Títol: Manegador d'esdeveniments per a sistemes UNIX

Volum: 1/1

Alumne: Álvaro Villalba Navarro

Director/Ponent: Juan José Costa Prats

Departament: Departament d'Arquitectura de Computadors

Data: June 19, 2012

DADES DEL PROJECTE

Títol del projecte:	Manegador d'esdeveniments per a sistemes UNIX
Nom de l'estudiant:	Álvaro Villalba Navarro
Titulació:	Enginyeria Informàtica
Crèdits:	37,5
Director/Ponente:	Juan José Costa Prats
Departament:	Departament d'Arquitectura de Computadors

MEMBRES DEL TRIBUNAL *(nombre y firma)*

President:	Yolanda Becerra Fontal
Vocal:	Montserrat Maureso Sánchez
Secretari:	Juan José Costa Prats

QUALIFICACIÓ

Qualificació numèrica:

Qualificació descriptiva:

Data:

Contents

Chapter 1

Introduction

Since IBM mainframes systems era, job schedulers[1] are an important part of the IT infrastructure. They are in charge of running background and unattended executions, and are typically used for system maintenance and administration jobs such as hard drive defragmentation, system updates check or system clock synchronization. However, they are also used by final system users to manage their jobs, like reminders or resource intensive processes.
Job schedulers must decide which job to run and when. There are some schemes or parameters that can be taken into account for taking these decisions[6].
Some of the most used are:

- Defined execution time
- Elapsed execution time
- Execution time given to user
- Job priority
- Compute resource availability
- Number of simultaneous jobs allowed for a user
- Availability of peripheral devices
- Occurrence of prescribed events

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

cron, probably the most popular job scheduler in the UNIX world, only considers the first parameter from the list, which is probably the simpler to use and the most functional. According to Wikipedia[2]:

> *"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."*

As we can read in this entry, *cron* is claimed to be a general-purpose job scheduler. But we can be more accurate and say that it is general-purpose in terms of 'action', but not in terms of 'reaction'. In other words, *cron* can execute any action, but as we said before, it only considers the defined execution time to do so, which we could refer to it as just one reaction parameter.

There are more popular services with job-scheduling capabilities based on other parameters like *udev*, which is a device manager that can run shell commands when the availability of some peripheral device changes. Or *syslog*, that can also run shell commands when it receives a log message from an application. Receiving a log message is not a listed scheduling parameter, but it is easy to notice that the message can be considered an event. In fact, most of those listed scheduling schemes can be defined as occurrences of prescribed events. The rest of them are states that need to be looked up.

Our software project, called *reactor* so far, is a general-purpose job scheduler and event handler. Its main goal is to be general-purpose on both 'action' and 'reaction' ways.

By now we have set out some basic concepts about job-scheduling that will be present for the rest of this final report. On the next sections to come we are going to expose a use case of this software project. We will also make a brief statement about the goals to achieve according to this use case, and explain the first decisions made to develop the solution.

reactor is intended to survive this final project, so there will be forecast goals that would not be implemented at the time this document is released. All this will be detailed.

As a final remark I want to confirm that English is not my mother tongue, but we found it useful to write this document in English so it could be useful to a wider range of people, if any.

1.1 Motivation

In the last years a big wave of mobile devices with integrated GPS sensors came in to stay. We are talking about smartphones, tablets, digital cameras... Handy and very portable devices. But nowadays you can also get geolocation information from not-so-portable devices like a laptop or a desktop computer. You can obviously connect your car's GPS to the computer, and if the controller allows it, you will be able to do exactly the same as with an smartphone, but this is not a usual use case. A most common situation is to be connected to the internet and expect location-related results from a web search, without extra peripherals and hassles. This is something that work thanks to Wi-Fi and ISP IP geolocation databases.
So we have a bunch of devices able to geolocate themselves. What can we do with such a feature? A lot of things have been done, but the major part of them seem to be a functionalities for the services providers more than for the users, for example showing location-related advertisements or creepy user tracking. We can make a good use of it by making a geolocation-based job scheduler, so the devices geolocation would be the reaction parameter for our job scheduler. For instance we could automatically set our smartphone in silent mode when we are at the theatre. This was the first idea that came in mind about a good personal project to develop. A more or less simple job scheduler daemon running along with cron if not with more job schedulers, without any communication between them. This was not enough, so the idea of the project began to become bigger and bigger, and so it became more abstract and modular. Also mobile devices have more sensors than GPS, like gyroscopes and accelerometers, that could be used too. We wanted a job-scheduler with the ability of reacting to several parameters, which would be able to interact between them. Something like 'cron meets GPS and more'.

Making money is usually the main motivation for a project, but this is not our case because there is no intention to sell it. The project will be FLOSS[2], so the real motivations are those that usually come with this kind of projects. Learning from the experience of developing a long-term personal project from scratch is the major one. Then the project should be useful to myself as by now I am the only target interested on it. Finally it may also be useful for other people not involved in the project development, what is expected and highly desirable to the point that the project is developed with them always in mind.

[2]Free/Libre Open-Source Software

1.2 Use case

Here we describe a use case for this project so we can have a reference in mind for the rest of the document to illustrate and help us to understand it.

For this use case we will think of a web developer who works at an office with his own laptop.
He begins the day at home, where with his laptop checks his personal mail account and sets some tasks and appointments to his personal calendar software. Also his smartphone is in normal mode (both ringtone and vibration are on).
Thanks to our software, when he arrives to the office the cellphone is in vibration mode. Also when he starts his computer there, the environment has changed. Now the email notifier is not checking personal email accounts, but work accounts. The calendar shown is the company one, and the wallpaper is more sober. It also automatically pulls the new commits from the remote server, asking for manual actions if required, and starts his favourite IDE. Afterwards, while he is working, every error that the company's production http server logs is notified to him.
When the working day is over the default behaviour is restored. The cellphone returns to the normal mode, he receives email notifications from his personal accounts and the personal calendar is enabled again. But is not until he leaves the office that he will stop receiving notifications from the server's system log and the company's email, and the company's calendar will be hidden.

We don't have any maintenance or system administration task done by our software in this use case, and this is done in purpose. We have seen before that job schedulers are typically use for those kind of tasks, but they are transparent for the user. So if we put administration tasks on the use case, the user would not be aware of the existence of our software and would not make direct use of it. We preferred to break with the cliché and show a tool useful not only for operating systems internals but also for the final user. Assuming that it is useful for maintenance is just going a tiny step further from the use case.

1.3 Brief goals description

The goals will be defined in detail in *section 2.2*, after explaining the problem to solve. In this section we want to show the main goals that can be extracted from the use case.

- Event driven

  It should react to abstract events, such as 'arrived to the office', 'error log in the http server' or 'beginning of the working day'...

- Execution of shell commands

  All the actions described in the use case can be performed by command-line. That is what makes it general purpose in 'action' terms.

- State aware

  Notice that in the use case our web developer can receive error logs notifications from the server when he is at the office, but not when he is not there. So our software must be able to know when he is at work and only then react to 'error log in the http server'. When we are out of this state these events must be ignored. In the use case there are more examples of this, but they are not so clear.

- Communication between systems with our job-scheduler

  Also in the communication of the error log, being the http server a remote machine, one can see that there is a network communication between systems.

- Multiple kinds of events

  The job-scheduler is little limited by the kind of events it can receive.

1.4 Preliminary decisions

Before the specification there are some decisions that were taken for a number of reasons like philosophy, ideals, learning goals... The fact that some of these decisions were made before we specified what we want to do may be taken as erroneous. Actually, those decisions could be done after the specification and it wouldn't change anything, some of them even after the implementation with the same result. But still, those decisions were personal requisites for the software project and for the final project. This is why we will

also justify the timing and not only the decisions themselves.
Those decisions are related to the software license election and the platform for the project to head.

1.4.1 License

From the beginning there was something that was out of discussion about this project, and it was the kind of license the software project was going to be under. As we said in a previous section, *reactor* is a FLOSS project, which according to the FSF[3] is software that follows four rules or freedoms to procure. These freedoms are[3]:

- The freedom to run the software, for any purpose.
- The freedom to study how the software works, and change it so it does your computing as you wish.

  Access to the source code is a precondition for this.
- The freedom to redistribute copies so you can help your neighbour.

  By doing this you can give the whole community a chance to benefit from your changes.
- The freedom to distribute copies of your modified versions to others.

This is not the only interpretation of what free software means, but probably is the most general and accepted one.
So yes, we agree with that and want it for our software. An idea is nothing more than a set of other ideas that other people had before, so they don't belong to anybody and they can not be sealed, hidden or restricted. Otherwise it would be a childish selfish behaviour. And software is nothing more than a written implementation of ideas.
But those four freedoms are not enough for what we want. We like the giving part of the deal, but we also want to get something. If someone takes advantage of the second freedom, we want these changes to be public, because they are probably improvements of a project we designed, so it is a derivative work, and we have the right to check on them and add them back to the initial project.

[3]Free Software Foundation is non-profit corporation that claims to have a '*worldwide mission to promote computer user freedom and to defend the rights of all free software users*'. The activities for which they are mainly known are the GNU Project, the GNU Licenses and pro-FLOSS activism[3]. For more information: `http://fsf.org`

That limits the choice of the software license. Doing a quick check on the major existing licenses and their main characteristics we easily arrive to the conclusion that the license that fits best to our needs is the GPL. It requires the derivative work to be released with the same terms of the license, without exceptions.
But there is another choice to make about the license and it is the version. The GPL has three versions. GPLv1 is the one that essentially protects the four freedoms stated by the FSF by forcing the distributors of the software to publish the source code and license under the same terms the modified versions, so the mix of licenses don't diminish the overall value of them. The GPLv2 adds some kind of protections to patent fees from software corporation to free software distributors. The GPLv2 licensed software only can be distributed without any condition or restriction like for example fees, if not it can not be distributed. The GPLv3 goes further on the software patent protection, and also states controversial clauses against the 'tivoization'. Tivoization is how the FSF calls the practice of limiting the execution of free software by the hardware. Its name comes from the TiVo device, which runs GPLv2 licensed software and follows the terms of use, but it doesn't let run your modified version of the code. GPLv3 states that the software must not be restricted by the hardware in which it comes[30]. This is controversial because some relevant developers think that telling how the hardware has to be in order to run GPLv3 software is too intrusive[16]. We finally stand for the GPLv2 for three reasons. The first one is that the GPLv2 is stronger than GPLv1 (which is basically deprecated). The second reason is that hardware intrusiveness arguments feels strong enough. We are making software and we don't care about the hardware design in which it runs, it is not our work. We may prefer open hardware, but this is a personal decision, not something we want people to be forced to. The last one is that we always can change our mind and upgrade to GPLv3 if we find it better, as it is designed to be easy to upgrade to.
There is one more special case to take into account and it is the license for the libraries, if any (and as we will see in the next chapters, we will have libraries). The problem of the GPL with the libraries is that the main program that links to a GPL library must be GPL too, and that is something that we don't want. The solution is using the LGPLv2, which is very similar to the GPLv2 but allows the code to be linked to any program.
So, in conclusion, our software project will be under GPLv2 and LGPLv2.

1.4.2 Platform

Taking by platform the hardware architecture and the software framework in which our project is going to run, here we are going to show the main platforms available, pros and cons, which one we choose and why. This will be centred on the software framework as the actual interface we are going to deal with. But we will choose it in regard to the hardware in which it can run.

So we have mainly two big families of software platforms to which we can focus our project to, Microsoft Windows and POSIX operating systems. Microsoft Windows is a widely used privative operating system on desktop and laptop form factors. More than a family, as UNIX-like operating systems are, they are all versions of the same OS as all of them are released by the same company and they drop support for old versions when they launch new products. However, its API is usually quite compatible between versions.

In the other hand we have POSIX operating systems, which can also be called UNIX-like operating systems. POSIX is a standard API definition for software compatibility with variants of Unix and other operating systems. So, long story short, UNIX-like operating systems are a big bunch of operating systems running in every hardware architecture, that share almost the same API. Into that category fall the popular Mac OS X, iOS, Linux, Solaris and HP/UX.

We prefer for our software the idea of using an standard API that works in many operating systems and devices than a API widely adopted but strictly restricted by a company and their products. That brings us a wide range of requirements to rely on. Also, we are interested on focus our project on Linux kernel, which is very stable and developer friendly, as well as GPLv2 licensed and very adopted on ultra-portable devices and mainframes. So the decision is made, our software project will be oriented to the UNIX platform.

There is also the possibility of making *reactor* cross-platform by choosing the correct API at compilation time with tools like Autoconf and using cross-platform libraries. The time of this final project is limited and that makes us choose not to support Windows from the beginning, as it would take time to learn an API that us, as developers, are not interested into. May be in the future if there are users interested.

This decision was made before the specification because we had a personal interest on learning POSIX and particularly on Linux API. Also the pros and cons we state before are valid for almost every project we could do.

Chapter 2

Specification and architecture

In order to be methodical and organized, first of all we need to specify *what* exactly our project has to do. We already have some hints about the problem to solve, which in this section we will formalize a bit more, so now is time for the solution. Being this a non-trivial software we need to help us to do that with a wide angle perspective explanation of the structure we choose in order to successfully achieve our goals.
First we will deepen on the problem so we can then make an informal but concise specification of the solution based on the architecture of the whole software project.

2.1 Problem

In *sections 1.1*, *1.2* and *1.3* the problem to solve by this project is already introduced. To summarize it in one phrase:
We want our devices to automatically react to events that they can notice.
This phrase sounds good as an statement, but is in fact what our devices are doing through their OS and programs, so it needs some further explanations. On the one hand, we must acknowledge that our solutions should give us the ability of using new kinds of events in a easy way for job scheduling, events we can not use for this purpose right now. On the other hand we want to centralize this job-scheduling by noticing all the events in one program and performing actions when defined sets of these events are received. These event sets may be combined using logical operators, like '*if noticed event_x or event_y then run z*'. All this should be done through the interface this program offers, for example a file similar to a crontab. In order to do that we also need a way to make this program

notice any kind of events. We must remember that these events are something abstract that can represent almost everything, from a sensor detection to an application failing. In the use case of *section 1.2* we saw events like the arrival to a GPS location, an error log from a server and cron-like events, so the sources can be very different. We have to keep in mind that in our UNIX system there are some hardly irreplaceable job schedulers, and we don't want to reinvent the wheel of have several job schedulers doing exactly the same. What we expect is the ability to use the existing job schedulers to send events to our centralized system.
We also expressed our interest on the state-awareness of our program. It should execute an action not every time that the set of events assigned to it are noticed, but also if a sequence of actions has been performed before.
A last remark about the problem is that it includes the need of noticing and reacting not only to local events, but also to events that happened on remote devices. In this regard we have in the use case the event on the http server that happens when an error message is logged.

2.2 Solution

In the *figure 2.1* we have a sketch of the solution in architecture terms we propose to the previously explained problem.

reactord is the main program, a background service, also known as daemon, that centralizes all the job scheduling. It has access to some files called in the figure "rules files". The rules files are the crontab-like files where the sequence of sets of events assigned to actions are defined by the user. Every set of rules assigned to an action will be called a *rule*. As we said before we need some kind of state awareness with this rules, so the best we found to achieve that is by making the rules define a deterministic state machine. To illustrate that concept in *figure 2.2* we can see the deterministic state machine of the web developers laptop use case. On the transitions between states be have the 'events/action' conjunction, where 'events' is a set of events using the AND (`&`) logical operator between them. This means that the transition will only be triggered when all the events are noticed. To perform an OR operator, so an action is executed if event 'x' or event 'y' are notices, is easy with this system. The only thing we have to do, as shown in the *figure 2.3*, is define a transition between the same states with the same action for each OR operand.

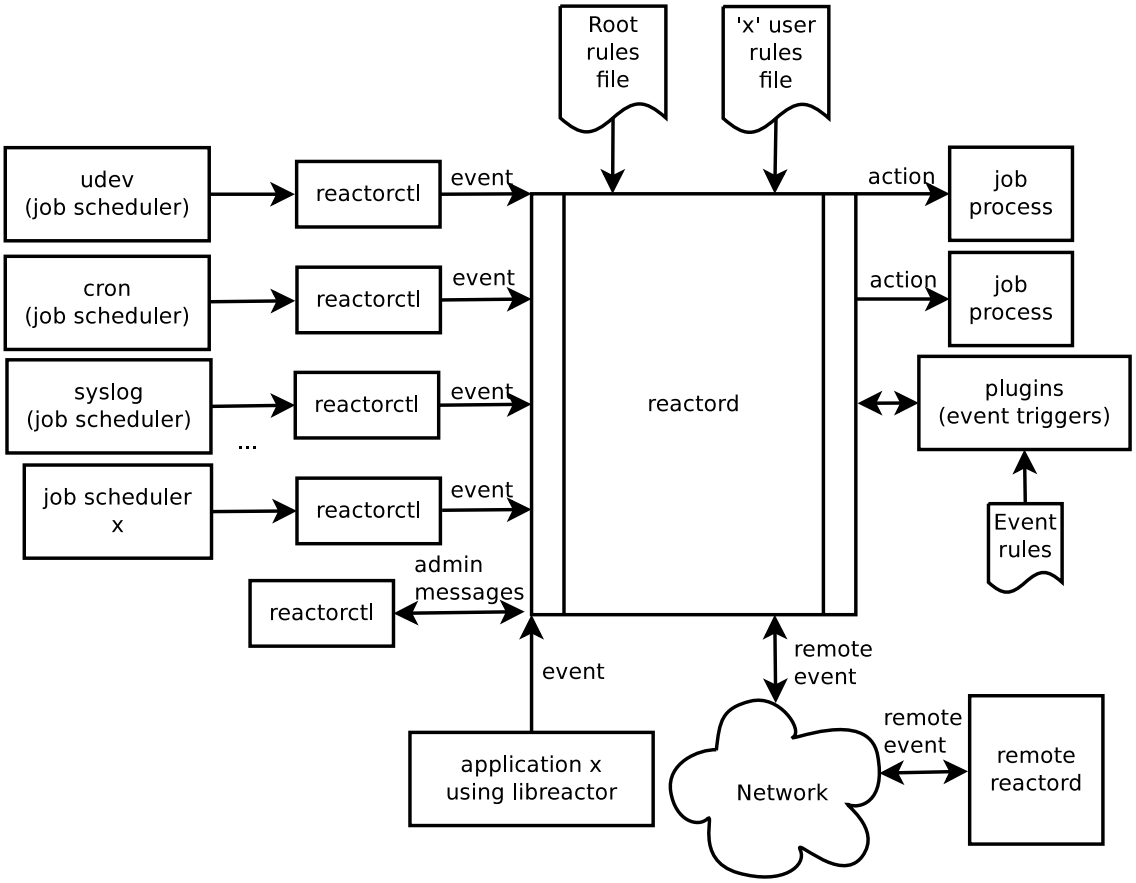


Figure 2.1: Diagram of reactor

We said that the state machines are deterministic, that means that we can have several transitions from the same state waiting for the same events to notice. If the user defines an indeterministic state machine, it will go through one of the transitions. Which one it will choose is an undefined behaviour. The syntax of these rules files is defined on *3.1.2*.

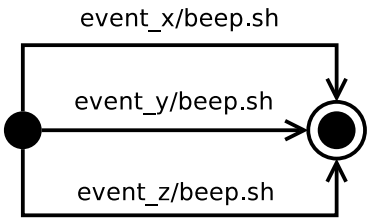


Figure 2.3: Example *reactord* state machine for "run beep.sh if event_x OR event_y OR event_z are noticed"

reactorctl is a daemon control program that acts as a command-line interface of the

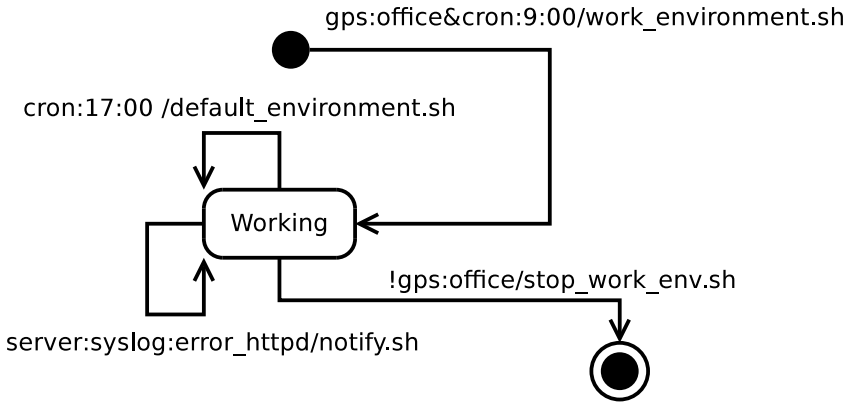


Figure 2.2: State machine of the laptops use case

daemon. Its options are:

- Add rule

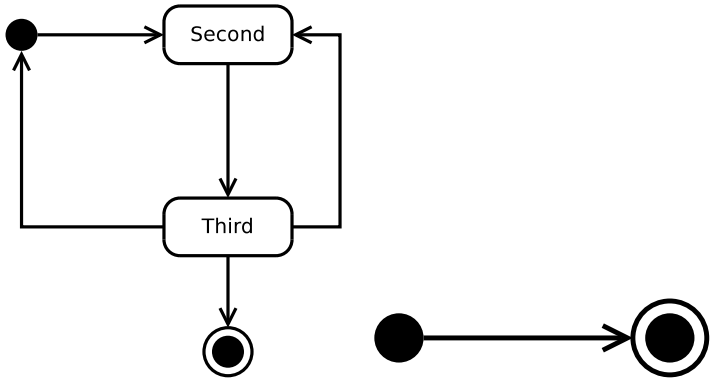
  Unlike adding rules through the rules file, those transitions will be active at once without need of restarting the daemon. Also, when the daemon is restarted the rules added with that method will disappear. The rule syntax will be the same as in the rules files.

- Send event

  It sends to the main daemon the specified event identification.

- Delete transition

  It deletes the specified transitions and all the dependant transitions and states of the state machine. An example is drawn in *figure 2.4*, where we can see how 'delete transition' not only deletes the target transition, the only one out from the 'Second' state, but also all the transitions depending on it. It is important to see in the example that if we simply follow the transitions beginning from the transition we want to delete in order to delete them, the result state machine would be empty. From the 'Third' state we go to the initial state which would be deleted, so then all the state machine would also be deleted. But an initial state does not depend on any transition more than the transitions out from it, so we keep it as it is expected. As in the 'add rule' option, when a transition is deleted the change will be effective instantly, but if the deleted transition is from a rules file, when *reactord* is restarted that transition will be restored.



(a) Before (b) After

Figure 2.4: Example of a transition deletion. The deleted transition is the only one out from the 'Second' state, so 'Second' becomes final. The initial state and the transitions from it are kept on the result because it does not depend on its entering states.

The 'send event' option needs special attention. Apparently there is not much use for this option than for debugging *reactor*, or testing the state machines. But it is in fact an option of great use. Its main purpose is to let external, independent and already existing job schedulers to propagate their events to *reactord*. The use is simple, *reactorctl* gives us a command to send events that we just have to put as a command action to those existing job schedulers, let's say for example *cron*:

```
1 0 * * * reactorctl -e cron_event_1
```

With that entry in a crontab, cron will send an event called 'cron_event_1' to *reactord* every night at 00:01.
And this can be used as well on other programs that are not mainly job schedulers, but do the job too and will be on our system we like it or not (and we do). Programs like *udev* or *syslog* have the ability of scheduling jobs and they are critic for the operating system, so reimplementing its functionality just for the sake of our project won't be very smart of us. That pretty much solves the 'reinventing the wheel' problem. This solution can also be seen in the *figure 2.1* schema.

These functionalities of the *reactorctl* program are isolated in a shared library called *libreactor*. This means that new programs, or plugins of a program can be built with

the ability of sending events directly to *reactord* whenever it finds an event occurred (including internal program events). That is a juicy potential source of events, because probably developers will not make their new programs with our project in mind, but if we are interested in a program with a plugin interface to send events to *reactor*, we can make a plugin to do so. And this is a great range of useful programs: Linux modules, Firefox, Thunderbird, Eclipse, LibreOffice... This is not the only use of *libreactor*. We also put here functions initially thought for the *reactord* internals but that actually are useful for the plugins. For example in the sketch we can easily see that both *reactord* and the plugins have rule files, son in both cases we need a rule parser that we may share. As you can imagine, the parser can't be exactly the same. This will be explained in detail in *section 3.3*

In the schema we can see two sources of events more. One is *reactord* itself running in a remote device. *reactord* has several kinds of actions to perform, not only command line executions. One of them is called *propagate*, and what it does is send all the event identifications that triggered the action to an specified address (which can be local, but it is quite useless). Those actions won't be propagated in any specified order, neither in the order of arrivals nor in the order in which they were written.
So for example in our use case, the http server should have our software installed. Then, the syslog should have a rule to execute *reactorctl* to send an event to its *reactord*. The last step would be set a *reactor* rule that propagates this event to an IP, maybe a broadcast for a subnet dedicated to developers and sysadmins where our web developer is.

The other sources of events left in the schema are the *reactord* plugins. Their initial purpose is to to trigger new kind of events to the system in the easiest way possible. *reactord* and *libreactor* compound a set of tools and interfaces that makes it easy to build a plugin. By now, their behaviour is not expected to be very different from the standalone job schedulers using *reactorctl*. Plugins will have their own rules files with their own syntax were it will be stated when an event should be triggered. This can be improved a lot, by for example letting use this rules directly on the *reactord* rules file, or letting us ask the plugin for an event. On *section 6.2* we will discuss future and more interesting functionalities.

2.3 State of the art

We are not alone and we are not the only ones who thought about the problem we are trying to solve, or at least similar ones. There are existing projects with similar purposes that have been around for some years, and are quite popular. We are going to explain why, even existing these good projects, ours is still needed for our purposes.
The project we are going to comment will be event driven job schedulers and similar projects that gave us ideas and have some approaches to our solution. We won't spend time explaining every job scheduler that exist, because this would take the whole report. The are a lot of job schedulers with a lot of different purposes, specially workload and calendar driven job schedulers, so in this field we will talk only about the event driven ones.

2.3.1 SOS JobScheduler

SOS JobScheduler[10] is probably the most similar project to *reactor*. It is an open source general purpose job scheduler which is mainly used to launch executable files and run database processes automatically. Has been written mainly in C++ and Java, its available for HP-UX(IA), IBM AIX, Linux, Solaris and Windows(2003/2008/XP/Vista/7) and it is licensed under GPL. It is also event driven, but its events are different from ours. Its events are called 'job starts', so an event is just that, an order to run a job sent by somebody. It internally detects two kind of events that are actually noticed by well known job scheduling capable programs, so it' reinventing the wheel. Those events are calendar and directory monitoring events, which are currently managed by *cron* and *at* for the first kind and *inotify-tools* for the second one[1]. It also has a library to build applications that can start jobs, a user interface with the capability for sending 'job starts' and 'job start' notification via IP.
The main difference with our project then is that it doesn't use state machines in order to execute its jobs. In *reactord*, noticing an event doesn't imply that an action is going to be launched. If this event is expected it will forward the state machines so it will be nearer to run an action. Instead, in *SOS JobScheduler* they don't have states but 'job chains'. A job chain is a sorted sequence of jobs launched by a single 'job start'. This sequence can have parallel jobs and dependencies between them. So it doesn't solve our problem, as it doesn't require a sorted execution of jobs, but a state driven execution of them.

[1]We are obviously talking about UNIX, probably Windows has its own similar services.

However, it has features to take into account that would be easy to develop for our project as plugins, like web service integration (receiving events from web services) or the timeslots. Timeslots are time periods in which the jobs should be executed. In *section 6.2* we explain our state machine oriented solution, which is not limited to time intervals.

2.3.2 Proprietary event driven job schedulers

We believe that being able to modify the code of the project to make it solve your specific needs, and being able to receive improvements on the software from unknown people interested on the project are two critical points that our solution must have. That's why we will acknowledge the proprietary job schedulers that we know in this section, they lack the same main requisite, being FLOSS.

Global ECS is a job scheduler by Vinzant Software that offers the following features[15]:

- Single point-of-control for monitoring and managing enterprise-wide job streams.

  Controller/Agent model uses the power of TCP/IP to simplify communications in a distributed enterprise environment.

- Global ECS has many capabilities that allow for a 'Management by Exception' approach to automating your production environment.

- Multiple Method Scheduling (MMS) allows for simple programming and management of tasks with widely varying repetition schedules.

- Role based security model.

- Launches and controls any command line, including graphical and text programs, batch files, command files and macros.

- Captures return codes to detect job success or failure and allow the system to take appropriate actions.

- Controls sequential execution and branching with sophisticated job dependencies.

- Full support for file and resource dependencies.

- GECS System Events to assist in scheduling and monitoring the production environment.

- Full featured browser-based client for remote console access.

As we can see it is very focused on monitoring, and it does nothing about state machines.

Cronacle and SAP Central Process Scheduling are job schedulers developed by Redwood. According to Wikipedia[9]:

> *"Cronacle is an event driven business and enterprise process automation solution. It was developed by Redwood Software in 1993 and is based on the use of business events to drive IT workload rather than more traditional time date based scheduling. Cronacle also supports a series of extensions for specific purposes, one of which, Insight, was introduced in 2011 as a business process monitor."*

> *"The SAP Central Process Scheduling application by Redwood delivers adaptive, real-time, event-driven job scheduling and process-automation capabilities. This product is sold by SAP AG."*

It is hard to get technical information about this projects on their companies web sites, and we are not really that interested on it. So we will just move forward, and take into account the features they claim to have.

2.3.3 udev

> *"udev is the device manager for the Linux kernel. Primarily, it manages device nodes in /dev. It is the successor of devfs and hotplug, which means that it handles the /dev directory and all user space actions when adding/removing devices, including firmware load."[13]*

Although *udev* is not a job scheduler, as we already said, it has job scheduler capabilities. Also it uses a similar architecture to the one that we use. The project core is the *udevd* daemon process which receives administration orders from the *udevadm* command line program, and 'uevents' from the kernel. These uevents are a kind of event sent through a netlink socket that informs about a device added or removed from the system. It has a library to make your application gain some *udev* functionalities and files were the user and the software distributions should write rules stating how the system must react to an

uevent.
As all the other projects it lacks the state machine running capability, and it only reacts to 'uevents'. So its clearly far from resolving our situation by itself. But instead of that, it is a great source of inspiration for our project. *udev* is a long-term project developed by experienced programmers and very close to the Linux kernel. We got a lot of ideas about the architecture of *reactord* from their code and solved some design issues as well.

Chapter 3

Design

Now we know exactly what we want our project to do, so it is time to design *how* we are going to do it. We already defined the architecture which will point the path to follow, and divide the design efforts in four parts: main daemon, command line daemon controller, library and the plugins interface.

3.1 Daemon

Called *reactord* is the central program of the project. It mainly has to:

- Locate the rules files.
- Parse the rules files.
- Receive control commands.
- Receive remote events.
- Register and run the plugins.
- Manage the state machines.

Before we start discussing these points in deep we will explain the users restrictions to use this daemon. We have to say these restrictions are not entirely implemented for the final project for lack of time, but they were kept in mind in order to easily satisfy them in a near future.

3.1.1 User restrictions

As you probably already know, UNIX is a multi-user system in which there is the *root* user, also known as *superuser*, who has permissions to access all the files, and then there are the other users, which could have more restrictive permissions. These users can be organized in groups so we can set permissions to a whole set of concrete users to restrict them to certain files.

We will make use of this and divide the potential users of *reactor* in three sets with different access to the main daemon. These groups will be *administrators*, *reactor group users* and *others*. The goal of this division is to give daemon access to the major numbers of users in a system while isolating some security issues and keeping it as a one instance only program.

Administrators is in fact just the *root* user, but we name it in plural because we are considering all the users with access to *root* user, either by *sudoers* file[1] or *wheel* group[2]. But the final user is always one, *root*.

By *reactor group users* we understand all the users that the system administrators choose to have some privileges that we will see in the following lines. They are all in a predefined user group, by default called *events*.

Others are simply the other users not considered by the two previous groups. Their access to *reactord* is not controlled by the system administrators so, for security reasons, this must be the most restricted group.

In *tables 3.1* we can see a summary of the restrictions for every kind of user. As we can see, *root* is the only user that can start the daemon. Also remember that *reactord* must check that there is not another instance running before it starts. This is a very common behaviour in UNIX systems.

The principal difference between the three sets of users is the location of the rule files, which every user has a different one. For the administrators state machines we have an standard location for *root* configuration files (`/etc/reactor.d/reactor.rules`). This file, and all the files in the same directory, will be readable by all the users but only writeable by the system administrators. The rule files for the *reactor* group users will be located in theirs *home* directory (`~`), and hidden (therefore the '.' prefix) so it does

[1] *sudoers* is a file which defines the users that can execute commands as superuser by using the `sudo` command.

[2] The *wheel* group is an inheritance of UNIX. It is a group that contains the users with access to `su` command.

	Users		
	Administrators	*reactor* group users	Others
Start *reactord*	✓	✗	✗
Rules file	`/etc/reactor.d/reactor.rules`	`~/.reactor.rules`	✗
Actions user	*root*	User	User
Plugins events notice	✓	✓	✓

(a) General restrictions

		libreactor messages send by		
		Administrators	*reactor* group users	Others
Valid SMs owners	Administrators	✓	✗	✗
	reactor group users	✓	✓ If same user	✗
	Others	✓	✓	✓ If same user

(b) Valid *libreactor* messages by user senders and owners of the receiving state machines.

Table 3.1: Summary of the *reactor* user restrictions

not bother the user by showing with his documents. Others do not have rule files so they can not have state machines running every time the operating system boots. In the other hand they can load state machines with *reactorctl* manually or using a session initialization script, in order to automatically load the rules every time the user logs in. May be it would be interesting for the system administrators that when these other users log out, their states machines would be automatically unloaded. A solution is described on section 6.2.

Some kinds of actions of the state machines we have may need an *uid*[3] to be performed. By now this is only the command actions, which needs an *uid* to make it the user owner of the process, and so restrict it to access the filesystem. Superuser obviously does not need any restriction on its state machine actions (actually it doesn't make sense). But the rest of the users must run their state machines and execute its actions as their own, so we do not put the integrity of the operating system in danger. Also we must not allow a state machine to have a transition to another user's state machine for the same reason.

[3]Stands for 'user identifier'. Is the number used in UNIX-like operating systems to identify a user.

There is one last thing to control about the users. Users can send control messages to the *reactord* through programs using the *libreactor* shared library, or the *reactorctl* program that, as it is explained later, is actually using *libreactor*. Therefore, for example, the *others* set of users may send an event that makes malfunction a *root*'s state machine, or simply delete all its state machines. To avoid this we should set credentials to the events send by *libreactor*. And that is what we can see at *table 3.1b*. Administrators state machines can only rely on their own control messages and not on the rest of the users'. 'events' state machines will work with both their own users' control messages and *root*'s, and finally *others* can rely on all the control messages, except for the other users without higher credentials. Notice a user can not send control messages in order to change or forward another user's state machine from the same set of users. But what about remote events? If there is no control for the remote events as well, the same *others* user may propagate an event to another device, or even to localhost, that makes it execute an unintended *root* command. This problem is trickier than the *libreactor*'s, because the connection only gives us the address of the remote *reactord* but neither the user, nor even the source program. A simple event message could be sent with a tool like `telnet` easily. To solve this we set several configurable security levels:

1. Leave all the plain events in.
2. Set the address which we expect events to arrive from.
3. Encrypt event messages with TLS.
4. Mutual authentication using TLS.

This solution is the same that they use in *syslog-ng*[17] for receiving remote log messages, and probably in many more projects.

3.1.2 Rule parsing

At the beginning of the section we made a list of the internal functionalities in which we can divide *reactord*, and we will follow them more or less to explain its design. The location of the rule files is explained in the previous subsection, we have a unique file for the administrators, and we have to check which are the members of the 'events' group in order to read the file from their 'home' directories. This is one of the first things the daemon has to do.

The next thing is to get and to process the information from those files, and parse them. At the beginning of the development the whole parser was a component of *reactord* but, when developing the plugins part, it became necessary to share it in *libreactor*, so it could be used for the plugins as well, or at least a part of it.
There are some properties that the rules from a plugin and from the daemon are more likely to share. For example a rule, that in our case is the minimal unit to define complete information about how we want our system to behave, will be one line only. Also, in general and as an implication one-line/one-rule property, those rules won't have a complex syntax. This syntax can be defined similarly to a markup language. We can define an expressions in a line by subexpression separator and expression end tokens. The next line is a valid *reactor* rule:

```
state_a state_b event1 &event2&event3 & event4 &event5 PROP myserver:6500
```

The two tokens from the beginning, `state_a` and `state_b`, are the out state and the in state of the transition we are defining. From the next token until `PROP` we have the list of events to notice before we forward to the in state, separated by the `&` character. At least one event is required. From `PROP` until the end of the line is the action to perform. `PROP` is the kind of the action, in this case it is a propagation action. We have three different kinds of actions:

- `NONE`
  Do not perform any action. It is not followed by any action definition.
- `CMD`
  Run a command in a shell. It is followed by the command itself.
  i.e.: `CMD echo ``OK > /tmp/test`
- `PROP`
  Propagate all the events from the rule to an IP address. It is followed by the address with the format *host:port*, or *host* for the default *reactor* port.
  i.e.: `PROP publicserver.com`

So the simplified syntax would be:

leaving_state destination_state list & of & events TYPE_OF_ACTION action

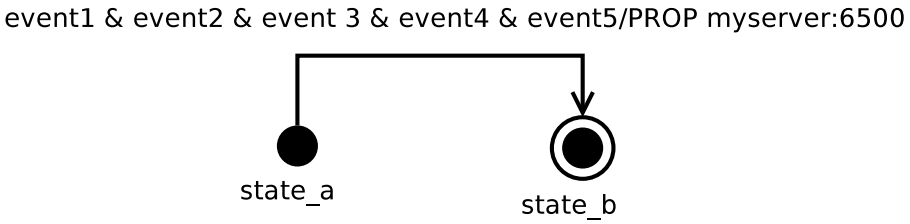


Figure 3.1: Resulting state machine from the example rule.

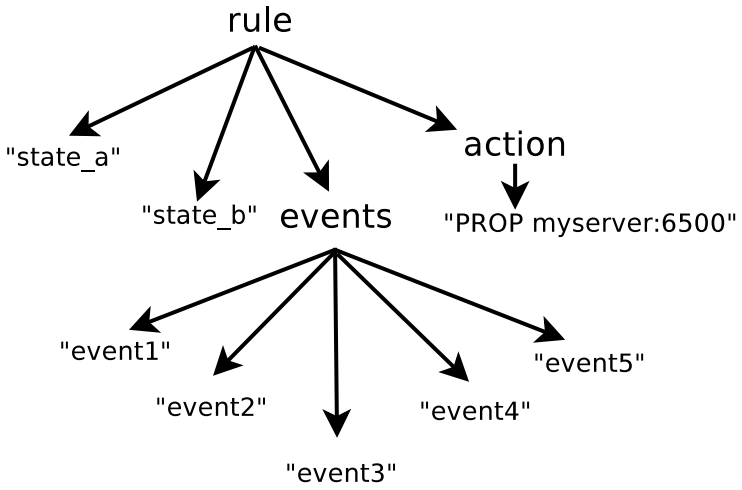


Figure 3.2: Resulting parse tree from the example rule with the explained grammar.

In *figure 3.1* we have the resulting state machine from the example rule.

With the simple kind of grammars we previously explained we can tokenize the rule and build an initial parse tree. The main expression has separated tokens with the space character, and with a `End-Of-Line` terminal character. The third token is the beginning of a subexpression tokens separated by the `&` character and with the space character as terminal. Finally, the fourth token (we are counting the subexpressions as a single tokens) has `End-Of-Line` as terminal character and the same as separator (it has no separators). With this grammar we get the parse tree in *figure 3.2*. As we know how the syntax of the plugins rules may look like (the subset of grammars), but not how exactly how they will be, we made a generic simple parser in which you specify the parameters explained before and we get a parse tree like the one in *figure 3.2*. But this one is not exactly the parse tree *reactord* needs. The 'action' branch gives the 'PROP' unparsed, and it needs the kind of action, the host and the port separated in different nodes. This can't be done with our initial parsing function because we do not support conditional subexpressions.

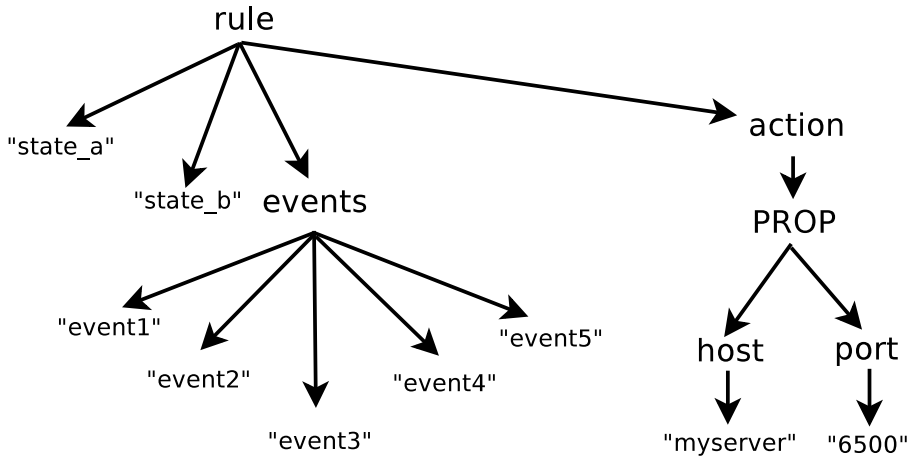


Figure 3.3: Resulting parse tree from final parse process in *reactord*.

That is we can not expect a subexpression after `PROP`, but a different one after `NONE` or `CMD`. We need the final part of the parsing process to be done in *reactord*. The process well be limited to receive the unfinished parse tree, go to the target branch (fourth from root) and parse it with the action kinds conditions. If its `NONE` there can't be nothing after it, so the action node will have only a leaf child with the content "NONE". If we have a `CMD` action it has to make a node with the content "CMD" and a single leaf child with the command in it. In the case of our example we expect two leaf children in the "PROP" node, one with the host and the other one with the port, as shown in *figure 3.3*. The parse tree also contain information about the line number and file of every rule, and informs about errors. This will be explained in detail in the plugin's *section 3.3.2*.
Now the parse tree is ready to be sent to a function which adds the actual transition to our states machines.

3.1.3 Control messages handling

As with the parser, the great part of code of the *reactord* control communication was moved to *libreactor* during the development. This is the code in charge of communicating *reactorctl* and other programs running in the same operating system with *reactord*. Once *reactord* has been initialized, it is permanently waiting for control messages. The format of a message will be detailed in *section 3.3.1* as the it is defined in *libreactor* as well, but by now we can consider that it has an identifier of the message type and the message data. Therefore when a message is sent to *reactord*, it is processed by a callback function. This

function checks the message expecting three kinds of message and act consequently:

- `EVENT`
  The data fields of the message are an event identifier and the credentials of the sender. This information is sent to the event handling function at once. The sender program expects an `ACK` message in return.

- `ADD_RULE`
  It comes with a rule in exactly the same form as in the rule files and the credentials of the sender. It is sent to the parser as a single rule instead of a whole file like in the previous section. The parse tree is sent to the transition adding function. The sender expects a response. If everything went fine it will receive an `ACK`, but if the rule was malformed or another error happened, it will receive `ARG_MALFORMED`. Errors will be logged using *libreactor*.

- `RM_TRANS`
  Contains an state identifier, a number to identify a leaving transition and the credentials of the sender. It is intended that in the future we have a way to visualize the state machines as graphs in which every transition has a number identifying it using graphviz[5] or/and similar tools. This number will simply be the order in which the transition was inserted (*section 6.2*). The data will be sent to the transition removing function. In case everything went well the sender must receive `ACK`, but `ARG_MALFORMED` otherwise. Errors will be logged using *libreactor*.

3.1.4 Remote events handling

The remote events handling uses the same tools from *libreactor* than the local control messaging system, but over the TCP/IP protocol.
Before the remote sender begins to send events to the local *reactord* a plain connection negotiation is needed. The local and the remote hosts need to compare their versions of the software so they can decide which connection-related features are available and so if the connection can be performed. Then the local host will inform about the conditions of the connection to the remote client, which can be accepted or rejected. Those conditions are extracted from a configuration file. When the conditions are accepted then connection begins. If the configuration states that the connection must be authenticated TLS, we should have the remote certificate linked to a local user in order to assign credentials.

Like with the `EVENT` case, the sender expects an ACK after every event received. The difference with the local control messages is that the connection keeps open until the sender has sent all the events. The sender will inform about the end of the set of events with an `EOM`[4] message. If the `EOM` is not received, it can be considered an error.

3.1.5 Plugins management

The *reactor* plugin system is formed by four parts:

- Plugin API.
- Plugin interface.
- Plugin manager.
- The plugins.

The **plugin API** is the set of functions available from *libreactor* and the callbacks sent by *reactord*. The main functionality from *libreactor* expected to be used in the plugins is the generic parser explained in *3.3.2*, and logging functions (*section3.3.4*). The *reactord* callbacks are the functions from the daemon that it set available for the plugin in the registration process. By now the only callback the plugin can expect is the event handling function, so it can make *reactord* notice events directly.
The **plugin interface** is also in *libreactor* and is the specification of the expected functions that the plugin has to implement. It also contains the data structures definitions shared by *reactord* and the plugins. Explained in *section 3.3.3*.
The **plugin manager** is the piece of code of *reactord* that is in charge of the communication with the plugin interface. It loads the plugins from the directory where they are supposed to be located, initializes them and sets them into a list. When we want to unload a plugin it is done by the plugin manager too. By now this is the only things it does, but when poll-like information is needed from the plugins, it will manage the plugins and the functions to call.
The **plugins** are implementations of the different functions from the interface. Explained in *section 3.3.3*.
In *figure 3.4* we can see the generic behaviour of the plugins system. The return variable

[4]EOM stands for End Of Message

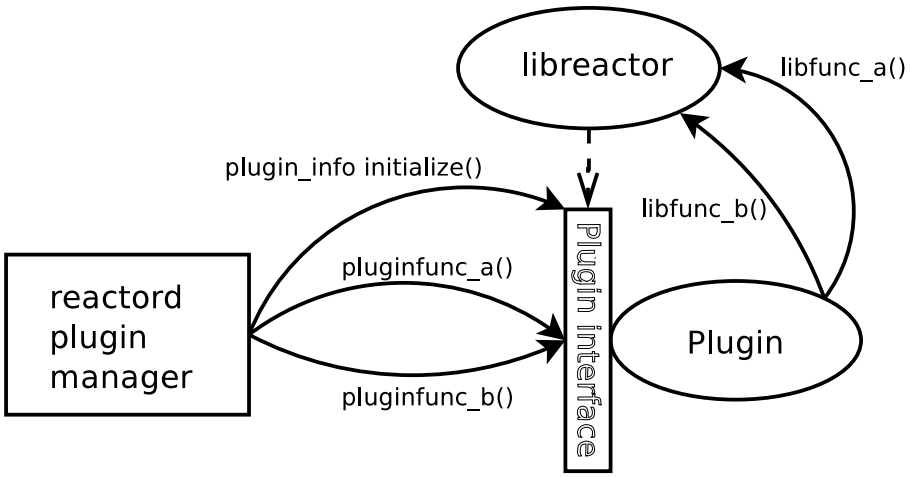


Figure 3.4: Generic behaviour of the *reactord* event system.

from the `initialize()` function is a set of plugin information values and callbacks to the plugins functions. In our case, the plugins are only launched as workers. This means that we will call a plugin main function as a thread which will be running until we stop it, or until we stop *reactord*. This function is expected to be the main function for the new events sensors, and it should call the *reactord* event handling function every time an expected event is detected. Of course by the initialization phase we are talking about the parse of the plugin's rules file.
So, the functions called after the `initialize()` function are those callbacks returned by the plugin, and in our case is the worker's main function.
The *libfunc_x()* functions is where the parser call would be.

3.1.6 State machine management

We have now management actions over the state machines ready to be performed. In order to perform them, we have three input handlers functions:

- Add-transition handler.
- Remove-transition handler.
- Event handler

The **add-transition** handler is a function that expects the parse tree of a rule and the credentials of its owner. It starts by checking for errors in the parse tree. Remember that

when the parser founds a syntax error in a rule, it writes the error on the parse tree. If there are any it will stop and return an error value. It checks next if the states of the transition already exists and in which state machines they are. If the states are in different state machines (and this includes an initial leaving state going to an existing non-initial state) the addition of the transition becomes illegal. We log the error and return an error code. The reason for this is that the state machines become a state machine with several initial states. In *reactor* an initial state is an execution of the state machine. If we have several initial states then we have several executions of the same state machine, and this makes the design a lot more complex. This is illustrated in *figure 3.5*.

Another illegal addition to check is the case in which the user is trying to add to connect two states machines of different users. You may understood that every transition connecting different states machines are illegal. But the case in which we are connecting an state to the initial state of another state machine is legal, because the result is an state machine with just one initial state. The problem comes when the user is trying to connect an state machine of his own with another of higher credentials, because he would be able to leave the other user's state machine unable to start by for example setting a new initial transition with an nonexistent event for its activation. We see it illustrated in *figure 3.6a*. In *figure 3.6b* we can see that the transition addition becomes legal by simply reverting the user roles. Once these checks are passed without problem we save the new states and link them with a transition. We also locate the events in a hash table in order to access them later when we receive their event identifiers. If the transition can be forwarded immediately (the leaving state is initial or is the current state of the state machine) then the transition is set in a current transitions list on the its events. This way the transitions are easily accessible every time we notice an event as well.

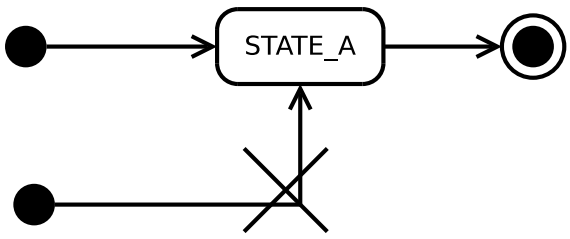
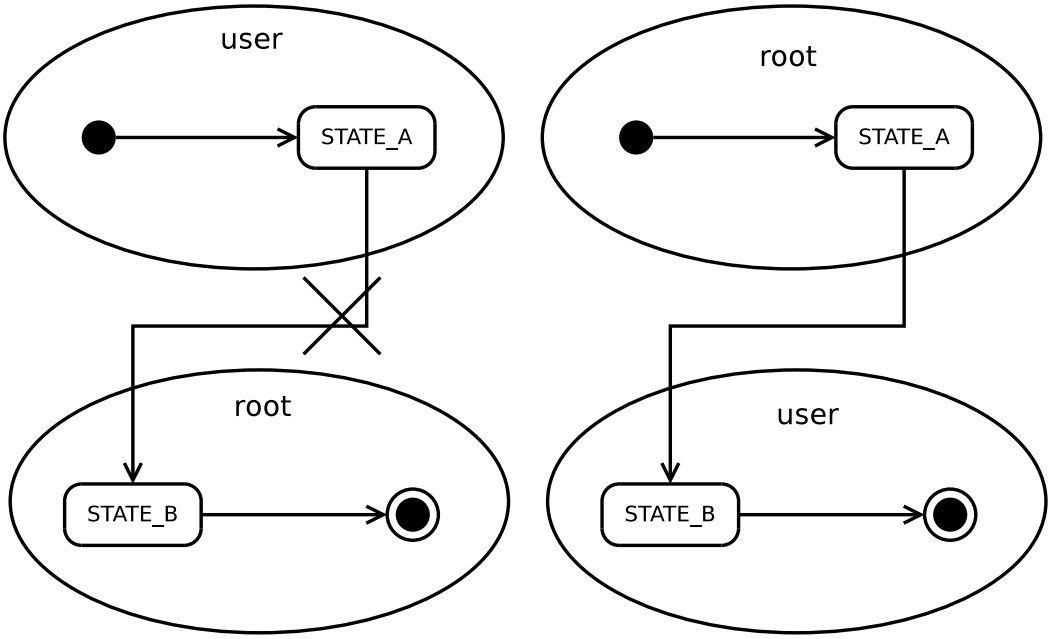


Figure 3.5: Illegal transition addition from an initial state to an existing non-initial state.



(a) Illegal transition addition from a lower credential user state machine to a higher credential user state machine.

(b) Legal transition addition from a higher credential user state machine to a lower credential user state machine.

Figure 3.6: State machine inter-user connection example.

The **remove-transition handler** needs a leaving state identifier with a number identifying one of its leaving transitions and the user credentials. It has to perform two checks. The first one is the credentials check, it means that a lower privileged user can't remove a transition from a higher privileged user. The other check is the existence of the transition. In both cases, if the checks fail, it will stop and return an error value.
Now it is good to remove the transition. To do so it unlinks the leaving state from the transition and the transition from the entry state. So we are removing a reference from the entry state. When we remove all the references from an state we remove this state as well, removing all its leaving transitions. As you can see this can lead to remove all the state machine if one of the following states is the initial state. As you will see in *section 3.1.7*, all the states have a reference to its state machine initial state. So before we can remove the initial state we need to remove all the other states. *Figure 3.7* shows the non-transition references with dotted lines in a state machine with four states. The states have the number of references over them.

There is a case in which this recursive remove action is not enough. When we have a

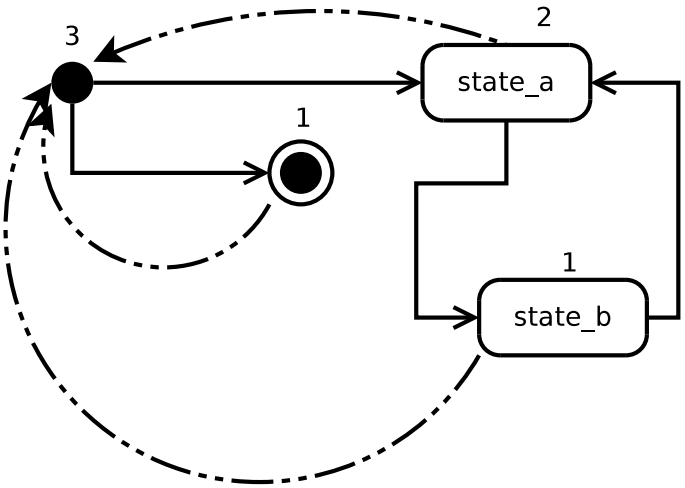


Figure 3.7: Non-transition references to the initial state. The states have the number of references over them.

cycle in the graph that represents the state machine, like the one that begins on `state_a` in *figure 3.7*, and we try to remove a transition before it, we will create two non-connex graph components. This graph will be composed by a subgraph with the states from the initial to the leaving state of the transition we removed, and another from the cycle to the end. *Figure 3.8* is the result of removing the transition from the initial state to `state_a`. It shows clearly the resulting graph with two non-connex components.

To solve this, every time we remove a transition and the destination state is not re-

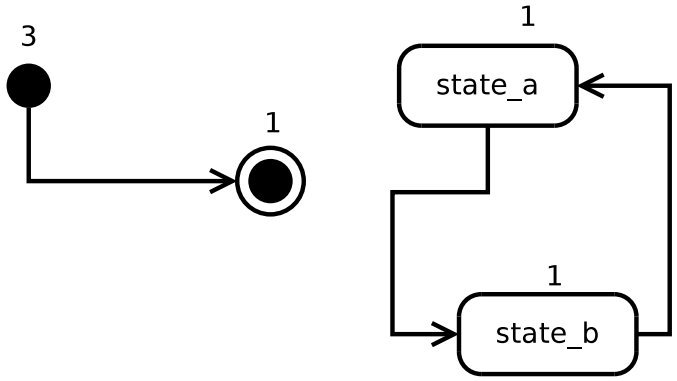


Figure 3.8: Graph with two non-connex components result of removing the transition from initial state to `state_a`.

moved, we check with the BFS algorithm if there is a path between the initial state and the destination state. If there is not, then we have found a cycle and we have to remove

the whole subgraph.

The **event handler** is the function that deals with the received events. When talking about the adding-transition handler we mentioned that the events from the transitions are stored in a hash table for quick access. Every time we receive an event we look in this hash table searching by its identifier. If it didn't find anything is because the event was not expected. Otherwise if we find an event in the hash table it means that this event is a requirement for at least one of all the transitions we have, but is not necessarily needed now. The event from the hash table contains references to the transitions that require it in order to forward to the next state. For every referenced transition the user that sent the event has credentials to, it adds a noticed event and when all the required events are noticed then it can run the action.
In order to run the action first we must check the kind of action the transition has assigned and then run its assigned launched according to the data it contains.
If the kind of the action is *NONE* it won't run any action launcher.
The *CMD* kind of action launcher will execute the stated command in a local shell with the permissions of the owner of the state machine.
For the *PROP* kind of action, the launcher will connect to the remote host, negotiate the configuration of the data transmission discussed in *section 3.1.1*, and send all the events. The host will decide the credentials of the received events by configuration and linking of users to TLS certificates.
Once the action has been executed it has to remove from the hash of events all the transitions that leave from the same state than the one it executed, because from now on these events are not expected by those transitions. We forwarded to the next state. This is probably the most ugly and costly part of the code we have. We go through all the transitions from the same state, which are in a list, get the events and search and dereference the current transitions. It is not hard to see that with a big number of transitions and events this could be a bottleneck.
If the action has not been executed yet, the clearing of the references to the current transitions is easier, because we only have to go to the received event and clear it, we don't have to take into account other events that are not being expected by other transitions.
The last step to perform is to reference the new current transitions to its events. This is done by exploring all the events of all the leaving transitions of the current state.

3.1.7 Conceptual model

In this section we will explore all the concepts from *reactord* we already explained from the abstract perspective that classes offer. *Figure 3.9* shows the UML conceptual mode of the daemon and we can see its relations. Probably you can recognize most of them by the previous explanations of the functions of *reactord*.

We explain every class stating the main attributes and methods, but ignoring irrelevant

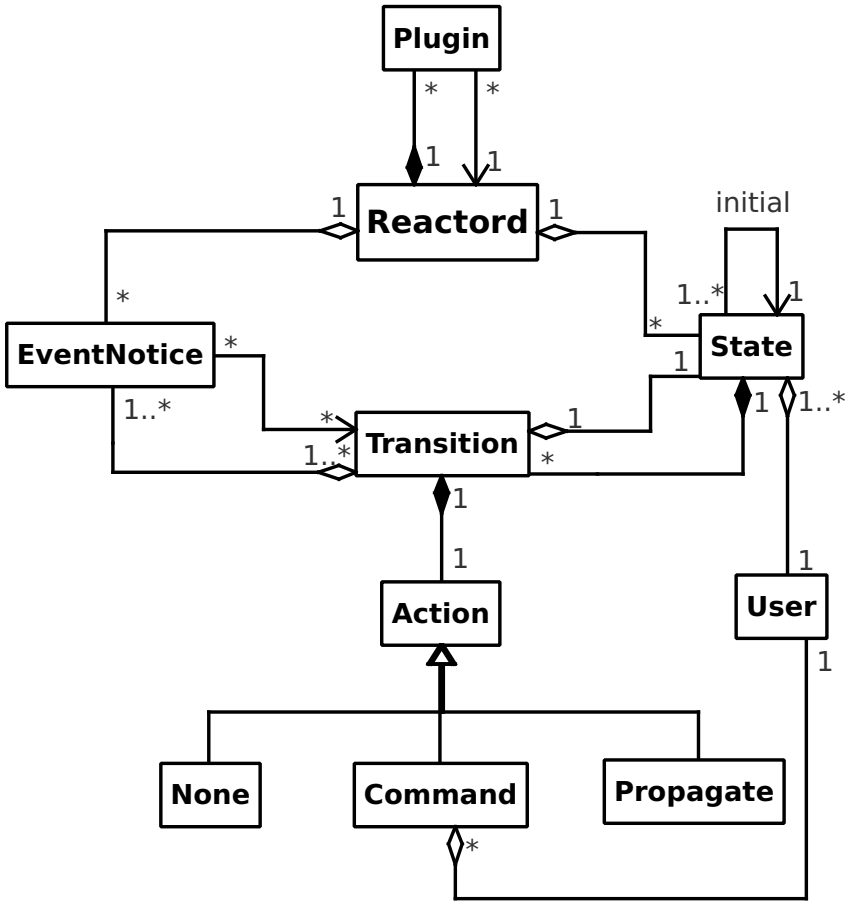


Figure 3.9: *reactord* UML conceptual model

functions for the design like getters, setters, constructors and destructors.

Reactord

This is the main class. Here we have the biggest part of the functionalities of *reactord* and is the piece of code that the will communicate most directly to. It contains the hash

tables of events and states using their own identifier as keys for fast searching, and the plugins loaded. It also has the communication structures for both local control messaging and remote events receiving. These structures will be covered with more detail in *section 4.4*.
The attributes are:

- **events**
  `Event`. Expected events in the state machines. Needed to notice them.

- **states**
  `State`. This is what we use to edit the state machines. Here we have all the states with their transitions.

- **plugins**
  `Plugin`. *reactord* plugins loaded during the initialization.

The functionalities in this class are:

- **Daemon initialization**
  The daemon needs some initial checks before it starts. It is in charge of loading locating and loading the rule files, so here is where *reactord* look for the members of the `events` user group. Also it loads the plugins from the default directory. There are more initializations done in here but they are more related to the implementation than to the design, so we won't cover them here.
  The functions are:

  `load_users`: Given a group name it returns all the members of this group in a list.

  `init_rules`: When called, *reactord* contains the state machines from all the privileged users rule files.

  `load_all_modules`: Given a directory it loads all the valid modules located in there.

  `load_module`: Given a module path, it checks if it is valid and loads it.

- **Final rule parsing**
  The functions that ask for the preliminary parsing of the rules to *libreactor* and make the final parsing of the action in the received parse tree. It is called in the

daemon initialization and the result parse tree is a parameter for the state machines machines management functions.
The functions are:

`parse_rule`: Receives a rule string as a parameter and returns a parse tree. If there were syntax errors they must be stated on the parse tree.

`parse_file`: Receives file path string as a parameter and returns a ordered list of parse trees with the same conditions for each parse tree than with the `parse_rule` function.

- **Local control messaging management**
This is only one function which for a local control message it selects de correct handler.

`attend_cntrl_msg`: Given a communication structure (socket) with data ready to be read, it gets the message with *libreactor* functions process it if needed and calls the correct handler. If there was any error, it sends an error message to the sender and returns an error code.

- **Remote events management**

As *libreactor* only contains the functions to communicate with local processes, we needed actions to make the same but through TCP/IP. These functions are wrappers of the local control ones to make them able to connect to remote hosts. The functions are:

`listen_remote`: It makes *reactord* able to receive remote events. The return value is the structure for communication (socket).

`connect_remote`: It returns a structure for communication (socket) for the client to send events to a remote host.

`receive_remote_events`: Given a communication structure (socket) with data ready to be read it returns a list of events received. For every event received an `ACK` message will be sent.

`send_remote_events`: Given a communication structure (socket) connected with the server and a list of events, it sends the events one by one and waiting for the `ACK`. When all the events are sent it sends a final `EOM` message.

`attend_remote_events`: Given a communication structure (socket) with data ready to be read, it gets the message and sends it to the event-noticing handler. If there was any error, it sends an error message to the sender and returns an error code.

Except for the `attend_remote_events` function, all the other functions could be in the *libreactor* in the future, if needed.

- **State machines management**
  These functions are the operations that can be performed over the states machines. They can also be called *input handlers* because these operations always use input received by *reactord* from external sources.

  `add_rule_handler`: Given a rule parse tree and the user credentials, it adds the transition if it is legal. If not it returns with an error code.

  `rm_trans_handler`: Given a state identifier, a transition number and the user credentials, it removes the transition and the part of the state machine depending on it. If there is some error occurred it returns with an error code.

  `event_handler`: Given an event and the user credentials, it notices the event in its current transitions.

- **Main loop**
  The main function simply enters in a loop waiting for external messages to react to. When a message is received, the correct "attend" function is called. It leaves the loop when the program is stopped.

One could think that some of these functionalities could be isolated in different classes. For example the management functionalities could be in a different class.
We put all these functionalities in the same main class because they do not use any data structure or their data structures depend on *libreactor*. Also, as you can see *Implementation section (4)*, we did not use a object oriented programming language, so class abstraction is not really critical for our project.

State

State is one of four principal components of the state machines, together with EventNotice, Transition and Action. It is the container and link between transitions and is the one that

permits our software react different in different situations.
The attributes are:

- **id**
  `String`. Unique identifier of the state. It is defined by the user using rules.

- **transitions**
  `Transition`. List of transitions leaving the state.

- **fsminitial**
  `State`. Is a reference to the initial, and it is used as an identifier of the state machine.

- **refcount**
  `Integer`. References to the state, by other states that have it as initial state or by transitions.

The functionalities in this class are only getters, setters, a constructor and an dereference based destructor. The cost of setting a new initial state to an state machine is high, because we have to go through all the states of the state machine using a BFS and changing the `fsminitial` attribute.

EventNotice

An EventNotice is a class that represents the expectation of an event by a transition in order to go to the destination state. When we receive an event, we use its identifier to find its EventNotice and obtain the transitions they are related to.
The attributes are:

- **id**
  `String`. Unique identifier of the event we expect to notice. It is defined in the rules.

- **currtrans**
  `Transition`. List of the transitions that are leaving the current states of all the state machines and have the EventNotice as a requirement for its advance.

- **refcount**
  `Integer`. References to the EventNotice by transitions that require it.

The functionalities in this class are only getters, setters, a constructor and an dereference based destructor.

Transition

It is the backbone of the whole project. It binds the other state machine classes together and contains the reacting conditions and the reaction in the same class.
The attributes are:

- **enrequisites**
  `EventNotice`. It is a list of EventNotices required to advance in the state machine by this transition.

- **eventnotices**
  `Integer`. It is a counter of EventNotices.

- **noticedevents**
  `Integer`. It is a counter of events already noticed. When this counter equals `eventnotices`, then we can advance and launch the action.

- **destination**
  `State`. `destination` is a reference to the destination State of the transition.

- **action**
  Reference to the action to be launched when all the required events are noticed.

It has several getters and setters, but the most important function of this class is:

`notice_event`: This function adds one to the `noticedevents` counter. If this counter equals `eventnotices` then it can launch the action and return `true`. If not it returns `false`.

Action

What we have to do when the transition is executed. As we can see and we already said in the model it can be of three kinds. NONE, CMD and PROP.
Action by itself does not have important attributes but it has an important method:

`action_do`: It checks the kind of action it is dealing with and calls the correct action launcher. In case of NONE Action, no launcher is called.

- **NONE**
  It has neither attributes nor methods. It is used when the user only wants to advance to the next State.

- **CMD**
  This kind of Action runs shell commands. The attributes are:

  - **user**
    `User`. User privileges that will be used to run the command.
  - **cmd**
    `String`. Command to run in the shell.

  And the methods:

  `cmd_execute`: It is the command launcher. It runs the command in a shell.

- **PROP**
  This kind of Action propagates all the events received by the Transitions at the same time. The attributes are:

  - **addr**
    `String`. Host IP address or address name in a string.
  - **port**
    `String`. Integer defining the port.
  - **enids**
    `String`. List of events identifiers to propagate.

  And the methods:

  `prop_execute`: Propagates the events to the remote host.

User

This class contains the information about a user, so *reactord* can load the 'events' group users rule files and control the credentials. As we already know we are going to use UNIX as our software platform, we can say that the attributes of the class will be simply the `passwd` struct offered by POSIX implementations[31]:

- **pw_name**
  `String`. User's login name.
- **pw_uid**
  `Integer`. Numerical user ID.

- **pw_gid**
  `Integer`. Numerical group ID.

- **pw_dir**
  `String`. Initial working directory.

- **pw_shell**
  `String`. Program to use as shell.

There are no methods instead of constructor and destructor.

Plugin

Here *reactord* manages the information about the loaded plugins. This information is retrieved by the plugin itself when it is loaded to the daemon. The attributes are:

- **modhandler**
  `Integer`. Reference to the loaded module handler.

- **name**
  `String`. The name of the plugin. It must be unique.

- **version**
  `Integer`. Version of the plugin named `name`. It can be useful for managing different versions of the same plugin.

- **pt**
  `Integer`. This is a reference to the `pthread` running the job-scheduler or event trigger worker.

By now, the methods are:

- `init`: This is the function that returns the set of function callbacks available in the plugin and information like the name and version.

- `start_worker`: This is the thread main function for the job-scheduler worker.

- `stop_worker`: It stops the the thread main function.

3.2 Command line program

This is a really little program called *reactorctl*. Its purpose is the communication between the user and the daemon. It is so simple that it is not even needed to show a conceptual model, because it uses *libreactor* to perform all the communication with the daemon.
It only has a main function that checks the input arguments of the program.

- `-e`: *Send an event to the daemon.* This argument is followed by an event identifier to send to the local *reactord*. When sent it is expected to receive an `ACK` message in response. If another kind of message or none is received, then it informs the user about an error in the communication.

- `-a`: *Add a rule.* It is followed by a rule using the syntax defined in *section 3.1.2*. There is no processing of this string in the *reactord* side, it simply sends it to the *reactorctl* and waits for an informational response. The expected messages are:

  `ACK` - Everything went fine.

  `ARG_MALFORMED` - There was a syntax error. It is logged using the system logger.

  If another kind of message or none is received, then it informs the user about an error in the communication.

- `-r`: *Remove a transition.* It is followed by a transition identifier formed by a transition leaving state identifier and the number of leaving transition, separated by a dot. i.e.: "`STATE_A.1`". There is no processing of this string in the *reactord* side, it simply sends it to the *reactorctl* and waits for an informational response. The expected messages are:

  `ACK` - Everything went fine.

  `ARG_MALFORMED` - The string is not of the form "string.number".

  `NO_TRANS` - The transition defined by the pseudo-identifier does not exist.

  If another kind of message or none is received, then it informs the user about an error in the communication.

3.3 Shared library

In this component of *reactor* is where we put every piece of code that we think that would be useful for developers that want to extend its functionalities, or make use of the existing ones in their applications. So we could divide its main use cases in two:

- *reactord* plugins.

- External programs.

Now we are going to discuss the design of the different parts in which the shared library is divided, and explain the expected uses of them.

3.3.1 Control

This component of the shared library is in charge of the communication of control messages between a server (*reactord*) and a local client, for example *reactorctl*. The messages are obviously divided by requests by the client and responses by the server.
This component of *libreactor* is specially not expected to be used *reactord* plugins because it does not make sense. The biggest advantage of the plugin system is that the communication is at function level, so this is not needed. It mainly has to:

- Make the caller listen for peers.

- Connect to a listening peer.

- Send a message in the correct format to the listening peer.

- Read a received message.

- Stop the caller from listening peers.

Wire protocol

The communication is performed through a simple wire protocol[21], based on two initial integers in the message defining the its size and its mutually known kind of message. *Figure 3.10* shows an scheme of the structure of a message. As we can see the **header** of the message only contains two fields:

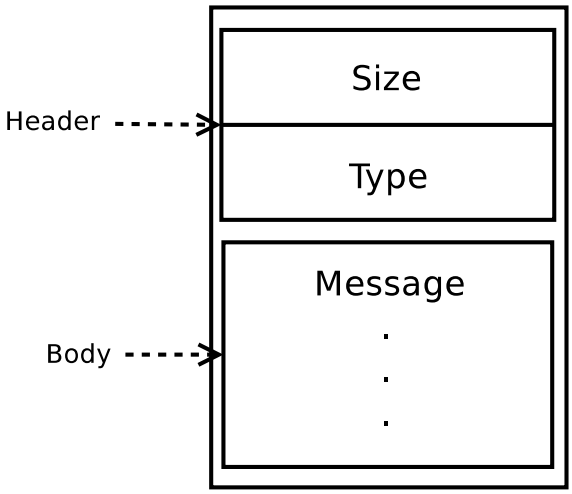


Figure 3.10: *reactor* wire protocol message structure.

- `Size`: This is the size of the message's body in bytes. It is limited by the size of the strings in the operating system, and the size of the field is the size of an integer in the operating system.

- `Type`: It is an integer that identifies the contents of the message's body. The possible types of message are:

  `EVENT` - The body message is an string with the identifier of an event. This type of message is sent by the client as a request.

  `ADD_RULE` - The body of the message is an string with a rule with the format defined in *section 3.1.2*. This type of message is sent by the client as a request.

  `RM_TRANS` - The body of the message is an string with a transition pseudo-identifier. The format is a leaving state identifier and the number of transition. i.e.: "STATE_A.1". This type of message is send by the client as a request.

  `EOM` - This type of message is used to mark the end of a sequence of messages. i.e. a list of events. This type of message is send by the client as the end of a request.

  `ACK` - It confirms that the actions requested by the client were performed as expected. It doesn't have body message. This type of message is send by the server as a response.

  `RULE_MULTINIT` - As response of an `ADD_RULE` message, it states that the rule is illegal because tries to create a state machine with several initial states. It doesn't

have body message. This type of message is send by the server as a response.

`ARG_MALFORMED` - The request to which this message is response to is malformed. It doesn't have body message. This type of message is send by the server as a response.

`NO_TRANS` - As a response of a `RM_TRANS` message, it states that the transition defined by its pseudo-identifier does not exist. It doesn't have body message. This type of message is send by the server as a response.

The message **body** only contains one field defined by the protocol:

- `Message`: Here is were the contents of the message are. The format is not defined by the protocol itself, but by the `Type` field.

Conceptual model

Figure 3.11 shows the conceptual model of the *libreactor*'s control component. `Message`, `Header` and `Body` have their attributes are the same as the message fields explained in *section 3.3.1.*

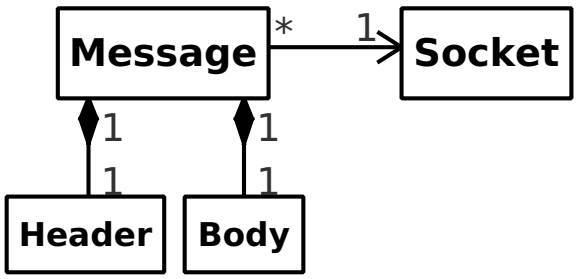


Figure 3.11: Conceptual model of the *libreactor*'s control component.

Message

The attributes are the same as the message fields explained in *section 3.3.1.*

Functions:

`send`: It sends the message using the `Socket` and waits for the response from the peer. If the response is `ACK` returns 0, but if it is different or nonexistent it will return an error code.

`receive`: It receives a message using the `Socket` and the wire protocol.

Header

The attributes are the same as the message fields explained in *section 3.3.1* and it has no methods beyond getters and setters.

Body

The attributes are the same as the message fields explained in *section 3.3.1* and it has no methods beyond getters and setters.

Socket

Bridge for the communication with the peer.

Attributes:

sfd

`Integer`. Reference to the operating system internal data structure to perform communication between programs.

Functions:

`listen`: Constructor. It returns a listening server socket.

`connect`: Constructor. It connects to the local *reactord* socket, and returns the client socket for the connection.

3.3.2 Parser

The *libreactor*'s parser generates parse trees from strings and simple grammar definitions. It is expected to be used in the *reactord* plugins development.

Grammar definition

This is an almost-generic parser. It means that it doesn't parse only one grammar, but several of them thanks to a previous configuration. This is possible because we think we only need quiet simple grammars that define unrelated expressions that can contain listed and nested inner expressions, but at least by now, we don't care about how the final tokens may look.
What defines an expression and configures the grammar are five parameters:

- **Subexpression separator** - Character used to separate subexpressions in the parent expression.
- **End mark** - Character used to mark the end of the expression.
- **Trim** - If used, it removes the useless initial and ending of an expression.
- **Subexpression number** - The number of the subexpression those parameters define in the parent expression.
- **Subexpressions** - Parameters for the inner expressions.

Also, this parser allows comments. Everything after a `#` character will be ignored by the parser.
As this is part of a shared library , let's take as example a crontab, instead of one of our use cases. We have seen before this crontab entry:

```
1 0 * * * reactorctl -e cron_event_1
```

This crontab entry means that every day at `00:01` it will execute the command `reactorctl -e cron_event_1`. In *table 3.2* we can see the meaning of the fields and the allowed values. Obviously, the last field is the command to execute. Our parser, as we said, does not care

Field name	Allowed values	Allowed special characters
Minutes	0-59	* / , -
Hours	0-23	* / , -
Day of month	1-31	* / , - ? L W
Month	1-12 or JAN-DEC	* / , -
Day of month	0-6 or SUN-SAT	* / , - ? L #

Table 3.2: Format of a crontab sorted by fields appearance.

about the allowed values. This must be checked lately. But we can see clearly three rules for the generation of a parse tree that our parser can use.
First of all, the whole expression is a line, so the end mark is the `EOL` character.
Also we can see that, except for the last field, all the fields are separated by the white space character. We also have to take into account that if between field we have several

white spaces, it doesn't mean that we have empty values, so we should trim it. Finally the rule always have six fields, so the sixth field is the last one no matter what it contains. The resulting configuration would be:

- **Subexpression separator**: ' '
- **End mark**: `EOL`
- **Trim**: Yes
- **Subexpression number**: -
- **Subexpressions**:

  Subexpression separator: `EOL`

  End mark: `EOL`

  Trim: No

  Subexpression number: 6

  Subexpressions: None

Using the end mark character as subexpression separator is the way to say that there is no separator.

Result

The result of the parsing process is a structure as it is defined by grammar definition. If the expression had a wrong syntax, then the result should inform about the error instead. The structure is a simple tree with the main exception at the root, the subexpressions as branches and the tokens as leaves. Our example's result would look like this:

- **Expression**:

  Subexpression 1: 1

  Subexpression 2: 0

  Subexpression 3: *

  Subexpression 4: *

  Subexpression 5: *

  Subexpression 6:

  Subexpression 1: reactorctl -e cron_event_1

Conceptual model

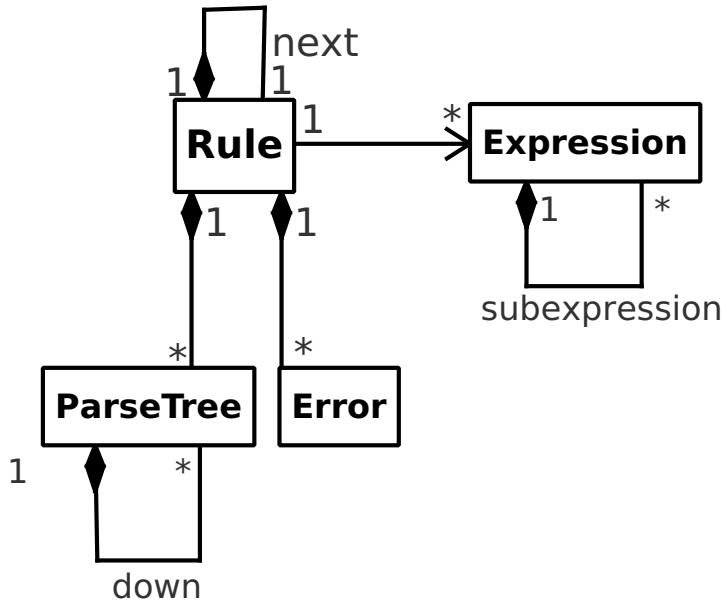


Figure 3.12: Conceptual model of the *libreactor*'s parser component.

Rule

The main class of the parser, contains the parse tree and the information related.

Attributes:

line

`String`. The unparsed rule.

expr

`Expression`. The definition of the grammar.

linen

`Integer`. Number of line of the rules file. -1 if the rule was not extracted from a file.

file

`String`. Path to the rules file. Empty if the rule was not extracted from a file.

ptree

`ParseTree`. Result parse tree from the `linen`. If `errors` is not empty, `ptree` is.

errors

`Error`. If in the parse process any error was detected it is specified here. If `ptree` is not empty, `errors` is.

next

`Rule`. Next rule in the same file.

Functions:

`parse_rule`: Parses `line`.

`parse_file`: Parses all the lines in `file`.

Expression

The simple grammar definition.

Attributes:

exprnum

`Integer`. Is the number of subexpression from its parent which grammar this *Expression* class defines. If it defines the root expression this value is not needed.

tokensep

`Character`. Subexpressions separator character.

end

`Character`. Expression end mark character.

trim

`Boolean`. If true, ignore all the white spaces at the beginning and at the end of the subexpressions when parsing.

subexpr

`Expression`. List of `Expressions` sorted by order of appearance.

The functions are only getters, setters, constructor and destructor.

Error

Rule malformed syntax information.

Attributes:

pos

`Integer`. Position where the beginning of the error is located.

msg

`String`. Message with information about the error.

The functions are only getters, setters, constructor and destructor.

ParseTree

The result of a correct rule parsing.

Attributes:

data

`String`. It is the content of this token.

down

`ParseTree`. List of subexpressions.

pos

`Integer`. Position in the unparsed rule string.

The functions are only getters, setters, constructor and destructor.

3.3.3 Plugins interface

Here we have data structures and the definition of functions needed both by *reactord* and the plugins. It does not contain any implementation, so we won't show any conceptual model here. In the other hand we will explain the data structures and function definition from both sides.

Data structures

We only have three shared data structures.

PluginInfo

- **version**

  `Version`. Version of the plugin.

- **name**
  `String`. Name of the plugin.

PluginServices

- **version**
  `Version`. Version of the plugin manager.

Version

- **major**
  `Integer`. Major version number.

- **minor**
  `Integer`. Minor version number.

Function definitions

We will divide this by two groups of functions.

- Implemented by *reactord*:

  `Event handler`: Given a an event identifier, the function notices it.

- Implemented by the plugins:

  `init_plugin`: Given a `PluginServices` it returns the plugin's `PluginInfo`.

  `main_thread`: It is called as a thread and it executes the plugin's job scheduler.

3.3.4 Data log

This part of *libreactor* for storing log messages. By now it is a simple wrapper to the *syslog* API[11]. We consider four levels of messages:

- `INFO`: An informational message. It only informs about something the software did in a normal workflow.

- `WARNING`: It is an alert about something unexpected that can lead to future malfunctions of the software.

- `ERROR`: Something went wrong and the software failed.

- `DEBUG`: Like `INFO`, but it is information only useful when debugging the software.

In order to do that *libreactor* offers a set of log functions:

- `info`: Given a string this function logs it as an `INFO` message.
- `warn`: Given a string this function logs it as an `WARNING` message.
- `err`: Given a string this function logs it as an `ERROR` message.
- `dbg`: Given a string this function logs it as an `DEBUG` message, only if the program is in debug mode.
- `dbg_e`: Given a string this function logs it as an `WARNING` message with the errno value, only if the program is in debug mode.
- `die`: Given a string this function logs it as an `ERROR` message and exits the program.
- `close_log`: Closes the connection with *syslog* if any.

Chapter 4

Implementation

First of all let's remember the most implementation-related preliminary decisions we took in *section 1.4* about the platform. We choose UNIX-like operating systems to focus from the beginning, and so we are using it's API (POSIX). For using this API we followed the indications of a book we think it worths a mention: *The Linux Programming Interface*[20]. This book, which has been a life saver for this project, not only specifies and explains with examples the Linux API, but also states which parts of the API are part of POSIX and in which versions, so it is easy to make software POSIX compliant.
As we don't want to copy all the code from the project here, we are going to comment the pieces of code that need special attention. Also we are going to list and explain the tools used and the code structure followed for each part of the project.
The source code of this project is available for review in the GitHub repository `https://github.com/alvarovn/reactor`[32].
In the repository we can also find the `README`[33], where we have a user manual and a developer manual for *reactor*.

4.1 Programming language

The programming language selection was made taking into account many points of view for our long term project. Being the project a daemon running constantly in our machines, first of all we considered the efficiency as a critical point. This almost discarded all the interpreted languages like Python or Ruby.
The possibility of making *reactor* a cross-platform project is desirable but not really

important, so keeping efficiency as more important we can also discard Java.
So we want to use a low-level compiled language. Probably the most important languages we have left are C and C++. The decision between them is not easy, and it is almost deciding between a low-level imperative programming language and a low-level object oriented programming language. From all the years in college I am more used to OOP for big projects design than imperative languages. Also as we can see in the previous section, the design uses classes. But this is not only about what I know to make this project more easy to develop, but to try also to make it useful to others, and learn. Finally we decided to use C because not having the abstract layer of OOP makes us develop more efficiency conscious software and because making a big project with an imperative language was a challenge. But maybe, the main reason for using C instead of C++, is because the biggest part of system daemons we checked were developed in C (*anacron*, *udev*, *syslog-ng*, *systemd*, *upstart...*), so it is easier to adopt their design decisions. It is also a GNU Standard recommendation, as C is a simpler language than C++ and more people knows it[19].

4.2 Tools

Apart of the language we used several tools for the implementation. In this section we are going to name, describe and explain the use we made of them. The tools we are talking about are from the compiler to the IDE.

4.2.1 GLIBC

Probably the most obvious tool is the C standard library. In our specific case we are using *GLIBC*[1] which as they say[26]:

> *"The GNU C Library is primarily designed to be a portable and high performance C library. It follows all relevant standards including ISO C99 and POSIX.1-2008. It is also internationalized and has one of the most complete internationalization interfaces known."*

So we have "system calls" and other basic facilities covered by it, such as string treatment functions or memory management.

[1] GNU C Library

4.2.2 GCC

GCC is the compiler we use for our C code. It gives some useful language extensions to the ANSI C standard[24], for example the attribute assignment to functions, variables and types, useful for defining visibility and many other things. It is considered the standard compiler for several UNIX-like systems such as Linux or BSD, and also it works on non-UNIX OSes like Windows. Also it is GPL-licensed.
Notice that as they say, *GCC* is a collection of compilers for several languages, including C++ and Java, although we use the C compiler[25]:

> *"The GNU Compiler Collection includes front ends for C, C++, Objective-C, Fortran, Java, Ada, and Go, as well as libraries for these languages (libstdc++, libgcj,...). GCC was originally written as the compiler for the GNU operating system. The GNU system was developed to be 100% free software, free in the sense that it respects the user's freedom."*

We choose this compiler because is the most standard choice, but there were others to take into account, even to consider for the future. For example *CLang*.
CLang is the C frontend of the *LLVM* compiler project. Its goal is to offer a replacement to the GNU Compiler Collection (GCC). Development is sponsored by Apple. Clang is available under the University of Illinois/NCSA License. Its main features are fast compiles with low memory use, more expressive diagnostics than *GCC* and *GCC* compatibility. It comes with a static analyser and it is written in C++.

4.2.3 GNU build system

The GNU build system is the popular name for a set of programs also known as *Autotools*. Its purpose is to assist in making source-code packages portable to many UNIX-like systems. This set is formed by:

- *GNU Autoconf*[22]

> *"Autoconf is an extensible package of M4 macros that produce shell scripts to automatically configure software source code packages. These scripts can adapt the packages to many kinds of UNIX-like systems without manual user intervention. Autoconf creates a configuration script for a package*

> *from a template file that lists the operating system features that the package can use, in the form of M4 macro calls."*

- *GNU Automake*[23]

> *"Automake is a tool for automatically generating 'Makefile.in' files compliant with the GNU Coding Standards. Automake requires the use of Autoconf."*

  Maybe the GNU Project description is not comprehensible enough.
  GNU Automake produces portable makefiles for use by the make program, used in compiling software. The makefiles produced follow the GNU Coding Standards.

- *GNU Libtool*[27]

> *"GNU libtool is a generic library support script. Libtool hides the complexity of using shared libraries behind a consistent, portable interface."*

  It also gives us a cross-platform wrap for the dynamic libraries management functions.

In this case we also had alternatives. This set is formed by:

- *CMake* - It is a cross-platform build system generator that is being widely adopted and proven for large scale software development. *CMake* uses its own scripting language. The implementation architecture is far more unified than GNU Autotools and it runs much faster.

- *Scons* - It is also cross-platform. Based on a full-fledged programming language, Python. This means you can make the build system do pretty much anything you can figure out how to program, if it doesn't do it already. This also means it doesn't reinvent the wheel, and uses a tried-and-proven syntax. Can be distributed with the software product, so users do not need to install it. This reduces the dependencies your users need to python, which almost everyone already has (or can easily get).

Although those are options to consider, and we will in the future, by now we decided to use *Autotools* because it is the most standard way to build cross-platform packages, and because by now is still the most adopted one.

4.2.4 GIT

This is our choice for version control system[4]:

> *"Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency. Git is easy to learn and has a tiny footprint with lightning fast performance. It outclasses SCM tools like Subversion, CVS, Perforce, and ClearCase with features like cheap local branching, convenient staging areas, and multiple workflows."*

Right now is the most adopted tool for that matter and it is being used for large projects. It also gave us the opportunity of using the `http://github.com` hosting service for projects source code.

4.2.5 KDevelop

This is the IDE choice[7]. As the IDE is very personal choice that does not concern other developers in the project, I did not choose the most "standard" or easy to use for other developers. After trying some different C/C++ IDEs like Eclipse, Netbeans and KDevelop, I stayed with KDevelop because it is the most friendly and with the most useful features for me.
It has strong support for *CMake*, which could be very useful in the future, and supports Autotools just fine.

> *"Kdevelop is a free, open source IDE (Integrated Development Environment) for Linux, Solaris, FreeBSD, Max OS X and other Unix flavours.*
> *It is a feature-full, plugin extensible IDE for C/C++ and other programming languages.*
> *It is based on KDevPlatform, and the KDE and Qt libraries."*

We should mention that KDevelop, as the other IDEs mentioned, uses *GDB* for debugging, which is pretty useful.

4.2.6 Valgrind

Another *reactor* lifesaver[14]:

> *"Valgrind is a GPL'd system for debugging and profiling Linux programs. With Valgrind's tool suite you can automatically detect many memory management and threading bugs, avoiding hours of frustrating bug-hunting, making your programs more stable. You can also perform detailed profiling to help speed up your programs."*

4.2.7 GLib

> *"GLib provides the core application building blocks for libraries and applications written in C. It provides the core object system used in GNOME, the main loop implementation, and a large set of utility functions for strings and common data structures."*

GLib[2] is mainly used by us for its generic data structures like hash tables and lists, and its operations. We wrapped it with our own interface so we can easily make our own specific data structures.

4.2.8 libevent

> *"The libevent API provides a mechanism to execute a callback function when a specific event occurs on a file descriptor or after a timeout has been reached. Furthermore, libevent also support callbacks due to signals or regular timeouts. libevent is meant to replace the event loop found in event driven network servers. An application just needs to call event_dispatch() and then add or remove events dynamically without having to change the event loop."*

We use *libevent*[8] in *reactord* to monitor the socket file descriptors to see if they are ready for I/O, instead of using directly the non-POSIX Linux's *epoll()*, or the POSIX compliant but less scalable *poll89*. *libevent* is a wrap for this functions that takes the decision of when is the best time to use which function, and makes it more easy to use. Also is cross-platform and works in Windows.

4.2.9 Check

For the most complex parts of the code, and the most likely to be changing we used Check[1] as unit testing framework.

[2]GNOME Library is not to be confused with glibc (GNU C Library).

> *"Check is a unit testing framework for C. It features a simple interface for defining unit tests, putting little in the way of the developer. Tests are run in a separate address space, so Check can catch both assertion failures and code errors that cause segmentation faults or other signals. The output from unit tests can be used within source code editors and IDEs."*

It is really simple and useful. The learning curve is very little even if you haven't used unit testing frameworks before.

4.3 Methodology

We did not followed a concrete implementation strategy for the whole project apart from iteration based methodology in which we changed the design of the project when we were implementing it. This way we fixed design mistakes detected, for example, when executing it. Also we added to the design features with a low difficulty/usefulness ratio whenever we thought of them.
Once implemented we tested manually most of the pieces of code. For the most complicated parts we did use a concrete methodology for the implementation, based on unit testing of the finished work. This methodology is called *test-driven development*, also known as "test a little, code a little", and as you can infer from its names is based on first defining the tests as function specifications, and then implementing the functions. This way we assure that our implementation comply with our functions specification, and that when we modify it we don't break anything.

4.4 Daemon

As we said in the previous sections, the daemon is mainly a server waiting for messages from different sources to manage them. For such a communication, in UNIX we have several ways to interact between applications like files, signals, pipes, shared memory or sockets.
We thought that for our purposes, which involves both local and remote communications the best and easiest way to achieve them is using sockets.
For control messages we use a UNIX domain stream socket, which only work in local.
For remote events we use an Internet domain stream socket.
We do not have any main-loop in our code for receiving messages, because as we said we

use *libevent* to manage the monitoring in the sockets.
There are two parts of the design that has not been implemented yet because of the lack of time. Those are the credential control and the BFS to control the two separe graphs issue. The lack of them does not break anything but the first one changes the behaviour of the workflow.

Using an imperative programming language instead of an object oriented programming language, the implementation differs from the design but keeping the essence and the functionality.

4.4.1 Code structure

The conceptual model (*section 3.1.7*) classes will be implemented in modules. Each class will be, at least, one module with an struct with the attributes, and the methods as functions. In case the class has some 'private' attributes, such as reference counters, we will use opaque structs and getter/setter functions. Even though that we will try to avoid the abstraction for the sake of abstraction.
In the `Reactord`'s class case, we have the `reactord` main module but we moved some functionalities into different modules. We have the `inputhandlers` module which contain the handlers for events, 'rm-transition' command and 'add-rule'. Also we have the remote communication wrapper functions in the `remote` module.

4.4.2 Code to consider

PROP execution

The execution of the propagation is done by calling the propagation function as a thread:

```
1 void action_do(struct r_action *raction){
2     pthread_t t1;
3     int s;
4     if(raction == NULL){
5         dbg("No action to run", NULL);
6         return;
```

```
7     }
8     switch(raction->atype){
9         case CMD:
10            cmd_execute((struct cmd_action *) raction->action);
11            break;
12        case PROP:
13            s = pthread_create(&t1, NULL,
14                 prop_execute_thread,
15                 (void *) raction->action);
16            if(s != 0)
17                dbg("Unable to create the thread to propagate the events",
18                  strerror(s));
19            break;
20        default:
21            /* CMD_NONE */
22            break;
23    }
24 }
```

We can see in line 13 that we call `prop_execute_thread` with `pthread_create`. We do this that way because if we propagate the events to localhost, then we enter in a deadlock where `prop_execute_thread` is waiting for an ack that can not arrive because it can't reach the code to attend the received events.

Action polymorphism

To implement the polymorphism-like behaviour of the actions, we simply use a `void` pointer in the `action` struct, and an identifier of the type of this `void` pointer content. The common functions for all the actions will check the type and call the correct function:

```
1 struct r_action{
2   enum a_types atype;
3   void *action;
4 };
```

4.5 Command line program

There is nothing really remarkable in the implementation of the command line program. It is just a single module with the main function checking for the entry arguments. The arguments control is performed with the `getopts` library from *glibc*. The option type is what sets the header of the message and the content is just sent along with the header without being checked.

4.6 Shared library

We use *Libtool* to build *libreactor*. Here is from *reactord*, *reactorctl* and any other application that communicates locally with *reactord* get thefile descriptor of the UNIX domain stream socket used for that communication.

4.6.1 Code structure

The code is organized by the functionalities of *libreactor*. We have four modules:

- `cntrl`
  Local communication logic.

- `parser`
  Rule parser logic.

- `log`
  Syslog wrappers.

- `util-private`
  By now this only contains wrappers to the `read` and `write` syscalls. This is useful for the unit testing so we can mock them without the need of really reading or writing in any file. Explained later.

Each of them has a header, and also there are headers for the plugin's interface and a global one that includes all of them.

4.7 Tests

The tests of this project are centred on the most complex pieces of code. For this we make unit testing with *Check* framework. The decision of making unit tests was taken when the project already began the development and had some parts already done. It was when we saw that some pieces of code were easily breakable by additions and modifications that we began to look for unit testing frameworks for programs written in C language.

We made tests for the control messages communication and for the parser.

4.7.1 Control

We check four things on this tests:

- **Connection** - Checks that we can both listen and connect correctly, so we obtain valid socket file descriptors
- **Correct message receiving** - Checks that in normal conditions the message arrives correctly to the user of `libreactor-cntrl`.
- **Shorter message receiving** - Simulates an unexpected disconnection. We receive a header with a size for the rest of the message, but the message is shorter. The result is supposed to be a NULL message.
- **Error message receiving** - Checks that `libreactor-cntrl` behaves as expected when `read` fails (returns -1).

What we are testing here are functions that make use of the syscalls `read` and `write`, and for our tests we want them to return some specific values, or act as they failed.
With this purpose we made wrappers for these syscalls in *libreactor*:

```
1 ssize_t reactor_read(int fd, void *buf, size_t count){
2     return read(fd, buf, count);
3 }
4
5 ssize_t reactor_write(int fd, void *buf, size_t count){
```

```
6     return write(fd, buf, count);
7 }
```

As you can see they do not change anything in the behaviour of the system calls, and they are used always `read` or `write` are needed.
But on the tests we reimplement these functions to make mocks of them. This means that in the tests, the software will act as the syscalls were real, but they are just dummy functions. So in the tests what it is being used is:

```
1  ssize_t reactor_read(int fd, void *buf, size_t count){
2      struct r_msg *msg = NULL;
3      int msgsize = 0;
4      int leastsize = 0;
5      switch(fd){
6          case READ_CORRECT:
7              msg = &eventmsg;
8              break;
9          case READ_SHORT:
10             msg = &badeventmsg;
11             break;
12         case READ_ERROR:
13             return -1;
14         default:
15             msg = &ackmsg;
16     }
17     if(msgp == NULL)
18         msgp = (void *) msg;
19     msgsize = sizeof(*msg);
20     leastsize = msgsize - (int) (msgp - (void *) msg);
21     if(count > leastsize)
22         count = leastsize;
23     memcpy(buf, msgp, count);
24     msgp+= count;
```

```
25      return count;
26  }
27
28  ssize_t reactor_write(int fd, void *buf, size_t count){
29      switch(fd){
30          case WRITE_CORRECT:
31              break;
32          case WRITE_SHORT:
33              count = count > 0 ? count -1 : count;
34          case WRITE_ERROR:
35              return -1;
36      }
37      return count;
38  }
```

We are using the file descriptor number to code what we want to receive from the system calls.

4.7.2 Parser

We test both the *reactord* parser, and the less specific *libreactor* parser. The things we check are:

- When there is a subexpression, the parser detects it fine whether if it has separators on it or just one token. i.e.: "`A B` **`e1`** `NONE`" and "`A B` **`e1&e2 &e3`** `NONE`".
- An empty subexpression is an error. i.e.: "`A B e1& & e2 NONE`".
- An empty rule is a NULL parse tree (empty line).
- Ignores comments.
- The `Action` part of the *reactord* parse tree is built correctly.

Chapter 5

Planning and economic study

In this section we will check the resources estimations and the final result of this project, being those resources time and money.

5.1 Tasks and temporal distribution

That is the planning followed 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:
December 10th, 2011 (**60 days**)

- *First specification iteration of the project*
  Define the main project's concept in which the rest of the project will be based, as a draft.

- *Define use cases*
  Use cases to support the concept.

- *Collect general information*
  Check the viability of the concept.

January 26th, 2012 (**16 days**)

- *First design iteration of the project*
  After checking the viability of the project it's time to look for the better solution. This includes an implementation point of view for the selection of technologies.

February 2nd, 2012 (**7 days**)

- *Makefile.am and configure.ac*
  Learn the basics of *Automake* and *Autoconf*.

- *Log functions*

- *'events' group users check*

February 16th, 2012 (**14 days**)

- *Main data structure of the daemon done and working*

- *Wrappers to all the third party data structures (mainly glib)*
  As it's expected to improve the efficiency of the software in the future, we want to wrap all the declarations of the third party basic data structures we use, so when we re-implement them the changes to any other non-related code will be minimal.

- *Daemon socket ready to receive events from other processes*
  The poll management performed by *libevent*.

- *Execute shell commands functionality*

- *Dummy first version of control program*
  Its purpose is only to send events, so we can test the daemon.

After this deadline what we have are the very most basic and principal functionalities of the project. This is, a daemon with some states machines defined in it that receives event messages for those states machines from a socket and executes shell commands if assigned to the transitions. There's no way to read the states machines from anywhere yet so it has some test hard-coded state machines.

February 29th, 2012 (**13 days**)

- *Rule files*

  Define syntax.

  Parser function.

  Define files location by user.

March 9th, 2012 (**9 days**)

- *Control program with basic functionalities*

  Add transition.

  Manual event trigger.

With this step done, we have the first basic but functional version of the project.
March 16th, 2012 (**7 days**)

- *Shared library*
  This is expected to be easy and fast to finish, because it has been already implemented for the control program.

March 30th, 2012 (**14 days**)

- *Propagate action*
  This was initially thought not to be an action, but to be an event filter with configuration files similar to the rule files.

April 13th, 2012 (**14 days**)

- *remote events*
  Daemon socket ready to send and receive IP event messages.

- *Plugin workers interface*

April 27th, 2012 (**14 days**)

- *Test plugin*

  As much useful as time we have left to implement it.

- *Current state storing*
  The idea is to let the user mark states which if *reactord* is stopped, it will save that it was in this state before. When *reactord* is started again, it will begin the state machine at this state instead of the initial state.

  Define syntax.

  Parser function.

  Saver function.

  Define files location by user.

  Add the option to the control program.

- *Trigger events on action finalization*
  We want a kind of event detected by *reactord* itself, because its *reactord* the software that generates it. This is the finalization of an action, so we can make a state machine wait until an action terminated. The events would be finalize, error and success.

- *Performance analysis*
  This is the last task because we want to test the whole software.

May 30th, 2012 (33 days)

- *Write this document*

So the final deadline of the project is 05-30-2012, and the total number of days of this final project is 201 days. We have to take into account that between these days there are holidays.

5.2 Final deviation

The final deviation from the initial planning is not very relevant, even though some tasks hasn't been finally performed. That was actually expected, and that's why in the last deadline we put the less important tasks.
Now we are going to list the deviations. We have to take into account that every time we change any deadline, the following deadlines are altered as well because they need the same time as before. We are not going to explain all the deadlines modifications, only the ones that needed a different amount of time than expected.
The deviations are explained as follows:

- **February 2nd, 2012** -> February 9th, 2011 (7 days more)
  I had never used *Autotools* before, and in the beginning its usage seems a little tricky. The problem also was finding the proper structure of the source files.

- **February 16th, 2012** -> March 1th, 2011 (7 days more)
  The problem here was a bad and hard implementation of the protocol between the daemon and the *reactorctl*. The right reimplementation took its time.

- **March 9th, 2012** -> March 20th, 2012 (3 days less)
  Once the protocol was defined, the control program was really easy to develop. All the hard work was already in the daemon.

- **March 16th, 2012** -> March 23th, 2012 (4 days less)
  Easier to make than expected. Once we learned the usage of *Libtool* the work was just moving code and changing includes.

- **April 27th, 2012** -> April 27th, 2012 (7 days less)
  This is the deadline for the implementation process, so we did not change it. But already were a week behind the schedule, so there were things that couldn't be done. Finally we only made the first point, a simple plugin implementation that can be used as an example.

5.3 Budget

Here we will deem the economic cost for the realization of this project in a real environment.

5.3.1 Software

In this section we will analyse the costs of the software needed to develop this project. As we are talking about software and its price depends on its license and the use that we make of it, we will specify the the license. *Table 5.1* shows these costs. This is an advantage that usually comes with FLOSS, its also payment free.

5.3.2 Hardware

In the *table 5.2* we show the hardware costs of our project. As we can see we only have a laptop and the source code hosting server. The server is a free service for open-source projects, and the laptop is a mid-class computer, enough for our goals.
The recovery field is a factor to multiply to the cost. To calculate it we make the inverse of the time in which the hardware cost will be recovered multiplied by the time it will be used for this project.

5.3.3 Personal

This is the cost of the work-power for this project. We separate it in three roles with different costs per hour. All of them were performed by me. The total number of hours stated by the rules of the final project is 600.
The results are shown in *table 5.3*.

	License	Cost
GCC	GPLv3	0€
GNU Build System	GPLv3	0€
GIT	GPLv2	0€
KDevelop	GPLv2	0€
Kile	GPLv2	0€
Valgrind	GPLv2	0€
GLib	LGPLv2	0€
libevent	BSD	0€
Check	LGPL	0€
Ubuntu	Mainly the GNU GPL and various other free software licenses	0€
Total		**0€**

Table 5.1: Software cost

	Units	Recovery	Cost	Total
Dell XPS M1330	1	0.25	1.100€	275€
Source code hosting (GitHub)	1	0	0€	0€
Total				**275€**

Table 5.2: Hardware cost

	Hours	Cost/hour	Total
Architect	190	60€	11.400€
Software analyst	160	50€	8.000€
Programmer	250	30€	7.500€
Total			**26.900€**

Table 5.3: Personal cost

Chapter 6

Conclusions and future work

In this chapter we will review what was expected to be done by the end of the final project, what has finally been done and how. Also we will talk about what was expected and has yet to be done, and ideas that came during the development process but can not be implemented because of the lack of time.

6.1 Goals review

Our goal at the beginning of this final project was to develop an event-driven job scheduler which could receive multiple kinds of events. Those jobs would shell commands because we want them to be general purpose. Its main feature would be that the execution of the jobs is state-aware, so it would run state machines that define the scheduling. This would permit us to control situations like "Execute 'x' if we haven't executed it yet". Also we wanted a client that resent events detected in the machine in which it is running to a server machine that could react to them.
Those goals are pretty much achieved. Our software is centred in a daemon which computes state machines. An state machine is a set of states connected by transitions. We go from one state to another by a transition when we receive a set of events specified in every transition. Also, when we change of state using a transition, this one may execute an action. In our case, the events of the transitions are the events that run jobs, and the transition actions are the jobs. So we transformed the job scheduling rule-making to an state machine definition. That makes the project an state-aware and event-driven job scheduler, so this point is solved too.

The version we release for this final project can receive several kinds of events, thanks to the command-line control program that can send events to the daemon. It permits to receive events from existing job-schedulers of different kinds such as *cron*, *syslog* or *udev*. But also we have two sources of events more that makes our project *potentially* able to receive lots of different kinds of events. This is the event-sending library and the plugin workers, which need some minimal development.
Our state machines can execute several kinds of actions like shell commands. Another kind of actions that we have is the 'propagate' action. Those actions, as we have seen, propagates all the events that triggered itself, so it achieves the client/server model goal.

6.2 Future work

There are features that we expected to have for this final project that finally we had no time to implement. Some of those does not have a design because it was intended to be done on a future development iteration. However, those features had low priority because they are not very related to the main goals of the project.
Other features were thought later, when the design and the planning were already defined so we had no time left to make them real.
We also have work to be done related to design improvements and bug-fixing.
This future work is divided into two categories, critical and new features. In critical will be bug-fixing and design changes. New features are just that, features that will make the software better and more useful.

6.2.1 Critical

Log `CMD` actions output

We have to design and implement a way to monitor the `CMD` actions and get the strings it tries to print in the standard output, in order to log them. For example, if we execute a command, it fails and writes the error information in the standard output, the user would want to know what this message said.
To do that we probably will need to make a thread that reads a pipe to the command programs.

Non-connex cycle component

We explained in *section 3.1.6* that when we remove a transition that is followed by a cycle, the cycle won't be removed and won't be connex to the beginning of the state machine. This is a problem to solve by using a BFS algorithm to check the connectivity of the state machine every time we remove a transition and it doesn't remove the destination state.

User credential control

In *section 3.1.1* we describe the design of the user credential control system of *reactor*. To summarize, we have three levels of users and the lower lever users can not reach the resources of the higher level users. This resources are mainly the state machines and the way to reach them are through control commands and events.
The implementation of this system has to be done yet, and the design of the remote events credential control system has to be detailed a lot more.

Graphical state machine visualization

The idea is to be able to generate a representation of the state machines we have loaded in a known format. This way it can be passed to a graph rendering program in order to visualize it. For example there is the DOT[28] language, which is pretty popular and is used by the also popular *Graphviz*[5] project.
This would let the user get a lot of useful information about the state machines running on *reactord*, like the current states or the identifier numbers of the transitions.

Current state storing

It can be seen in the planning (*5.1*), but it doesn't appears neither in the specification nor in the design of the project, because of the lack of time. What we want is to allow the user select states to make them able to be saved. This means that when the daemon stops it will save that it was stopped in this state. When the daemon is started again it will put as current states the ones saved, instead of the initial states.
One way of mark these states would be using the key character `*`. i.e.:

```
STATE_A STATE_B* e1 & e2 NONE
```

If the state machine that contains that transition is in `STATE_B` when *reactord* is stopped, then this would be saved, and the next time *reactord* starts the state machine will began

in `STATE_B` instead of its initial state. However if this happens in `STATE_A` nothing will be saved and in a posible *reactord* restart the state machine would begin normally at the initial state.

Performance analysis

This is probably the most important work to be done. It was planned to be performed by the last deadline of the project, but the final lack of time did not allow us to do it.
This is not a new feature or a bug-fix by itself, but it will help us to detect the bottlenecks of *reactor* and find solutions. It is critical because our project is based on a daemon running with root permissions that could be performing a lot of actions and have lots of huge state machines. It is important to push *reactor* to the limit and measure its behaviour.
By the theory we can sense that removing a transition from a huge state machine would be very costly, as it would be to put a new initial state. Also, going forward from a state with a lot of leaving transitions would be slow.
But that's all we have by now, theory and speculation and we need to solve this in order to improve our project.

6.2.2 New features

Future useful features for our project.

Empty set of events

As we will see later in *6.2.2*, there are situations in which we would like to have an empty set of events in order to run an action and go to the next state. For this we will define the symbol `-`, which will mean empty event set. The transitions with this symbol as a set of events will be executed when we arrive to its leaving state.

State machine intercommunication

By now we do not plan to make *reactord* able to join different state machines with a transition, because as we said, it results a very complex software design and it doesn't provide really important features.
But there is another kind of state machine intercommunication much more useful and easy to develop. We are talking about making a transition able to ask if we are in a concrete state that could be from another state machine.

Right now, when we are executing the action of a transition, we do it because we received some events when we were in a concrete state in its state machine. The idea is to extend this to check states of other state machines when we receive an event in order to go forward. *Figure 6.1* is an example of this. In the last state machines only transition, where the

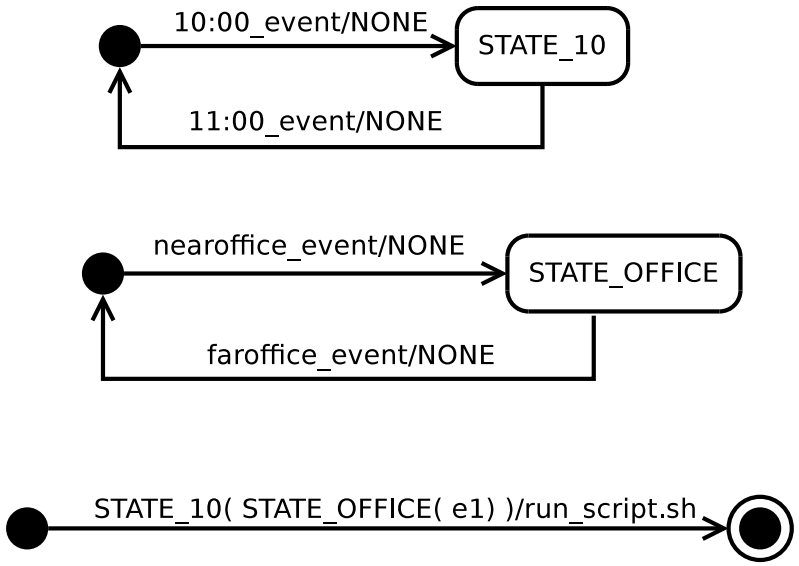


Figure 6.1: A state machine making use of the intercommunication feature

events are supposed to be, we have "`STATE_10( STATE_OFFICE( e1 ) )`". `STATE_10` and `STATE_OFFICE` are states from the other state machines. The way to read this is "if we receive `e1` while we are in state `STATE_OFFICE` and while we are in state `STATE_10`, then run the action and go to the next state". That makes us able to have independent state machines doing nothing more than controlling states in the daemon, useful for other more functional state machines.
We could have the events field look like "`STATE_10( STATE_OFFICE( - ) )`", so we don't wait for any event, but to the state machines to be in those states at the same time.
We could make it also able to ask for states in remote machines.

Extend plugins interface

The easiest way to add new features to *reactor* is to extend the plugins interface. To be precise, we have two extensions in mind:

- **Black-box states**
  The ability to ask to the plugins if we are in a specific state, like in *section 6.2.2*.

The difference is that the user or *reactord* knows nothing about the supposed state machine in which the asked state is. We can ask if we are in the "between 10:00AM and 11:00AM" state, but we don't know how this is controlled. Probably the plugin will just ask to the system clock, which makes much more sense than having to control it all the time with our state machine.

- **Parser accessible to *reactord***
  Having the *reactord* rules in one file and then the plugins rules in several files can become a mess. One way to solve that is to use as event identifier in the *reactord* rules the plugin rule that triggers the event. i.e:

  ```
  STATE_A STATE_B @cron:``0 1 * * *'' CMD script.sh
  ```

  The only event in this rule is `@cron:``0 1 * * *''`. `@cron:` identifies the plugin and what is between `` `` `` and `''` is a rule of the the 'cron' plugin.
  Internally those rules would be loaded by the plugin when *reactord* starts and would make a hash-like identifier for it.

User monitor plugin

This feature would not be a *reactor* internal, but a useful plugin for the system administrator. As it is explained on this documentation, the lowest privileged *reactor* users have to load their state machines manually, so they need to login in the system in order to make use of *reactor*. The problem is that when they logout their state machines will be running and maybe this is something the system administrator doesn't want. To solve this we can make a plugin to notice users login and logout events. This way the system administrator can make a state machine waiting for user monitoring events to remove the state machines of other users once they logout.

`CMD` action finish events

A `CMD` action finishing and its return value is a source of events very useful for our system. It is so because we can make an state machine wait in a state until an action finishes with it. And also we can continue with a different flow depending on if the command returned 'success' or 'fail. In *figure 6.2* we have a simple example of it. With the first transition it upgrades de system. It waits until the process is finished and, if everything went fine, then reboots the system. If not it notifies the user.

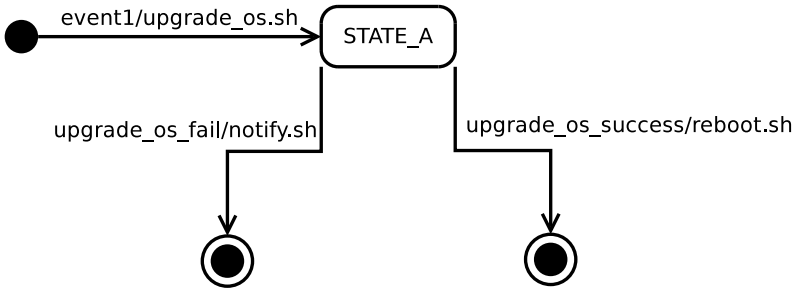


Figure 6.2: A state machine making use of the `CMD` action finish events.

Chapter 7

Bibliography

[1] Check. http://check.sourceforge.net/. [Accessed 06-07-2012].

[2] cron - Wikipedia. http://en.wikipedia.org/wiki/Cron. [Accessed 05-1-2012].

[3] Free Software Foundation. http://fsf.org. [Accessed 05-14-2012].

[4] Git. http://git-scm.com/. [Accessed 06-07-2012].

[5] Graphviz - Graph Visualization Software. http://www.graphviz.org/. [Accessed 05-17-2012].

[6] Job scheduler - Wikipedia. http://en.wikipedia.org/wiki/Job_scheduler. [Accessed 05-1-2012].

[7] KDevelop. http://kdevelop.org/. [Accessed 06-07-2012].

[8] libevent. http://libevent.org/. [Accessed 06-07-2012].

[9] Redwood Software Products - Wikipedia. http://en.wikipedia.org/wiki/Redwood_Software#Products. [Accessed 05-16-2012].

[10] SOS JobScheduler. http://www.sos-berlin.com/scheduler. [Accessed 05-16-2012].

[11] *syslog(3) - Linux man page.* http://linux.die.net/man/3/syslog [Accessed 05-18-2012].

[12] systemd - Wikipedia. http://en.wikipedia.org/wiki/Systemd. [Accessed 05-16-2012].

[13] udev - Wikipedia. http://en.wikipedia.org/wiki/Udev. [Accessed 05-16-2012].

[14] Valgrind. http://valgrind.org/. [Accessed 06-07-2012].

[15] Vinzant Software - Global ECS. http://www.vinzantsoftware.com/global_ecs.php. [Accessed 05-16-2012].

[16] Dual-Licensing Linux Kernel with GPL V2 and GPL V3. https://lkml.org/lkml/2007/6/9/11, 2007. Thread from the Linux kernel mailing list where main kernel developers discuss the non-adoption of GPLv3 with other contributors. [Accessed 05-14-2012].

[17] BalaBit IT Security Ltd. *The syslog-ng 3.0 Administrator Guide*, twelfth edition, July 2005. http://www.balabit.com/dl/guides/syslog-ng-v3.0-guide-admin-en.pdf [Accessed 05-18-2012].

[18] GNOME. GLib. http://developer.gnome.org/glib/. [Accessed 06-07-2012].

[19] GNU Project. GNU Coding Standards - Source Language. http://www.gnu.org/prep/standards/standards.html#Source-Language. [Accessed 05-11-2012].

[20] Michael Kerrisk. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. Number 978-1-59327-220-3. No Starch Press, 1st edition, October 2010.

[21] PC Magazine. PC Magazine Encyclopedia - Wire protocol definition. http://www.pcmag.com/encyclopedia_term/0%2C2542%2Ct%3Dwire+protocol&i%3D54750%2C00.asp. [Accessed 05-11-2012].

[22] GNU Project. Autoconf. http://www.gnu.org/software/autoconf/. [Accessed 06-07-2012].

[23] GNU Project. Automake. http://www.gnu.org/software/automake/. [Accessed 06-07-2012].

[24] GNU Project. C Extensions - Using the GNU Compiler Collection (GCC). http://gcc.gnu.org/onlinedocs/gcc/C-Extensions.html. [Accessed 06-06-2012].

[25] GNU Project. GCC, the GNU Compiler Collection. http://gcc.gnu.org. [Accessed 06-06-2012].

[26] GNU Project. GLIBC, the GNU C Library. http://www.gnu.org/software/libc/. [Accessed 06-06-2012].

[27] GNU Project. Libtool. http://www.gnu.org/software/libtool/. [Accessed 06-07-2012].

[28] Graphviz Project. The Dot Language. http://www.graphviz.org/doc/info/lang.html. [Accessed 05-17-2012].

[29] Gigi Sayfan. Building Your Own Plugin Framework. http://www.drdobbs.com/cpp/204202899, November 2007. [Accessed 05-10-2012].

[30] Richard Stallman. Why Upgrade to GPL Version 3. http://gplv3.fsf.org/rms-why.html, May 2007. [Accessed 05-14-2012].

[31] The IEEE and The Open Group. <pwd.h> - The Open Group Base Specifications Issue 6. http://pubs.opengroup.org/onlinepubs/000095399/basedefs/pwd.h.html, 2004. [Accessed 05-11-2012].

[32] Álvaro Villalba Navarro. reactor - GitHub source code repository. https://github.com/alvarovn/reactor. [Accessed 06-06-2012].

[33] Álvaro Villalba Navarro. README. https://raw.github.com/alvarovn/reactor/master/README. README file of the *reactor* project. Contains an user manual and a plugin developer manual. [Accessed 06-07-2012].