UNIVERSITY OF HAWAII AT MĀNOA
Institute for Astrononmy

Pan-STARRS Project Management System

PanTasks Software Design Description
The IPP Scheduler and Controller system

Grant Award No. : **F29601-02-1-0268**
Prepared For : **IPP**
Prepared By : **Eugene Magnier**

Document No. : **PSDC-430-017**
Document Date : **January 22, 2006**
Revision : **DR**

DISTRIBUTION STATEMENT
Approved for Public Release – Distribution is Unlimited

Submitted By:

[Insert Signature Block of Authorized Developer Representative] Date

Approved By:

[Insert Signature Block of Customer Developer Representative] Date

Contents

1 Overview

This document discusses the design of PanTasks. PanTasks is the IPP tool which manages the sequencing of data analysis steps and, with the related tool 'PControl', distributes the data processing across a cluster of computers. **uses the 'opihi' shell-scripting system (TBD)**

The purpose of PanTasks is to manage the automatic construction and execution of inter-related (often repetative) operations. PanTasks uses a set of rules to define UNIX commands, and their corresponding command-line arguments, to be performed on some regular, repeated basis. The utility of PanTasks is that it can easily define and manage an analysis system which is completely state-based, as opposed to an event-driven system.

Consider, for example, a telescope which obtains a collection of images over the course of a night. Every minute or two, it takes an image and writes the image to some disk. An event-driven analysis system would involve having the telescope initiate a process at the end of the exposure. This process would perform an analysis, write some output, then send trigger another process. This type of operation works very well for a simple set up with reliable hardware. Such a system becomes more difficult to maintain when hardware failures occur or when multiple systems need to interact with each other. When failures occur, the triggering information (the events) is easily lost, thus some mechanisms are needed to detect these failures and either re-send the trigger or send an alternative failure-mode trigger. Or, if two systems need to interact, one or the other system must block for results from the first. Stopping and restarting such an analysis system is very delicate since the appropriate triggers must be set up some how, eg by noticing which images have not succeeded and restarting them at the appropriate stage. All of these types of methods of handling complexity and failures are essentially state-based rules. PanTasks allows the easy definition of a totally state-based analysis system.

In a state-based system, some mechanism examines the state of the system and decides which actions to perform based on the current state. In the illustration above, the mechanism could examine the images available (either by examining the disk or by examining the state of a data table) and decide to perform an operation based on what images are available. This makes it very easy to handle complexity and errors. If an analysis fails, the state either is not successfully updated or the error state is recorded, both situations being easy to detect and easy to handle. Restarting the system simply involves starting the state-monitoring mechanism. Combining results from multiple input sources simply involves watching for the multiple inputs to be available. PanTasks provides a mechanism to define state monitors, and to define the actions which are performed when those states occur. PanTasks action consist of initiating UNIX commands, where the arguments of those commands may depend on the results of the state tests.

1.1 Tasks vs Jobs

The two basic units of PanTasks operation are the 'task' and the 'job'. A 'job' is simply a command the user would execute on a command line; it consists of a command along with optional command line arguments. A 'task' is a generic description of a type of job in which the details of any specific piece of data are omitted. The task defines the UNIX command which corresponds to the job, and it provides rules for determining the identity of data for the job. The task also defines tests which are used to decide if the job may be executed. Finally, it defines a polling frequency in which PanTasks should attempt to construct a new job for the task.

For example, we may want to regularly copy files from one location to another. Perhaps the name of the next available file is available in a database table, and can be retrieved with the command 'nextFile'. Perhaps we wish to check for new files every 60 seconds. Thus, the task is to copy files, while a specific job of this task might be of the form `cp newfile newpath`. With PanTasks, we would define a copy task somewhat like the following:

```
task copyfile
  periods -exec 60.0
```

```
  task.exec
    $file = ‘nextFile‘
    if ($file == "none")
      break
    end
    command cp $file $newpath
  end

  task.exit 0
    queuepush copied $file
  end

  task.exit 1
    queuepush failure $file
  end
end
```

In this simple example, the task is attempted every 60 seconds. If there is no new file (output of ’nextFile’ is ’none’), then no job results from this task. If there is a new file, the copy command is performed. This example is deceptively simple because one could easily imagine writing a stand-along program which performs the same thing. The important advantages of using PanTasks for this type of operation are:

- the output from one job can be used to spawn a number of other tasks.
- the success or failure state of each job can be used to spawn other tasks.
- the details of the job and data test are kept separated from the rules which connect tasks together.
- the relationships between tasks are kept together in a single location.

In addition to these organizational advantages, PanTasks also provides a direct connection to the tool for monitoring parallel jobs, pcontrol. Thus the jobs spawned by PanTasks can be defined to run in the background locally or on any of the computers in the parallel processing cluster.

1.2 Parallel vs Local Job Processing

Jobs which are generated by PanTasks tasks may either be run locally (forked in the background on the same machine as PanTasks) or run on the IPP parallel process controller, pcontrol. The default is for the job to be run locally. If a job should be run on the parallel controller, this can be specified by including the command host (hostname) in the definition of a task. If the value of (hostname) is ’anyhost’, then pcontrol may select any of its host computers to run the job according to its own rules. If the value of (hostname) is one of the computers managed by pcontrol, then that machine will be selected for the job, if it is available. This amounts to a preference to use that machine, but pcontrol is allowed to substitute a different machine if it chooses. If the host command is given the option -required, then pcontrol is forced to use the named host, even if the machine is down, unknown, or otherwise unavailable. If the machine is not available, pcontrol will simply hold onto the job until the machine is available or the job is deleted. Note that PanTasks may delete jobs from pcontrol if they remain pending for too long.

It is possible to interact directly with the parallel processor to examine the current status, halt the parallel processor, etc. Commands to the parallel processor are defined under the controller command. The following controller commands are available:

- controller host (command) (hostname): Manage the parallel controller collection of hosts. This command can be used to add a new host, the delete one of the existing hosts, to turn a host on or off, and to check the status of a host
  - controller host add (hostname): add a new host.
  - controller host delete (hostname): delete a host.
  - controller host on (hostname): tell pcontrol that the host is on.
  - controller host off (hostname): tell pcontrol that the host is off.
  - controller host retry (hostname): tell pcontrol to retry the host connection.
  - controller host check (hostname): check the current status of a host.
- controller exit: stop controller execution.
- controller status: report controller current status.
- controller check: check job or host status.
- controller output: print accumulated messages from the controller.

It is also possible to specify a host for a task which has not been identified to the controller. If such a host is required, the controller will simply keep the associated jobs in the pending state until such a machine exists. See the pcontrol section below for further discussion of the controller manipuation of jobs and hosts.

1.3 Task Restrictions

Tasks may have restrictions on when they create jobs and how frequently they create jobs. The task command trange is used to specify a valid or invalid time range for a task. A valid time range limits the task evaluation to that time period. An invalid time range excludes task evaluation from the time period. Any number of time range restrictions may be defined, and the union of all restrictions will define if a job may be created. By default, the time range is an inclusive time range: the task is evaluated only if the current time falls within the specified time range. Alternatively, if the -exclude flag is given, the time range is exclusive, in which case the task is not evaluated if the current time falls within this range.

The time range may be given as a range of absolute dates as follows:

```
trange YYYY/MM/DD,HH:MM:SS YYYY/MM/DD,HH:MM:SS
```

where the two dates specify the start and end of the time range. In either of these date representations, the least-significant elements of the date and time may be dropped, defaulting to 00 (in the case of hours, minutes, and seconds) or 01 (in the case of day and months). Rather than specifying an end date, it is also valid to specify a time interval from the starting date. The time interval is specified as a number followed by a unit indicated by a single letter: d (days), h (hours), m (minutes), s (seconds).

The time range may also be specified as a repeated period of time, either as a time of day or a day and time of week. In the first case, the time range is specified as follows:

```
trange HH:MM:SS HH:MM:SS
```

where again the least-significant elements may be dropped and default to 00. This type of restriction defines a time range which is valid every day. The alternative is to specify a time range within the week, in the following form:

```
trange DAY@HH:MM:SS DAY@HH:MM:SS
```

where the value of DAY may take on any of the three letter day-of-week names (Sun, Mon, Tue, etc). This restriction specifies a start and end time within a week which is evaluated for each week.

Below are several examples of valid time range restrictions

```
trange 2005/01/01 2005/12/31   (only run during 2005!)
trange 18:00 00:00             (only run from 6pm until midnight)
trange 00:00 06:00             (only run from midnight until 6am)
trange Mon@08:00 Fri@17:00     (only run between Mon morning and Fri afternoon)
trange -exclude 12:00 13:00    (skip 1 hour from noon)
```

Note that the current definition of trange does not include time zone information. This means that all times are relative to UT. This should be addressed by adding a timezone environment variable to PanTasks and by allowing the trange to define a timezone offset.

It is also possible to restrict the total number of jobs which are spawned for a given task. This is done with the nmax command, which is given as part of the task definition. Once a task has constructed nmax jobs, it stops task evaluation. It is possible to redefine the value of nmax at any time by redefining the task. Any time the task is redefined, the new values for any task concept will override the existing values for the task concept.

1.4 Inter-Task and Inter-Job Communications

There are several ways in which the results of jobs may be used to influence other jobs. These include:

- external communications
- job exit status
- job stdout/stderr parsing

It is always possible for the interprocess communication to be performed externally: all jobs may simply write results to an external data source which is queried as part of the task evaluation. PanTasks may interact with UNIX programs using Opihi system interaction functions. These interaction methods include: the backticks for setting Opihi variables:

```
$variable = ‘UNIX Command‘
```

The exec command (which executes a UNIX command) and the backticks both receive the UNIX command exit status, setting the variable $STATUS. It is also possible to set a variable list to the output of a UNIX command:

```
list var -x "UNIX Command"
```

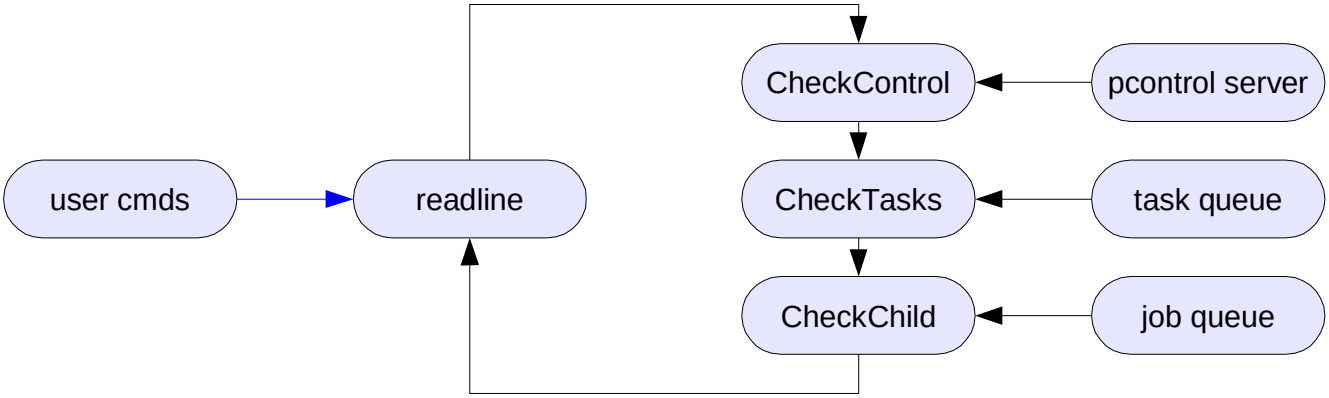

Figure 1: PanTasks Monitor Loop

In this last case, the values `$var:0 - $var:N-1` are set to the value of the stdout lines from the UNIX command, and the value `$var:n` is set to the number of output lines.

Fine-grained control over the job exit status is available with the task.exit macro command. This allows a task to define an exit macro which is performed for different exit status conditions. The argument to the task.exit command is the exit status value which triggers the macro. This may consist of any valid numeric exit status value (0-255). It may also have the value crash, in which case the macro is executed if the program exited as a result of a signal (ie, segmentation fault, etc). Finally, if may have the value default, in which case, the macro is run if no other macro describes the exit status.

Jobs may transmit their results back to PanTasks for further evaluation through the standard output and standard error streams. Whenever a job exits, the complete stdout and stderr streams from the job are pushed onto the PanTasks queues `stdout` and `stderr`. The job exit macros may then parse these queues, moving the results into other PanTasks / Opihi data containers (queues, variables, vectors, whatever is appropriate). Note that currently, the output data is simply pushed onto these output queues. It is currently the responsibility of the PanTasks programmer to use or dispose of the data in these queues. This may change in the future: the queues may be flushed for each job completion.

1.5 PanTasks Monitoring Loop

Figure 1 illustrates the operation of PanTasks. Currently, PanTasks is run as a stand-alone foreground process, not a server. In this current operating mode, PanTasks accepts commands from the user (left side) via a GNU readline interface. Readline provides a mechanism to schedule an background process which is executed on a regular basis, or in the interval after each keystroke. This background processing is represented as the loop on the right. In this loop, PanTasks checks on the status of pcontrol and on any locally spawned background jobs. It also checks the list of tasks for tasks which are ready to be executed. The tasks and the background jobs are kept in rotating queues. Every time the loop is processed, PanTasks runs through a number of the background jobs and tasks, attempting to evaluate as many as possible, but stopping after a limited number of milliseconds. This test stage cycles through the tasks and/or jobs in their queue, but stops if it examines all of the entries in the queue. If this testing process runs out of time before the entire queue is searched, it leaves the sequence intact for the next pass. The timeout is set to keep the keyboard small enough to keep the human interaction from being troublesome.

Figure 2 shows the interactions performed for each check of the list of tasks. The task timers are examined to see if the specific task is ready to be executed. If so, then the task exec macro is executed. If this macro exit status is false, the loop goes to the next task. If the task exec macro is true, the job command is constructed and the job is submitted to the correct location, either the controller (pcontrol) or to the queue of local, background jobs.

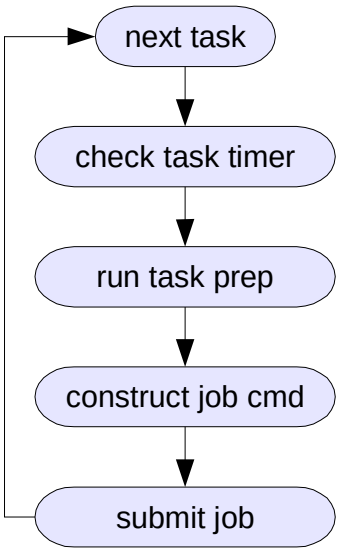

Figure 2: PanTasks Task Check Loop

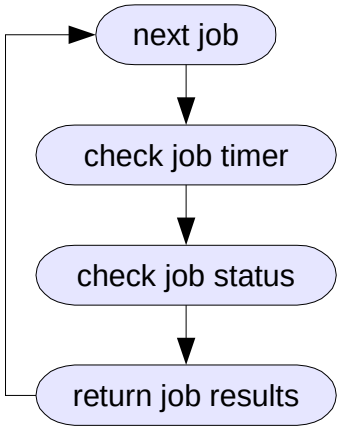

Figure 3: PanTasks Job Check Loop

Figure 3 shows the interactions performed for each check of a single background job. The job timer is checked to see if the job has timed out. Next, the job run status is examined to see if the job has finished or not. If the job has finished, the appropriate exit macro is executed and the job results are returned to the rest of the PanTasks data structures (ie, stderr and stdout queues are filled out). Users should not define exit macros which perform long running jobs, or PanTasks will become quite sluggish to keyboard strokes.

PanTasks also checks the current status of pcontrol on each loop. In this test, PanTasks requests from pcontrol a list of the jobs which have finished. It then attempts to harvest the finished jobs one by one and place the results in the approrpiate locations. This loop is also limited to a fixed amount of time; any pcontrol jobs which remain unharvested are left for the next pass by PanTasks.

1.6 Running the scheduler

Once a set of tasks has been defined, the scheduler can be started. The scheduler will run in the background, at regular intervals examining the collection of tasks and jobs. In these periods, the scheduler attempts to construct new jobs and checks on the status of jobs which may have finished, either locally or on the controller. To start the scheduler, give the command run. To stop the scheduler, given the command stop. The current status of the scheduler, controller, and any jobs which have been spawned are listed with the status command.

It is also possible to kill or delete individual jobs by hand with the commands kill (jobID) or delete (jobID). Other features

1.7 PanTasks Command Summary

```
controller                -- controller commands
task                      -- define a schedulable task
host                      -- define host machine for a task
nmax                      -- define maximum number of jobs for a task
trange                    -- define valid/invalid time periods for a task
task.exit                 -- define exit macros for a task
task.exec                 -- define pre-exec macro for a task
command                   -- define executed command for a task
periods                   -- define time scales for a task
run                       -- run the scheduler
stop                      -- stop the scheduler
pulse                     -- set the scheduler update period
status                    -- get system status
kill                      -- kill job
delete                    -- delete job
verbose                   -- set/toggle verbose mode
```

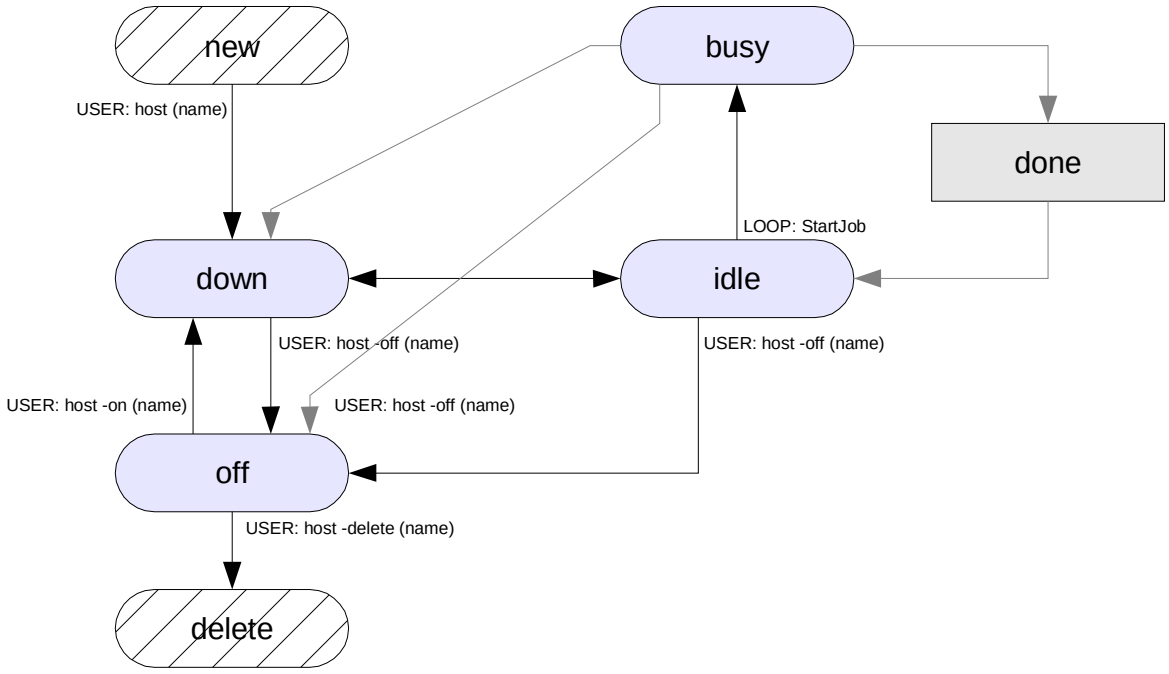

Figure 4: PanTasks Host States

2 pcontrol : the PanTasks parallel controller

The IPP uses a group of computers to store and process images and to manipulate collections of detections. These computers perform any of a large number of analysis stages or other processing tasks without significant interprocess communication. It is necessary to have a mechanism which initiates computing tasks on the different computers, which monitors the tasks as they are executed, which handles the output and the errors from these tasks, and which reacts to the failure of any of the computing nodes. The system responsible for the tasks in the IPP is pcontrol.

2.1 Host States

pcontrol maintains a table of available processing computers (hosts) and tracks their status. Hosts managed by pcontrol are allowed to be in one of several states: `off`, `down`, `idle`, `busy`, and `done` (see Figure 4). There are also two virtual states: `new` and `delete`. These states have the following meanings:

If the host is `off`, it is known to pcontrol, but pcontrol does not have an active connection to the machine. Hosts which are `off` are not available for jobs, and pcontrol does not attempt to initiate a connection to them. A pcontrol user may force a host to transition to the `off` state with the command host `off~(hostname)`. (Note that this command will set only one of the connections to the named host to `off`. If multiple connections to a machine have been defined, multiple `off` commands must be sent).

When pcontrol is told to consider a machine on, the machine is moved from the `off` state to the `down` state. Pcontrol attempts to initiate a connection to the host. Connections are made by running a remote client on the host, using the specified connection method. The connection method may be ssh, rsh, or an equivalent remote shell connection. The choice is specified by the `COMMAND` Opihi variable. The remote connection starts a dedicated remote client which must accept the pcontrol client commands and respond appropriately. The provided remote client is called *pclient*, though in

principal other equivalent programs could be used by setting the Opihi variable `SHELL`. This feature more generally allows a user to specify a path to the remote client, if it is not in the user's path.

If the remote connection is successful, the connected host is moved by pcontrol from the `down` state to the `idle` state. If the connection is unsuccessful, pcontrol will try again after a certain period of time. If the connection continues to be unsuccessful, the retry period is doubled for each successiver connection attempt. If the user wants to force pcontrol to retry the connection to a machine (if, for example, the timeout is now very long, but the user knows the machine's ethernet cable has been re-inserted...), this can be achieved with the command host `retry (hostname)`. A host which is `down` is in the limbo state between `off` and `idle`.

Once pcontrol has made a successful connection to the host, the host is in the `idle` state. At this point, it is ready to accept jobs from pcontrol for execution. Pcontrol periodically queries the hosts to check that they are still alive. If a host is discovered to be unresponsive, and particularly if the remote pipe connection has closed, then the machine is moved back to the `down` state.

Hosts which are `idle` may accept a job from pcontrol. A job simply consists of a bare UNIX command, without redirection of standard input or standard output. The host will initiate the job, and pcontrol will place the host into the `busy` state. The remote client, pclient, runs the job in the background and will continue to accept input from pcontrol. pcontrol will continue to check the status of the host, and now also the status of the specific job. As before, if the connection breaks, pcontrol will migrate the host to the `down` state. Any job already initiated on a host which goes down will be returned to the pool of unexecuted jobs for later processing, so the job will not be lost.

When the job exits, pclient tells pcontrol that the job is completed, and specifies the exit status. At this point, pcontrol will move the host from `busy` to `done` state. It will stay in this state until pcontrol can determine the ending conditions and reset the remote client. pcontrol requests the standard error and standard output from the job from pclient. pcontrol stores this data with its information about the completed job, and send a reset command to the remote client. Once these cleanup tasks are successfully completed, pcontrol will move the host to the `idle` state, ready for further jobs.

Each physical computer may have multiple processors. pcontrol treats each processor independently. It is up to the system configuration if each computer needs to reserve one of its CPUs to manage background tasks or if pcontrol should attempt to send one task per CPU and let the operating system handle the I/O load. some of this behavior will probably be eventually more intelligent. For example, the commands which turn a host on or off should be able to do the same operation to all host connections for the same machine name.

A machine may be completely removed from pcontrol's host tables with the command host delete (hostname).

2.2 Jobs

The pcontrol accepts new jobs with the command `job ...`, in which the ellipsis represents the command and arguments of a valid UNIX command. The commands are run under `sh`, and are executed in the user's home directory. Users should be wary of the conditions under which the remote jobs are run. If the nodes in question all cross-mount the same home directories, multiple jobs which interact with the same named file may produce unexpected results. The controller cannot enforce good behavior on the part of the remote jobs; it is the responsibility of the user to ensure that conflicts do not arise by, eg, always using unique output file names.

Other issues may arise from the fact that pcontrol may be choosing any of the hosts to run the job. Typical failures arise if the user does not realize that specific jobs do not behave the same on all machines, or if a necessary resource (eg, some input data file) is only available or accessible from some of the hosts. It is also the responsibility of the task to wait for network lags (ie, NFS delays).

pcontrol gives each task a unique internal identifier (Job ID) equivalent to the process ID used in UNIX. When a job is

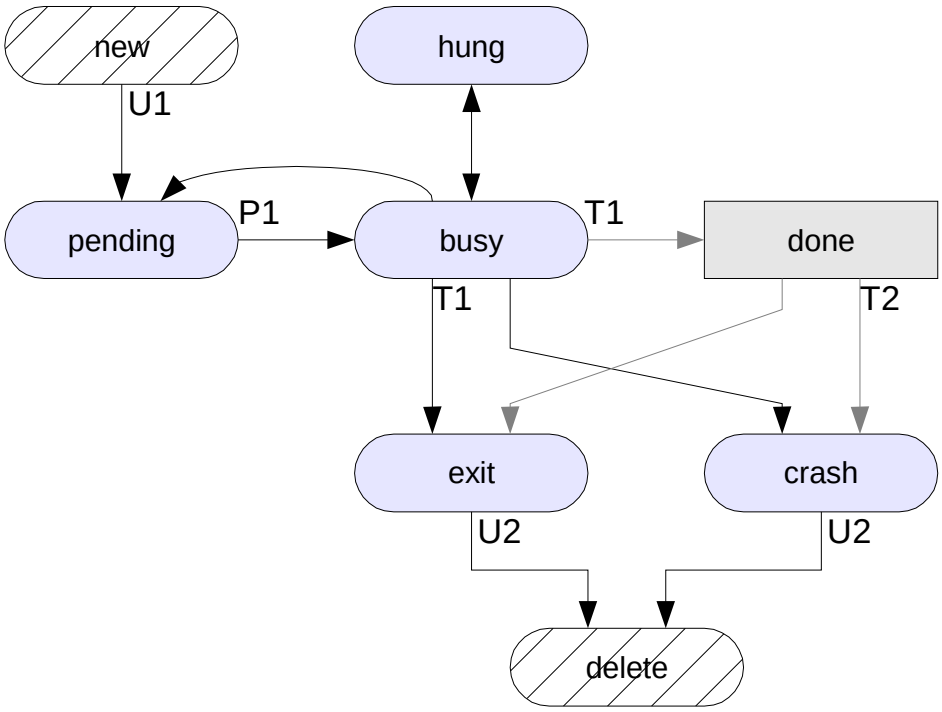

Figure 5: pcontrol job states. Transitions labeled Ux are issued by the pcontrol user (including PanTasks). Transitions labeled Px are initiated by pcontrol. Transitions labeled Tx are the result of the job completion

submitted to pcontrol, the command echoes back the Job ID. This ID may be used by other pcontrol commands to obtain information about or interact with the job. For example, PanTasks uses the Job ID to examine specific jobs.

A job may specify a specific host for the task execution. The host specified for a job may be required, or desired. In the first case, pcontrol, will only run the job on the specified host, waiting until it is available before attempting the job. In the second case, pcontrol will attempt to send the job to the specified host, but if the host is unavailable (how long? what conditions?), pcontrol will allow the job to be sent to an alternative host. pcontrol attempts to honor the requests for required and desired hosts, giving priority first to required-host jobs, then to the desired-host jobs, and finally to all other jobs. To specify a host for a job, the following commands are used:

```
job -host (command and arguments...)
job +host (command and arguments...)
```

The first case specifies a desired host, while the second specifies a required host. It is also possible to specify the special host name anyhost, which is equivalent to not specifying a host at all.

Job priority / urgency levels are not implemented at this time.

I/O vs CPU tasks are not currently distinguished by pcontrol

pcontrol stores the stdout and stderr for each completed job. To retrieve these data from these streams, the user issues the commands stdout (JobID) and stderr (JobID). The result is a single line specifying the number of bytes to expect, followed by a dump of the buffers, followed by the prompt. It is the user's responsibility to relieve pcontrol of this data load by deleting jobs once they are no longer needed. Job deletion is performed with the command delete (JobID).

Jobs are moved between the following states by pcontrol (see Figure **??**):

- pending: the job has not yet been executed.
- busy: the job is currently being executed.
- done: the job has completed, but the stdout/stderr has not been processed by pcontrol.
- exit: the job has completed with a valid exit status
- crash: the job has completed with a crash status (exit on signal).

2.3 Miscellaneous Commands

It is possible to check the status of a single host or job with the user command `check`.

pcontrol continuously examines the stack of jobs, adjusting their state as needed and extracting their output when it is ready. These checks are performed in the background, with pcontrol ready to accept further commands from the user in the foreground. These checks are performed after every keystroke, and also after an inactivity timeout. The interrupt interval defaults to 1 second, but may be adjusted with the pulse command, which takes as an argument, the number of microseconds for the timeout.

The pcontrol system status may be examined with the command status. This provides a dump of the job stacks and the host stacks.

It is possible to list the jobs currently in a specific stack, corresponding to the list of jobs with a given state. This is done with the command jobstack (stackname). The valid stack names are pending, busy, exit, crash, and done. The result is a list of all jobs on the specified stack. This is useful to determine quickly which jobs have exited or crashed.

A specific job may be killed with the command `kill (JobID)`. This command is only valid for a job in the busy state. Any job in the pending, exit, or crash state may be deleted with the `delete (JobID)` command. This is necessary to free the memory associated with the job and its output streams. PanTasks automatically performs these job harvesting functions.

The command `verbose (mode)` turns the verbosity of the pcontrol operations on or off.

The pcontrol and Nebulous have related needs for information from the combined storage-and-processing nodes regarding which nodes are available. Currently, the two systems independently examine the hardware to judge the availability. It is not yet clear if this information is best stored in a single location (either pcontrol or Nebulous), which provides the information to other systems on demand. The current implementation allows the IPP system as a whole to distinguih nodes which are available for processing from those that are available to serve data as part of Nebulous.

Figure 6 illustrated the pcontrol job monitor loop. This is very similar to the loop in PanTasks. A background process launched by readline during idle periods cycles through the list of hosts (children) and migrates jobs to and from them as needed.

2.4 pcontrol Command Summary

```
check               -- get job or host status
delete              -- delete job
host                -- add / delete / modify host
job                 -- add job
jobstack            -- list jobs for a single stack
kill                -- kill job
pulse               -- set system pulse
```

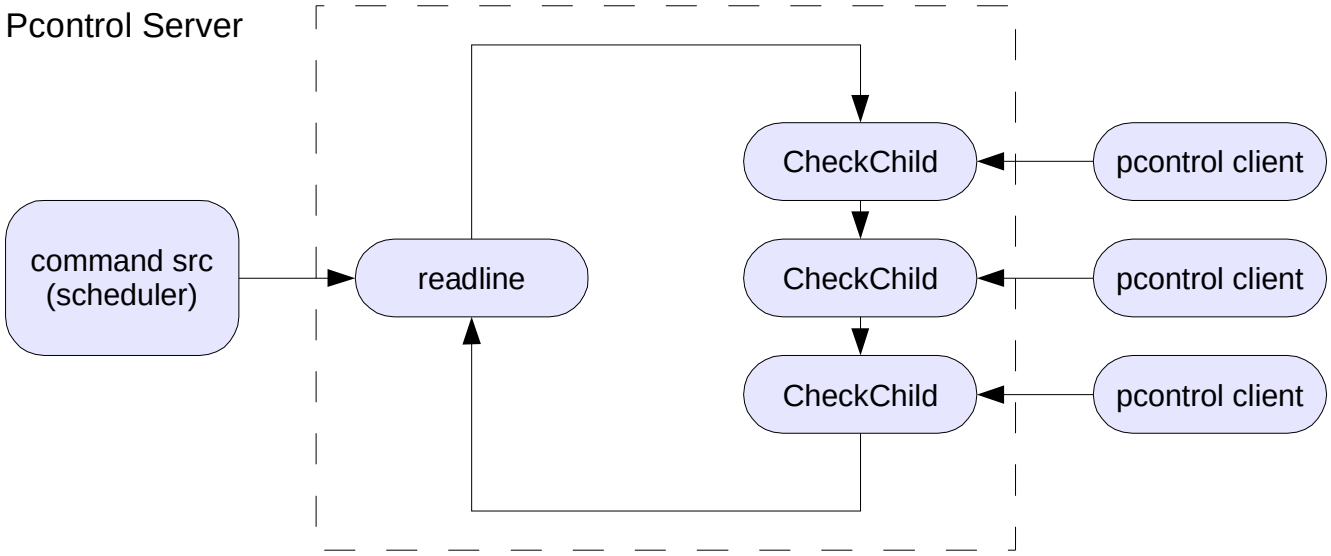

Figure 6: PControl Job Monitor Loop

```
status              -- get system status
stderr              -- get stderr buffer for job
stdout              -- get stdout buffer for job
verbose             -- set the verbose mode for job
```

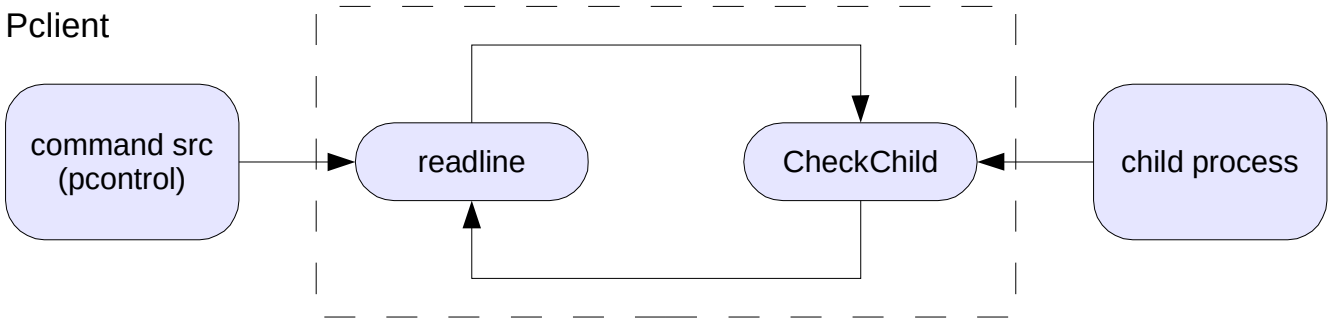

Figure 7: PanTasks queues and MDDB tables

3 pclient

pclient is the remote process monitor for pcontrol, the parallel process controller.

The program pclient is used to support the remote jobs which are run on the remote hosts by pcontrol. The concept of pclient is to act as a buffer between the job running on the remote host and pcontrol. The pcontrol design uses (by default) ssh connections initiated by pcontrol to the remote hosts. These connections execute the remote program of pclient. The use of a remote login process lets the UNIX system take care of the user authentication issues. In this case, the recommended practice is to set up ssh to allow the connection to the remote host without additional authentication using the appropriate authorized keys. The security is maintained by the security of ssh logins to the computers used by PanTask and pcontrol.

It is convenient to keep a continuous connection to the remote hosts. This avoids incurring the overhead of authentication for each command which is executed, while keeping a high-quality user authentication process in place.

pclient acts as a buffer between pcontrol and the remote background process, allowing the continuous connection to remain viable without samping pcontrol with output from the jobs.

4 Command Summary

pclient has a very limited command set, as follows:

```
job           : start the job (UNIX command) in the background.
check         : return the current job status
status        : return the current job status (?)
stdout        : dump the stdout stream accumulated from the job
                back to the calling program.
stderr        : dump the stderr stream accumulated from the job
                back to the calling program.
reset         : kill (if needed) the job and reset to accept
                another job.
```

A Example : Pcopy.pro

Below is an example script for psched which demonstrates the scheduling system. This parallel-copying script implements the Pan-STARRS image copying system, which requests images from the summit and copies them to the appropriate computer. The first task in the script queries an external system for new image names with the function new.images. In the case of Pan-STARRS, this would be a request from OTIS, the observatory controlling system. The second task initiates the individual image copies, with separate CCDs being copied to separate computers. This script uses the concept of having specific machines assigned to specific CCDs (as Pan-STARRS intends to operate). The association is determined by calling the external function chip.host, providing the identifier of the chip in question. This returns an appropriate host. The copy.image function copies the file and also sends a message to the summit system to inform it that the image has been successfully copied.

```
verbose on
 queueinit newImages
 exec echo 0 > new.last
 exec cp -f raw.list new.list

 controller host add po01
 controller host add po02
 controller host add po03
 controller host add po04

 # identify the images ready for copy
 # new entries are added to queue newImages
 # need to compare the new list with the ones already being processed
 task        new.images
   command      new.images
   host         local

   periods      -poll 1
   periods      -exec 5
   periods      -timeout 5

   # success
   task.exit    0
     local i j Nstdout Nimages
     # compare output with new.image queue
     # keep only new entries
     queuesize stdout -var Nstdout
     for i 0 $Nstdout
       queuepop stdout -var line
       queuepush newImages -uniq -key 0 "$line"
     end
   end

   # locked list
   task.exit    1
     echo       "new.images: exec failure"
     $new.image.failure ++
   end

   # default exit status
   task.exit    default
     echo       "new.images: unknown exit status: $EXIT"
     $new.image.failure ++
   end

   # operation times out?
   task.exit    timeout
     echo       "new.images: timeout"
     $new.image.failure ++
   end
```

```
end

# copy new images, sending job to desired host
task        copy.images
  periods      -poll 0.2
  periods      -exec 1
  periods      -timeout 5

  task.exec
    queuesize  newImages -var N
    if ($N == 0) break
    # if ($network == 0) break
    # if ($filesystem == 1) break

    queuepop newImages -var line
    list tmp -split $line
    $filename   = $tmp:0
    $chip       = $tmp:1
    $state      = $tmp:2
    if ($state == new)
      # copy this image
      queuepush newImages -replace -key 0 "$filename $chip run"
    else
      # ignore this image
      queuepush newImages -replace -key 0 "$filename $chip $state"
      break
    end
    # echo $chip
    $host = `chip.host $chip`
    # echo $host
    host $host
    # echo "starting copy for $filename on $host..."
    command copy.image $filename $chip
  end

  # can I have access to argc,argv?

  # success
  task.exit    0
    echo "done copy..."
    queuepop stdout -var line
    list tmp -split $line
    $filename   = $tmp:0
    $chip       = $tmp:1
    exec mark.image $filename
    queuepush newImages -replace -key 0 "$filename $chip copy"
  end

  # default exit status
  task.exit    default
    echo       "new.images: unknown exit status: $EXIT"
    $new.image.failure ++
  end

  # operation times out?
  task.exit    timeout
    echo       "new.images: timeout"
    $new.image.failure ++
  end
end
```