
3.4.4.3.2 Dolphin Directions

You can do about the same thing with Dolphin as what you did in Nautilus. So check out those directions. However, I prefer this "Saved" way of doing it.

Look at your "Places" on the left hand side and notice the "Network" link. Click that to bring up your Network connections. You want to add one. Click "Add Network Folder".



That will bring up a box wanting you to choose the type (protocol) of the Network Folder. You want ssh, so select that and hit Next.


Network Folder Wizard — KDE Network Wizard

Add Network Folder

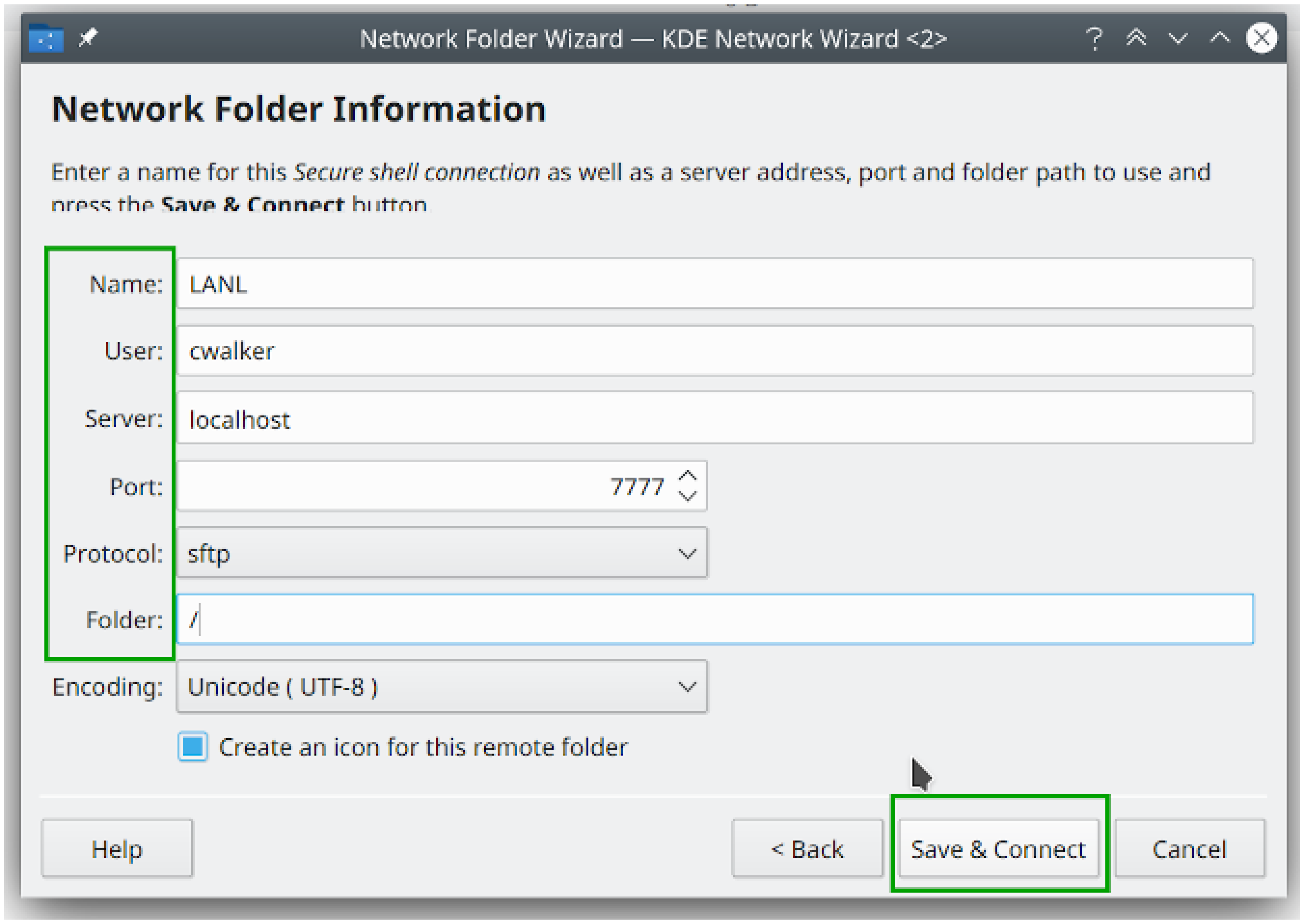
Select the type of network folder you wish to connect to and press the Next button.

- WebFolder (webdav)
- FTP
- Microsoft® Windows® network drive
- [✓ Secure shell (ssh)]
- Recent connection: LANL

That will bring up a box wanting you to enter the connection information. Enter all information and hit Save & Connect.

Enter This Information

```
Name:       [This folder is saved so you can click it and go right to your sftp connection.
                Choose a name for this folder]
User:       [The username you connect to comely with.]
Server:     [The server you are connecting to. This is just ``localhost'' for us]
Port:       [The port you are connecting on. This is the port we forwarded: 7777]
Protocol:   [Make sure this is set to sftp]
Folder:     [Once connected, what folder on the connected server should we start at?]
            [I just use ``/''. You could use your home folder if you wanted, for example: /home/cwalker/ ]
```



Network Folder Wizard — KDE Network Wizard <2>

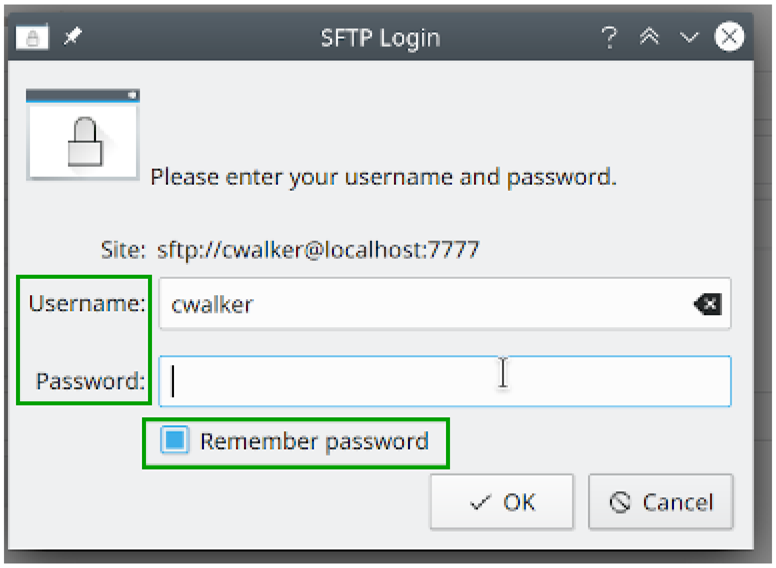
Network Folder Information

Enter a name for this *Secure shell connection* as well as a server address, port and folder path to use and press the **Save & Connect** button

Name:	LANL
User:	cwalker
Server:	localhost
Port:	7777
Protocol:	sftp
Folder:	/
Encoding:	Unicode (UTF-8)

☑ Create an icon for this remote folder

You will need to enter your password again. If you want your password saved Check "Remember Password".

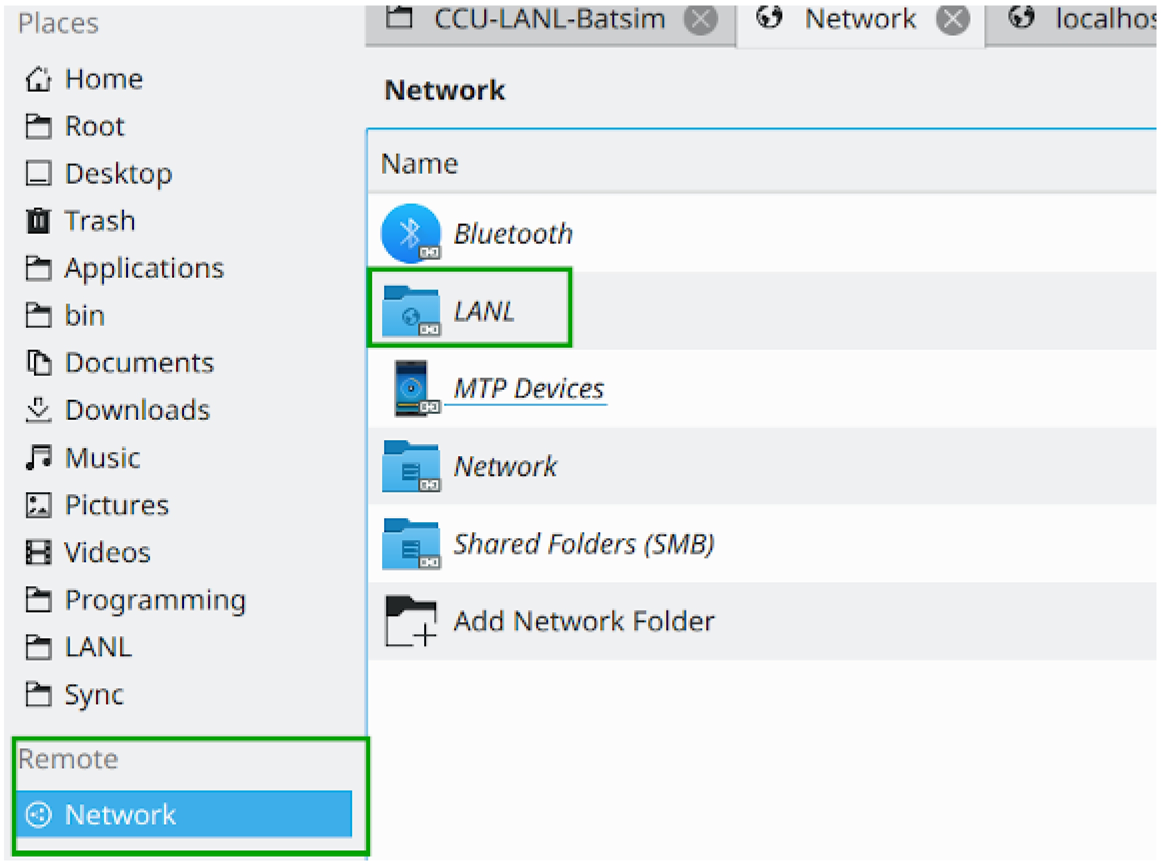

SFTP Login
Please enter your username and password.
Site: sftp://cwalker@localhost:7777
Username: cwalker
Password:
Remember password
OK
Cancel

Hit "OK" and you will be taken to the folder you set as your initial folder. Now you can easily move, copy, paste,upload, download files.

To get back to this in the future, make sure you are connected to proton with command:

```
ssh LANL_proton
```

Now go back to your "Network" link under "Places". Click that to be presented with your Network Connections. Whatever "Name" you chose for your folder will be there. Double-click it and if you chose to "Remember Password" you will now be in your initial folder. Otherwise, first enter your username and password.



3.4.4.4 Jupyter Notebook

You should be on the ssh command line on your proton or gamma remote computer. I prefer to run either “tmux” or “screen” when using ssh. So start a screen or tmux session if you desire. First get into your python environment and issue:

```
jupyter notebook --port=8888
```

Now it is runnning, and make sure it is running on port 8888. It should tell you in the text that appears after issuing “jupyter notebook” what port it is running on.

Through our tunnel we created with our .ssh/config file , we will see jupyter notebook on port 9999.
Open up your Internet Browser on your normal machine and type in the address bar:

```
http://localhost:9999
```

You should be good to go. I have had problems with the jupyter notebook being slow to start the cells, however. ymmv.

3.4.5 Sweeps

There are times when an experiment takes the form: Run a series of simulations increasing parameter X from a to b. This is what we call a ”sweep”. We have made this fairly easy to do with a json config file. It is also somewhat easy to sweep over any new parameters by making your own ”sweep module”. It should be mentioned that sweeping over multiple parameters is at the heart of the config file.

1. First I will go over making a json config file
2. Next I will go over running your sweeps
3. Finally I will go over making your own sweep module

3.4.5.1 Json config

First, you should be familiar with json do’s and don’ts. If an error occurs in running your sweep, it may be a json error and it will tell you where the problem occurs.

Next, to run a simulation you will always need some kind of workload, so always include one in your input config after any sweeps.

You should know that sweeps are much like loops in reverse order. For example, take the following json:

```
"sweep1":{
  },
"sweep2":{
  },
"sweep3":{
  },
...
```

The point of sweeps is to make a whole bunch of experiments. sweep1 will make x experiments, sweep2 will make y experiments, and sweep3 will make z experiments. But they are looped so by running sweep2 you will have $x * y$ experiments making xy experiments. sweep3 comes last and will make $xy * z$ experiments making xyz experiments.

It is often best to start with a node-sweep as this value is often used in other sweeps.

3.4.5.1.1 node-sweep

So let's start with the easiest sweep, a node sweep: (notice I use comments, however, valid json does not have comments)

```
#Start with the name of the whole experiment.
#This will be the folder name that all your sub-experiments fall under.
"Test_Experiment":{

}

#You need an input and output.
#Input deals with what goes into the simulation,
#Output for what comes out of the simulation.
"Test_Experiment":{
               "input":{

               },
               "output":{

               }

}
#Now add in an iterated node-sweep.
"Test_Experiment":{
               "input":{
                  "node-sweep":{
                     "min":20,
                     "max":50,
                     "step":5
                  }
               }
}
# this starts at 20 nodes and goes to 50 with a step of 5. step can be negative, and max is
    inclusive.

# we can also specify a range of known values
                  "node-sweep":{
                     "range":[20,50,100]
                  }
# we can even specify a formula using a single variable "i". "i" can be iterated or in a range
                  "node-sweep":{
                     "formula":"5*i",
                     "min":20,
                     "max":50,
                     "step":10
                  }
#or this does the same thing
                  "node-sweep":{
                     "formula":"5*i",
                     "range":[20,30,40,50]
                  }
# for both formulas we get node values: 100,150,200,250
```

3.4.5.1.2 Remember to include a workload

So that wasn't so bad, but we need to remember our minimal config needs a workload so lets do that:

```
# this is the simplest of workloads
"Test_Experiment":{
                "input":{
                   "node-sweep":{
                      "range":[1490]
                   },
                   "grizzly-workload":{
                      "time":"4-10-2018:6-12-2018",
                      "input":"grizzly_data.csv"
                   }
                }
}
```

So this does one simulation with 1490 nodes and using a workload from april 10 to june 12, using the data found at "grizzly_data.csv".

3.4.5.1.3 Include the output

But we still need an output so lets do that:

```
"Test_Experiment":{
                "input":{
                   "node-sweep":{
                      "range":[1490]
                   },
                   "grizzly-workload":{
                      "time":"4-10-2018:6-12-2018",
                      "input":"grizzly_data.csv"
                   }
                },
                "output":{
                   "AAE":true,
                   "raw":1
                }
}
```

This outputs an Average Application Efficiency and raw post-processing data.

3.4.5.1.4 Other options applied to all

If there are any options you want to apply to all experiments they can come after the sweeps like so:

```
"Test_Experiment":{
                "input":{
                   "node-sweep":{
                      "range":[1490]
                   },
                   "checkpointing-on":true,
                   "seed-failures":true,
                   "grizzly-workload":{
                      "time":"4-10-2018:6-12-2018",
                      "input":"grizzly_data.csv"
                   }
                },
               "output":{
                  "AAE":true,
                  "raw":1
               }
}
```

This turns checkpointing on and also makes failures completely random by seeding the random generator with the current time. Otherwise the experiments would be deterministic.

There are many options and sweeps. For info on these consult "../python-scripts/generate_config.py" by typing:

```
python3 generate_config.py --config-info <type-of-info>

where <type-of-info> can be:
               general | sweeps |
               node-sweep | SMTBF-sweep | checkpoint-sweep | performance-sweep |
               options|
               grizzly-workload | synthetic-workload |
               output
```

You can have multiple experiments

If you want to run multiple experiments from a single config file, that is no problem.
Just format your file like so:

```
"Test_Experiment_1":{
                      "input":{
                        ...
                      },
                      "output":{
                        ...
                      },
                    },
"Test_Experiment_2":{
                      "input":{
                        ...
                      },
                      "output":{
                        ...
                      },
                    }
```

Experiments in Test_Experiment_1 will be in a folder with that name, and
experiments in Test_Experiment_2 will be in a folder with that name.

3.4.5.2 Running a sweep

So now we have a valid sweep json config file. Let's run it:

Long Wait

This command can take a long time to complete depending on how many experiments and runs you are doing.
Above 10,000 jobs is on the order of 30 minutes.

```
python3 generate_config.py -i config_file_name.config -o /home/$USER/Experiments/folder_name
    --basefiles /home/$USER/Batsim/base/ --output-config
```

This will generate all the folders we need in /home/$USER/Experiments/folder_name/. It will generate the workloads and platforms you need and put them in –basefiles/workloads and –basefiles/platforms (so make sure you have those folders there). It will also output the config file into the /home/$USER/Experiments/folder_name/ folder under the same name as the original.

Next we want to run the experiments:

Long Wait

Our SLURM was set to only take 10,000 jobs
at a time, so it may be queuing jobs for some time if you are above that.

```
python3 run_experiments -i /home/$USER/Experiments/folder_name/
```

This will create the slurm batch jobs and queue them up.

3.4.5.3 Make a sweep module

There are currently 4 sweeps one can use: node_sweep,checkpoint_sweep,performance_sweep,SMTBF_sweep. If your needs go beyond that, you will ultimately need to make your own sweep. Here we will go over how to do this.

Keep in mind that although generate_config.py creates the folders needed for all the experiments and creates the workloads and platforms needed, it's main purpose is to create config files. It takes the information in the json and creates the necessary config file for each experiment and puts it in it's "input" folder. So, a simple example of this: We sweep from 1490 to 1500 nodes. That is 11 experiments. 11 config files will be created, the first one having an entry "nodes":1490, the next one "nodes":1491 and so on...

Let's look at a sweep that is already made so we can show how to make our own. We'll look at node_sweep. It is just a normal .py file.
node_sweep.py

```python
from sweeps.sweepFunctions import *
import numpy as np

def nodeSweep(nodeSweepInput,ourInput):
   myRange = nodeSweepInput["range"] if dictHasKey(nodeSweepInput,"range") else False
   myFormula = nodeSweepInput["formula"] if dictHasKey(nodeSweepInput,"formula") else False
   if type(myRange) == bool:
      #ok so we are going to have a min,max,step
      minimum = float(nodeSweepInput["min"])
      maximum = float(nodeSweepInput["max"])
      step = float(nodeSweepInput["step"])
      if myFormula:
         #ok so we have a formula
         formula_range = list(np.arange(minimum,maximum+step,step))
         nodeRange = [eval(myFormula) for i in formula_range]
      else:
         nodeRange = list(np.arange(minimum,maximum+step,step))
   elif myFormula:
      formula_range = myRange
      nodeRange = [eval(myFormula) for i in formula_range]
   else:
      nodeRange = myRange
   currentExperiments = len(ourInput.keys())
   #if there were no sweeps before
   if currentExperiments == 0:
      count = 1
      for i in nodeRange:
         ourInput["experiment_{count}".format(count=count)]={"nodes":i}
         count+=1
   #there were sweeps before
   else:
      tmpInput = ourInput.copy()
      count = 1
      # update the current experiments first
      for i in ourInput.keys():
         data = ourInput[i].copy()
         data["nodes"] = nodeRange[0]
         ourInput[i] = data
         count+=1
      for i in nodeRange:
         if not i == nodeRange[0]: #skip the first, we already did it
            for j in tmpInput.keys():
               data = tmpInput[j]
               data["nodes"] = i
               ourInput["experiment_{count}".format(count=count)] = data
               count+=1
```

It may be helpful to see what is in sweepFunctions so here it is:
sweepFunctions.py

```
__all__=["dictHasKey"]

def dictHasKey(myDict,key):
   if key in myDict.keys():
      return True
   else:
      return False
```

This is just code to check if the json has a certain option or not. You should be familiar with numpy, so we have covered the imports. So let's move on:

```
def nodeSweep(nodeSweepInput,ourInput):
   myRange = nodeSweepInput["range"] if dictHasKey(nodeSweepInput,"range") else False
   myFormula = nodeSweepInput["formula"] if dictHasKey(nodeSweepInput,"formula") else False
   if type(myRange) == bool:
      #ok so we are going to have a min,max,step
      minimum = float(nodeSweepInput["min"])
      maximum = float(nodeSweepInput["max"])
      step = float(nodeSweepInput["step"])
      if myFormula:
         #ok so we have a formula
         formula_range = list(np.arange(minimum,maximum+step,step))
         nodeRange = [eval(myFormula) for i in formula_range]
      else:
         nodeRange = list(np.arange(minimum,maximum+step,step))
   elif myFormula:
      formula_range = myRange
      nodeRange = [eval(myFormula) for i in formula_range]
   else:
      nodeRange = myRange
```

So we start with the name of the function. We can call it anything. The arguments to a sweep are the json inside the sweep, called nodeSweepInput here, and the accumulated configuration

3.4.6 Process Sweeps

4 Internals