Job Scripting

Job Specification vs Job Authoring in Python

NOTE See the Python Job Authoring API discussion for details on creating and spooling jobs from your own Python scripts.

For simple single-command jobs such as rendering a network mounted RIB file, you can also use tractor-spool command-line options to create the job for you:

```
tractor-spool -c prman -Progress /net/myfiles/frame1.rib
or
tractor-spool -c /usr/bin/printenv
```

The section below describes a *different* concept: The format and syntax of the "classic" job files used to statically define a single job's structure, as *a data file.* Job files like this are usually written by other programs and then spooled into the Tractor job queue using the tractor-spool program.

Static Job Specification Synopsis

Tractor Job files are typically represented as TCL scripts using special job definition operators recognized by the Tractor job parser. This format was first introduced for the Alfred system, and Tractor remains compatible with nearly all Alfred constructs.

Tractor also supports an evolving set of JSON job specification formats, described in a separate document.

Tractor job input scripts (in alfscript format) are composed of expressions using the following Tcl operators:

```
Job [options]

Task {title} [options]

RemoteCmd {launch_expr} [options]

Cmd {launch_expr} [options]

Instance {task_title_to_reference}

Iterate varname -from n -to m -by i -template {script} [options]

Assign varname {value_string}
```

See the Operator Details for detailed descriptions of options and operator syntax.

Introduction

A job script describes the work to be dispatched and monitored by Tractor (or Alfred). Job scripts are typically created by application programs as they spool new rendering jobs. That is: the following syntax discussion will probably be useful only to people developing Tractor-compliant applications.

Job scripts support a simple hierarchical structuring scheme that can encode dependencies that, in turn, determine the order of task execution; job scripts are somewhat similar to *makefiles* in this regard. The predefined job script operators are: Task, Cmd, andInstance. There are also some special conventions for embedding certain run-time values into command expressions. As the job script is parsed during job initialization, Tractor constructs both the internal execution hierarchy for the engine/blade as well as the display hierarchy for the Dashboard. The engine then continuously traverses this internal representation looking for opportunities to launch new work.

The expression *syntax* consists of some simple list-formation rules. Specifically, each operator accepts a list of argument strings, some of which may be optional. Strings are delimited by either double-quotes or curly-braces; they may contain newlines, and they are also allowed to be empty. Case is usually important, while whitespace is generally not. Newlines and semicolons, when not in strings, are end-of-expression markers. Backslash is used (as in C) to produce literal versions of syntactically significant characters and other special character codes (and to disable newline processing). Leading hash-signs (#) introduce one-line comments. See the special characters discussion for more information.

Hierarchical nesting is accomplished by embedding child Task descriptions within the subtask field of the parent Task operator. It is only possible to create nested strings using the curly-brace delimiters.

A Simple Example

Here is a simple script that defines an example task hierarchy. Assume we have two shadow maps to compute, and a final frame that references them; in this case the shadow maps must be completed before rendering can begin on the final frame. The "reserved words" are *Job*, *Task*, and *RemoteCmd*, and these operators can take options, such as -subtasks that begin with a leading hyphen:

```
Job -title {A Simple Job} -subtasks {

    Task {Frame One} -subtasks {

        Task {Shadow A} -cmds {
            RemoteCmd {render light.1.rib} -service {PixarRender}
        }

        Task {Shadow B} -cmds {
            RemoteCmd {render light.2.rib} -service {PixarRender}
        }

    } -cmds {
        RemoteCmd {render beauty.1.rib} -service {PixarRender}
    }
}
```

This job script describes a two-level execution tree in which the task named *Scene One* will launch its command (render frame.1.rib) after the two tasks upon which it depends have completed successfully (the *Shadow*s). The nested task descriptions define the (depth-first) order of execution; hence, the render command for light.1.rib is launched first, then the one for light.2.rib follows. Then, when they are both complete, frame.1.rib is rendered. Commands are launched as separate sub-processes, in the order that they appear in the script.

Note that the two shadow commands are not dependent on each other, i.e. they aren't nested with respect to each other; hence, they represent a parallel processing opportunity. Given sufficient resources, Tractor will typically launch both commands and allow them to execute concurrently. Hence, the script defines a *launching order*, some commands may still be running when another is launched, and these concurrent commands may complete in any order. In no case, however, will a Task launch its own commands until *all* of its sub-tasks have completed their commands successfully.

The assumptions made in this example are:

- that there is an executable named *render* somewhere in the current search path.
- that the two shadow map images are created on disk where frame.1.rib expects to find them (i.e. file system references within those the RIB files are consistent).

Tractor will *not* attempt to confirm these sorts of assumptions, that is the domain of the job generator. If an error does occur, the task becomes blocked, the corresponding UI widget converts to the error state, and all tasks that depend upon the broken one will remain unexecuted (others that have no dependency on it may proceed however).

Things to note about Tasks:

- the title string will appear in the Dashboard, so it's handy to choose a brief but descriptive title.
- child tasks are listed in the -subtasks field; if there are no dependencies it can be empty or missing.
- the RemoteCmd list in the -cmds field describes the actual actions to take when the dependencies have been met; it can be empty or missing; it can also contain multiple (newline separated) RemoteCmds, which will be processed in sequence.

Process Launch and Tracking

Remote Servers, Launch Expressions, and Runtime Substitution

The *launch expression* argument to the **RemoteCmd** operators specifies the actual executables that Tractor should launch. As in the simple example above, these can be straightforward command strings that simply specify an executable and its arguments. Many times the named executable is just a locally written shell script.

Tractor, like Alfred, also provides other key dispatching features beyond hierarchical ordering of launches, most notably *locating* available remote servers. Consider this command that might be typed in a terminal:

```
prman -Progress /net/my_ribs/a.rib
```

If we would like Tractor to execute that same command on an available machine in our render farm, we can create a Tractor job like this:

```
Job -title {a render test} -subtasks {
  Task -title {render a.rib} -cmds {
    RemoteCmd {prman -Progress /net/my_ribs/a.rib} -service {PixarRender}
  }
}
```

Service Keys

The RemoteCmd option *-service {PixarRender}* in the example above is called a *service key expression* and it is used describe which **type** of blade is acceptable to run that command. In this example, the prman executable should be launched on blades that advertise the service key "PixarRender". Tractor simply treats these keys as "filters" when matching commands to blades. Matching ignores capitalization when comparing command keys to blade capabilities.

Having created a job script that specifies a particular service key, then you need to "attach" a matching service key to the appropriate blades on your farm, namely those on which you want commands of that type to run. For example, it might be important to select hosts on which "prman" is actually installed! For tractor, this means going to the blade.config file and adding the appropriate key to the "Provides" entries for each type of blade profile. There are defaults shared by all profiles at the top of the file.

The particular choice of keywords to use in your jobs is arbitrary, you just need to adopt conventions for job scripts and blade profiles that agree on what an advertised keyword implies in terms of a particular blade's capabilites. By convention, Pixar applications like *RenderMan for Maya* generate Tractor jobs that user certain service keywords, like PixarRender. These keywords are intended to express the type of sibling Pixar software required on the blades in order to run certain commands. Here is a brief list of some common service keys from RfM-generated jobs, and the requirements they are intended to express:

PixarRender	cmd requires RPS (prman) to be installed on the blades
PixarNRM	cmd requires RPS to be installed, and a running *netrender* server (tractor-blade.py is also a netrender server, so in practice this usually equivalent to PixarRender, but it helps distinguish netrender commands)
RfMRender	cmd requires an appropriate intalled Maya on the blade, plus RfM or RMS must also be installed; the cmd consumes a render license
RfMRibGen	cmd requires Maya and RfM/RMS, no render license needed

Note: in addition to the keywords specified in blade.config, each server blade always "provides" three additional implicit keywords: the blade's **host name,** t he blade's **IP address** (dotted quad), and the name of the blade.config **profile** that it is using. This feature is seldom used, but by specifying a profile or host name as the --service key for a RemoteCmd, scripts can restrict execution of a particular command to a specific host, or class of host matching a specific profile by name. Usually it is best to refer to the more abstract type of service keys.

There is also an "advanced" mode in tractor in which a site-defined python plug-in for the blade can dynamically change the services advertised by that blade on a request by request basis. The site plug-in might refer to an external production database, for example, when making these sorts of dynamic decisions.

Service Key Expressions

Service keywords can be combined using a simple syntax to further restrict the blades that match a given command. For example:

```
RemoteCmd {prman -Progress vast.rib} -service {PixarRender,BigIron}
```

In this example the command will only match blades that provide PixarRender **AND** BigIron as keywords in their profile.

Individual key names can also be preceeded with an exclamation point, like "!xyz" to indicate a negation, meaning that the given key must **NOT** be present in order for the blade to be acceptable. For example, to avoid blades that are providing a key named "Irix" you might have a job script containing "RemoteCmd -service {PixarRender,!Irix}". Or this same effect might be applied to every command in a job by adding that key to the top-level Job specification, or in the Dashboard job attributes editor; or it can also be given at spool time on the tractor-spool command line:

```
tractor-spool --service="\!Irix" myjob.alf
```

Service keys are usually site-defined abstract keywords such as those given in the "Provides" specification of blade profiles in blade.config. They can also be a specific hostname or the name of an entire profile. So you could avoid rendering on a host named "darth" by adding the service key "!darth" to the Job's service keys.

Service Key Expression Operators

Operator	Use	Example
keyname	keynames are replaced with 1 if they match a name in the blade's "Provides" list, or 0 if they do not	PixarRender
" *keynamePattern* **"**	like keynames, above, with additional support for pattern matching using the wildcard characters * and *?*	"rack-15?" || '192.168.0.*' (*Note the required double or single quotation marks*)
&& **,** *(comma)*	boolean AND	PixarRender && Linux PixarRender,NorthAnnex
||	boolean OR	Linux || macOS
!	boolean NOT	PixarRender && !Desktops
(*subexpr* **)**	parenthetical sub-expressions	PixarRender && (Linux || macOS)
@. *blade_metric*	numeric blade metric value	see table below
+ - * /	numeric add, subtract, multiply, divide	for blade metrics, see table below
< <= == != >= >	numeric comparison, less, greater, equal, etc	for blade metrics, see table below

Service Key Expression Blade Metrics

Metric	Meaning	Example

@.disk	available disk space, in gigabytes	PixarRender && @.disk > 5
@.mem	available physical RAM, in gigabytes	PixarRender && ((1024 * @.mem) > 2048)
@.nCPUs	number of CPU cores reported by the OS	Windows7_32bit && (@.nCPUs >= 4)
@.cpu	current CPU usage, normalized by nCPUs	@.cpu < .75
@.sa	number of abstract "slots" available	(@.sa > 2) && (PixarRender || PixarNRM)

(Note: boolean OR, parenthetical subexpressions and blade metrics expressions were first introduced in tractor-engine 1.5)

Unusual netrender-style client / server commands

Some commands, such as *netrender*, are client applications that execute on the **local** spooling host but which also require the *names* of one or more **remote** servers to be provided as part of their command-line options. Applications such as scp or ftp are similarly clients of remote servers. For example, consider a typical invocation of netrender from a terminal:

```
netrender -h antigone -h percival /net/my_ribs/a.rib
```

In this case, the single RIB file is parceled out to two rendering servers that are assumed to be available and already be running a netrender server (i.e. each has a running tractor-blade since that has built-in netrender support). The local netrender client application contacts each of hosts listed on its command line and initiates a rendering there, using custom netrender protocol. It is often desirable to have Tractor find available netrender servers from a pool, rather than typing in specific hostnames (such as antigone and percival above). An example job script for requesting this collection of servers might be:

```
Cmd {netrender %H /net/my_ribs/a.rib} -service {PixarNRM} -atleast 2
```

Note: It is important to remember that the Cmd directive does *not* actually send work to remote hosts selected in this manner. It launches the local application, such as netrender or ssh, which then manage the interactions with their remote server themselves. RemoteCmd should be used to send commands directly to a remote host with no local client.

Substitutions

The following symbols are expanded at launch time when they occur within launch or message expressions:

~	home directory expansion, as in csh(1)
%h	substitute the hostnames bound to the current Cmd via the dynamic -service mechanism. This is a simple blank-delimited list of hostnames (useful for rsh, etc).
%H	like %h, but formatted as -h hostname pairs (as required by netrender).
%n	converted to the count of bound slots; an integer indicating how many slots were bound to this command.
%j	expands to the internal dispatcher Job identifier (aka "jid") for the current job. Tractor engine creates unique ids for jobs in its queue.
%t	expands to the Task identifier (aka "tid") for the current task, it is unique only within each job.
%c	expands to the Command identifier (aka "cid") for the current command, it is unique only within each job.
%r	the recover mode, expands to 0 when a task is beginning a fresh start, and to an integer greater than zero when the user or system is requesting a recover from checkpoint
%R	expands to the "loop count" for commands that are being restarted due to the -resumewhile construct
%q	the quality hint, expands to the value 1.0 when a task is being executed due to final quality runs from the subtasks it depends upon; it will be less than 1.0 if some subtask only reached a checkpoint during the current resumewhile loop
% *idref* **(host)**	like %h but using the hostnames from the command whose *-id* value is *idref*.
% *idref* **(-host)**	as above, formatted as *-h hostname* pairs.
%D (*path*)	*DirMaps*, apply per-architecture remapping of paths using a site-defined mapping table.
%%	a single percent-sign is substituted.

Job Context Environment Variables

The following environment variables are set by tractor-blade just before it launches each command. They may be useful in some cases where the application itself needs to know some Tractor-related context information. See the Custom Environment Handlers discussion for details on controlling custom environment variables globally or on a per-project or per-command basis.

TR_ENV_JID	The job ID number for the job from which the command was dispatched.
TR_ENV_TID	The task ID number, within the job.
TR_ENV_CID	The command ID number, within the job.
TR_ENV_JOB_PROJECT	The project affiliation for the job. The name (or names) are specified at job submission time, usually by the job creation application.
TR_ENV_TRACTOR_ROOT	The path to the top of the Tractor install tree for the running blade.

Sharing One Server Check-Out Among Several Commands

Sometimes it can be useful to acquire a remote server and run several commands in sequence on that server. This is called a *shared server* scenario. The mechanism for accomplishing this goal is to add a server check-out specification to the enclosing *Task* and then reference the task-id on each *Cmd* or *RemoteCmd* that needs the server slot.

For example:

```
Task {SharedServer example} -id {bob} -service {pixarRender} -cmds {
      Cmd {rsh %h mkdir /tmp/somedir} -refersto {bob}
      Cmd {rsh %h render -Progress some.rib} -refersto {bob}
} -cleanup {
      # Clean-up commands go here.
      # This example uses RemoteCmd just to illustrate its use as an
      # alternative to rsh above.  Note that there is an implicit '%h'
      # used to determine where the command should be run.
      #
      RemoteCmd {/bin/rm -rf /tmp/somedir} -refersto bob
} -subtasks {
      # the description of any nested dependent tasks go here
      [...]
}
```

The lifetime of the check-out is governed by two factors. The initial task-level check-out is done lazily in the sense that it only occurs when one of the commands that references it actually becomes the next command to be executed. A reference count is used to determine when it is safe to check the slot back in, it is freed when the last command to reference it has completed or errored out.

Syntax Note: the characters allowed in the *idref* names are letters, numbers, period (.), and underbar(_). For all of the % substitutions above, braces may be used delimit names, as in csh or Tcl; for example, if the current schedule has a service of type "renderserver" defined for the host named "percival", then a Task containing this command:

```
RemoteCmd {renderer -use %{h}.settings} -service {renderserver}
```

might cause the following command to be launched:

```
renderer -use percival.settings
```

The braces are required in this case because the simpler string "%h.settings" would cause the substitution mechanism to look for a Task or Cmd with "-id h.settings" and use its current values.

Special Characters and Escapes in Tractor Scripts

Tractor makes two passes through the scripts submitted to it. The first, a spool-time *parsing* pass, is used to construct the "shape" of the job; it sets up the hierarchy of tasks. The second pass (and possibly subsequent identical passes) is the dispatching, run-time, pass during which commands are launched and run-time substitutions are made, etc.

The initial parsing of the cript is done using a system built around John Ousterhaut's TCL interpreter. As a result, the TCL syntax rules apply to command arguments and string formation. Scripts consist of calls to the **Job**, **Task**, **Cmd**, and **Instance** commands (operators), which in turn take strings as arguments. In the case of Task, the -subtasks option can contain additional scripts, which are parsed recursively. In general, the TCL rules, as they apply to operator arguments, are much simpler than those for the Bourne shell (sh) or csh(1). Shell programmers should be aware of several things:

- launch expressions are tokenized and passed directly to execvp(2); hence semi-colon separated command sequences, command pipelines, environment variable references, and filename meta-characters (e.g. *,?,[]) are not directly supported (instead use an appropriate sub-shell).
- Tractor (and Alfred) will do tilde-expansion, as in csh
- single-quotes have no special meaning (i.e. they don't group words together), use braces - {} - for grouping.
- only one tokenizing pass is made through each argument string.

For example, consider using find(1) to remove files that start with "preview" from a directory tree. In a csh(1) script the command might look this way:<

```
find ~bob -type f -name 'preview.*' -exec /bin/rm {} \;
```

that is, the asterisk must be escaped from the shell filename expansion because it is used directly by the find command; similarly with the semicolon after the exec expression. The curly braces are untouched by the shell (since they're not part of a variable expression). If a similar command was part of a Tractor/Alfred script (as a Task's clean-up command for example), it would look like this:

```
Cmd {find ~bob -type f -name preview.* -exec /bin/rm \{\} ;}
```

Shell Pipelines and Other Expressions

Given the restrictions just described above on Cmd launch expressions there are nonetheless occasions when shell constructs, such as command pipelines or run-time filename expansion, can be very useful. In these situations, a simple solution is to launch an appropriate shell as the Cmd, passing the pipeline expression to the shell via command-line arguments:

```
Cmd {/bin/sh -e "cd /tmp/frames; ls -1 | xargs -I+ cp + /DDR"}
```

Script authors should read the documentation for their shell of choice to understand the implications of various invocation options. For example, you must decide whether the user's .cshrc or .profile should be executed when the dispatcher launches a command like the above.

Another approach is to use the Cmd -msg option to pass arbitrary expressions to a launched shell. This is essentially equivalent to the above approach, but not as compact; however, it does allow for persistent reuse of the shell, if that's desired. Consider the following examples which send mail, which can sometimes be handy at the end of job. Recall that the -s option to Mail looks for the next "word" as the subject, so spaces need to be escaped (using csh (1) syntax this time):

```
Cmd {/bin/csh -fc "/usr/sbin/Mail -s 'job done' jean &lt; ~/errlog"}
Cmd {/bin/csh -fet} -msg {/usr/sbin/Mail -s 'job done' jean &lt; ~/errlog}
```

Often the *simplest solution* for complex expressions is to write a short shell script in your favorite language and launch the script from Tractor:

```
Cmd {myscript}
```

If a remote server is required, the run-time host selection can be passed to the script as an argument:

```
Cmd {myscript %h} -service {someServerType}
```

Progress or Percent-done Indication

Tractor also scans the stdout of its launched apps for strings of this form:

```
TR_PROGRESS nnn%
ALF_PROGRESS nnn%
```

The integer *nnn*, in the range 0-100, is sent to the UI and used to control the percent-done bars drawn on active task nodes.

Exit Status

Tractor also scans for this directive:

```
TR_EXIT_STATUS nnn
ALF_EXIT_STATUS nnn
```

The integer *nnn* is used to determine success or failure rather than the actual program exit status. This override is most often used to express that an job-blocking error has occurred from within a library, even if the calling application is known to exit with status code zero. (0 indicates successful completion, any non-zero value indicates failure and results in a blocking task error). Conversely, some wrapper scripts will emit a zero code this way when the launched app is known to exit with a non-zero code in non-fatal cases. All legal application exit codes are are integers between 0 and 255 inclusive. Tractor will report exits due to interrupting signals using a negative number, namely the negative signal value. When tractor.config is configured to enable automatic task retries on error, all non-zero exit codes will cause a retry to occur except -2, -15, and -9 which are considered to be intentionally terminal due to user action. Note that if an application issues TR_EXIT_STATUS but does not actually exit within two seconds, then tractor-blade will automatically begin shutting down the application process group; this automatic sweep can be disabled in blade.config by adding this profile entry: "TR_EXIT_STATUS_termination": 0