Unix Systems Event Manager
Initial report

Álvaro Villalba Navarro
Director: Juan José Costa Prats

March 9, 2012

Abstract

In this initial report we will explain the problems of general purpose job scheduling and abstract events handling in Unix systems, and describe one solution implemented through this final project. It's important to understand that this is not only an engineering final project, but also a personal project. As the nature of the final project is time-limited we won't cover the implementation of the whole solution but the most important functionalities, so the goals of this project will be stated explicitly.
Furthermore this document will state which are the goals that have been achieved and which need to be achieved yet. This will be done by defining a planning divided into tasks with assigned deadlines, a final explanation about the work being done and a forecast about the future of the project.
The reason for this document to be written in English, which is not my native language, is that the project is under the free software terms. This decision is expected to help its diffusion and maybe catch the interest of somebody.

1 Introduction

In Unix-like operating systems there is a service which exists in almost every distribution or version and it's treated as an standard. This service is *cron*, which according to Wikipedia:

> *"cron is a time-based job scheduler in Unix-like computer operating systems. cron enables users to schedule jobs (commands or shell scripts) to run periodically at certain times or dates. It is commonly used to automate system maintenance or administration, though its general-purpose nature means that it can be used for other purposes, such as connecting to the Internet and downloading email."*

It is a job scheduler[1], which means that it is a service in charge of running background executions. Job schedulers must be based on something to decide when is the right moment to run a job. Most of job schedulers are workload-based, so they schedule their jobs by taking into account the available resources, to run them when the computer is idle.
However, as we have seen on the Wikipedia article, cron is time-based, so it will run its jobs on defined instants.
There are more services with job-scheduling capabilities based on other kinds of events, like *udev* which is a device manager that can run shell commands when events such as a 'new device connection' occur. Or *syslog*, that can also run shell commands when it receives a log message from an application.

[1]Should not be confused with process scheduling, which is the assignment of currently running processes to CPUs by the operating system.

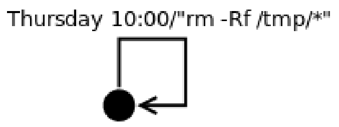

Figure 1: cron state machine example.

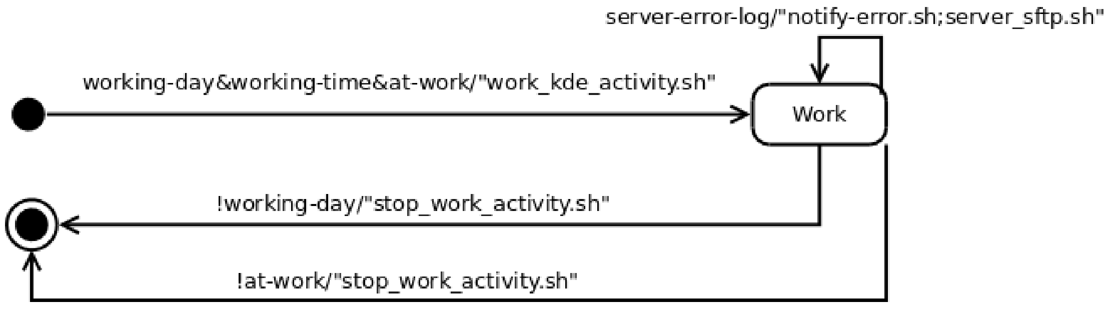
server-error-log/"notify-error.sh;server_sftp.sh"
working-day&working-time&at-work/"work_kde_activity.sh"
Work
!working-day/"stop_work_activity.sh"
!at-work/"stop_work_activity.sh"

Figure 2: Proposed service state machine example.

1.1 Problem

This project intends to solve the problem of scheduling jobs using an abstract concept of *event*, so we are not that much limited by the nature of the events that trigger the jobs, such as time or workload.
That alone is not a problem to solve as we have services for each kind of events. But what we want is a service that:

- Can have multiple events as a requirement for a job.
- Adds the ability of detecting new kinds of events to the system.
- Runs a state machine.
- Propagates events through IP.

The first point means that we want our service to notice various events of different kinds to execute a single job. It should also permit logical operations between those event notices as condition to execute a job, like '*if noticed event_x or event_y then run z*'.
The second point refers to the fact that as we said before, Unix-like systems have several job schedulers for various kinds of events, generally each for one kind. But as we want to be able to combine this kinds of events, it's also desirable to have more than we currently have. For example geolocation, system sensors, weather forecast or file monitoring events could be quite useful.
If we receive the event “10:00AM” and later the event “11:00AM”, we can define the period of time “between 10AM and 11AM”. This period is a state, and we would like to execute jobs only in this state every time we receive some random event, but not when we receive it at 12AM. This is simply defining a *state machine*, and it's explained in depth in the following section.
The last entry of the list means that a local service could propagate an event to the same service in another computer in order to inform of that event and usually running a job there.

1.1.1 Example

Cron is a simple and known enough example of a job scheduler, so we will use it to illustrate what we have and what we want. If we see every entry of a crontab as a state machine we

would have something like shown on *figure 1.* As we can see there is only one state (initial) and a single *transition.* Over the transition we have an event and a job separated by the character */.* In a state machine a job is called *action*, so that is the way we will call them from now on.
This is what cron can do, but what we actually would like to have on our service is something like *figure 2*, where we can see multiple states, events of different kinds, and transitions. Imagine for this example that we have a system administrator working in a office with his own laptop. This state machine detects when he is working by waiting for a geolocation event that sites the device at the office, *and* two time events[2] that define the beginning of the working day. What if we start the service after the time the event would be received? A solution is discussed in section 1.2.3.
The first action is a script that would change the current KDE[3] Activity[4] to one configured for work, with a sober and relaxing wallpaper, and the applications needed to do his job.
Once he is working, he will be notified every time the server he is in charge of logs an error message. When he accept the notification it will open the working directory of the server. As this event has been generated on the server, this is a remote event received through IP.
To detect the end of our working day and close the 'work' Activity the service must detect that he is no more in a working day (time event) *or* that he is out of the office (geolocation).
A remark about this state machine is that it has a final state, so it won't start again until the service is restarted. This behaviour can be easily changed to the cron's behaviour by removing the final state and sending the last two transitions to the initial state.

1.2 Solution

In *figure 3* we have a sketch of the solution we propose to the previously explained problem. The project is, by now, globally called *reactor.* reactor is formed by four principal components:

1.2.1 Daemon

reactord in *figure 3*, is the main service that would receive events from the *event sources.* It also has the state machine structure specified by some editable files (*state tabs*) similar to crontabs or the udev rules files. Transitions can also be added by the user during the runtime of the service using the command line program (*section 1.2.3*) . This state machine is defined by transitions with a list of event notices identifications, an action, and '*from*' and '*to*' state identifications. If 'from' identification is empty, then it's the initial state. If 'to' identification is not the 'from' identification of any other transition, then it is a final state.
reactord could also act as a *remote event source* sending events through a network to a remote reactord. Those events are the events received by local sources that are stated on the *propagation rules files* to be resend to an IP address. We could have a system to send propagation rules of the remote events we are interested to the remote reactord, although this won't be part of the final project.

1.2.2 Workers

Workers[5] or *event triggers* in the sketch, are one kind of local event sources. Their goal is to generate events of kinds which we don't have any job scheduler, so reactord must provide a

[2]It could be in fact just one event as in *figure 1*, but this way is more comprehensive.
[3]KDE is a popular software compilation for Linux operating systems, which include a desktop environment.
[4]A KDE Activity is a desktop environment configuration, which includes the wallpaper, desktop widgets and applications running.
[5]A worker is a child process that runs in parallel with its father doing some specific job.

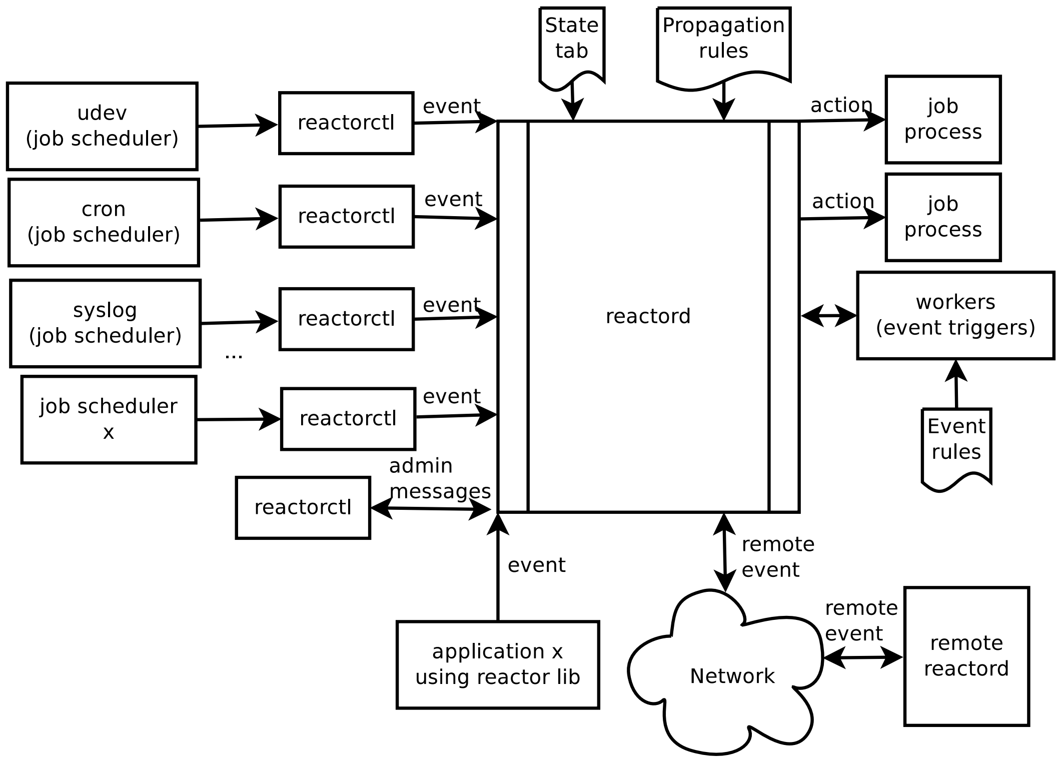

Figure 3: Diagram of reactor.

good interface to make them easy to develop.
For doing that they will act as an external job scheduler would, with its *event rules* files to know when to trigger an event, but with the advantage of being completely integrated with reactord. We mentioned in the example the problem that if an event that defines the beginning of a state happens before the service started, we lose that information and reactord won't know we actually aren't in the initial state. The daemon workers gives us a solution: We send to the workers the events we are currently waiting for and which the workers are in charge. The workers will check if the last time those events happened was after the last time next states leaving events happened. If that's true those events are triggered and we enter the correct state. That's something other event sources can't do, because they are not integrated with the daemon.

1.2.3 Command line program

reactorctl in the sketch, is the typical administration or control application for the daemon. It has two differenced functionalities: sending administration commands and manually triggering events to the daemon.
The administration commands would be adding transitions to the state machine without saving them on disk, ask the workers to trigger the past events to enter in valid current states and get the state machines in a interpretable format[6].
The manually triggering events functionality is useful not only for debugging purpose, but also because it gives the ability to use existent job schedulers for sending events to reactord, so we don't have to reinvent the wheel. Far from being part of the final project, it could be able to set the rules of well known job schedulers to send events to reactord without having to edit

[6]Those formats could be DOT, GML or TGF.

manually the rules files of each one.

1.2.4 Library

In the diagram we can see one last component which is a random application sending events to reactord. It should be done by sharing the main reactor functions in a shared library, giving access to any application to easily sending events or adding transitions. This is intended to help to reduce the number of daemons on the system loaded by applications that are just waiting for something to occur to show simple alerts or alarms when the application is closed.

2 Planning

That is the planning to follow in order to finish the project in time. Here is also where the goals of the final project are clearly stated. It's mainly a list of tasks to perform ordered from most to less priority, and assigned deadlines to sets of those tasks:

January 26th, 2012

- Design of the project.

February 2nd, 2012

- Makefile.am and configure.ac.
- Log functions.
- reactor users check.

February 16th, 2012

- Main data structure of the daemon done and working.
- Daemon socket ready to receive events from other processes.
- Execute shell commands functionality.
- Dummy first version of control program.

February 29th, 2012

- *State tabs.*

March 9th, 2012

- *Control program with basic functionalities.*
- Delivery this report.

March 16th, 2012

- Shared library.

March 30th, 2012

- Propagation rules data structure.
- Propagation rules files.

April 13th, 2012

- Daemon socket ready to send and receive IP event messages.
- Plugin workers interface.

April 27th, 2012

- Test plugin.
- Current state storing.
- Trigger events on action finalization.
- Trigger currently valid events.

Right now we are at at March 9th, 2012 deadline and we haven't achieved all the goals that we anticipated, even though we have most of the tasks done. The task that needs more work to be done and was foreseen to be already done is the state tabs task, that includes a syntax definition (already done), files destination (already done), parser and builder.
reactorctl has the basic functionalities done but it lacks a good argument entry for the user. Recently we noticed a malfunction on the state machine structure, and the socket communication between reactorctl and reactord can be improved.

Because of the experience in this project until now, we believe that this planning may be altered. That's why we consider the last deadline tasks as optional and they will be done if there is time left.