# GXP3 User's Guide

January 2007

**Kenjiro Taura**

**University of Tokyo**
**7-3-1 Hongo Bunkyo-ku Tokyo, 113-0033 Japan**

# 1 Getting Started

## 1.1 Prerequisites

In order to play with GXP, you need

- a Unix platform
- Python interpreter

To do anything interesting with GXP, you should be able to remote login (e.g., via SSH, RSH, QSUB, etc.) computers you want to use.

GXP has been developed on Linux and tested on Linux and FreeBSD. It has been tested with Python 1.5.2, 2.2.2, 2.3.4, and 2.4.2. There are high chances that it runs with later versions of Python.

Python is a very popular programming language that is installed on most Linux platforms. For remote logins, it's very likely that it accepts SSH logins. Thus there are chances that the above prerequisits are already satisfied in your platform.

## 1.2 Installation

You can obtain the latest version (tarball) from sourceforge:

```
http://sourceforge.net/projects/gxp/
```

and then unpack the tarball by something like:

```
tar zxvf gxp3.xx.tar.bz2
```

The exact file name (and the directory name) depends on the current version of GXP. Perhaps you want to make a symlink `gxp3` to the directory you obtained, or directly rename the directory to `gxp3`.

If you have cvs, you can use it to obtain the latest copy under development.

```
cvs -d :pserver:anonymous@gxp.cvs.sourceforge.net:/cvsroot/gxp co gxp3
```

Either way, you will have directory called `gxp3`. Since GXP and accompanying tools are all written in Python, you need no compilation. Instead, all you need to do is either to:

1. make a symlink from anywhere in your path to `gxp3/gxpc`. For example,

   ```
   ln -s /absolute/path/to/gxp3/gxpc /usr/local/bin/gxpc
   ```

   or,
2. add `gxp3` directory to your PATH environment variable. For example, add the following line in your `.bashrc`.

   ```
   export PATH=$PATH:/absolute/path/to/gxp3
   ```

In the former, it is important to make a symlink to, not a copy of, `gxpc`. Otherwise it fails to find accompanying files under `gxp3` directory.

To test your installation, type `gxpc` to your shell prompt and see something like this.

```
$ gxpc
gxpc: no daemon found, create one
/tmp/gxp-you/gxpsession-hongo019-you-2007-07-03-13-56-21-3561-19041263
```

See Chapter 6 [Troubleshooting], page 37, if you fail.

# 2 Tutorial

## 2.1 Testing Your Installation

The most basic function of GXP is to run commands on many hosts. To this end, you first need to learn how to run commands and how to acquire those hosts.

GXP is run from a regular shell prompt. For those who have used prior versions of GXP, this has been changed in this version.

## 2.2 Running commands via e command

Given you put gxpc in your PATH in one way or another, type the following to your shell prompt.

```
$ gxpc e whoami
```

In this manual, `$` means a shell prompt, and is not a part of the input.

You will see something like the following.

```
$ gxpc e whoami
gxpc: no daemon found, create one
tau
$
```

The `gxpc e` command lets GXP run whatever follows it, which is, in this example, `whoami`. As you can see in the first line, a GXP "daemon" is brought up, which takes the request to run `whoami`. GXP daemon stays running background. Thus, if you issue another gxpc `e` command again, you will see result immediately this time.

```
$ gxpc e whoami
tau
$ gxpc e uname
Linux
$
```

The daemon stays running even if you exit the shell or even logout. If you exit or logout and then run your shell or login again, the daemon should be still running.

`gxpc quit` is the command to terminate the daemon.

```
$ gxpc quit
$
```

Issueing gxpc `e` command again, the daemon will bring up again, of course.

```
$ gxpc e uname
gxpc: no daemon found, create one
Linux
```

A primitive way to check if the daemon is running is `ps` command. For example,

```
$ ps w
  PID TTY      STAT   TIME COMMAND
 9273 pts/1    S      0:00 /bin/bash --noediting -i
 9446 pts/1    S      0:00 python /home/tau/proj/gxp3/gxpd.py --no_stdin -
-redirect_stdout --redirect_stderr
```

```
 9486 pts/1    R      0:00 ps w
```

`gxpd.py` is the daemon running for you.

Or you can run `gxpc prompt` command. If no daemon is running, it does not print anything. If a daemon is running, it prints a short string indicating its status. For example,

```
$ gxpc prompt
[1/1/1]
```

We will detail the meaning of these three numbers ([1/1/1]) later. For now, remember `gxpc prompt` command as a way to check if the daemon is running.

In general, `e` command will run any shell command on *all hosts selected for execution.* In this example, you only have one host, which is the local host you issued `gxpc` command on, and that node is selected. See Section 2.3 [Getting more hosts with use/explore commands], page 3, for how to have more hosts and how to select a set of hosts for execution.

## 2.3 Getting more hosts with use/explore commands

With GXP, you will probably want to use many hosts in parallel. You need to have a host you can remote-login. In this example, we assume you are originally on host `hongo000` and you have another host, `hongo001`, which you can remote-login from `hongo000` via SSH. Furthermore, we assume you can do so without typing password every time you login `hongo001`. See Section 2.4 [Introduction to SSH for GXP Users], page 7, if you are not familiar with necessary SSH setup to do so. For those who already know it, you need to have public/private key pairs and either (1) use `ssh-agent` and `ssh-add` (`eval ‘ssh-agent‘ && ssh-add`) and input the phassphrase to `ssh-add`, so SSH does not ask you it again (recommended), or (2) set the encryption passphrase of the private key empty. Though the method (2) uses secure public-key protocol for authentication, when your account on the host storing the private key is compromised, the intruder may be able to access other hosts via the private key.

Either way, you should have an environment where something like the following silently succeeds without your password/passphrase being asked by SSH client.

```
$ ssh hongo001 hostname
hongo001
$
```

You may use remote-execution commands other than SSH, including RSH, QRSH, and QSUB as well as the local shell (sh) to spawn multiple daemons on a single host. You can customize exact command lines used or even add your own custom rsh-like command in the repertoire. ??? Explore Settings for Various Environment, for details.

Once you have such an environment, try the following to acquire hongo001.

```
$ gxpc use ssh hongo000 hongo001
$ gxpc explore hongo001
reached : hongo001
$
```

The first line reads “`gxpc` [can] `use ssh` [from] `hongo000` [to login] `hongo001`.” The second line instructs GXP to go ahead and really login `hongo001`. If `hongo000` and `hongo001` share your home directory, this command will not take much longer than the regular SSH.

After the explore command has finished, issue an `e` command again. This time, you will see the command being executed on the two hosts.

```
$ gxpc e hostname
hongo000
hongo001
$
```

`gxpc stat` command shows hosts (daemons) connected.

```
$ gxpc e hostname
hongo000
hongo001
$
```

If you want to grab a third host, say `hongo002`, try the following.

```
$ gxpc use ssh hongo000 hongo002
$ gxpc explore hongo002
reached : hongo002
$ gxpc e hostname
hongo000
hongo001
hongo002
```

You could continue this way, issueing `use` command and then `explore` command to get a single host at a time, but this is obviously not so comfortable if you have hundred hosts, say `hongo000` – `hongo099`. The first trick to learn is that, both the second and the third arguments to `use` command are actually *regular expressions* of hostnames. Therefore, the following single line:

```
$ gxpc use ssh hongo000 hongo0
```

says that "`gxpc` [can] `use ssh` [from any host matching] `hongo000` [to login any host matching] `hongo0`." Note that any host that begins with `hongo0` matches the regular expression `hongo0`.

Second, explore command can take multiple hostnames in a single command. For example, after the above command, you may grab three hosts with a single stroke by:

```
$ gxpc explore hongo003 hongo004 hongo005
reached : hongo003
reached : hongo004
reached : hongo005
$
```

By default, GXP won't grab the same host multiple times, so issuing the following once again will have no effect.

```
$ gxpc explore hongo001
$
```

If you have many hosts, this is still painful. An even better way is to use a special notation `[[`*xxx*`-`*yyy*`]]`, which represents a set of numbers between *xxx* and *yyy* (inclusive). So,

```
$ gxpc explore hongo[[000-014]]
$
```

is equivalent to

```
$ gxpc explore hongo000 hongo001 hongo002 ... hongo014
$
```

If you wish to exclude some hosts from the range, use a notation `;`*nnn* or `;`*nnn*-*mmm*. For example,

```
$ gxpc explore hongo[[000-014;007]]
$
```

will explore hongo000 ... hongo014 except for hongo007.

```
$ gxpc explore hongo[[000-014;003-006]]
$
```

will explore hongo000 ... hongo014 except for hongo003-hongo006.

Since GXP gracefully handles (ignores) dead or non-existent hosts, you normally do not have to exclude every single non-working hosts this way, but doing so is sometimes useful to make explore faster.

Instead of typing hostnames in a command line, you may have a file that lists targets and give it to explore command by `--targetfile` or the shorter `-t` option. Let's say you have a file like the following.

```
$ cat my_targets
hongo[[000-014]]
kashiwa[[000-009]]
edo[[000-012]]
```

Then

```
$ gxpc explore -t my_targets
```

is equivalent to

```
$ gxpc explore hongo[[000-014]] kashiwa[[000-009]] edo[[000-012]]
```

Now you reached 15 hosts in total. The `stat` command shows hosts (daemons) connected.

```
$ gxpc stat
/tmp/gxp-tau/gxpsession-hongo000-tau-2007-01-27-23-15-04-29004-97439271
hongo000 (= hongo000-tau-2007-01-27-23-15-04-29004)
 hongo007 (= hongo007-tau-2007-01-27-23-16-30-24842)
 hongo001 (= hongo001-tau-2007-01-27-23-16-27-1205)
 hongo013 (= hongo013-tau-2007-01-27-23-48-28-20179)
 hongo014 (= hongo014-tau-2007-01-27-23-48-14-11435)
 hongo005 (= hongo005-tau-2007-01-27-23-16-30-9412)
 hongo006 (= hongo006-tau-2007-01-27-23-16-28-24083)
 hongo003 (= hongo003-tau-2007-01-27-23-16-29-16040)
 hongo002 (= hongo002-tau-2007-01-27-23-16-28-22077)
 hongo008 (= hongo008-tau-2007-01-27-23-48-12-4070)
 hongo012 (= hongo012-tau-2007-01-27-23-48-14-32519)
 hongo010 (= hongo010-tau-2007-01-27-23-48-14-12066)
 hongo011 (= hongo011-tau-2007-01-27-23-48-15-15126)
 hongo009 (= hongo009-tau-2007-01-27-23-48-13-21582)
 hongo004 (= hongo004-tau-2007-01-27-23-16-31-8776)
$
```

The first line shows the name of *a session file*, which is explained later (??). Below hongo000, hongo001–hongo014 are indented by a single space, which means hongo000 issued logins to all these hosts. This happened because we have previously said

```
$ gxpc use ssh hongo000 hongo0
```

indicating that (only) hongo000 can login hongo0.

You can alternatively say

```
$ gxpc use ssh hongo0 hongo0
```

indicating gxpc can use ssh from any host matching hongo0 to any host matching hongo0. This can be abbreviated to:

```
$ gxpc use ssh hongo0
```

In this case, gxpc will try to reach these hosts in a more load-balanced fashion. To see this let's quit and try it again from the beginning.

```
$ gxpc quit
$ gxpc use ssh hongo0
gxpc: no daemon found, create one
$ gxpc explore -h /etc/hosts hongo00 hongo01[0-4]
reached : hongo003
reached : hongo008
reached : hongo001
reached : hongo004
reached : hongo002
reached : hongo005
reached : hongo007
reached : hongo006
reached : hongo009
reached : hongo013
reached : hongo010
reached : hongo011
reached : hongo012
reached : hongo014
$ gxpc stat
/tmp/gxp-tau/gxpsession-hongo000-tau-2007-01-28-00-10-51-311-66768183
hongo000 (= hongo000-tau-2007-01-28-00-10-51-311)
 hongo006 (= hongo006-tau-2007-01-28-00-10-33-29696)
 hongo005 (= hongo005-tau-2007-01-28-00-10-35-13470)
 hongo003 (= hongo003-tau-2007-01-28-00-10-34-16875)
 hongo002 (= hongo002-tau-2007-01-28-00-10-33-27494)
 hongo004 (= hongo004-tau-2007-01-28-00-10-36-9479)
 hongo009 (= hongo009-tau-2007-01-28-00-10-33-21614)
 hongo008 (= hongo008-tau-2007-01-28-00-10-31-4102)
  hongo011 (= hongo011-tau-2007-01-28-00-10-34-15158)
  hongo012 (= hongo012-tau-2007-01-28-00-10-33-32551)
  hongo010 (= hongo010-tau-2007-01-28-00-10-33-12098)
```

```
      hongo013 (= hongo015-tau-2007-01-28-00-10-34-20211)
      hongo014 (= hongo014-tau-2007-01-28-00-10-34-11467)
     hongo007 (= hongo007-tau-2007-01-28-00-10-35-26002)
     hongo001 (= hongo001-tau-2007-01-28-00-10-32-1979)
```

The indentations indicate that `hongo008` issued logins to `hongo010`–`hongo014`. `hongo000` issued logins to `hongo001` –`hongo009`. In general, GXP daemons will form a tree. By default, a single node tries to keep the number of its children no more than nine.

## 2.4 Introduction to SSH for GXP Users

todo

## 2.5 Quick Reference of Frequently Used Commands

-
  ```
  gxpc prompt
  ```

  will show a succinct summary of gxp status. It is strongly recommended to put 'gxpc prompt 2> /dev/null' (note for the backquotes, not regular quotes) as part of your shell prompt. For example, if you are a bash user, put the following into your .bashrc.

  ```
  export PS1='\h:\W`gxpc prompt 2> /dev/null`% '
  ```

  Then you will always see the succinct summary in your shell prompt. e.g.,

  ```
  $ export PS1='\h:\W`gxpc prompt 2> /dev/null`% '
  $ gxpc
  hongo:tmp[1/1/1]%
  ```

  '2> /dev/null' is to make gxpc silently exit if should be there any error.

-
  ```
  gxpc use rsh-name src target
  ```

  tells gxp it can login from *src* to *target* via *rsh-name*.

-
  ```
  gxpc explore target ...
  ```

  attempts to login specified *target*'s by methods specified by `use` commands. There is a convenient notation to represent a set of targets. e.g.,

  ```
  $ gxpc explore hongo[[000-012]]
  ```

  is equivalent to

  ```
  $ gxpc explore hongo000 hongo001 hongo002 ... hongo012
  ```

-
  ```
  gxpc e whatever
  ```

  will run a shell command *whatever* on all selected hosts. e.g.,

-
  ```
  gxpc cd [directory]
  ```

  will change the current directory of all selected hosts.

-

```
gxpc export VAR=VAL
```

will set the environment variable for seqsequent commands.

-
  ```
  gxpc smask
  ```

  will select the hosts on which the last command succeeded for the execution targets of subsequent commands.

-
  ```
  gxpc rmask
  ```

  will reset the selected hosts to all hosts.

-
  ```
  gxpc savemask name
  ```

  is similar to smask, but it remembers the set of selected hosts for future reference (by `-m` option of `gxpc e` commands).

-
  ```
  gxpc e -m name whatever
  ```

  is similar to `gxpc e` *whatever*, but runs *whatever* on hosts that have been set by *name* by `gxpc savemask` *name*.

-
  ```
  gxpc -m name cd [directory]
  gxpc -m name export VAR=VAL
  gxpc -m name e whatever
  ```

  `-m` can actually be written immediately after `gxpc`.

-
  ```
  gxpc e -h hostname whatever
  ```

  is similar to `gxpc e` *whatever*, but runs *whatever* on hosts whose names match regular expression *hostname*.

-
  ```
  gxpc e -H hostname whatever
  ```

  is similar to `gxpc e` *whatever*, but runs *whatever* on hosts whose names do not match regular expression *hostname*.

-
  ```
  gxpc -h name cd [directory]
  gxpc -h name export VAR=VAL
  gxpc -h name e whatever
  gxpc -H name cd [directory]
  gxpc -H name export VAR=VAL
  gxpc -H name e whatever
  ```

  Both `-h` and `-H` can actually be written immediately after `gxpc`.

-
  ```
  alias e='gxpc e'
  alias smask='gxpc smask'
  ```

```
    ...
```

Finally, it is recommended to put aliases to some frequently used commands into your shell startup files, so that you do not need to type 'gxpc' everytime.

# 3 Using GXP for Parallel Processing

todo

# 4 Command Reference

## 4.1 cd

**Usage:**

```
gxpc cd [OPTIONS ...] DIRECTORY
```

**Description:**

Set current directory of the selected nodes to *DIRECTORY*. Subsequent commands will start at the specified directory. Options are the same as those of ‘`e`’ command.

## 4.2 e, mw

**Usage:**

```
gxpc e  [OPTION ...] CMD
gxpc mw [OPTION ...] CMD
gxpc ep [OPTION ...] FILE
```

**Description:**

Execute the command on the selected nodes.

**Options:**

(for `mw` only):

`--master 'command'`
: equivalent to `e --updown` '3:4:command' ... if `--master` is not given, it is equivalent to `e --updown` 3:4 ...

**Options:**

(for `e`, `mw`, and `ep`):

`--withmask,-m` *MASK*
: execute on a set of nodes saved by `savemask` or `pushmask`

`--withhostmask,-h` *HOSTMASK*
: execute on a set of nodes whose hostnames match regexp *HOSTMASK*

`--withhostnegmask,-H` *HOSTMASK*
: execute on a set of nodes whose hostnames do not match regexp *HOSTMASK*

`--withgupidmask,-g` *HOSTMASK*
: execute on a set of nodes whose gupid (shown by `gxpc stat`) match regexp *HOSTMASK*

`--withgupidnegmask,-G` *HOSTMASK*
: execute on a set of nodes whose gupid (shown by `gxpc stat`) do not match regexp *HOSTMASK*

`--withtargetmask,-t` *HOSTMASK*
: execute on a set of nodes whose target name (shown by `gxpc stat`) match regexp *HOSTMASK*

`--withtargetnegmask,-T` *HOSTMASK*
: execute on a set of nodes whose target name (shown by `gxpc stat`) do not match regexp *HOSTMASK*

`--up FD0[:`*FD1*`]`
: collect output from *FD0* of *CMD*, and output them to *FD1* of `gxpc`. if :*FD1* is omitted, it is treated as if *FD1* == *FD0*

`--down FD0[:`*FD1*`]`
: broadcast input to *FD0* of `gxpc` to *FD1* of *CMD*. if :*FD1* is omitted, it is treated as if *FD1* == *FD0*

`--updown` *FD1*`:FD2[:`*MASTER*`]`
: if :*MASTER* is omitted, collect output from *FD1* of *CMD*, and broadcast them to *FD2* of *CMD*. if :*MASTER* is given, run *MASTER* on the local host, collect output from *FD1* of *CMD*, feed them to stdin of the *MASTER*. broadcast stdout of the *MASTER* to *FD1* of *CMD*.

`--pty`
: assign pseudo tty for stdin/stdout/stderr of *CMD*

`--rlimit rlimit_xxx:soft[:hard]`
: apply setrlimit(rlimit_xxx, soft, hard)

By default,

- stdin of `gxpc` are broadcast to stdin of *CMD*
- stdout of *CMD* are output to stdout of `gxpc`
- stderr of *CMD* are output to stderr of `gxpc`

This is as if '`--down` 0 `--up` 1 `--up` 2' are specified. In this case, stdout/stderr are block-buffered by default. You may need to do setbuf in your program or flush stdout/err, to display *CMD*'s output without delay. `--pty` overwrites this and turn them to line-buffered (by default). both stdout/err of *CMD* now goto stdout of `gxpc` (they are merged). *CMD*'s stdout/err should appear as soon as they are newlined.

**See Also:**

```
smask savemask pushmask rmask restoremask popmask
```

## 4.3 ep

**Usage:**

```
gxpc e  [OPTION ...] CMD
gxpc mw [OPTION ...] CMD
gxpc ep [OPTION ...] FILE
```

**Description:**
Execute the command on the selected nodes.

**Options:**

(for `mw` only):

`--master 'command'`
: equivalent to `e --updown` '3:4:command' ... if `--master` is not given, it is equivalent to `e --updown` 3:4 ...

**Options:**

(for `e`, `mw`, and `ep`):

`--withmask,-m` *MASK*
: execute on a set of nodes saved by `savemask` or `pushmask`

`--withhostmask,-h` *HOSTMASK*
: execute on a set of nodes whose hostnames match regexp *HOSTMASK*

`--withhostnegmask,-H` *HOSTMASK*
: execute on a set of nodes whose hostnames do not match regexp *HOSTMASK*

`--withgupidmask,-g` *HOSTMASK*
: execute on a set of nodes whose gupid (shown by `gxpc stat`) match regexp *HOSTMASK*

`--withgupidnegmask,-G` *HOSTMASK*
: execute on a set of nodes whose gupid (shown by `gxpc stat`) do not match regexp *HOSTMASK*

`--withtargetmask,-t` *HOSTMASK*
: execute on a set of nodes whose target name (shown by `gxpc stat`) match regexp *HOSTMASK*

`--withtargetnegmask,-T` *`HOSTMASK`*
: execute on a set of nodes whose target name (shown by `gxpc stat`) do not match regexp *HOSTMASK*

`--up FD0[:`*`FD1`*`]`
: collect output from *FD0* of *CMD*, and output them to *FD1* of `gxpc`. if :*FD1* is omitted, it is treated as if *FD1* == *FD0*

`--down FD0[:`*`FD1`*`]`
: broadcast input to *FD0* of `gxpc` to *FD1* of *CMD*. if :*FD1* is omitted, it is treated as if *FD1* == *FD0*

`--updown` *`FD1`*`:FD2[:`*`MASTER`*`]`
: if :*MASTER* is omitted, collect output from *FD1* of *CMD*, and broadcast them to *FD2* of *CMD*. if :*MASTER* is given, run *MASTER* on the local host, collect output from *FD1* of *CMD*, feed them to stdin of the *MASTER*. broadcast stdout of the *MASTER* to *FD1* of *CMD*.

`--pty`
: assign pseudo tty for stdin/stdout/stderr of *CMD*

`--rlimit rlimit_xxx:soft[:hard]`
: apply setrlimit(rlimit_xxx, soft, hard)

By default,

- stdin of `gxpc` are broadcast to stdin of *CMD*
- stdout of *CMD* are output to stdout of `gxpc`
- stderr of *CMD* are output to stderr of `gxpc`

This is as if '`--down` 0 `--up` 1 `--up` 2' are specified. In this case, stdout/stderr are block-buffered by default. You may need to do setbuf in your program or flush stdout/err, to display *CMD*'s output without delay. `--pty` overwrites this and turn them to line-buffered (by default). both stdout/err of *CMD* now goto stdout of `gxpc` (they are merged). *CMD*'s stdout/err should appear as soon as they are newlined.

**See Also:**

```
smask savemask pushmask rmask restoremask popmask
```

## 4.4 explore

**Usage:**

```
gxpc explore [OPTIONS] TARGET TARGET ...
```

**Description:**

Login target hosts specified by *OPTIONS* and *TARGET*.

**Options:**

`--dry` dryrun. only show target hosts

`--hostfile,-h` *HOSTS_FILE*
: give known hosts by file

`--hostcmd` *HOSTS_CMD*
: give known hosts by command output

`--targetfile,-t` *TARGETS_FILE*
: give target hosts by file

`--targetcmd` *TARGETS_CMD*
: give target hosts by command output

`--timeout` *SECONDS*
: specify the time to wait for a remote host's response until gxp considers it dead

`--children_soft_limit` *N* `(>= 2)`
: control the shape of the `explore` tree. if this value is *N*, `gxpc` tries to keep the number of children of a single host no more than *N*, unless it is absolutely necessary to reach requested nodes.

`--children_hard_limit` *N*
: control the shape of the `explore` tree. if this value is *N*, `gxpc` keeps the number of children of a single host no more than *N*, in any event.

`--target_prefix` *PATH*
: specify the directory in remote hosts in which gxp files are installed. default is ~/.gxp_tmp, meaning a temporary directory like ~/.gxp_tmp/RANDOM_NAME/gxp3 will be created and all files are installed there. automatically created if it does not exist. `use e`.g., /tmp/YOUR_NAME or something if you have ridicuously slow home directory.

`--verbosity` *N* `(0 <=` *N* `<= 2)`
: set verbosity level (the larger the more verbose)

`--set_default`
: if you set this option, options specified in this `explore` becomes the default. for example, if you say `--timeout` 20.0 and `--set_default`, timeout is set to 20.0 in subsequent explores, even if you do not specify `--timeout`.

`--reset_default`
: reset the default values set by `--set_default`.

`--show_settings`
: show effective `explore` options, considering those given by command line and those specified as default values.

Execution of an `explore` command will conceptually consist of the following three steps.

(1) Known Hosts: Know names of existing hosts, either by `--hostfile`, `--hostcmd`, or a default rule. These are called 'known hosts.' `-h` is an acronym of `--hostfile`.

(2) Targets: Extract login targets from known hosts. They are extracted by regular expressions given either by `--targetfile`, `--targetcmd`, or directly by command line arguments. `-t` is an acronym of `--targetfile`.

(3) `gxpc` will attempt to login these targets according to the rules specified by '`use`' commands.

Known hosts are specified by a file using `--hostfile` option, or by output of a command using `--hostcmd`. Formats of the two are common and very simple. In the simplest format, a single file contains a single hostname. For example,

```
hongo001
hongo002
hongo004
hongo005
hongo006
hongo007
hongo008
```

is a valid *HOSTS_FILE*. If you specify a command that outputs a list of files in the above format, the effect is the same as giving a file having the list by `--hostfile`. For example,

```
--hostcmd 'for i in `seq 1 8` ; do printf "%03d\n" $i ; done'
```

has the same effect as giving the above file to `--hostfile`.

The format of a *HOSTS_FILE* is actually a so-called `/etc/hosts` format, each line of which may contain several aliases of the same host, as well as their *IP* address. `gxpc` simply regards them as aliases of a single host, wihtout giving any significance to which columns they are in. Anything after '#' on each line is a comment and ignored. Lines not containning any name, such as empty lines, are also ignored. The above simple format is obviously a special case of this.

It is sometimes convenient to specify `/etc/hosts` as an argument to `--hostfile` or to specify 'ypcat hosts' as an argument to `--hostcmd`. As a matter of fact, if you do not specify any of `--hostfile`, `--hostcmd`, `--targetfile`, and `--targetcmd`, it is treated as if `--hostfile /etc/hosts` is given.

Login targets are specified by a file using `--targetfile` option, `--targetcmd` option, or by directly listing targets in the command line. Format of them are common and only slightly different from *HOSTS_FILE*. The format of the list of targets in the command line is as follows.

```
TARGET_REGEXP [N] TARGET_REGEXP [N] TARGET_REGEXP [N] ...
```

where *N* is an integer and *TARGET_REGEXP* is any string that cannot be parsed as an integer. That is, it is a list of regular expressions, each item of which may optionally be followed by an integer. The integer indicates how many logins should occur to the target matching *TARGET_REGEXP*. The following is a valid command line.

```
gxpc explore -h hosts_file hongo00
```

which says you want to target all hosts beginning with hongo00, among all hosts listed in hosts_file. If, for example, you have specified by '`use`' command that the local host can login these hosts by ssh, you will reach hosts whose names begin with hongo00. If you instead say

```
gxpc explore -h hosts_file hongo00 2
```

you will get two processes on each of these hosts.

If you do not give any of `--targetfile`, `--targetcmd`, and command line targets, it is treated as if a regular expression mathing any string is given as the command line target. That is, all known hosts are targets.

Format of targets_host is simply a list of lines each of which is like the list of arguments just explained above. Thus, the following is a valid *TARGETS_FILE*.

```
hongo00 2
chiba0
istbs
sheep
```

which says you want to get two processes on each host beginning with hongo00 and one process on each host beginning with chiba0, istbs, or sheep. Just to illustrate the syntax, the same thing can be alternatively written with different arrangement into lines.

```
hongo00 2 chiba0
istbs sheep
```

Similar to hosts_file, you may instead specify a command line producing the output conforming to the format of *TARGETS_FILE*.

We have so far explained that target_regexp is matched against a pool of known hosts to generate the actual list of targets. There is an exception to this. If *TARGET_REGEXP* does not match any host in the pool of known hosts, it is treated as if the *TARGET_REGEXP* is itself a known host. Thus,

```
gxpc explore hongo000 hongo001
```

will login hongo000 and hongo001, because neither hosts_file nor hosts_cmd hosts are given so these expressions obviously won't match any known host. Using this rule, you may have a file that explicitly lists all hosts and solely `use` it to specify targets without using separate *HOSTS_FILE*. For example, if you have a long *TARGETS_FILE* called targets like:

```
abc000
abc001
  ...
abc099
def000
def001
  ...
def049
pqr000
pqr001
  ...
pqr149
```

and say

```
gxpc explore -t targets
```

you say you want to get these 300 targets using whatever methods you specified by '`use`' commands.

Unlike *HOSTS_FILE*, an empty line in *TARGETS_FILE* is treated as if it is the end of file. By inserting an empty line, you can easily let `gxpc` ignore the rest of the file. This rule is sometimes convenient when targeting a small number of hosts within a *TARGETS_FILE*.

Here are some examples.

1.

```
gxpc explore -h hosts_file chiba hongo
```

Hosts beginning with chiba or hongo in hosts_file become the targets.

2.

```
gxpc explore -h hosts_file -t targets_file
```

Hosts matching any regular expression in targets_file become the targets.

3.

```
gxpc explore -h hosts_file
```

All hosts in hosts_file become the targets. Equivalent to '`gxpc explore -h` hosts_file .' ('.' is a regular expression mathing any non-empty string).

4.

```
gxpc explore -t targets_file
```

All hosts in targetfile become the targets. This is simiar to the previous case, but the file format is different. Note that in this case, strings in targets_file won't be matched against anything, so they should be literal target names.

5.

```
gxpc explore chiba000 chiba001 chiba002 chiba003
```

chiba000, chiba001, chiba002, and chiba003 become the targets.

6.

```
gxpc explore chiba0
```

Equivalent to '`gxpc explore -h /etc/hosts` chiba0' which is hosts beginning with chiba0 in `/etc/hosts` become the targets. Useful when you `use` a single cluster and all necessary hosts are listed in that file.

7.

```
gxpc explore
```

Equivalent to '`gxpc explore -h /etc/hosts`' which is in turn equivalent to '`gxpc explore -h /etc/hosts .`' That is, all hosts in `/etc/hosts` become the targets. This will be rarely useful because `/etc/hosts` typically includes hosts you don't want to `use`.

## 4.5 export

**Usage:**

```
gxpc export VAR=VAL
```

**Description:**

Set environment variable *VAR* to *VAL* on the selected nodes. Options are the same as those of '`e`' command.

## 4.6 help

**Usage:**

```
gxpc help
gxpc help COMMAND
```

**Description:**

Show summary of `gxpc` commands or a `help` on a specific *COMMAND*.

## 4.7 i

**Usage:**

```
gxpc i
```

**Description:**

Enter a new shell with a `gxpc` session.

## 4.8 log_level, log_base_time

**Usage:**

```
gxpc log_level LEVEL
gxpc log_base_time
```

**Description:**

Command `log_level` will set the log level of the selected nodes to the specified *LEVEL*. 0 will write no logs. 2 will write many. Command `log_base_time` will reset the time of the selected nodes to zero. Subsequent log entries will record the time relative to this time.

## 4.9 make

**Usage:**

```
gxpc make GNU-MAKE-ARGS [ -- GXPC-MAKE-ARGS ]
```

**Description:**

Parallel and distributed `make`.

**Options:**

## 4.10 makectl

**Usage:**

```
gxpc makectl (leave|leave_now|join)
```

**Description:**

*GXP* daemons who receive this command will leave, leave immediately, or join the gxp `make` computation.

**Options:**

## 4.11 makeman

**Usage:**

```
gxpc makeman
```

**Description:**

Generate command reference chapter of gxp manual.

## 4.12 mapred

**Usage:**

```
gxpc mapred GNU-MAKE-ARGS [ -- GXPC-MAKE-ARGS ]
```

**Description:**

This is a map-reduce framework built on top of *GXP* `make`. You can run map-reduce without writing any Makefile by yourself. You specify various options in GNU-MAKE-ARGS in the form of var=val. *A* simple example:

```
gxpc mapred -j input=big.txt output=out.txt \
  mapper=./my_mapper reducer=./my_reducer
```

You will find the '-n' option of `make` useful, since it tells you which commands are going to be executed by this command line.

```
gxpc mapred -n input=big.txt output=out.txt \
  mapper=./my_mapper reducer=./my_reducer
```

**Options:**

You will probably want to specify at least the following. input=<input filename> (default: "input") output=<output filename> (default: "output") mapper=<mapper command> (default: "ex_word_count_mapper") reducer=<reducer command> (default: "ex_word_count_reducer")

<input filename> and <output filename> are filenames. <mapper command> is a command that reads anything from stdin and writes key-value pairs. <reducer command> is a command that reads key-value pairs from stdin in sorted order and writes arbitrary final outputs.

You will frequently want to specify the following. n_mappers=<number of map tasks> (default: 4) n_reducers=<number of reduce tasks> (default: 2) reader=<reader command> (default: "ex_line_reader") int_dir=<intermediate directory> (default: "int_dir") keep_intermediates=y will keep all intermediate files in int_dir small_step=y will execute the entire computation in small steps dbg=y equivalent to keep_intermediates=y small_step=y (useful for debugging)

More options are: partitioner=<partitioner command> (default: "ex_partitioner") pre_reduce_sorter=<sort command> (default: "sort") final_merger=<merge command> (default: "cat")

pre_reduce_sorter is a command that runs before each reducer. It takes in the command line mappers' output filenames and should output to the stdout the sorted list of key-value pairs. final_merger takes the filenames of all the reducers' output and outputs the final result.

## 4.13 ping

**Usage:**

```
gxpc ping [LEVEL]
```

**Description:**

Send a small message to the selected nodes and show some information. The parameter *LEVEL* is 0 or 1. Default is 0. This is useful to know the name and basic information about all or some nodes. It is also useful to check the status (liveness) of nodes and `trim` non-responding nodes.

**See Also:**

```
trim smask
```

## 4.14 popmask

**Usage:**

```
gxpc popmask
```

**Description:**

Pop the set of nodes on the top of the stack. The next entry that is used to be referred to by name '1' will now be the top of the stack, and thus become the default set of nodes selected for execution.

**See Also:**

```
pushmask
```

## 4.15 pp

**Usage:**

```
gxpc pp GNU-MAKE-ARGS [ -- GXPC-MAKE-ARGS ]
```

**Description:**

This is a simple parameter parallel framework built on top of *GXP* `make`. You can run a simple parameter-sweep type parallel applications without writing any Makefile by yourself. You specify various options in GNU-MAKE-ARGS in the form of var=val. *A* simple example:

```
gxpc pp -j cmd='./my_cmd -a $(a) $(f) ' a="1 2 3 4" f="a.txt b.txt c.txt" paramet
```

This will execute "./my_cmd -a $(a) $(f)" for all the 4x3=12 combinations of a and f. `parameters=` is mandatory. For all names listed in `parameters`, you need to specify at least one value. You will find the '-n' option of `make` useful, since it tells you which commands are going to be executed by this command line.

```
gxpc pp -n cmd='./my_cmd -a $(a) $(f) ' a="1 2 3" f="a.txt b.txt c.txt"
```

## 4.16 prof_start, prof_stop

**Usage:**

```
gxpc prof_start FILENAME
gxpc prof_stop
```

**Description:**

Start/stop profiling of the selected nodes. Stats are saved to the *FILENAME*.

## 4.17 smask, savemask, pushmask

**Usage:**

```
gxpc smask    [-]
gxpc savemask [-] NAME
gxpc pushmask [-]
e.g.,
gxpc  e  'uname | grep Linux'
gxpc  smask
```

**Description:**

All three commands have a common effect. That is to modify the set of nodes that will execute subsequent commands. When '-' argument is not given, nodes that executed the last command and succeeded are selected. With argument '-', nodes that executed the last command and failed are set. The definition of sucess or failure depends on commands, but in the case of 'e' command, a node is considered to succeed if the command exits with status zero.

These commands can be used to efficiently choose the nodes to execute subsequent commands by various criterion.

**Examples:**

1.

```
gxpc  e  'uname | grep Linux'
gxpc  smask
```

This will set the execution mask of Linux nodes.

2.

```
gxpc  e  'which apt-get'
gxpc  smask  -
gxpc  e  hostname
```

This will set the execution mask of nodes that do not have apt-get command, and the last command will show their hostnames.

To see the effect of these commands, it is advised to include the gxp3 directory in your *PATH*, and add something like the following in your shell (bash) prompt, which will show the number of nodes that succeeded the last command, the number of nodes that is currently selected, and the number all nodes.

```
export PS1='\h:\W`which gxp_prompt 1> /dev/null && gxp_prompt`% '
```

With this, you can see the effect of these commands in shell prompt.

```
[66/66/66]% e 'uname | grep Linux'
Linux
Linux
Linux
Linux
Linux
Linux
Linux
Linux
Linux
Linux
[10/66/66]% gxpc smask
[10/10/66]%
```

In addition to setting the execution mask, `savemask` saves the set of selected nodes with the specified name. Sets of nodes hereby saved can be later used for execution by giving

`--withmask` (`-m`) option. This is useful when your work needs several, typical set of nodes to execute commands on. For example, you may save a small number of nodes for test, all gateway nodes to compile programs, all nodes within a particular cluster, and really all nodes.

Command `pushmask` is similar to `savemask`, but the set is saved onto a stack. The newly selected set of nodes are on the top of the stack and named '0'. Previously selected nodes are named by the distance from the top.

**See Also:**

```
showmasks rmask restoremask popmask
```

## 4.18 quit

**Usage:**

```
gxpc quit [--session_only]
```

**Description:**

Quit gxp session. By default, all daemons will exit. If `--session_only` is given, daemons keep running and only the session will cease.

## 4.19 reclaim

**Usage:**

```
gxpc reclaim tid
```

**Description:**

Command `reclaim` will `reclaim` task tid unconditionally.

## 4.20 restoremask

**Usage:**

```
gxpc restoremask NAME
```

## 4.21 rmask

**Usage:**

```
gxpc rmask
```

**Description:**

Reset execution mask. Let all nodes execute subsequent commands.

**See Also:**

```
showmasks smask savemask restoremask pushmask popmask
```

## 4.22 rsh

**Usage:**

```
gxpc rsh [OPTIONS]
gxpc rsh [OPTIONS] rsh_name
gxpc rsh [OPTIONS] rsh_name rsh_like_command_template
```

**Description:**

Show, add, or modify rsh-like command explore/use will recognize to the repertoire. 'gxpc `rsh`' lists all configured rsh-like commands. 'gxpc `rsh` rsh_name' shows the specified rsh-like command. 'gxpc `rsh` rsh_name rsh_like_command_template' adds or modifies (if exist) the specified rsh-like command to `use` rsh_like_command_template.

```
The rsh_name is used as the first parameter of 'use' command (e.g.,
'gxpc use ssh src target' or 'gxpc use rsh <src> <target>').
```

By default, the following rsh-like commands are builtin.

```
ssh, ssh_as, rsh, rsh_as, sh, sge, torque, and pbs.
```

**Options:**

`--full` when invoked as `gxpc rsh --full` (with no other args), show command lines of all available rsh-like commands.

**Examples:**

```
1.
  gxpc rsh ssh
  (output) ssh : ssh -o 'StrictHostKeyChecking no' ... -A %target% %cmd%
```

This displays that an rsh-like command named 'ssh' is configured, and gxp understands that, to run a command a host via ssh, it should `use` a command:

```
ssh -o 'StrictHostKeyChecking no' ... -A %target% %cmd%
```

with %target% replaced by a target name (normally a hostname) and %cmd% by whatever commands it wants to execute on the target.

2. `gxpc rsh` ssh ssh -i elsewhere/id_dsa -o 'StrictHostKeyChecking no' -A %target% %cmd%

```
This instructs gxp to use command line:
```

ssh -i elsewhere/id_dsa -o 'StrictHostKeyChecking no' -A %target% %cmd%

```
when it uses ssh.  You can arbitrarily name a new rsh-like command.
For example, let's say on some hosts, ssh daemons listen on customized
ports (say 2000) and you need to connect to that port to login those
hosts, while connecting to the regular port to login others. Then you
first define a new rsh-like command.
```

3. `gxpc rsh` ssh2000 ssh -p 2000 -o 'StrictHostKeyChecking no' -A %target% %cmd%

```
And you specicy ssh2000 label to login those hosts, using 'use' command. e.g.,
```

`use` ssh2000 <src> <target>

**See Also:**

```
use explore
```

## 4.23 set_max_buf_len

**Usage:**

```
gxpc set_max_buf_len N
```

**Description:**

Set maximum internal buffer size of gxp daemons in bytes. default is 10KB and any value below the default is ignored. If no argument is given, `use` the default value.

## 4.24 show_explore

**Usage:**

```
gxpc show_explore SRC TARGET
```

**Description:**

Show command used to `explore` *TARGET* from *SRC*.

## 4.25 showmasks

**Usage:**

```
gxpc showmasks [--level 0/1] [NAME]
e.g.,
gxpc showmasks
gxpc showmasks --level 1
```

**Description:**

Show summary (`--level` 0) or detail (`--level` 1) of all execution masks (if ‘name’ is omitted) or a specified excution mask named ‘name.’ The name of the current default execution mask is ‘0’.

**Examples:**

```
gxpc showmasks
gxpc showmasks --level 0     # show summary of all exec masks
gxpc showmasks --level 0 0   # show summary of the current mask
gxpc showmasks --level 1 0   # show detail of current exec mask
gxpc showmasks --level 1 all # show detail of exec mask named ‘all’
```

Without any argument as in the first line, a summary like the following is shown.

```
 0 : 1
 1 : 66
 six : 6
 ten : 8
 half : 27
```

This says there are three execution masks, called ‘0’, ‘1’, ‘six’, ‘ten’, and ‘half.’ The right column indicates the number of nodes that will execute a command when the mask is selected by a `--withmask` (`-m`) option. ‘0’ is the current, default execution mask, used when `--withmask` (`-m`) option is not given. These masks are created by `savemask` or `pushmask` command.

When option `--level` 1 is given, details about the execution mask will be shown, like the following.

```
six : 6
 idx: name (gupid)
   -: hongo-lucy-tau-2006-12-31-22-40-24-31387
   -:  hongo002-tau-2006-12-31-13-39-17-338
   0:   hongo010-tau-2006-12-31-22-25-58-20559
   -:  hongo006-tau-2006-12-31-22-27-19-11951
   1:   hongo020-tau-2006-12-31-22-34-06-11026
   -:  hongo001-tau-2006-12-31-22-34-38-15517
   2:   hongo030-tau-2006-12-31-22-44-07-4931
   -:  hongo007-tau-2006-12-31-22-30-54-21738
   3:   hongo040-tau-2006-12-31-22-47-55-32666
   -:  hongo004-tau-2008-12-31-22-42-33-31254
   4:   hongo060-tau-2006-12-31-22-42-59-3899
```

```
-:  hongo005-tau-2006-12-31-22-38-30-1150
5:   hongo050-tau-2006-12-31-22-40-55-18088
```

This is a subtree of the whole tree of gxp daemons. Nodes that will actually execute commands will have an index number (0, 1, ..., 5) on the first column. Nodes marked with '-' will not execute the command, but are part of the minimum subtree containing those six nodes.

**See Also:**

```
smask rmask savemask restoremask pushmask popmask
```

## 4.26 stat

**Usage:**

```
gxpc stat [LEVEL]
```

**Description:**

Show all live gxp daemons in tree format. *LEVEL* is 0, 1, or 2 and determines the detail level of the information shown.

## 4.27 trim

**Usage:**

```
gxpc trim
```

**Description:**

Trim (release) some subtrees of gxp daemons. This is typically used after a `ping` cmd followed by `smask`, to prune non-responding (dead) daemons. Specifically, `trim` command will be executed on the selected nodes, and each such node will throw away a children $C$ if no nodes under the subtree rooted at $C$ are selected for execution of this `trim` command. This effectively prunes (trims) the subtree from the tree of gxp daemons. For example,

```
gxpc ping
```

```
# If there are some non-responding daemons, this command will hang.
# type <Ctrl-C> to quit.
gxpc smask
gxpc trim
```

## 4.28 use, edges

**Usage:**

```
gxpc use          [--as USER] RSH_NAME SRC [TARGET]
gxpc use --delete [--as USER] RSH_NAME SRC [TARGET]
gxpc use
gxpc use --delete [idx]
```

e.g., `gxpc use` ssh your_hostname compute_node_prefix

**Description:**

```
  Configure rsh-like commands used to login targets matching a
particular pattern from hosts matching a particular pattern. The
typical usage is 'gxpc use RSH_NAME SRC TARGET', which says gxp can
use an rsh-like command RSH_NAME for SRC to login TARGET. gxpc
remembers these facts to decide which hosts should issue which
commands to login which hosts, when explore command is issued. See the
tutorial section of the manual.
```

**Examples:**

```
gxpc use           ssh abc000.def.com pqr.xyz.ac.jp
gxpc use           ssh abc000 pqr
gxpc use           ssh abc
gxpc use           rsh abc
gxpc use --as taue ssh abc000 pqr
gxpc use qrsh      abc
gxpc use qrsh_host abc
gxpc use sge       abc
gxpc use torque    abc
```

The first line says that, if `gxpc` is told to login pqr.xyz.ac.jp by `explore` command, hosts named abc000.def.com can `use` 'ssh' method to do so. How it translates into the actual ssh command line can be shown by '`show_explore`' command (try '`gxpc help show_explore`') and can be configured by '`rsh`' command (try '`gxpc help rsh`').

*SRC* and *TARGET* are actually regular expressions, so the line like the first one can often be written like the second one. The first line is equivalent to the second line as long as there is only one host begining with abc000 and there is only one target beginning with pqr. In general, the specification:

```
gxpc use RSH_NAME SRC TARGET
```

is read: if `gxpc` is told to login a target matching regular expession *TARGET*, a host matching regular expression *SRC* can `use` *RSH_NAME* to do so.

Note that the effect of `use` command is *NOT* to specify which target `gxpc` should login, but to specify *HOW* it can do so, if it is told to. It is the role of `explore` command to specify which target hosts it should login

If the *TARGET* argument is omitted as in the third line, it is treated as if *TARGET* expression is *SRC*. That is, the third line is equivalent to:

```
gxpc use ssh abc abc
```

This is often useful to express that ssh login is possible between hosts within a single cluster, which typically have a common prefix in their host names. If the traditional `rsh` command is allowed within a single cluster, the fourth line may be useful too.

If `--as` user option is given, login is issued using an explicit user name. The fifth line says when gxp attempts to login pqr from abc000, the explicit user name 'taue' should be given. You do not need this as long as the underlying rsh-like command will complement it by a configuration file. `e`.g., ssh will read ~/.ssh/config to complement user name used to login a particular host.

qrsh_host uses command qrsh, with an explicit hostname argument to login a particular host (`i.e.`, qrsh -l hostname=...). This is useful in environments where direct ssh is discouraged or disallowed and qrsh is preferred.

qrsh also uses qrsh, but without an explicit hostname. The host is selected by the scheduler. Therefore it does not `make` sense to try to speficify a particular hostname as *TARGET*. Thus, the effect of the line

```
gxpc use qrsh abc
```

is if targets beginning with abc is given (upon `explore` command), a host beginning with abc will issue qrsh, and get whichever host is allocated by the scheduler.

**See Also:**

```
explore rsh
```

## 4.29 version

**Usage:**

```
gxpc version
```

## 4.30 vgxp

**Usage:**

```
gxpc vgxp
```

# 5 Tools Reference

todo

# 6 Troubleshooting

todo

# 7 Environment Variables Reference

todo

# 8 Key Stroke Reference

todo

# Indices

## Function Index

(Index is nonexistent)

## Variable Index

(Index is nonexistent)

## Data Type Index

(Index is nonexistent)

## Program Index

(Index is nonexistent)

## Concept Index

(Index is nonexistent)

# Table of Contents