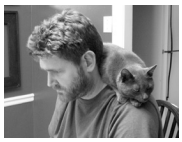
iVoyeur

Crom

DAVE JOSEPHSEN



Dave Josephsen is the author of *Building a Monitoring Infrastructure with Nagios* (Prentice Hall PTR, 2007) and is senior systems engineer at DBG, Inc., where he maintains a gaggle of geographically dispersed server farms. He won LISA '04's Best Paper award for his co-authored work on spam mitigation, and he donates his spare time to the SourceMage GNU Linux Project.
dave-usenix@skeptech.org

In 2006 I joined a tiny little company that was in the process of moving from the California Bay Area to Texas. Their business was about half database outsourcing and half niche Web site hosting. Their production infrastructure was primarily Linux, but they'd had all sorts of sysadmins, so there were a few pieces of SCO here, some HPUX there, etc.

The various sysadmins had also left their mark on the haphazard bowl of spaghetti that was their back-end processing automation. This unruly mob of code that extracted and imported, encrypted and decrypted, compressed and uncompressed and sent hither and yon the data that was the lifeblood of the company was written in all manner of languages, and never did the same thing the same way twice. When I joined the company no one had a clear idea of what it was all doing, much less how it managed to do it.

There were several hundred scripts in all, written in TCL, Perl, C, shell, and Java. The DBAs knew where to drop things off and where to pick things up, and beyond that nobody wanted to touch any of it. But now that the company was moving, it all needed to get untangled, and the untangling had fallen to me.

It's rarely much fun to inherit another sysadmin's (or, in this case, gaggle of sysadmins') mess, but I was actually kind of fascinated by the problem. It was pretty obvious that all of this stuff was doing the same subset of tasks over and over again. Extract the file, encrypt the file, send the file. Repeat. My plan was to write a library that encompassed all of those tasks, as well as enforce some standardization, and then re-write all of the existing scripts using that library. None of this was exactly rocket science, and transparency was important, so I took the LCD approach and wrote the library in shell.

It was a commendable effort. The library enforced a common runtime directory structure, so that everything was in a predictable place. It included its own logging functions to ensure that all of the logging and error-handling was centralized and in a common format. It even trapped signals and responded accordingly, such that if anyone were to, for example, hit Ctrl-C in the middle of a script execution, the library would gracefully exit. All of the scripts would use cron for scheduling. It solved a lot of the problems that the original bowl of spaghetti presented, and even changed the way I write shell scripts to this day (for the better), but ultimately, I think, the effort was a failure.

There are several reasons I think I missed the mark on this problem, but really they could be summed up by saying that I hadn't given the company a solution. I'd

rewritten their automation in the manner I thought it ought to have been done in the first place, but for everyone other than myself, these scripts are still a black box of mystery. Were I to leave the company tomorrow, the admin to replace me would be more likely to write the next script in his or her language of choice (Ruby or Lua, or whatever you kids are using this week) than to dig into my code to learn how my boring, probably obsolete shell library worked. I hadn't added to the mess, but neither had I provided a means to ensure it didn't reoccur. And really that's why the problem existed in the first place.

Also, there were aspects of bad engineering about it. Yes, there was a library of reusable code there, and all those common tasks were represented as functions within it, but I still needed to port the old scripts to new scripts, and those new scripts all still did the same subset of things again and again. So there remained an abhorrent amount of silly code redundancy—100 scripts to call different combinations of the same 15 functions on 100 different files. Had I written a few proof-of-concept scripts instead of being so focused on finishing the magical library of wonder, I would have noticed it earlier. Once the lib was done, I'd ported about two TCL scripts to it before realizing my mistake, but by then I was committed to the design and nearly out of time. I paid for it in the mind-numbing 72-hour port-fest that ensued, feeling stupider and stupider with each newly ported shell script.

Needless to say, the seed of a mental image of the correct answer formed in my mind that night, but as these things go, it was a couple of years before I was able to revisit the problem. That seed had plenty of time to germinate, and I was determined to get it right this time. The library wasn't a bad idea at all, I just wasn't thinking big enough. The correct answer to this problem was, I think, just a single layer of abstraction up from where I'd started. I had written a library to enforce a common way to do things, but I needed a framework, and a set of common interfaces for people to use that library (in a way that didn't force them to write their own shell scripts). I call that framework "Crom." And while Crom is dry fodder for conversation, and only peripherally related to systems monitoring, it's also about all I've worked on for the last several months, so I'm afraid we're stuck with it, dear reader. My apologies.

Crom has a few operational assumptions about your job. First, it assumes that your job can be broken down into tasks. Next, it assumes that you want to schedule those tasks to run on a recurring schedule of some sort. Crom uses the UNIX `at` command to perform the actual job scheduling, and it is written in 100% shell, so it requires only /bin/sh, at, and the usual slew of shell commands like `date`, `cut`, `grep`, and `sed`.

Although the similarities are unintentional, Crom's architecture is quite similar to Nagios [1]. It's a task-specific scheduling and notification engine, but instead of scheduling little monitoring plugins to collect metrics or check availability, Crom schedules individual tasks that make up a larger Job. These tasks deal with some little piece of automation, like loading data into an Oracle database or sending a file via FTP to a remote host. Figure 1 shows a typical Crom job definition.

```
meta{
JOBID=4019
JOBNAME=exampleJob
DESCRIPTION="An example job for the wonderful readers of ;login magazine"
NOTIFYONERRORS='cromerrors@domain.com'
}
```

```
task0{
DESCRIPTION="extract the file from DB1"
TASKTYPE='extract'
SCHEDULE='1 0 * * 2'
SOURCE="$(cat ${CTL}/${JOBID}/db1schema)@DB1:"
ORA_PROC=$CTL/$JOBID/file_extract.proc
ORA_ERROR='halt'
}

task1{
DESCRIPTION="scp the file from coke"
TASKTYPE='pull'
SCHEDULE='runafter:0'
PROTO='sftp'
SOURCE='oracle@DB1.domain.com:/data01/outgoing/Post*'
DESTINATION=%NEXT%
SKEY="${KEYS}/oracle_DB1_dsa"
ARCHIVESOURCE='1'
}

task2{
DESCRIPTION="add a date to the filename"
TASKTYPE='custom'
SCHEDULE='runafter:1'
DESTINATION='%NEXT%'
SOURCE='%THIS%/Post*'
INCLUDE="${CUSTOM}/${JOBID}/rename.sh"
}

task3{
DESCRIPTION="sftp the file to xyz bank"
TASKTYPE='push'
SCHEDULE='runafter:2'
PROTO='sftp'
DESTINATION='dbguser@1.2.3.4:in'
SOURCE='%THIS%/Post*'
DKEY="${KEYS}/${JOBID}/xyz_dsa"
}
```

Figure 1: Crom job definition

Except for the surrounding brackets, each attribute is shell syntax. In fact, when Crom reads in some piece of the job definition, it does so by extracting the section it's interested in between the brackets via sed, and then sourcing it. The attributes are then available as shell variables. Since we source in the attributes, we can use nested execution blocks to hide sensitive info such as passwords. In the example, the SOURCE attribute in task 0 is reading in its Oracle schema and password from an external file using `$()`. Crom inherits its directory structure from my original back-end processing library and provides environment variable shortcuts to useful directories at runtime. These may be used by any script that sources the library. For example, task 0 in Figure 1 is using the ${CTL} shortcut provided by Crom to locate an Oracle procedure file.

Each job is identified by a unique job number, called the JOBID, and each task is numbered sequentially. All tasks are required to have a schedule. There are three valid types: (1) Crom can parse schedules in standard cron syntax using its own parser (also written in shell); (2) a task may be scheduled to be run subsequent to the successful completion of another task with the "runafter" keyword (any number of tasks may "runafter" the same parent task in parallel); and (3) Crom supports the "never" keyword as a valid schedule, for tasks that are never intended to be run automatically (such as break-fix or debug tasks).

The Crom library supports macros in its definition files for those variables that aren't necessarily known at runtime. For example, task 1 in Figure 1 is making use of the "%NEXT%" macro, which will resolve to "cromhome/run/today/files/4019/2" (since 2 is the number of the next task). Since that specific directory is actually known at runtime, we could have just specified that instead of using the Macro %NEXT%, but the macro is preferable in that if the task is ever renumbered, %NEXT% will continue to work without modification. %NOW% is another macro supported by the library, which will resolve to the current time in seconds since epoch format. Users may specify their own macros by placing their values in a file named for the macro in Crom's "macro" subdirectory. It's common practice for tasks to share data with each other by setting up macros in the jobs macro directory.

Each task is required to have a task type, which is similar to a plugin in Nagios. Crom uses the TASKTYPE variable to locate the actual shell script to execute. Since task 0 in Figure 1 is specified as an "extract" task, Crom will check cromhome/bin for an executable named "extract". If it finds "cromhome/bin/extract", it will schedule an "at" job at the next occurrence specified by the tasks schedule passing the JOBID as argument 1 and TASKID as argument 2. Expanding Crom is as easy as writing a shell script and placing it in cromhome/bin.

Strictly speaking, the executable is not required to be a shell script: it can be any type of executable, and Crom will gladly schedule it for you. However, Crom provides a litany of useful shell functions for tasks that are shell scripts, such as functions for sourcing in the task definition from the job file, and job control and logging functions. A task that sources the Crom libs can, for example, call the "'halt" function, which halts the current execution of the task, generates an error to the logs, emails the recipient list specified by the NOTIFYONERROR variable, and prevents any other tasks within the job from being executed or rescheduled. In fact, tasks that source the Crom libs may make eight function calls relating to job handling and logging alone: debug, notify, info, stop, warning, error, nonfatal_error, and halt.

If you have a Java program and want to run it as a task under Crom, the wiser thing to do is to use the "custom" task, which takes the name of a shell script as an attribute called INCLUDE. The custom task will source in the script specified by INCLUDE and will check for a function therein called "runCustom", which it will run. This way, we don't need to port our Java code, but we retain full functionality with the job control system at the cost of only a few lines of shell. The custom function can also be used to build tasks that are not easily encompassed by a more generic task type. Renaming a file and setting up a custom macro for a subsequent task to use are good examples.

Crom was written with the expectation that large parts of it would be ripped out and replaced wholesale. For example, it currently reads in its job definitions from

files in the cromhome/jobs directory, but the plan has always been to replace the jobs directory with an Oracle database table. The library is, therefore, modular and extensible, and just nice to work with to an extent I find difficult to articulate. Perhaps the best evidence of this is the fact that I find myself using it to write things, such as supporting tools, that I normally wouldn't bother with. Figure 2, for example, is the output of the "cq" tool, which uses library function calls to summarize the scheduling queue. I usually lose interest and move on long before I'd consider writing something like this.

```
JOBID,TASKID    ATID      SCHEDULE
-----------------------------------------------
4007,0          2420      Mon May 30 01:00:00 2011 a crom
4007,2          2422      Mon May 30 01:01:00 2011 a crom
4007,4          2425      Mon May 30 01:04:00 2011 a crom
4007,6          2426      Mon May 30 01:10:00 2011 a crom
4007,8          2427      Mon May 30 01:12:00 2011 a crom
4008,0          2309      Tue May 31 08:00:00 2011 a crom
4008,3          2334      Wed Jun 1 08:00:00 2011 a crom
4010,0          2415      Mon May 30 00:15:00 2011 a crom
```

Figure 2: Output from cq, the Crom queue command

Most core functions, and every default behavior in the lib, are overridable by defining a custom function or setting an environment variable. Several of the supporting tools I've written, in fact, make use of override variables. The runtask script, for example, is what I use for manual intervention when something goes wrong. This script takes a JOBID and TASKID as its arguments and uses them to force-run that task immediately, overriding states like halt, which would normally make the task refuse any attempt at execution. Even within the default tasks we've written, the error handling behavior is usually overridable. Task 0 in Figure 1, for example, specifies an ORA_ERROR variable, which is there to override the default behavior of the extract task when it encounters an error running sqlplus (changing it from its default value of "error" to "halt").

Crom can log to a flat-file log (which rotates daily), syslog (to a user-configurable facility and priority), a FIFO (which we use to push log lines into a database with a separate script), or any combination thereof. The log lines contain all of the information you'd expect, plus a few fields I find especially useful at times, such as the "run number" field, which uniquely identifies each iteration of a task that runs, for example, every other minute all day long. Crom has built-in functions for sending email notification and automatically notifies recipients when a task calls warning, error, nonfatal-error, and halt.

Well, thanks for letting me gush over my shell script. This is about the only tool I've written where I have no doubt I'm reinventing the wheel and it's working so wonderfully I just don't care. It's also the kind of tool that's esoteric enough that I'm not sure if I'm scratching an itch that nobody else has (but again, it's still working so wonderfully that I don't care). If so, I've probably just bored you to death. Sorry about that. Stay tuned for something more monitoring-related next time.

Take it easy.