qpy The Queue Management System in Python

User, Administrator, and Developer manual

version 0.0, 2018

Conceived and created by Pradipta Kumar Samanta and Yuri Alexandre Aoto

With the kind help and support of Andreas Köhn and Arne Bargholz

November 13, 2018

Contents

1 For users

q*py* is used for submitting jobs from a main server and executing them on the nodes of a compute cluster, according the availability or the options given by the user. It can handle both single-processor and multiprocessor jobs and has several options for user-friendly interaction. If you are using q*py* for the first time, please read carefully the Section ??, *Basics*, just below. For a complete list of commands and their options, see the Section ??, *Commands*.

1.1 Basics

1.1.1 Installation

The main installation will probably be done by your system administrator. You might have to do, however, the following:

Put the following lines in your ~/.bash_profile or ~/.bashrc:

```
export PATH=<qpy_dir>:$PATH
source <qpy_dir>/bash_completion.sh
```

Your system administrator should tell you which directory should be used in <qpy_dir> and might also send you a few files to be placed in your q*py* directory, ~/.qpy/. This directory hosts the main configuration files, but in general you do not have to directly access them.

1.1.2 Initialization

The first q*py* command one should run is:

```
$ qpy restart
```

This command will set the background environment for using q*py*. It should also be executed when the version of q*py* is updated (you will be informed by the administrator), or if the master node crashes.

1.1.3 Basic concepts and usage

Every job submitted by q*py* has an ID and a status. The ID, a fixed integer for a particular job, is unique and identifies your job in the pool of (your) jobs. The status can be:

- queue - the job is waiting for allocation;
- running - the was allocated and is running;

- `done` - the job has run and is finished;
- `killed` - the job had its execution killed by the user;
- `undone` - the job has not been executed (it was killed while in the queue).

When you submit a job, it initially has the status `queue`. When a node is allocated for it, it is executed and the status changes to `running`. If the job terminates normally, the status changes to `done`. If you kill the job (that is, stops its execution), the status changes to `killed`. If you kill a job that is still waiting for allocation, it will never be executed and the status changes to `undone`.

You can submit a job using the command `sub`:

```
$ qpy sub <job's execution command>
```

This means that you put, on your queue of jobs, a new job that has to be executed with the command `<job's execution command>`. q*py* will handle it for you from here on!

By the way, all commands in q*py* are like this:

```
$ qpy <cmd> [options]
```

where `<cmd>` is one of the q*py* commands and `[options]` are the otions for that command. Of course, these options depend on the actual command `<cmd>`, some of them are optional, some not. See below for a complete list of commands and the explanation of their options.

Now that your job is on the list, you probably want to be able to:

- check the status of your job (or jobs);
- check the situation of the nodes;
- eventually kill an unsatisfactory job.

This can be done by the following commands:

```
$ qpy check
```

This shows a list of all your jobs. If you are interested only on a portion of these jobs, say, the ones that are currently running, try:

```
$ qpy check running
```

Another useful q*py* command is:

```
$ qpy status
```

It will show all the available nodes, users, how many jobs each one is running, and some extra information. If your jobs are waiting for too long on the queue, this command should provide an indication why.

Sometimes you are not so happy with the development of a job and you want to terminate its execution, without waiting to its normal termination. You can do this with:

```
$ qpy kill <job_ID>
```

where `<job_ID>` is the ID number of the job. Attention! This is not reversible. q*py* will not ask you if you are sure to kill the job, so use the command with care.

There are several commands and options in q*py*. Every command in q*py* can be completed automatically by using the key `<TAB>`. It also gives the possible next arguments which, we hope, will be very helpful. Moreover, if you type:

```
$ qpy kill ? <TAB> <TAB>
```

where `<TAB> <TAB>` indicates that you should type the `<TAB>` key two times, you will receive a brief explanation on the command `kill`.

1.2 Commands

This section presents a list of all the commands and options of q*py*. Remember that they should be used as:

```
$ qpy <cmd> [options]
```

where `[options]` depends on the command `<cmd>`.

1.2.1 `restart`

To start the q*py* background environment. This command also works as an easy way to finish the current session of q*py* and to start a new one, which is most useful when you want to update the q*py* to its latest version. If you are restarting q*py* because the background environment has crashed, you might have to remove the file `master_connection_port` from the q*py* directory before starting a new q*py* session.

- Options: there are no options.
- Examples:

  Restart the q*py* background environment:

```
$ qpy restart
```

1.2.2 `finish`

If you feel like done using q*py*, please finish the existing q*py* environment, which is running in background, by using this command.

- Options: there are no options.
- Examples:

  Finishes the background q*py* environment:

  ```
  $ qpy finish
  ```

1.2.3 `sub`

Used to submit any executable or command. The standard output and standard error of that command will be written in the files `job_<job_ID>.out` and `job_<job_ID>.err` respectively.

By default, q*py* will allocate one core and 5 GB of RAM for such submission. To submit a job that uses more cores (say `<N>` cores) or a different amount of memory (say `<M>` GB), one can add this information in two ways. The first way is using the options `-n <N>` or `-m <M>` after `qpy sub`, but *before* the executable (or the options will be used as options to the command). The other way can be used when submitting a bash script as command. In this case, the following lines can be added in the script, that will be recognized by q*py*:

```
#QPY n_cores=<N>
#QPY mem=<M>
```

- Options: the executable and its arguments; optionally, the the following flags *before* the executable:
  - `-n <N>` the number of cores
  - `-m <M>` requested memory, in GB
- Examples:

  Executes the command `hostname`. Allocates the default one cores and 5 GB of memory for it:

```
$ qpy sub hostname
```

Executes the script `./script.sh` (that must have the permission to be executed!). Allocates three cores and 10 GB of memory for it:

```
$ qpy sub -n 3 -m 10 ./script.sh
```

Executes the command `ls -ltr`:

```
$ qpy sub ls -ltr
```

Using a submit script is highly recommended. q$_{py}$ offers a number of environment variables that can be useful in this context:

- `$QPY_JOB_ID` holds the job number of your run. You can use it to create unique directories for scratch files or for output files
- `$QPY_NODE` is the name of the allocated node (should be equal to `$HOSTNAME`)
- `$QPY_N_CORES` is the number of cores reserved on that node (can be passed to the argument line of the parallel job)
- `$QPY_MEM` is the reserved memory in Gb.

1.2.4 `check`

Checks the status of all the submitted jobs, printing a list of jobs. If no option is given, the command gives the list of all jobs submitted. One can also list just specific jobs, by their status, ID, or directory of submission. This information can be passed by the options, and if multiple options are given the jobs that satisfy any of the required property is listed. For example a status and a directory are given, this command will list all the jobs with that status plus the jobs that were submitted from the directory.

- Options: statuses, job IDs, and/or directories. The job IDs can be passed individually or as a range, such as 100-120.
- Examples:

  Lists the jobs that are currently running:

```
$ qpy check running
```

Lists the jobs that have been submitted from the current directory

```
$ qpy check .
```

Lists the jobs with ID in the range 100-120, plus the jobs with ID in the range 130-135, plus the job 137:

```
$ qpy check 100-120 130-135,137
```

1.2.5 `kill`

Kills a particular job or set of jobs. One can kill several jobs by providing several job IDs (with a range, for instance, like in `check`. A status can be used to kill all jobs with that status (`running` or `queue`). Use `all` to kill all the jobs (be careful!).

- Options: one or more job IDs, a status (`queue` or `running`), or `all`.
- Examples:

  Kills the jobs with ID 100, 101, 102, 103, 104, 105, and 108:

  ```
  $ qpy kill 100-105 108
  ```

  Kills all the jobs with status `queue` (that is, not yet allocated):

  ```
  $ qpy kill queue
  ```

  Kills all jobs:

  ```
  $ qpy kill all
  ```

1.2.6 `clean`

Cleans the list of jobs that q$_{py}$ has currently in memory. One can only clean jobs with status `done`, `killed`, or `undone`. Similarly to the `kill` command, the user can clean some specific jobs by the ID, all the jobs with some status, or all jobs. One can also clean by the directory of submission. It has no effect

- Options: one or more job IDs, a status (`done`, `killed`, or `undone`), `all`, or a directory.
- Examples:

  Cleans the jobs with ID 100 to 200:

```
$ qpy clean 100-200
```

Cleans all the jobs with status `undone` (that ones that have been killed before running):

```
$ qpy clean undone
```

Cleans all the jobs that can be cleaned:

```
$ qpy clean all
```

1.2.7 `status`

This command is mainly useful for checking the multi-user environment. It writes the number of jobs running for each users, and in each of the available nodes.

- Options: There are no options.
- Examples:

```
$ qpy status
```

1.2.8 `config`

This command, if run without option, gives information about the current settings of q*py*. Feel free to check, it is quite self explanatory. These settings can be changed by the same command, giving as option a keyword and a value (or soemtimes a sequence of values):

```
$ qpy config <keyword> <value> [<value2> ...]
```

In the following is a list of possible keywords and what they mean.

- `colour true|false`

  The output of check is coloured (`true`) or not (`false`). Coloured output might not work for some terminals.

- `coloursScheme <c queue> <c running> <c done> <c killed> <c undone>`

  Sets the colours to be used in a coloured output of check. Give always five colours, following the order above, that can be: `grey`, `red`, `green`, `yellow`, `blue`, `magenta`, `cyan`, and `white`.

- `checkFMT '<pattern>'`

  When the command `check` is used, the jobs are printed in a specific pattern. One can change this with this keyword, where `<pattern>` is a string (the single quotes above are important!') that may contain one of the "per cent" modifiers below, that will be replaced by the appropriate content of the job.

  - %j : job ID
  - %s : job status
  - %c : command you used to submit the job
  - %d : working directory of your job
  - %n : node allocated for your job
  - %N : number of cores for your job.
  - %R : running time of the job. (time in queue if the job is in queue)
  - %Q : actual time when the job is submitted
  - %S : actual time when the job has started
  - %E : actual time when the job has finished

  It might seem a little complicated, but an example should clarify this. The default value is:
  `%j (%s):%c (on %n; wd:  %d)\n`
  Let us suppose that one has only one job, running a command `bash my_script.sh`, with ID 100, submitted from `/home/user/`, and it is currently running on the node `comp1`.

```
$ qpy check
100 (running):bash my_script.sh (on comp1; wd: /home/users/)
$ qpy config checkFMT '%j: %s\n'
Check pattern modified to '%j: %s\n'
$ qpy check
100: running
$ qpy config checkFMT '%j (%s)\n\tSubmitted from %d\n\tStarted at %S\n-----\n'
Check pattern modified to '%j (%s)\n\tSubmitted from %d\n\tStarted at %S\n-----\n'
$ qpy check
100 (running)
        Submitted from /home/users/
        Started at 2018-10-04 18:01:45.948201
```

```
-----
$ qpy config checkFMT default
Check pattern restored to the default value: '%j (%s):%c (on %n; wd: %d)\n'.
$ qpy check
100 (running):bash my_script.sh (on comp1; wd: /home/users/)
```

The `\n` is important, and it means "new line". Without it, all jobs are printed in the same line. One can also use `\t` to insert a tabulation, as the example shows.

- `copyScripts <true|false>`

  Bash scripts, generally, stop if changes are made to the script file while they still are running. Hence, it is sometimes advantageous to have q*py* running a copy of the script, which allows to change and reuse the original bash script. If this keyword is set to `true`, qpy will copy the script to the local folder `~/.qpy/scripts` and use this copy when running the actual calculation. To enable this option, use `qpy config copyScripts true`. By default `copyScripts` is set to `false`.

  The same thing can also be done for a specific job if the submitted script has the line `#QPY cpScript true` or if the job is submitted using the command `qpy sub -c [script_name]`. If the `copyScripts` is set True, then the reverse can be done by adding the line `#QPY cpScript false` in the script or by submitting the script using the command `qpy sub -o [script_name]`.

- Options: a keyword and a value (or values) for this keyword.
- Examples: See above for examples for each keyword.

1.2.9 `ctrlQueue`

This command controls and move jobs in the queue. First the options `pause` and `continue` can be used to pause the submission of jobs from the queue to running, or to continue. Also, it can be used to move jobs within the queue (`jump`), and reorganize the order of submission.

- Options: `pause`, `jump`, `continue`
- Examples:

  Consider the situation where you have a long list of jobs in the queue but you have some job(s) of top priority that needs to be done as soon as possible. The command ctrlQueue can be used in this situation to move the priority job(s) up in the queue. These are what you have to follow: The queue must first be paused, then "jump" the job `<job ID>` to `<target>`:

```
qpy ctrlQueue pause
qpy ctrlQueue jump <job ID> <target>
qpy ctlrQueue continue
```

The `<target>` can be a job ID, `begin`, or `end`.

1.2.10 `tutorial`

Shows a tutorial, that is basically a text version of the "For users" section of this manual.

- Options: Optionally, a keyword can be given to be searched in the tutorial
- Examples:

  Opens the tutorial and goes to the first occurrence of `ctrlQueue`:

  ```
    qpy tutorial ctrlQueue
  ```

2 For administrators

To work as administrator of q*py*, it is useful to know a few facts about how q*py* works. q*py* has three levels:

1. the `qpy`
2. the `qpy-master`
3. the `qpy-multiuser`

The first, `qpy`, takes care of the interaction of the user with q*py*, and its usage is described in the Section ?? of this manual. The second is the background environment of q*py* *of each user*, and controls the user's jobs. The third is the global (or multi-user, as we prefer to say) background environment of q*py*, that decides when and where a job can run, by allocating cores and memory to each `qpy-master`.

All this said, the duties of the administrator of q*py* are:

- Maintain the program `qpy-multiuser` running;
- Add/remove new users and machines;
- Assist the users with any problems;
- Update q*py* to newer versions.

This section explains how this is done. First of all, it is a good idea to have a user dedicated to the administration of q*py*, what keeps everything more organized. We of course strongly discourage using the root for this task, for security reason. The administrator of q*py* does not need any special permission.

2.1 Installation

We assume that you have a copy of q*py*, otherwise you wouldn't be reading this manual. To set up a new q*py* environment, the following steps must be taken:

- Make sure that every user that will use q*py* has permission to read the directory where q*py* is installed. This directory will be (and has been, in Section ??) called `<qpy_dir>`.
- Create the configuration's directory in the administrator's home directory:

```
$ mkdir ~/.qpy-multiuser/
```

- Create the following files in this directory:

```
$ vi ~/.qpy-multiuser/distribution_rules
$ vi ~/.qpy-multiuser/allowed_users
$ vi ~/.qpy-multiuser/nodes
$ vi ~/.qpy-multiuser/multiuser_connection_address
```

  The content of these files are described in section ??. Please, read such section carefully and create these files according to your needs before proceeding.

- Finally, start the multiuser environment of q$_{py}$:

```
$ python <qpy_dir>/qpy-access-multiuser.py start
```

All the interaction of the administrator with q$_{py}$ is made with the following command, plus some options:

```
$ python <qpy_dir>/qpy-access-multiuser.py
```

It is a good idea to define an alias for this by, for example, adding the following in the q$_{py}$ administrator `~/.bashrc`:

```
 alias qpy-admin='<qpy_dir>/qpy-access-multiuser.py'
```

2.2 Add new user

Let us suppose that a user with username `<user>` will start using q$_{py}$. As q$_{py}$ usually runs across several machines, first make sure that `<user>` has the ssh keys properly configured, that is, that the user is be able to connect from the master node (where q$_{py}$ is running) to the slave nodes without password. Because the home directory of the master node and the slave nodes must have the same home directory for a correct q$_{py}$ environment, setting the ssh keys just once should be enough. To allow for the new user in q$_{py}$, add the username in the file `~/.qpy-multiuser/allowed_users`, one user in each line:

```
$ cat ~/.qpy-multiuser/allowed_users
<user 1>
<user 2>
```

```
...
<user>
```

After, you have to send the following files to the new user, to be put in his/her qpy directory. Something like this:

```
$ cp ~/.qpy-multiuser/multiuser_connection_address /home/<user>/.qpy/
$ cp ~/.qpy-multiuser/multiuser_connection_port /home/<user>/.qpy/
$ cp ~/.qpy-multiuser/multiuser_connection_conn_key /home/<user>/.qpy/
```

In addition, if you have a dedicated machine to run qpy, copy also the following file file to the user's qpy directory:

```
$ cat ~/.qpy-multiuser/master_connection_address
<hostname>
$ cp ~/.qpy-multiuser/master_connection_address /home/<user>/.qpy/
```

If this file is not present in the user's qpy directory, qpy runs locally.

2.3 Add and remove a node

To add or remove a node, edit the file `~/.qpy-multiuser/nodes` (see Section **??**), and run the following commands:

```
$ python <qpy_dir>/qpy-access-multiuser.py nodes
$ python <qpy_dir>/qpy-access-multiuser.py distribute
```

2.4 Update version

When a new version of qpy is released, depending where the new changes have been done, you might have to restart the multiuser background environment:

```
$ python <qpy_dir>/qpy-access-multiuser.py finish
$ python <qpy_dir>/qpy-access-multiuser.py start
```

And/or ask all the users to restart their own masters, with:

```
$ qpy restart
```

Whatever is the case, this will be informed in the release's notes.

2.5 Commands

This is a full list of the commands that are available to the administrator:

- `start`

  Starts a new qpy-multiuser instance. Run this if you are starting $\mathtt{q}_{py}$ for the first time, if a update is needed or if the machine where $\mathtt{q}_{py}$ runs has crashed.

  ```
  $ python <qpy_dir>/qpy-access-multiuser.py start
  ```

- `finish`

  Finishes the multiuser background environment.

  ```
  $ python <qpy_dir>/qpy-access-multiuser.py finish
  ```

- `nodes`

  Loads the content of the file ~`/.qpy-multiuser/nodes`.

  ```
  $ python <qpy_dir>/qpy-access-multiuser.py nodes
  ```

- `distribute`

  Distributes the cores among the users.

  ```
  $ python <qpy_dir>/qpy-access-multiuser.py distribute
  ```

- `variables`

  Lists several internal variables of qpy-multiuser. Used mainly for debugging.

  ```
  $ python <qpy_dir>/qpy-access-multiuser.py variables
  ```

- `status`

  Show the current status of the users, nodes and cores. It is the same as the command `status` accessible to the users.

  ```
  $ python <qpy_dir>/qpy-access-multiuser.py status
  ```

- `saveMessages`

  Saves messages from the internals of qpy-multiuser, that will be shown in the "variables" command. Mainly for debugging.

```
$ python <qpy_dir>/qpy-access-multiuser.py saveMessages
```

There are some “cheating” commands that the administrator can run, such as artificially adding or removing running jobs of users. These commands are not supposed to be used in a normal run, but only if something went wrong and must be manually fixed. **Use only if you know exactly what you are doing!!**

- `__user`

  Adds a new user to qpy-multiuser. *In a normal run, it is automatically done when the user restart his/her* $\mathtt{q}_{py}$*.*

  ```
  $ python <qpy_dir>/qpy-access-multiuser.py __user <user_name> \
    <address> <port> <conn_key>
  ```

- `__req_core`

  Asks for a slot (cores plus memory) to run a job. *In a normal run, it is done by the user's master whenever there is a job in his/her queue.*

  ```
  $ python <qpy_dir>/qpy-access-multiuser.py __req_core <user_name> \
    <jobID> <n_cores> <mem> <queue_size>
  ```

- `__remove_job`

  Tells the multiuser background environment that a job has finished. *Ina a normal run, it is done by the user's master whenever the job has finished.*

  ```
  $ python <qpy_dir>/qpy-access-multiuser.py __remove_job <user_name> \
    <job_ID> <queue_size>
  ```

2.6 Files

This is a list of the files in the qpy-multiuser directory. Some of them the administrator should edit to control the behavior of $\mathtt{q}_{py}$, some others not.

- `distribution_rules`

  This file defines how the cores are distributed to the users. The basic syntax is one of the following:

```
even minimum <n_cores>
```

This means an even distribution among the users, with at least `<n_cores>` granted for each.

- `allowed_users`

  A list with all the users that can use the q*py* environment:

```
<user_1>
<user_2>
<user_3>
```

- `nodes`

  A list with all the nodes available in the q*py* environment. Each line has the information of one node, as shown below:

```
<node_1> <n_cores> [M]
<node_2> <n_cores> [M]
<node_3> <n_cores> [M]
```

  First is the hostname of the node, followed by the number of cores this node has (or is available to q*py*) and, optionally, a "M", to indicate that that node has preference for multicore jobs.

- `multiuser_connection_address`

  This file simply contains the address where the `qpy-multiuser` instance will run. If you do not set this, q*py* automatically uses the local machine (`localhost`).

```
<hostname for qpy-multiuser>
```

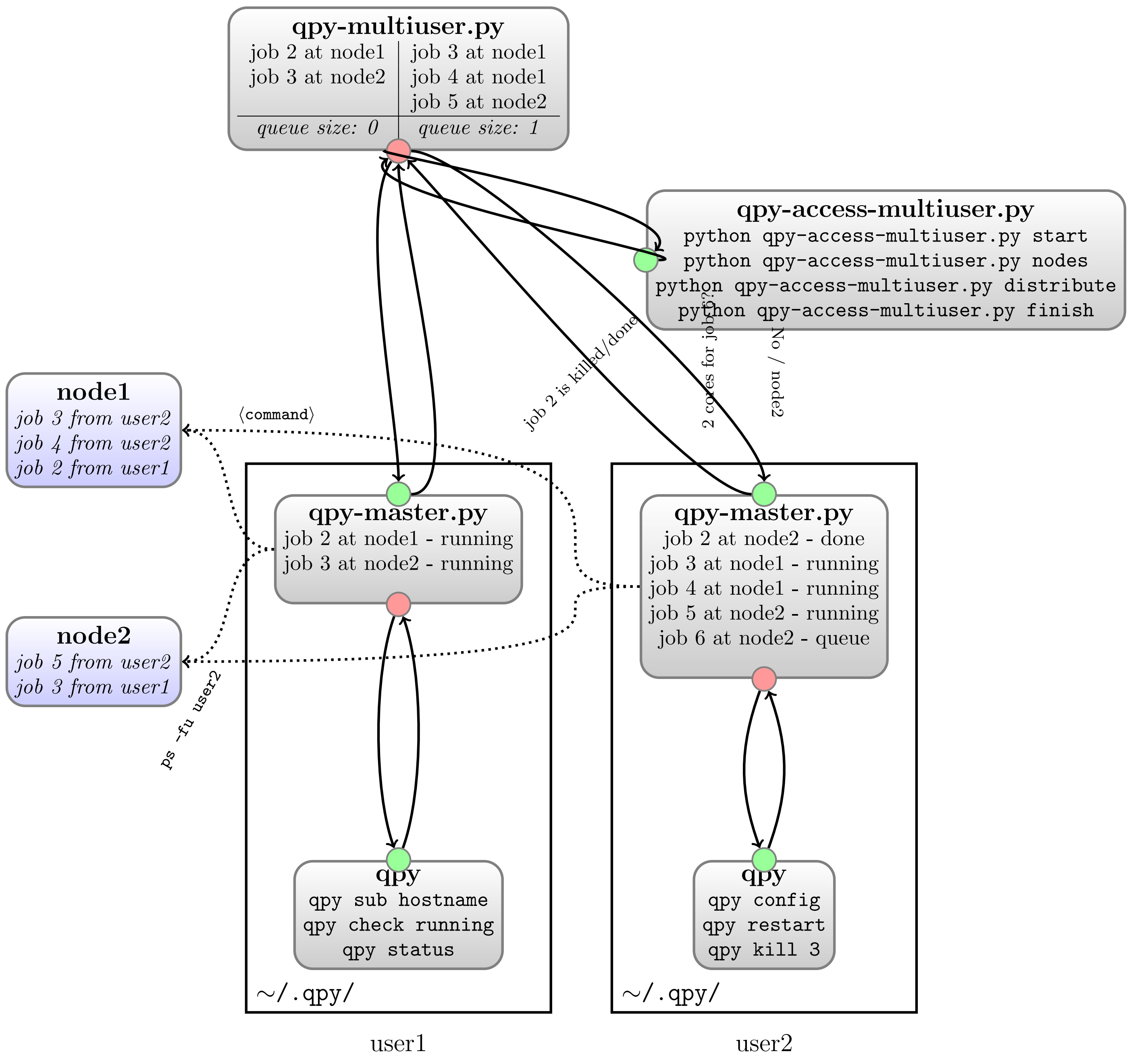
  Optionally, for the sake of organization mainly, the programs `qpy-multiuser` and the `qpy-master` of each user can run on a machine different than the node where the users work and submit their jobs from. For instance, you might have a machine (even a virtual machine) dedicated to q*py*. To do this, `<hostname for qpy-multiuser>` must be the hostname of this machine In this case, the files `master_connection_address` that the users have in their q*py* directory should have the correct hostname.

The following files are generated and used by q*py*, but the administrator should not alter or delete them. Moreover, these files should not be shared, because they have the information required to make all the message transfers. q$_{py}$ takes care of the permission of the directories `~/.qpy-multiuser/` and `~/.qpy/` of each user (whose content should also not be shared).

- `~/.qpy-multiuser/multiuser_connection_port`
- `~/.qpy-multiuser/multiuser_connection_key`
- `~/.qpy-multiuser/connection_<user>_key`
- `~/.qpy-multiuser/connection_<user>_key`
- `~/.qpy-multiuser/connection_<user>_key`

3 For developers

3.1 Program design

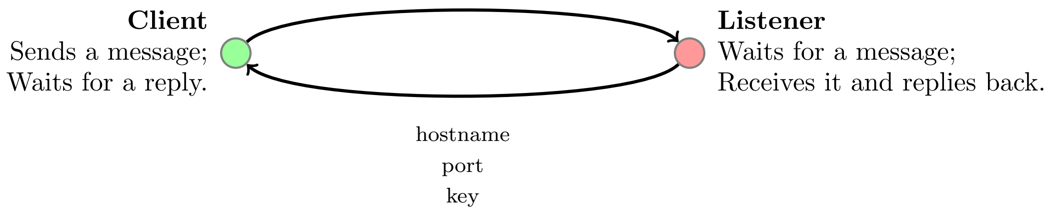


```
python qpy-access-multiuser.py start
python qpy-access-multiuser.py nodes
python qpy-access-multiuser.py distribute
python qpy-access-multiuser.py finish
```

```
qpy sub hostname
qpy check running
qpy status
```

```
qpy config
qpy restart
qpy kill 3
```

Python's multiprocessing: Listener/Client connection



3.2 Testing

This section describes the procedure for (locally) testing a modified code. A full protocol for release of a code for production run is not yet developed.

3.3 Single-User Testing

For testing the changes as a single user, without interfering with your currently running q*py* version, do the following:

It is assumed that you have checked out a local copy of the q*py* code from the git repository. We assume the code is in the directory `${TEST_QPY_DIR}`

Now, issue the following commands:

```
$ touch ${TEST_QPY_DIR}/test_dir
$ mkdir ~/.qpy-multiuser-test/
```

and create the files

`~/.qpy-multiuser-test/distribution_rules`

`~/.qpy-multiuser-test/allowed_users`

`~/.qpy-multiuser-test/nodes`

with the contents as described in the administration section, see section **??** and related. The file `test_dir` will tell the program to run in test mode and to access the directories `~/.qpy-multiuser-test` (and `~/.qpy-test`) instead of the standard directories without the `-test` extension.

In `allowed_users` it is sufficient to only add yourself, the `nodes` directory may be copied from the running version (and modified, if required).

Now, you can start the local multiuser kernel by

```
$ python ${TEST_QPY_DIR}/qpy-access-multiuser.py start
```

Check the log file `~/.qpy-multiuser-test/multiuser.log` for problems.

After that, you create your local test user directory

```
$ mkdir ~/.qpy-test/
```

and copy the files

`~/.qpy-multiuser-test/multiuser_connection_address`

`~/.qpy-multiuser-test/multiuser_connection_port`

`~/.qpy-multiuser-test/multiuser_connection_conn_key` to ~/.qpy-test/.

Finally start also the local master kernel by

```
$ ${TEST_QPY_DIR}/qpy restart
```

Errors can be found in ~/.qpy-test/master.log.