GXP ver.3 Tutorial

Kenjiro Taura

What GXP intends to be

- A tool to interactively use hundreds of machines
- Many users' "shortest path" from "just a bunch of Unix machines" to "parallel computation"

Philosophy

- More SW you assume, more likely you fail
- More admins you ask, more likely you fail

$\forall \Rightarrow$ Trust (rely on) as little SW as possible

- ssh and python
- Lightweight, explicit, and transparent

Requirements (1)

- The very minimum requirements
  - Python
  - A Unix (Linux, BSD, Solaris, etc.) host

Requirements (2)

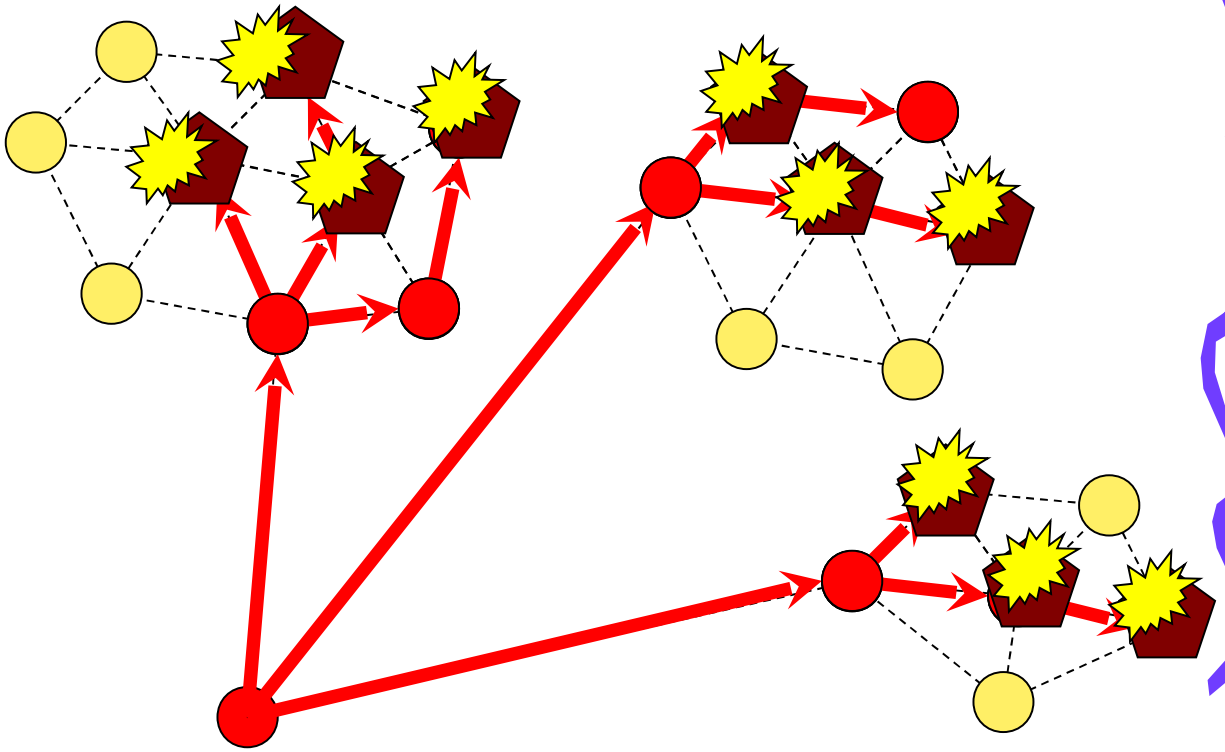
- Perhaps you want to use many hosts with GXP. Then you need,
  - an rsh-like command (rsh, ssh, …), and
  - an authentication settings for it (to be discussed in detail later)

Installation

- Step 1:
  - visit `http://www.logos.ic.i.u-tokyo.ac.jp/gxp`
- Step 2: either:
  - add <gxpdir> to your PATH in your shell start up file, or
  - ln –s <gxpdir>/gxpc /a/dir/in/your/PATH/gxpc

1. you say which hosts can login which hosts and how (use)
2. say which hosts you want to acquire (explore)
3. select hosts on which you want to execute commands (smask etc.)
4. execute commands (e etc.)

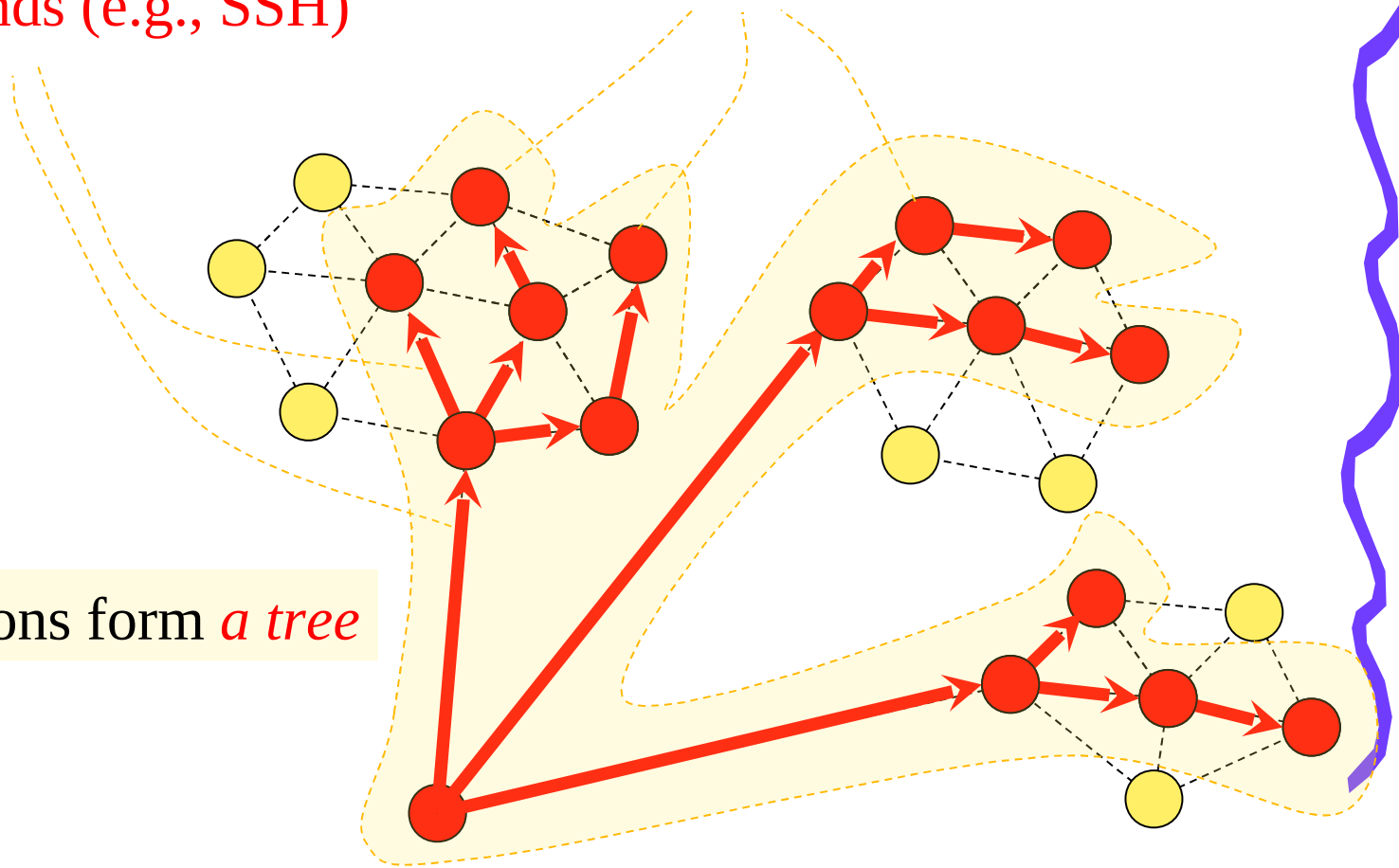
You can execute any of them at any time, in any order



Basic concepts

3. A daemon creates other daemons by an underlying rsh-like commands (e.g., SSH)

1. GXP *daemons* (*gxpd.py*) stay running



2. Daemons form *a tree*

A Note on Terminology

- GXP invoke some GXP daemons and talk to them (“*exec this command*” etc.)
- You typically invoke a single daemon on each host, but not necessarily (you may invoke multiple daemons on a single host)
- When we say “a host,” it most of the time really means “a daemon”, though it’s not totally accurate

Scenario

- Playing with a single host
- A recommended prompt setting
- use & explore
- Playing with many hosts
- Selecting target hosts
- Embarrassingly parallel computation
- Accompanying tools (bomb and bcp)

Playing with a single host

```
$ gxpc e hostname
gxpc: no daemon found, create one
hongo000
```

A daemon starts automatically and stays running

Command syntax

gxpc [*global_options*] *gxp_command* [*command_options*] *args* …

`e, use, explore,` etc.

- **gxpc** is the *only* program you need to run from the shell prompt

help

```
$ gxpc help
```

lists all available gxp commands

```
$ gxpc help gxp_command
```

shows a detailed help on a gxp command

Daemons stay running

- A primitive way to see your daemon

```
$ ps -ef | grep gxpd
```

- A better way

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

saying 1 daemon is in the tree

Terminating daemons

```
$ gxpc quit
```

terminates daemons

```
$ gxpc prompt
```

now prints nothing

A recommended setting

- include `gxpc prompt` in your shell prompt!
- bash example (in your .bashrc):

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

include [A/B/C] in your prompt

use backquotes

doesn't matter (whatever is your taste)

doesn't bother you on error (e.g., "command not found")

tcsh, zsh?

- zsh:

```
precmd () {
     export PS1="%m:%c$(gxpc prompt 2>
/dev/null)$ "
    }
```

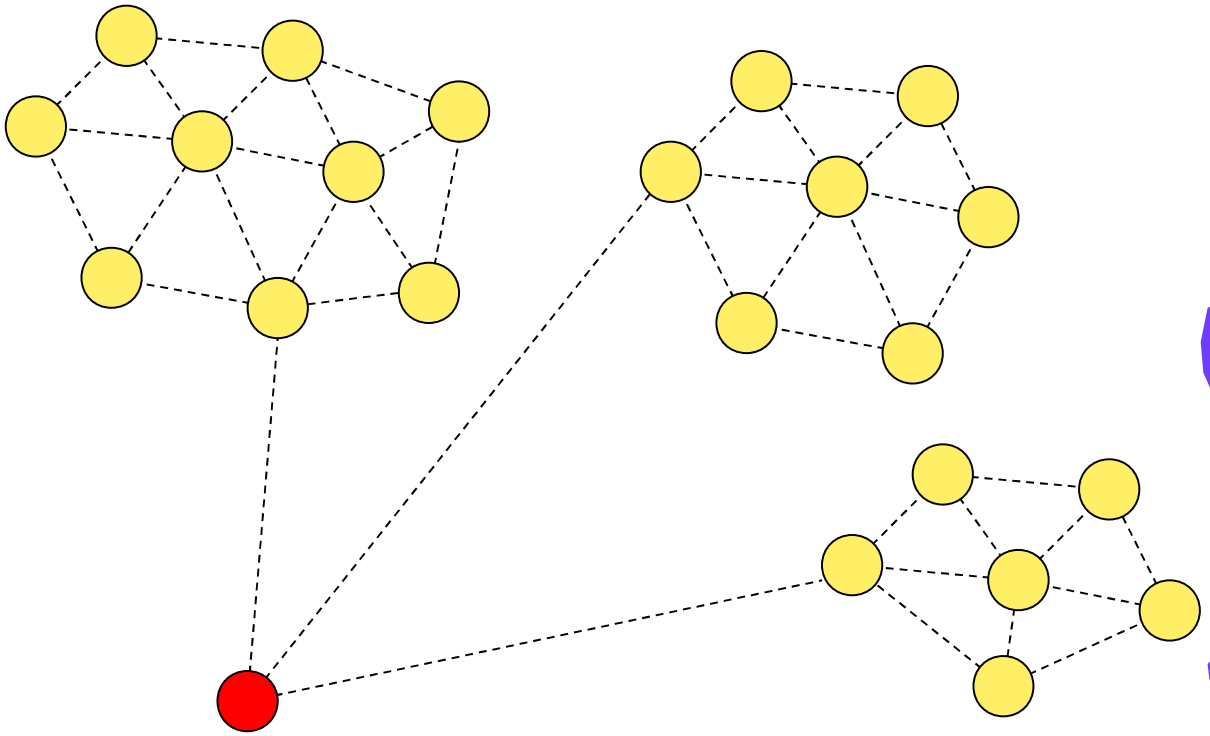
- tcsh (no good way to redirect stderr only?):

```
alias precmd 'set prompt="%m:%c`which gxpc >&
/dev/null && gxpc prompt`$ "'
```

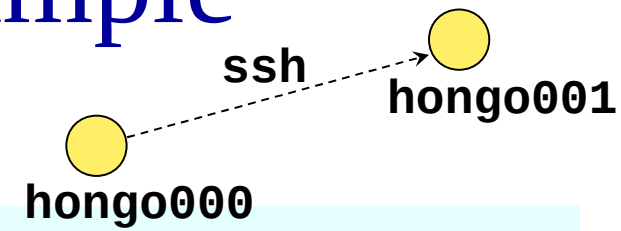
(all in a single line)

use

- **`gxpc`** **`use`** command tells GXP which hosts can remote-login which hosts, with which rsh-like commands



use example



- A simplest example

```
$ gxpc use ssh hongo000 hongo001
```

reads:

gxp [can] use **ssh** [from] **hongo000** [to] **hongo001**

- in short, the following will succeed *without your password/passphrase being asked*

(on **hongo000**):
$ **ssh hongo001** *arbitrary_shell_command*

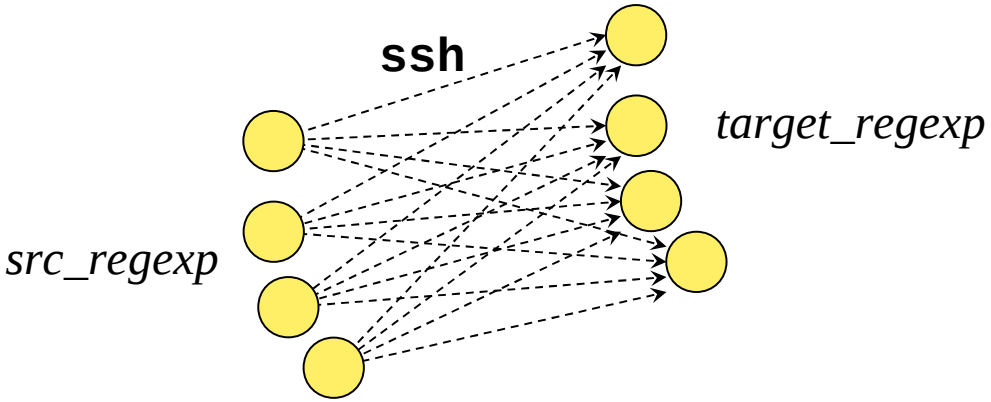
More general use (1)

- In more general,

```
$ gxpc use ssh src_regexp target_regexp
```

reads

gxp [can] use **ssh** [from any host matching] *src_regexp* [to any target matching] *target_regxp*



More general use (2)

- Furthermore,

```
$ gxpc use ssh src_regexp
```

is an abbreviation of

```
$ gxpc use ssh src_regexp src_regexp
```

- That is,

ssh among [hosts matching] *src_regexp* will succeed

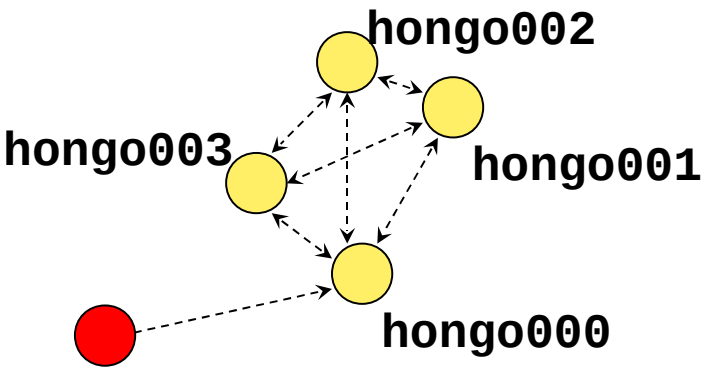
Practical examples

```
$ gxpc use ssh hongo
```

reads: **ssh** among **hongo*** will succeed (often useful in clusters)

```
$ gxpc use `hostname` hongo000
```

reads: **ssh** from your host to **hongo000** will succeed (often useful for a gateway of a remote cluster)



What about other remote shells?

- Built into GXP
  - **`rsh`** : a legacy remote shell
  - **`sh`** : local shell to run multiple daemons on a single host (useful in multi-processors)
  - **`sge,n1ge,troque`** : batch queuing system
  - **`qrsh`** : an interactive shell over SGE
- Exact command lines are customizable

Which remote shells to use?

- RSH
  - ☺ sometimes available within a cluster and easiest to setup (~/.rhosts)
  - ☺ even better, your admin may have written /etc/hosts.equiv for some reasons
  - ☹ rarely allowed across clusters
- SSH
  - ☺ ubiquitous, both across and inside clusters
  - ☹ need a public-key setup, but it’s necessary anyway across clusters

Which remote shells to use?

- RSH/SSH
  - ☹ Your cluster may not allow you to run heavy jobs using them
  - ☹ Some admins may even ban them
- Batch queuing system (SGE, torque, etc.)
  - ☺ may be the only choice in environments that are really actively shared by many groups

SSH settings

- The goal is to make the following succeed, without your intervention (like password or alike)

(on host **A**):
$ **ssh B** *arbitrary_shell_command*

- The following few slides explain how for OpenSSH (the concept is common in all SSH implementations)

SSH authentication basics

- SSH supports, among others,
  - *password* authentication

```
$ ssh B arbitrary_shell_command
tau@A's password:
```

  - public-key authentication
- Password authentication always asks you to input password interactively

⇒GXP needs public-key authentication

SSH public-key authentication

- For the following to succeed,

  (on host **A**):

  `$ ssh B` *arbitrary_shell_command*

  - **A** must have a private key (normally in `~/.ssh/id_{dsa,rsa}`)
  - **B** must have the corresponding public key in `~/.ssh/authorized_keys`
- **`ssh-keygen`** generates a key pair

  (on host **A**):

  `$ ssh-keygen -t dsa` (or **rsa**)

Passphrase-less authentication

- You may encrypt the private key by a *passphrase*

```
$ ssh-keygen -t dsa
Generating public/private dsa key pair.
Enter file in which to save the key
(/home/tau/.ssh/id_dsa):
Enter passphrase (empty for no passphrase):
```

- But then you are, normally, asked to input it interactively on every ssh

- Two ways not to be asked
  - empty passphrase
  - ssh agent

```
$ ssh B
Enter passphrase for key
'/home/tau/.ssh/id_dsa':
```

Empty passphrase

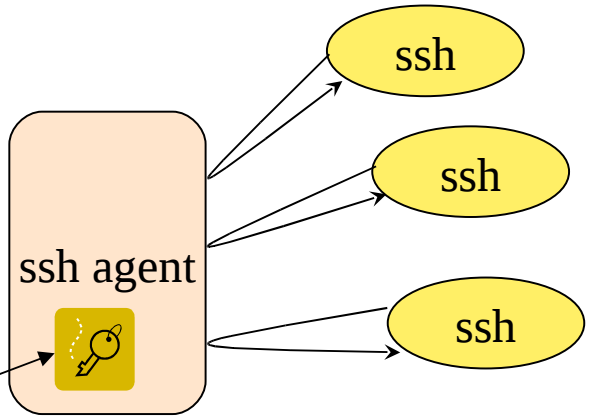
- do not encrypt the private key (leave the passphrase empty when **`ssh-keygen`** asks)
- simpler, but less secure
  - risk **B** if host **A** is compromised

SSH agent

- let “ssh agent” remember it and supply it on your behalf

1. run an agent and tell your shell where it is running

```
$ eval `ssh-agent`
Agent pid is 6753
```



3. now ssh asks the agent without bothering you

2. decrypt and supply your private key

```
$ ssh-add
Enter passphrase for /home/tau/.ssh/id_dsa:
Identity added: /home/tau/.ssh/id_dsa
(/home/tau/.ssh/id_dsa)
```

So how to setup everything? (Empty passphrase)

- Step 1: generate public/private key pair if you have not done so

```
(on host A):
$ ssh-keygen -t dsa
Generating public/private dsa key pair.
Enter passphrase (empty for no passphrase):↵
Enter same passphrase again: ↵
Your identification has been saved in
/home/tau/.ssh/id_dsa.
Your public key has been saved in
/home/tau/.ssh/id_dsa.pub.
The key fingerprint is:
28:bd:66:f6:9f:c4:f2:88:ce:a0:8b:a7:7a:a6:d0:57
tau@hongo000
```

- Step 2: install the public key in **B** (append **~/.ssh/id_dsa.pub** of **A** to **~/.ssh/authorized_keys** of **B**)
- if **A** and **B** share home directory (**~/.ssh**, for that matter), it is simple

(on host **A**):

```
$ cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys
```

- otherwise you first copy id_dsa.pub to B somehow and work on B

(on host **B**, after copying **A**'s id_dsa.pub somehow to **B**:

```
$ cat id_dsa.pub copied from A >>
~/.ssh/authorized_keys
```

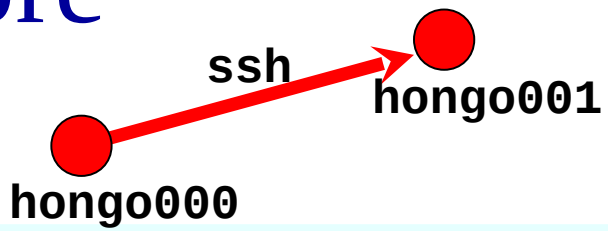
So how to setup everything? (ssh agent)

- Do the same as before up to now (except you input non-empty passphrase), and then

(on host **A**):

```
$ eval `ssh-agent`
$ ssh-add
Enter passphrase for /home/tau/.ssh/id_dsa:
Identity added: /home/tau/.ssh/id_dsa
(/home/tau/.ssh/id_dsa)
```

explore



- A small example:

(on **hongo000**):

```
hongo000:tau$ gxpc use ssh hongo000 hongo001
hongo000:tau[1/1/1]$ gxpc explore hongo001
reached : hongo001
hongo000:tau[2/2/2]$
```

indicates you reached hongo001

- Now e will run commands on the two hosts

```
hongo000:tau[2/2/2]$ gxpc e hostname
hongo000
hongo001
hongo000:tau[2/2/2]$
```

Detailed daemon status

```
hongo000:tau[2/2/2]% gxpc stat
/tmp/gxp-tau/gxpsession-hongo000-tau-2007-07-04-18-41-08-22896-
39385997
hongo000 (= hongo000-tau-2007-07-04-18-41-08-22896)
 hongo001 (= hongo001-tau-2007-07-04-18-41-15-16457)
```

2 daemons are running

target names

globally unique identifiers of daemons (generated)

indentation shows tree structure (**hongo001** is a child of **hongo000**)

session file (where most status information is kept)

Multiple daemons on a host

(on **hongo000**):

```
hongo000:tau$ gxpc use ssh hongo000 hongo001
hongo000:tau[1/1/1]$ gxpc explore hongo001 2
reached : hongo001
reached : hongo001
hongo000:tau[3/3/3]% gxpc stat
/tmp/gxp-tau/gxpsession-hongo000-tau-2007-07-04-18-41-08-
22896-39385997
hongo000 (= hongo000-tau-2007-07-04-18-41-08-22896)
 hongo001 (= hongo001-tau-2007-07-04-18-57-52-16680)
 hongo001 (= hongo001-tau-2007-07-04-18-41-15-16457)
```

- Note this issues **`ssh`** *twice* from **`hongo000`** to **`hongo001`** (somewhat inefficient)

explore counts hosts already acquired

(on **hongo000**):

```
hongo000:tau$ gxpc use ssh hongo000 hongo001
hongo000:tau[1/1/1]$ gxpc explore hongo001
reached : hongo001
hongo000:tau[2/2/2]$ gxpc explore hongo001
hongo000:tau[2/2/2]$ gxpc explore hongo001 2
reached : hongo001
hongo000:tau[3/3/3]$
```

no effect

only add one daemon

explore in more general

```
$ gxpc explore target [n] target [n]…
```

tries to reach specified *targets*, *n* times

- It can reach hosts multiple hops away

- (not truly accurate) analogy:
  - use is like specifying a graph
  - explore is like calculating a spanning tree involving all specified *targets*

Target expansion rules

- Nobody wants to type:

```
$ gxpc explore hongo000 hongo001 hongo002 hongo003 hongo004
hongo005 hongo006 hongo007   …   hongo069
```

- Three expansion rules to save you:
  - numeric set notation
  - target file
  - host file + regular expression

Numeric set notation

```
hongo[[000-005]]
→ hongo000 hongo001 hongo002 hongo003 hongo004 hongo005
```

```
hongo[[000-005:003]]
→ hongo000 hongo001 hongo002 hongo004 hongo005
```

```
hongo[[000-005:003-004]]
→ hongo000 hongo001 hongo002 hongo005
```

```
hongo[[000-005:003-004,010-012]]
→ hongo000 hongo001 hongo002 hongo005, hongo010, hongo011,
hongo012
```

$A,B \Rightarrow A \cup B$ (union)

$A:B \Rightarrow A - B$ (set difference)

Target file

$ **gxpc explore –t** *file* …

→ **gxpc explore** ⟨*whatever is in file*⟩ …

- An example:

```
$ cat machines
hongo[[000-005]]
chiba[[000-004]]
$ gxpc explore –t machines
→ gxpc explore hongo000 … hongo005 chiba000 … chiba004
```

Host file + regular expression

$ **gxpc explore –h** *file target* …

→ **gxpc explore** ⟨*whatever matches target in file*⟩ …

- The format of *file* is that of **/etc/hosts**
  - In fact you may use exactly **/etc/hosts**
- Example (assume **/etc/hosts** has entries **hongo000, hongo001**, …)

```
$ gxpc explore –h /etc/hosts hongo0
→ gxpc explore hongo000 hongo001 …
```

So how explore determines the targets?

- *target* is actually a regular expression, extended with a special notation [[…]]
- explore determines the set of targets as follows (next page)

1. process files specified with **`-h`**
   ⇒*a pool of known hosts*
2. process files specified with **`-t`** and all arguments directly in the command line
   ⇒*target expressions*
3. expand numeric set notation **`[[…]]`** in each target expression
   ⇒*target regular expressions*

- apply each target regular expression to the pool of known hosts
  - > 0 matches found ⇒take them into the final target list
  - no matches found ⇒take the target regular expression literally into the final target list

A practical example

```
hongo000:tau$ gxpc use ssh hongo
gxpc: no daemon found, create one
hongo000:tau[1/1/1]$ gxpc explore hongo[[000-029]]
reached : hongo001
reached : hongo002
reached : hongo003
  ...
reached : hongo017
reached : hongo018
reached : hongo015
hongo003-tau-2007-07-04-19-32-41-11432 heard from hongo014 : ssh:
connect to host hongo014 port 22: No route to host
failed  : hongo014 <- hongo003-tau-2007-07-04-19-32-41-11432
hongo003-tau-2007-07-04-19-32-41-11432 heard from hongo016 : ssh:
connect to host hongo016 port 22: No route to host
failed  : hongo016 <- hongo003-tau-2007-07-04-19-32-41-11432
2 failed logins:
 hongo014
 hongo016
hongo000:tau[28/28/28]$
```

hongo003-… issued **ssh** to reach **hongo014**, and **ssh** said this

gxp decided login from **hongo003-…** to **hongo014** failed

list of targets that failed

Tips for troubleshooting

- explore transparently displays whatever the underlying remote-exec command (e.g., **`ssh`**) said
- it also shows which host attempted and failed to reach a target
  - to diagnose, you can login the source host and issue **`ssh`** etc. manually there

Common reasons for SSH to fail

- The machine is dead or does not exist
  - “No route to host” or timeout
- Wrong authentication settings
  - “Permission denied”
- Firewall blocks connections
  - **`gxpc`** gives up after timeout
- Other network-related errors (DNS lookup failures, etc.)
  - an error message from **`ssh`**

Diagnosing SSH failures

- For all error types described so far, diagnose by issuing **`ssh`** manually

GXP “unreachable targets” error

```
hongo000:tau$ gxpc explore hongo[[000-004]]
gxpc: no daemon found, create one
4 unreachable targets:
 Use `use' command to specify how.
 Or, consider specifying --children_hard_limit N to
increase the maximum number of children of a single host.
e.g., explore --children_hard_limit 50 ....
 hongo001
 hongo002
 hongo003
 hongo004
hongo000:tau[1/1/1]$
```

GXP does not know how to reach them

it makes sense because the daemon has just brought up and you have not told anything to it

“unreachable targets” (1)

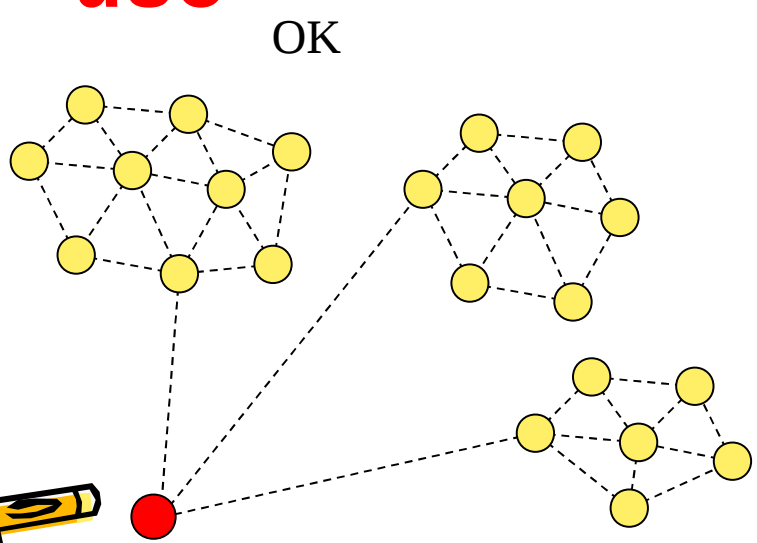
- Most often, you simply forgot to issue necessary **use** commands
  - note: when you quit gxp, it forgets the settings
- To see what you told to gxp, run **use** without arguments

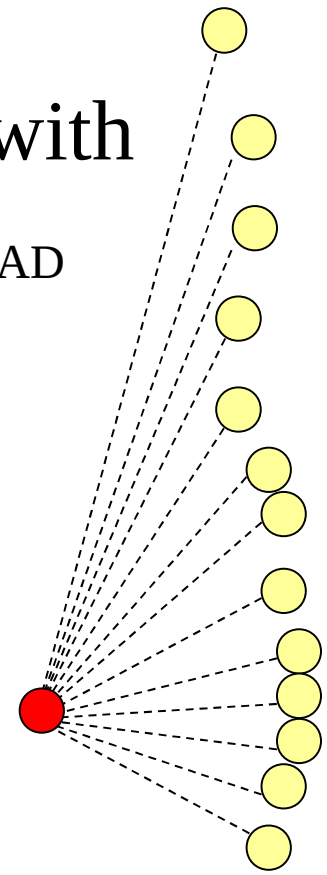
```
hongo000:tau[1/1/1]$ gxpc use ssh `hostname`
chiba000
hongo000:tau[1/1/1]$ gxpc use ssh chiba
hongo000:tau[1/1/1]$ gxpc use ssh hongo
hongo000:tau[1/1/1]$ gxpc use
0 : use ssh hongo hongo
1 : use ssh chiba chiba
2 : use ssh hongo000 chiba000
hongo000:tau[1/1/1]$
```

“unreachable targets” (2)

- The other way for this to happen is gxp tries to avoid making a host with too many children

⇒rule of thumb: make dense graphs with **use**





--children_hard_limit option

- explore has an option

```
--children_hard_limit N
```

- to set the maximum number of children of a single host to *N* (default = 40)

```
hongo000:tau$ gxpc use ssh hongo000 hongo
gxpc: no daemon found, create one
hongo000:tau[1/1/1]$ gxpc explore --children_hard_limit 3 hongo[[000-009]]
reached : hongo001
reached : hongo003
reached : hongo002
6 unreachable targets:
 Use `use' command to specify how.
 Or, consider specifying --children_hard_limit N to increase the maximum
number of children of a single host. e.g., explore --children_hard_limit 50
....
 hongo004
 hongo005
 hongo006
 hongo007
 hongo008
 hongo009
hongo000:tau[4/4/4]$ gxpc explore --children_hard_limit 10 hongo[[000-009]]
reached : hongo004
reached : hongo005
reached : hongo007
reached : hongo006
reached : hongo008
reached : hongo009
hongo000:tau[10/10/10]$
```

Other useful explore parameters

```
$ gxpc help explore
```

will show you all options

- **`--timeout`** *S* (seconds) : set time to wait until a host is considered to be dead (default = 15.0)
- **`--verbosity`** *N* (*N* = 0, 1, or 2) : controls verbosity of explore (default = 0)
- **`--show_settings`** : only show current explore parameters and quit

Explore with batch scheduler

- GXP can use batch scheduler's job submission command (e.g., **`qsub`**) instead of **`ssh`**
  - More precisely, GXP accompanies a tool called "**`qsub_wrap`**," which mimics ssh-like interactive behavior on top of **`qsub`**
- Different batch schedulers (Sun Grid Engine, Torque, etc.) have different flavors
- GXP currently supports **`torque`**, **`sge`**, and **`n1ge`**.

Explore with torque

say **`torque`** instead of **`ssh`**

```
hongo000:tau[1/1/1]$ gxpc use torque hongo
hongo000:tau[1/1/1]$ gxpc explore hongo 30
reached hongo
reached hongo
  …
reached hongo
hongo000:tau[31/31/31]$
```

say **`explore`** the same target (**`hongo`**) many times and let the scheduler select hosts for you

Time to explore

Remote installation

Recommended settings

- Before moving on, set aliases for frequently used **gxpc** commands. In bash,

```
alias e='gxpc e'
alias mw='gxpc mw'
alias smask='gxpc smask'
alias rmask='gxpc rmask'
alias savemask='gxpc savemask'
alias explore='gxpc explore'
alias use='gxpc use'
```

Playing with many hosts

```
$ gxpc e shell_command
```

normally runs "*shell_command*" on all hosts

```
hongo000:tau[10/10/10]$ gxpc e hostname
hongo000
hongo003
 …
hongo005
hongo004
hongo000:tau[10/10/10]$
```

results from all hosts

Ctrl-C will cleanup

- Pressing Ctrl-C while an e command is running (sending SIGINT to gxpc) will kill (send SIGINT to) processes run by the e command on all hosts

Meaning of gxpc prompt

```
hongo000:tau[A/B/C]$
```

the number of daemons in the tree

the number of daemons currently selected for execution by default (next command will run on this number of hosts by default)

the number of daemons that finished the last command successfully (with exit status = 0)

Meaning of gxpc prompt

```
hongo000:tau[10/10/10]% gxpc e 'hostname | grep hongo000'
hongo000
hongo000:tau[1/10/10]%
```

'hostname | grep hongo000' succeeded only on one host (**hongo000**)

Selecting hosts to run commands on

- Three ways
  - by names
    - straightforward and convenient when it fits
  - by exit status of commands
    - powerful but needs an extra step
  - by saved masks

Selecting hosts by names (1)

```
$ gxpc e -h regexp shell_command
```

runs *shell_command* on hosts matching *regexp*

runs *shell_command* on hosts not matching regexp

```
$ gxpc e -H regexp shell_command
```

Actually, matches are performed against globally unique identifiers of daemons

Selecting hosts by names (2)

```
hongo000:tau[10/10/10]% gxpc e -h hongo00[3-5] hostname
hongo004
hongo003
hongo005
hongo000:tau[3/10/10]% gxpc e -H hongo00[3-5] hostname
hongo000
hongo001
 ...
hongo006
hongo000:tau[7/10/10]%
```

- Note: **`hongo00[3-5]`** is a regular expression, not a special notation for numerical ranges

Selecting hosts by exit status

- You run some commands and then issue

```
$ gxpc smask
```

to set the hosts selected by default. To reset it to the all hosts, do

```
$ gxpc rmask name
```

set default exec mask

smask/rmask example

```
hongo000:tau[10/10/10]% gxpc e 'hostname | grep 003'
hongo003
hongo000:tau[1/10/10]% gxpc smask
hongo000:tau[1/1/10]% gxpc e hostname
hongo003
hongo000:tau[1/1/10]% gxpc rmask
hongo000:tau[10/10/10]% gxpc e hostname
hongo000
hongo001
...
hongo009
hongo007
hongo000:tau[10/10/10]%
```

Saving Exec Mask

```
$ gxpc savemask name
```

does the same thing as smask, and saves the exec mask to name for future use, as follows

```
$ gxpc e -m name …
```

savemask example

```
hongo000:tau[10/10/10]% gxpc e 'hostname | grep 003'
hongo003
hongo000:tau[1/10/10]% gxpc savemask h3
hongo000:tau[1/1/10]% gxpc rmask
hongo000:tau[10/10/10]% gxpc e hostname
hongo000
hongo001
...
hongo009
hongo007
hongo000:tau[10/10/10]% gxpc e –m h3 hostname
hongo003
```

A non-trivial example

- How to select one host from each group of hosts sharing the home directory?
  - You often need this to do things that modify your directory (one winner per each distinct directory)
  - You may do this based on hostnames, but if you don't know it, the following does the trick:

```
$ gxpc cd
$ gxpc e mkdir some_non_existing_dir_name
$ gxpc smask
```

Execution environment

- **$GXP_NUM_EXECS** : number of hosts selected for execution
- **$GXP_EXEC_IDX** : serial number (0, 1, …, $GXP_NUM_EXECS – 1) assigned to hosts that are selected for execution
- **$GXP_GUPID** : globally unique identifier of the daemon
- **$GXP_HOSTNAME** : hostname
- **$GXP_DIR** : where gxp is installed on that host

Execution Environment

```
hongo000:tau[1/1/1]$ gxpc e env | grep GXP
GXP_GUPID=etch-tau-2007-06-15-05-55-46-9200
GXP_DIR=/home/tau/proj/gxp3
GXP_EXEC_IDX=0
GXP_HOSTNAME=etch
GXP_NUM_EXECS=1
```

```
hongo000:tau[5/5/5]$ gxpc e echo '$GXP_EXEC_IDX'
0
1
3
2
4
hongo000:tau[5/5/5]$
```

e’s friends

- mw
  - similar to e, but connect processes in a particular way
  - useful to launch communicating processes
- ep
  - built on top of mw, it provids a straightforward way to do embarrassingly parallel tasks (detailed later)

Other shell things besides e

- The following two things do what you expect and affect the environment in which subsequent commands run

```
$ gxpc cd [directory]
```

changes directory on all daemons

```
$ gxpc export var=val
```

sets environment variable on all daemons

Note on cd/export

```
$ gxpc e cd directory
```

is different from

```
$ gxpc cd [directory]
```

Similarly,

```
$ gxpc e export var=val
```

is different from

```
$ gxpc export var=val
```

So they are built in

Common pitfalls

- You sometime need to be aware that your local shell (bash, tcsh, zsh, etc.) is in front of gxpc
  - your local shell does shell expansion ($VAR, ~/, etc.)
  - pipe and redirections

Pitfall examples (1) shell expansion

```
$ gxpc cd ~/tmp
```

does not work if your home directory is different among hosts. Instead use

```
$ gxpc cd '~/tmp'
```

Similar examples. Perhaps,

```
$ gxpc e $USER
$ gxpc e echo `hostname` `date`
```

will not be what you want to do (quote them)

Pitfall examples (2)
shell parsing priority

```
hongo000:tau[5/5/5]$ gxpc e hostname | grep 003
hongo003
hongo000:tau[5/5/5]$
```

```
hongo000:tau[5/5/5]$ gxpc e 'hostname | grep 003'
hongo003
hongo000:tau[1/5/5]$
```

ep

- EP : embarrassingly parallel
- gxp’s built-in command to do ep tasks straightforwardly

GXP ep command

```
$ gxpc ep [tasks_file]
```

- reads task descriptions from *tasks_file* (default: **`tasks`** in the current directory), and dispatch them to selected hosts
- you normally do something like

```
$ gxpc explore targets …
$ gxpc ep [tasks_file]
```

ep example

```
hongo000:tau[1/1/1]$ gxpc explore target ...
reached ...
# arrange execution environment (working dir etc.)
hongo000:tau[100/100/100]$ gxpc cd ...
# you somehow select hosts for execution
hongo000:tau[100/100/100]$ gxpc e ...
hongo000:tau[93/100/100]$ gxpc smask [or savemask name]
hongo000:tau[93/93/100]$ gxpc ep [-m name]
```

tasks_file format

- Each line contains:

taskname commandline

taskname can be any string, but cannot contain spaces or slashes (you normally name tasks after filenames to process)

- An example:

```
a zchaff a.cnf
b zchaff b.cnf
c zchaff c.cnf
```

Where are results?

- Status summary *of all tasks*

  ⇒**`status`** in the current dir *of the local host*

- output (stdout/stderr) of a task

  ⇒**`output/`***taskname*.{**`out,err`**} of the host that executed that task

  ∴ if hosts do not share **`output`** directory, outputs are spread across machines

Format of status file

- Each line is a record of a *finished* task

taskname status etime worker output_prefix finish_time

- *status* : exit status of the command (normally, 0 for "success")
  - *etime* : elapsed time
  - *worker* : name of the worker that executed the task (its prefix contains hostname)
  - *output_prefix* : file name of the task's stdout/err (append .out or .err to this)
  - *finish time* : human readable time the task finished at

status file example

```
hel 0 0.01 etch_0 output/helo Fri Jun 15 06:59:03 2007
una 0 0.01 etch_0 output/un Fri Jun 15 06:59:03 2007
hna 0 0.01 etch_0 output/hn Fri Jun 15 06:59:03 2007
usr 0 0.01 etch_0 output/usr Fri Jun 15 06:59:03 2007
upt 0 0.37 etch_0 output/up Fri Jun 15 06:59:04 2007
```

exit status (seems OK)

Simple fault tolerance

- If you press Ctrl-C while running an **ep** command, all executing tasks will terminate
- If you issue the same ep command again, tasks that have not finished will not run again
- More precisely, tasks that appear in **`status`** will not run again

screen is your friend

- When running long-running tasks, run ep from a stable (non-mobile) hosts and within a **`screen`** command

Watch out for NFS-write hogs

- If your tasks write a lot, make **`output`** directory local, though putting it under NFS is more convenient for accesses
- If you put **`output`** directory in a NFS-shared directory, keep watching the load avg of the NFS server (ganglia, VGXP, etc.)

NFS-read hogs

- If your tasks read a common large volume of data, consider copying them to all hosts beforehand
  - bcp is a reasonably fast file broadcast utility accompanying GXP
- But the choice is not as obvious as NFS-write hogs because
  - copying consumes more disk spaces
  - when data are not too large, caching may circumvent the problem automatically

How to choose execution hosts

- Remove misconfigured hosts
  - commands you need do not run anyhow
- Remove hosts you are not supposed to run heavy tasks on
  - file server etc.
  - nodes reserved for interactive tasks
- In some environments you need to find and choose lightly loaded hosts
  - nodefind

How to access outptut files conveniently

- GXP does not provide a particular mechanism for it
- There is no such a thing as universally-available-highly-reliable-scalable-globally-distributed-file system

Possibilities

- http server
  - seems a practical idea for manually watching task progress and copying files via wget
- fuse and sshfs
  - convenient for ad-hoc file sharing of file system semantics
- gfarm
- other projects under way

Accompanying tools

- bomb : kill all your processes
- bcp : file broadcast utility
- nodefind
- psfind

bomb

- kill all processes except GXP daemons and children
- the ultimate weapon to make sure you do not leave any process (so you won't be blamed)

```
hongo000:tau[100/100/100]$ gxpc e bomb
```

- (reality): we use bomb routinely, especially after you type Ctrl-C

bcp

- a convenient utility for 1 to *N* file copy (broadcast)

- Limitations (hopefully fixed soon…):
  - if firewall blocks connections to user-level ports, it may fail

    ⇒use this within a single cluster at a time (mw –h …)
  - no support for recursive directory copy

    ⇒use tar

```
hongo000:tau[100/100/100]$ gxpc mw bcp host:src_file tgt_file
```

nodefind

- Find hosts with various criteria (succeed on hosts that meet the criteria)
  - combine it with smask, savemask, etc. to choose hosts to run commands on

```
hongo000:tau[100/100/100]$ gxpc e nodefind
hongo000:tau[93/100/100]$ gxpc smask
hongo000:tau[93/93/100]$
```

by default, it succeeds on hosts whose 1 min. load average < 0.1 (very lightly loaded hosts)