11falsenonelisttrue

storageN

The use of `storageN` within these documents indicates that any storage platform can be used.

Current available storage platforms:

storage1

storage2

General Guidelines and Tips

LSF puts a lot of hardware at your fingertips, but it must be used wisely so you don't affect others' work. This page contains guidelines and tips for using LSF smartly.

Don't submit 1000 jobs until you've seen 1 job finish successfully.

Job submission has overhead. Avoid submitting jobs that complete in just seconds. If you can't avoid it, look into job arrays.

Limit the total number of files you place in a single directory. On the order of thousands at most. Normal filesystem operations become unweildy with directories of millions of files.

Every LSF jobs gets its own temporary directory that gets cleaned up for you! Do your work there, if possible. The path is: `/tmp/$JOBID.tmpdir`.

Don't monopolize queues with large numbers of long-running (multi-day) jobs. Use job groups to limit your running jobs if necessary.

This document is just for quick reference and examples and does not cover all IBM LSF features. Refer IBM Spectrum LSF official documentation for a more detailed and extensive features.

Launch Interactive Jobs

From the `man bsub` manual page:

It is possible to submit an interactive job from within an already interactive job. Take note that the first job must be started with "host networking" for this to work:

LSF_DOCKER_NETWORK=host bsub -G ${group_name} -Is -q general-interactive -a 'docker(python:slim)' /bin/bash Job <54727> is submitted to queue . <> <> slim: Pulling from library/python bc51dd8edc1b: Pull complete dc4aa7361f66: Pull complete f7d31f0d4202: Pull complete edfb78ef674e: Pull complete 9630f24835d2: Pull complete Digest: sha256:73f3903470a6e55202a6bb989c23b047487eb1728feba655410076da24106838 Status: Downloaded newer image for python:slim docker.io/library/python:slim ${compute_username}@compute1-exec-124:~$ > bsub -Is -G ${group_name} -a 'docker(ubuntu:bionic)' /bin/bash Job <54728> is submitted to default queue . <> <> bionic: Pulling from library/ubuntu Digest: sha256:8d31dad0c58f552e890d68bbfb735588b6b820a46e459672d96e585871acc110 Status: Downloaded newer image for ubuntu:bionic docker.io/library/ubuntu:bionic ${compute_username}@compute1-exec-79:~$]]>

Start an Interactive Job With Access to All My Data

As the client login machines are only indended for light work, you will likely want access to an interactive shell for heavier computing such as development or testing which could require access to all the data you have available.

There are three major storage locations for data:

Your home directory with scripts and config: /home/${compute_username} or patterned like: /home/IDC-ID-123456

Your storageN allocation: /storageN/fs1/${STORAGE_ALLOCATION}/Active

Your scratch1 space: /scratch1/fs1/${COMPUTE_ALLOCATION}

Noting that the current working directory is always mounted by the container automatically, and the `LSF_DOCKER_VOLUMES` variable requests other volumes to be mounted, this will start an interactive job similar to the compute client host, but running on more substantial hardware:

cd # switch to your home directory > export LSF_DOCKER_VOLUMES='/storageN/fs1/${STORAGE_ALLOCATION}/Active:/storageN/fs1/${STORAGE_ALLOCATION}/Active /scratch1/fs1/${COMPUTE_ALLOCATION}:/scratch1/fs1/${COMPUTE_ALLOCATION}' > bsub -G ${group_name} -q general-interactive -Is -a 'docker(ubuntu)' /bin/bash]]>

Using "Condos" or Landing Jobs on Specific Hosts

Users who are members of “condo” groups may submit jobs to their “condo” by specifying the condo queue, noting the difference between interactive and non-interactive jobs: `-q labname-interactive` or `-q labname`

Also note that if a user is a member of **more than one** compute group, use the `-G` argument to specify which group you are submitting for: `-G compute-labname`

bsub -G compute-labname -Is -q labname-interactive -a 'docker(alpine)' date Job <54732> is submitted to queue . <> <> ...]]>

Users who are not “condo” customers can still choose a specific host to land a job on, provided access controls allow them to do so.

To land a job on a specific host, use the `-m` flag:

is submitted to queue . <> <> ... [user@compute1-exec-174 ~]$]]>

Difference Between Interactive and Batch Mode

An interactive mode of operation provide you access to the shell of the compute node. For e.g.
Below job requests for an interactive session by including **-Is** flag and using an interactive queue (**-q).** This is evident in the above example where the shell prompt changed from `user@compute1-client-1` to `user@compute1-exec-174`

On contrast a batch job does not provide you access to shell. When a user submits a job it goes into `PENDING` state, awaiting resources availability and finally once the resources are available the job gets `DISPATCHED` to a compute host and goes into `RUNNING` state. Once the job completes the output is shared with the user in an email and/or configured output files.

Assign Execution Priority

Use the `-sp` flag to set a numeric priority to order the execution of your jobs:

Note that you can modify the priority of jobs in a PENDING state by using `bmod -sp`. See `man bmod`:

bmod -sp 99 1024]]>

Job Groups

It frequently happens that we want to run a resource-intensive process on hundreds or thousands of data sets. In some cases, the load this imposes on the system will become apparent to other users. They will experience increased latency if their data lives on the same filesystem. Using job groups you can create the effect of a “personal queue”.

First, create a job group and assign it a limit of N running jobs:

bgadd -L 10 /${compute_username}/${group_name} Job group was added.]]>

To check the status of jobs in the job group:

bjgroup -s /${compute_username}/${group_name} ${group_name} NJOBS PEND RUN SSUSP USUSP FINISH SLA JLIMIT OWNER /${compute_username}/${group_name} 0 0 0 0 0 0 () 0/20 ${compute_username}]]>

Assign jobs to this group at submission time, using `-g`:

bsub -g /${compute_username}/${group_name} -J ${job_name} ...]]>

List jobs in this job group using bjobs:

bjobs -g /${compute_username}/${group_name}]]>

List a summary of jobs in the job group using bjgroup:

bjgroup -s /${compute_username}/${group_name}]]>

Alter the number of running jobs in the job group:

bgmod -L 5 /${compute_username}/${group_name}]]>

Default Job Group

If you do not supply a job group with the `bsub -g` argument, a default job group named `/${compute_username}/default` will be created. This group has a default of 5 jobs, meaning only 5 jobs will run at a time, even if you are not at the max vCPU of 500.

Arrays

When submitting a lot of similar jobs, they should submitted as an array. This allows you to easily submit and manage all of the jobs as a single entity. A single LSF bsub command is used to submit and can be altered with a single command as well. Each job in the array can also be accessed and modified individually if necessary.

The maximum number of jobs allowed in a single array is 1000.

To submit one thousand jobs as a single job array called myArray, you would issue the following command:

bsub -J 'myArray[1-1000]' SCRIPT]]>

Let's say the above command created a job with the job ID 123. You can then refer to the entire array using the name, myArray, or the job ID, 123. You can refer to individual jobs using a subscript, e.g., to refer to the tenth job in the array you could use myArray[10] or 123[10]. Each job can figure out what element of the job array it is by using the LSB_JOBINDEX environment variable.

If each job needs an input file, the input files should all have the same base name with an extension equal to the job in the array the input file corresponds to, e.g., input.1, input.2, input.3,…

To submit 100 jobs that run the script multiblaster in a single array named blast_array:

bsub -J 'blast_array[1-100]' multiblaster]]>

To create a job array named ipr_array with 800 jobs that run run_ipr and each job has a different input file, ipr.1 to ipr.800:

bsub -J 'ipr_array[1-800]' run_ipr ipr.\$LSB_JOBINDEX]]>

To create a job array named chimp_submit that has 500 jobs, each running the command ncbi_submit chimp and each expecting the contents of file named chimp_stdin.N in the current directory on its standard input where N is the number of the job in the array:

bsub -J 'chimp_submit[1-500]' -i 'chimp_stdin.%I' ncbi_submit chimp]]>

To do the same as above but also write the standard output and standard error to the file chimp_out.M.N, where M is the array job number and N is the number of the job in the array:

bsub -J 'chimp_submit[1-500]' -i 'chimp_stdin.%I' -o 'chimp_out.%J.%I' ncbi_submit chimp]]>

Notifications

By default, completed jobs will send an email notification to the `mail` address of the execution $USER. Control this by using the `-u` command line argument, see `man bsub` for more about `-u`.

Resource Requirements

Different physical compute nodes will be equipped with different "resources" such as number of CPUs or GPUS, amount of physical memory, perhaps a processor type, etc. Rather than asking for a specific host, using resource requests allow your job to land on one of many hosts that might satisfy your needs. Use the `-R` flag with a space separated string of resource requirements:

bsub \ -n 2 \ -q foo-condo \ -m foo-condo \ -g my_group \ -J my_name \ -M 8000000 \ -W 10 \ -N \ -u myemail@wustl.edu \ -i input_file \ -oo ${HOME}/path/to/output_file \ -R 'select[mem>8000 && tmp>2] \ rusage[mem=8000, tmp=2, matlab=1] span[hosts=1]' \ /usr/bin/some_program]]>

All of these arguments are optional, but they indicate some of the available features. In English, this means the job…

- is submitted to the foo-condo queue (-q foo-condo)
- is limited to running on the host group foo-condo (-m foo-condo)
- uses the job group my_group (-g my_group)
- has name my_name (-J my_name)
- requires 2 processors (-n 2)
- will be killed if its memory usage exceeds 8GB (-M 8000000 in KB)
- will be killed if it runs for longer than 10 minutes (-W 10)
- sends an email when it's done, even with other output options specified (-N)
- any email sent goes to myemail@wustl.edu (-u)
- reads input_file into STDIN (-i input_file)
- overwrites output to `${HOME}/group/path/output_file` (-oo)

will only run on a host with more than 8000 MB of RAM and 2 GB of local temp disk (-R ‘select[mem>8000 && tmp>2]’)

will consume 8000 MB of RAM, 2 GB of local temp disk (-R ‘rusage[mem=8000, tmp=2]’)

will span only a single host (-R ‘span[hosts=1]’)

will execute /usr/bin/some_program with no arguments

The default unit for limits and reservations is KB.

bsub Job Files

The options for the `bsub` command can be placed within a job file. However, the LSF environment variables will need to be appended either at the beginning of the `bsub` command to submit the job file or exported prior. The previous options would look like the following within a file named `resources_example.bsub`.

8000 && tmp>2] rusage[mem=8000, tmp=2, matlab=1] span[hosts=1]' /usr/bin/some_program]]>

The `#!/bin/bash` line needs to refer to the native interpreter of the Docker image. The most common interpreters are either BASH (`#!/bin/bash`) or Bourne (`#!/bin/sh`). This line is often referred as shebang.

The job would then be submitted in the following manner.

You can also use a job file like this to start up interactive jobs or run jobs interactively.

Output Files

By default for each batch job an email notification is sent to the user upon job completion. This feature is not available for interactive jobs since the user can interactively see job output. However, if a user wishes to redirect job output into a text file one can specify this using `-oo` and `-eo` arguments. `-oo` argument redirect all output to the given file path, while `-eo` argument will redirect error output to the given file path. Users can use also use `%J` which gets substituted by the submitted job id.

bsub -G ${group_name} -q foo-condo -oo ${HOME}/output.%J.out -eo ${HOME}/error_out.%J.out -a 'docker(alpine)' /bin/true]]>

Note: Shell variable can be used by enclosing them in `${}`. Refer above example that uses `$HOME`shell variable in the bsub command.

Memory Resources

The default memory reservation for a compute job is 4GB, and will be killed if it goes over that reservation. One can specify a custom memory reservation with the `rusage` expression, and the the kill threshold with the -M option:

bsub -G ${group_name} -q foo-condo -R 'rusage[mem=8GB]' -M 6GB -a 'docker(alpine)' /bin/true]]>

The above example reserves 8GB of memory, but the job will be killed if it uses more than 6GB. If `-M` is missing, it will default to match the reservation amount.

Try to be as accurate as possible with the memory reservation. Using an artificially high reservation wastes a shared resource, and makes it more difficult to get your job scheduled.

CPU Resources

By default, a job will use any CPU that is available to run the job. If a job has specific CPU architecture resource requirements, one can specify a CPU architecture with the `rusage` expression.

bsub -q foo-condo -R 'select[model==Intel_Xeon_Gold6242CPU280GHz]' -a 'docker(alpine)' /bin/true]]>

Where `model=…` is the host’s CPU architecture.

Find the CPU architecture of a host via `lshosts -w host_name`" The “host_name” variable can be a space-separated list of hosts.

To find the “host_name” for each host in a condo or host group, use `bhosts -w group_name`

Current CPU architecture available in the general host group: `Intel_Xeon_Gold6242CPU280GHz, Intel_Xeon_Gold6148CPU240GHz`

Slot Limits:

The Compute Platform has a limit of 500 slots per user. This essentially means 500 vCPUs. This is across all jobs. For example, you can have 500 jobs running with 1 vCPU each, or 10 jobs running with 50 vCPUs each, or 1 job running with 500 vCPUs, and if you are using 490 vCPUs and a job you submit needs 15, it will sit in queue until your total is 485 or less.

GPU Resources

To use a GPU, one must include the `-gpu` flag and the `-R 'gpuhost'` option as part of the job submission:

If you are using other options that are part of the `-R` option, you will need to list `gpuhost` first.

If you are using `select` for another option, you will need to place `gpuhost` within that and first.

```
-R select[gpuhost && port8888=1]
```

bsub -q foo-condo -R 'gpuhost' -gpu "num=1:gmodel=TeslaV100_SXM2_32GB" -a 'docker(alpine)' /bin/true]]>

Where `gmodel=…` is the model of GPU to use, and `num=…` is the number of GPUs required for the job.

List all available GPUs via `bhosts -gpu -w general`. Replace “general” with the name of a condo or host group to see the GPU information for that group.

Current GPU models available in the general host group:

Double Precision

```
TeslaV100_SXM2_32GB

NVIDIAA100_SXM4_40GB
```

Single Precision

```
NVIDIAA40
```

We provide services for the entire WashU campus and as such, we suggest a best practices of not over allocating how many resources a single user utilizes. To that point, we suggest that a single user take up no more than 10 GPUs at a time.

Jobs using GPUs should reserve an appropriate amount of GPU memory with the gmem resource in the `-gpu` argument. The system will impose a default reservation of 2G if the job submission does not specify something different. For example:

bsub -q foo-condo -R 'gpuhost' -gpu "num=1:gmem=8G" -a 'docker(alpine)' /bin/true]]>

GPU jobs automatically **have** the `j_exclusive=true` flag set, meaning that all GPU jobs, by default, are exclusive to the GPU or GPUs requested and no other jobs will use that GPU. If a user wishes to allow multiple jobs to run on a GPU, the flag can be set to false as part of the -gpu option.

Attach to an Already Running Container

Sometimes it’s desirable to run a new process within an already running container, for example to investigate a hung process. One can submit a new job that runs within a container created by an already running job with the `docker_exec` application profile.

First, find the ID for the running job:

bjobs JOBID USER STAT QUEUE FROM_HOST EXEC_HOST JOB_NAME SUBMIT_TIME 2044 ${compute_username} RUN general compute1-ex compute1-ex /bin/proce Oct 17 16:54]]>

Then, start a new job to inspect the contents of a log file within that container using the `docker_exec` application profile and pass in the job ID:

bsub -G ${group_name} -I -q general -a 'docker_exec(2044)' cat /path/to/log/file]]>

Or to start an interactive shell within that container:

bsub -G ${group_name} -q general-interactive -Is -a 'docker_exec(2044)' /bin/bash]]>

Please note that you will need to attach to the running container using the same queue as the job that created it. For instance, if the job that created the container used the `general-interactive` queue, you will need to attach to it using the `general-interactive` queue. This also means that you cannot connect interactively to a job submitted to the `general` queue.

Currently, only connecting to an interactive job using the `general-interactive` queue is supported.

Some LSF jobs started with more than one slot with `bsub -n N` execute on multiple hosts. When attaching to the container of one of these jobs, you might see this message:

which means the `-m hostname` option is required in order to tell the system which host’s container to attach to. First, find the list of hosts assigned to the job:

bjobs -w 2044 JOBID USER STAT QUEUE FROM_HOST EXEC_HOST JOB_NAME SUBMIT_TIME 2044 ${compute_username} RUN general compute1-client-1.ris.wustl.edu 1*compute1-exec-174.ris.wustl.edu:1*compute1-exec-178.ris.wustl.edu sleep 10000 Oct 17 16:54]]>

The `EXEC_HOST` column (6th) contains the list of hosts, separated by colons. Choose which host to connect to, and include it in bsub’s `-m` option:

bsub -G ${group_name} -m compute1-exec-174 -q general-interactive -Is -a 'docker_exec(2044)' /bin/bash]]>

You may have to wait some time for the new job to be scheduled if the host is busy.

Kill Running Jobs

Jobs can be stopped using `bstop` and later resumed with `bresume`.

In general, we recommend you simply kill jobs with `bkill`.

To kill the job with Job ID 9753:

bkill 9753]]>

To kill all your jobs:

bkill 0]]>

To kill all of your jobs that are running on the host group `foo-condo`:

bkill -m foo-condo 0]]>

To kill all the jobs from the job group named `my_group`:

bkill -g my_group 0]]>

To stop jobs 10, 20 and 55-65 from the job array named blast_array with ID 456:

bstop 'my_array[10, 20, 55-65]' > bkill -s STOP '456[10, 20, 55-65]']]>

To resume the above job:

bresume 'blast_array[10, 20, 55-65]' > bkill -s CONT '456[10, 20, 55-65]']]>

Host Resources

To see which (if any) hosts have enough resources to handle jobs you'd like to submit, use the bhosts -R command. For example, the below query lists all hosts in the `foo-condo` host group that have at least 25GB of unreserved memory, 25GB of free memory and 10GB of local disk space.

bhosts -R 'select[mem>25000 && free_memory_mb>25000 && tmp>10]' foo-condo HOST_NAME STATUS JL/U MAX NJOBS RUN SSUSP USUSP RSV compute1-exec-1 ok - 24 1 1 0 0 0 compute1-exec-2 ok - 24 0 0 0 0 0 compute1-exec-3 ok - 24 0 0 0 0 0 compute1-exec-4 ok - 24 0 0 0 0 0 compute1-exec-5 ok - 24 0 0 0 0 0]]>

To query total host resources instead of what's currently available, use the `lshosts` command. This will only accept a host name, not a host group name. Extra bash is needed if you want to query an entire host group.

bhosts -w foo-condo | tail -n +2 | awk '{print $1}' | xargs lshosts -w -R 'select[mem>25000 && free_memory_mb>25000 && tmp>10]' HOST_NAME type model cpuf ncpus maxmem maxswp server RESOURCES compute1-exec-1 LINUX64 Intel_EM64T 60.0 12 96671M - Yes (docker) compute1-exec-1 LINUX64 Intel_EM64T 60.0 12 96671M - Yes (docker) compute1-exec-1 LINUX64 Intel_EM64T 60.0 12 96671M - Yes (docker) compute1-exec-1 LINUX64 Intel_EM64T 60.0 12 96671M - Yes (docker) compute1-exec-1 LINUX64 Intel_EM64T 60.0 12 96671M - Yes (docker) compute1-exec-1 LINUX64 Intel_EM64T 60.0 12 96671M - Yes (docker) compute1-exec-1 LINUX64 Intel_EM64T 60.0 12 96671M - Yes (docker)]]>

Host Status

A host's status can change as it gets busy, is closed or goes down. Use the bhosts -w command to see a host's full status (without the wide format flag, the status will be abbreviated). For example:

bhosts -w foo-condo HOST_NAME STATUS JL/U MAX NJOBS RUN SSUSP USUSP RSV compute1-exec-1 ok - 8 4 4 0 0 0 compute1-exec-2 closed_Full - 8 8 8 0 0 0 compute1-exec-3 unavail - 8 0 0 0 0 0 compute1-exec-4 closed_Adm - 8 0 0 0 0 0 compute1-exec-5 closed_Busy - 8 0 0 0 0 0]]>

- `ok` - the host is open for jobs and has resources available.
- `closed_Full` - all job slots are currently occupied by running jobs.
- `unavail` - LSF can't talk to some of the services on the host. Any jobs on the host are likely in an unknown state and no new jobs will be scheduled.
- `closed_Adm` - an administrator has closed the blade to new jobs.
- `closed_Busy` - some resource on the blade is over-consumed and it won't accept new jobs until that resource is freed.

Expose Ports From Within Containers

Normally, processes that open ports within containers are not reachable outside the container. docker enables access to these ports with its `-p` option, and the compute cluster exposes that functionality with the `LSF_DOCKER_PORTS` environment variable, and through LSF resources with names such as `port8765`.

Ports 8000-8999 are available for containers to use.

For example, to start an apache container and allow outside access from port 8001 and forward it to the web server listening on port 80. Note the use of the `-R` option to run the job on a host where port 8001 is available.

LSF_DOCKER_PORTS='8001:80' bsub -G ${group_name} -q general-interactive -R 'select[port8001=1]' -a 'docker(httpd)' /usr/local/bin/httpd-foreground]]>

`bsub`'s -R option does not have a syntax for specifying ranges of resource names. For jobs using a range of ports, the resource request must mention each port by name, and joined with &&:

LSF_DOCKER_PORTS='8001-8003' bsub -G ${group_name} -q general-interactive -R 'select[port8001=1 && port8002=1 && port8003=1]' -a docker(ubuntu) /bin/true]]>

More detail is available in the LSF_DOCKER_PORTS detail section.

Set container-specific environment variables

Most environment variables are preserved between the shell the job was submitted from, and within the job's container. To set some environment variables exclusively inside the container without setting them in the submission environment, create a text file with the variables and their values, and put the file names in the LSF_DOCKER_ENV_FILE environment variable.

More information can be found in the Docker Wrapper Environment Variables section of the manual.

Set container-specific host definitions

The `--add-host` option of `docker run` is used to add host names in the container's /etc/hosts file to allow those names to resolve to custom IP addresses. Its functionality is exposed in two ways.

First, the `LSF_DOCKER_ADD_HOST` environment variable can contain a list of space-separated `hostname:ipaddress` pairs:

The two host names, host_a and host_b, will then appear in the container's /etc/hosts file and resolve to the given IP addresses.

The second way is for the `LSF_DOCKER_ADD_HOST_FILE` to name a file containing one or more `hostname:ipaddress` entries, one per line. The equivalent host file for the above command could be put into a file named `my_host_file` and would look like:

and would be put into use with:

Application Profiles

Note that the compute cluster offers a number of applications. These are pre-configured by the RIS team and offer a user interface to the compute cluster. One requests an application profile by using the `-a "application(arguments)` flags with the `bsub` command. Note that there are two ways IBM LSF exposes this capability: bsub -a and bsub -app which differ slightly.

This is functionally equivalent to bsub -app docker:

We write our examples and documentation using the -a argument:

bsub -G ${group_name} -Is -q general-interactive -a 'docker(alpine)' date]]>

But this is functionally equivalent to:

LSB_CONTAINER_IMAGE=alpine bsub -G ${group_name} -Is -q general-interactive -app docker date]]>

See available application profiles with the `bapp -l` command:

bapp -l APPLICATION NAME: docker -- Runs docker jobs via EXEC_DRIVER https://confluence.ris.wustl.edu/display/RSUM/02%3A+RIS+Compute+Management STATISTICS: NJOBS PEND RUN SSUSP USUSP RSV 0 0 0 0 0 0 PARAMETERS: CONTAINER: docker[image($LSB_CONTAINER_IMAGE) options(--rm)] EXEC_DRIVER: context[user(lsfadmin)] starter[/opt/ibm/lsfsuite/lsf/10.1/linux2.6-glibc2.3-x86_64/etc/docker1_starter.py] controller[/opt/ibm/lsfsuite/lsf/10.1/linux2.6-glibc2.3-x86_64/etc/docker-control.py] monitor[/opt/ibm/lsfsuite/lsf/10.1/linux2.6-glibc2.3-x86_64/etc/docker-monitor.py] ---------- APPLICATION NAME: docker0 -- Runs docker jobs via JOB_STARTER https://confluence.ris.wustl.edu/display/RSUM/02%3A+RIS+Compute+Management STATISTICS: NJOBS PEND RUN SSUSP USUSP RSV 0 0 0 0 0 0 PARAMETERS: JOB_STARTER: /opt/ibm/lsfsuite/lsf/10.1/linux2.6-glibc2.3-x86_64/etc/docker_run_v2.py ---------- APPLICATION NAME: docker1 -- Runs docker jobs via EXEC_DRIVER https://confluence.ris.wustl.edu/display/RSUM/02%3A+RIS+Compute+Management STATISTICS: NJOBS PEND RUN SSUSP USUSP RSV 0 0 0 0 0 0 PARAMETERS: CONTAINER: docker[image($LSB_CONTAINER_IMAGE) options(--rm)] EXEC_DRIVER: context[user(lsfadmin)] starter[/opt/ibm/lsfsuite/lsf/10.1/linux2.6-glibc2.3-x86_64/etc/docker1_starter.py] controller[/opt/ibm/lsfsuite/lsf/10.1/linux2.6-glibc2.3-x86_64/etc/docker-control.py] monitor[/opt/ibm/lsfsuite/lsf/10.1/linux2.6-glibc2.3-x86_64/etc/docker-monitor.py] ----------]]>

Numbered Application Profiles

Users will note that some application profiles have a number, like “docker0” and “docker1”, as well as one “docker”, without a number. This is a “versioned interface”. The un-numbered application will always refer to the latest profile. Recall the example used above:

bsub -G ${group_name} -Is -q general-interactive -a 'docker(alpine)' date]]>

This example uses “docker” with no number. Since “docker1” is the latest available application interface, “docker” can be thought of as a link to “docker1”. Numbered applications like docker0 to dockerN represent “versioned APIs”, so that use cases designed to work with specific implementations are not broken by future releases.

Users should use the “docker” application profile to always use the latest implementation. But users **have the option** to refer to the specific interface that works with there software. We hope this provides stability.

Docker1 Application Usage Notes

LSF ensures environment variables for jobs match the environment when the job was submitted, with a few exceptions:

HOSTNAME - containers inherit the hostname of the host they land on

LSB_INTERACTIVE - prevents nested jobs from defaulting to interactive

In addition, bash functions are implemented as envrionment variables, and are removed from the container’s environment. It’s fine if those functions are re-defined inside the container after it starts, whether through a .profile script (or equivalent) or some other mechanism.