upstart

Service Management for Linux

Design and Specification

Scott James Remnant

May 15, 2006

1 Introduction

Linux has traditionally relied on a number of different daemons inherited from both BSD and System V to manage the task of booting a machine, starting and stopping services and running user tasks at the appropriate time.

The current version of Ubuntu, for example, includes `sysvinit`, `atd`, `crond`, `anacron` and `netkit-inetd`; all of which are configured differently, yet are arguably performing the same kind of job.

To make matters worse, jobs such as starting and stopping services and running user scripts are also performed by packages such as `udev`, `acpid`, `apmd`, `ifupdown` and even `module-init-tools`. It truly is a nightmare for a system administrator to know how to make sure a task runs at the right time.

upstart is a single replacement for the standard UNIX services, combining their functionality into a single daemon which can be easily configured by a system administrator. It also introduces powerful new abilities, such as dependencies on other tasks completing first and service management.

Additionally compatibility is retained for the traditional tools so that users do not need to adapt immediately, and so that tasks required by vendor-supplied software can be supported.

2 Basic Design

The basic design of upstart is that tasks exist in one of three states:

Waiting This state is where all tasks start off, they are waiting for some event or dependency before they can be run. Once that event occurs, they are moved into the next state.

Running This state is for all of those tasks that are currently running, ie. those for which there are one or more processes on the system. Those processes are supervised and once they vanish, the task is moved into the next state.

Dead This state is where tasks come when they die, or once they have finished. They exist here for as long as the reaction to that is handled, and once finished the task is moved back into the *waiting* state, or if its termination was not expected and it is desired to keep it alive, the *running* state.

A task can only be in one state at a time, so if a service is running it can only be stopped; a second copy of it cannot be run. This simple rule prevents timed scripts from overlapping without requiring complicated locking.

Where a task handles a network connection, or simply needs to be started many times, it instead requests that a sub-task be spawned. That sub-task is placed in the *running* state and is destroyed once *dead*.

3 Events

Tasks may be manually started and stopped by the system administrator or users given the privilege to do so by the task or system configuration.

They may also be automatically started and stopped by events tracked within upstart, which fall into the following two categories [1]:

Edge Events
: These represent the vast majority of events, and include such events as the system starting or the lid button being pushed.

  Tasks can be started by any of a list of edge events, and also stopped by any of a further list of edge events.

Level Events
: These are associated with such things as hardware, and include both an event and a value. The state of a network interface is considered a level event, as it can be either "up" or "down".

  Tasks can be started by a level event being set to a certain value, and may be stopped when the value changes again if desired.

  If the task indicates no particular value, then it is started whenever the value changes; ie. they behave as edge events.

Temporal Events
: These are a separate class of events that allow tasks to be started and stopped at particular times, or after particular periods.

  Full cron-like specifications are permitted such as "at thirteen minutes past the hour", "at four-fifteen in the morning", etc. as well as anacron-like periods.

  Times can also refer to other times or events, such as "fifteen minutes after startup" or even relative to the tasks own starting time.

  If a temporal event is missed, usually due to the system being down, it is automatically caught up unless configured otherwise by the task.

Events, and for level events their values, are represented by strings which may be namespaced for clarity. A standard for their naming is highly recommended.

Any edge event may be triggered, or level event's value changed by upstart itself or by external programs. Triggering or setting the value of an event hithero unknown to upstart causes it to be created.

[1] terms stolen from PCI

Level events also exist for every task in the system, such that tasks can be started or stopped based on the state of other tasks.

3.1 Startup and Shutdown

The `startup` and `shutdown` events deserve particular special mention, given their critical role in bringing a system up and down.

Both are edge events, the first is triggered as soon as upstart is running and ready to work. The latter is triggered when there is a request to bring the machine down.

upstart normally makes no attempt to stop running tasks during shutdown, other than sending them the kill signal to allow disks to be unmounted.

Tasks are expected to be resilient to crashes, and given that the power is about to go anyway, no particular need to clean up their memory usage.

If a task isn't resilient, it should be registered as being automatically stopped by the `shutdown` event; this should be the exception, rather than the norm, though.

4 Dependencies

As well as specifying the events that trigger the automatic start of a task, the task can also specify dependencies that must be fulfilled before it can be started.

These include manual start by a system administrator or privileged user, if a task's dependencies have not been fulfilled yet then it cannot be started manually.

Dependencies are simply a list of events that must occur first, which may be edge, level or temporarl. Once the event has occurred, that dependency is considered fulfilled and removed from the list within the task.

Once a task has been stopped and returned to the *waiting* state, its dependencies are still considered to have been fulfilled and it may be started again.

5 Service Supervision

The core of a task is a particular process that needs to be run, this may be a fully-fledged daemon or just a user script. These processes are supervised by upstart, which is a fancy way of saying that it keeps an eye on them.

5.1 Foreground Processes

Processes that run in the foreground and do not change their session can be supervised trivially. Many daemons can operate in this fashion, often either for debugging purposes or simply to support service supervision.

The process is run in a forked child, with the standard input set to `/dev/null` and standard output and error either the same or diverted to a logging system of some kind.

When the daemon dies, the supervisor receives the `SIGCHLD` signal from the kernel and the exit status can be obtained. The process can then be moved into the *dead* state so it is restarted if necessary.

5.2 Background Processes

Many daemons have not been modified to run in the foreground and instead perform actions on startup to deliberately distance themselves from the process that spawned them, usually forking into the background and changing their session.

The kernel will not send `SIGCHLD` to the supervisor for these processes, so it is necessary for the supervisor to locate and watch the processes itself.

It scans `/proc` for all processes that share the same executable as the one run, probably including a check of device and inode in case the executable is replaced. Once the number of processes reaches zero, it knows that the service has died so it can be moved into the *dead* state.

Obviously with this method, there is no way to determine the exit status of the process; and a process that spawns children that fail to die will still be recorded as running.

6 Task Lifetime

While a task sits in the *waiting* state, it is idle and there is no activity on it; except the removal of dependencies, if present in the configuration.

Once an event that triggers the task occurs, assuming all dependencies have been met, the task is moved into the *running* state. At this point the task's level event is set to the `start` value, and after any tasks triggered by that event have started, the task's start hook (if any) is run.

The start hook is a piece of shell code executed to prepare the system for the task, usually creating any necessary directories and such. It is executed with the full environment of the task.

Once that has completed, the task level event is set to `running` and the task itself is run. This may be either the path to an executable that is to be supervised, or shell code that is run within a supervised shell.

The supervised task then runs until it exits normally, abnormally, one of the specified stop events occurs or the task is stopped manually. At this point the task is moved into the *dead* state, and if the process is still running it is sent the `SIGTERM` and then `SIGKILL` signals.

In the case of a normal termination or explicit stop, the task level event is set to `stop` and the task's stop hook (if any) is run.

Like the start hook this is shell code that is intended to clean up after the service and is executed within the same full environment.

Once the stop hook has finished, the task level event is set to NULL and the task is returned to the *waiting* state.

In the case of abnormal termination, where a restart is required, the task level event is set to `restart` and the task's restart hook (if any) is run.

The service is then restarted and the task level event is set to `running` again.

7 Task Environment

The environment of the task's hooks and process itself is specified within the task, and includes resource limits, environment variables and working directory.

The environment can also include variables that are associated with the event, passed from the external trigger tool through upstart and placed into the running environment.