SPONSORED BY

GEEK GUIDE

Beyond Cron, Part II:

Deploying a Modern Scheduling Alternative

LINUX™ JOURNAL

Table of Contents

MIKE DIEHL has been using Linux since the days when Slackware came on 14 5.25" floppy disks and installed kernel version 0.83. He has built and managed several servers. Mike has written numerous articles for *Linux Journal* on a broad range of subjects, and he has a Bachelor's degree in Mathematics with a minor in Computer Science. He lives in Blythewood, South Carolina, with his wife and four sons.

GEEK GUIDES:
Mission-critical information for the most technical people on the planet.

About the Sponsor

Skybot, a Division of HelpSystems

HelpSystems has more than 30 years of experience in providing enterprise scheduling and automation solutions. Part of the HelpSystems family of brands, Skybot provides an affordable solution for cross-platform enterprise job scheduling, allowing businesses to integrate workflows across servers and critical business applications and monitor them from a central interface. Skybot Scheduler incorporates your disparate job schedules to help you build a unified enterprise schedule based on cross-server dependencies. For cron users, in particular, Skybot Scheduler allows you to import existing UNIX crontab data and use the cron expression to schedule new jobs using familiar cron syntax, helping to connect your UNIX cron job scheduling to enterprise operations. Skybot Scheduler also includes reporting, auditing and security capabilities to ensure that your enterprise job schedule is well documented and reliable.

For more information on Skybot Scheduler, see http://www.helpsystems.com/skybot.

Beyond Cron, Part II:

Deploying a Modern Scheduling Alternative

MIKE DIEHL

One of the best things about the UNIX environment (aside from being stable and efficient) is the vast array of software tools available to help you do your job. Traditionally, a UNIX tool does only one thing, but does that one thing very well. For example, grep is very easy to use and can search vast amounts of data quickly. The find tool can find a particular file or files based on all kinds of criteria. It's pretty easy to string these tools together to build even more powerful

tools, such as a tool that finds all of the .log files in the /home directory and searches each one for a particular entry. This erector-set mentality allows UNIX system administrators to seem to always have the right tool for the job. Cron traditionally has been considered such a tool for job scheduling, but is it enough?

This ebook considers that very question. The first part builds on my previous Geek Guide, *Beyond Cron* (http://geekguide.linuxjournal.com/content/beyond-cron-how-know-when-youve-outgrown-cron-scheduling-and-what-do-next), and briefly describes how to know when it might be time to consider upgrading your job scheduling infrastructure. The second part presents a planning and implementation framework.

Part I, Why Should You Upgrade?

Times Have Changed—Cron Has Not: The cron job scheduling tool has been around since the early days of UNIX, but oddly enough, it has maintained backward-compatibility throughout its development. In fact, aside from a few optimizations and some scheduling prefixes, cron hasn't changed much, but the average operating environment certainly has.

An IT enterprise used to consist of an email server and a file server or two—maybe a handful of each. Today's IT enterprises often add web servers, authentication servers, database servers (sometimes with many different databases), calendaring servers, business process management servers, CMS servers, revision control servers, backup servers and so on. Those servers may be located at

So modern system administrators are faced with a two-fold problem. They have to manage a wider variety of servers as well as a larger number of servers that may be scattered all over the place.

the office or spread across the globe. Enterprises even have servers to monitor other servers.

This is where automation comes in. Chances are you've already built many of the software tools you need to keep things running smoothly and just need to make sure that they do, in fact, run when they are supposed to run. Almost all of you probably use cron for this purpose. Cron doesn't know how to do server backups, for example, but it does know how to run the tool that does know how to run the backups.

So modern system administrators are faced with a two-fold problem. They have to manage a wider variety of servers as well as a larger number of servers that may be scattered all over the place. And, they have to do all of this with a scheduling tool that hasn't changed noticeably in 30 years.

But hey, it works, right? Sure, cron does what you ask it to do. But you can ask cron to do only things that you know it *can* do. You don't ask cron to manage task dependencies, because you know it can't. You don't ask cron to manage tasks across servers, because you know that it runs only on a given server. So, you typically use a lot of other tools

in your toolbox to work around the shortcomings in your main, if not only, scheduling tool.

For example, you might write a script to check whether a file actually exists before you attempt to process it. You might ping a server before you attempt to back it up. There are lots of other "pre-task" tests that you might perform, but you get the idea.

The problem comes when one or more of those "pre-task" checks fails. Should your scheduled job simply fail? Should it try again? And if so, how do you arrange for that? Sure, you can write a script to keep checking until all of the "pre-task" tests return positive results, but most people don't. And those who do eventually realize they're working too hard. All of these checks simply add complexity to a process that should "just work".

The reality is that most sysadmins simply perform backups with a script like this:

```
#/bin/bash
/usr/bin/rsync
root@remotehost:/home /backups
```

There's not much error checking (actually none) going on, but most of the time, it just works. And when it doesn't, hopefully someone notices. It would be nice if there were a convenient way of not only knowing that a given job didn't run, but also *why* it didn't run.

Ease of Use: Nobody ever said cron was difficult to use, but let's face it, it's a point-and-click world now. Editing a raw ASCII file with an editor like vi or nano is simple, but

it's also an easy way to make a mistake that often is difficult to spot. It's really easy to press the wrong key on your keyboard. It's not hard to specify the schedule incorrectly. Finally, cron doesn't perform any validation on the actual script that you request be run. For example, is your script in /usr/bin/ or /usr/local/bin? Did you remember to check? The editor surely didn't check for you. These are all simple, "only human" mistakes. It would be nice if your tools could watch your back when you used them.

FIGURE 1. Skybot Enterprise Job Scheduler

Multi-Server: Life would be easy if you had only a handful of servers to manage, but life usually isn't that easy. From a technical point of view, it's not too difficult to replicate a crontab file to any given number of servers—it's just a bit of scripting. But then it gets worse—not every server needs the same crontab file. A web server requires different jobs to run from what a database server would need to run. Some departments in your organization may have jobs that need to run on their servers before jobs in other departments can run. However, all the servers in a given organization may share a core set of jobs that run across the entire enterprise. Managing something like this could prove to be painful.

In an ideal world, sysadmins would like to be able to categorize servers based on things like operating system, function, department or development stage. Cron won't let you do that easily.

Dependency Management: One area of job scheduling that cron just doesn't do well is job dependency management. Some jobs simply can't run until other jobs have completed.

A classic example is the database server that requires scheduled maintenance each night. It doesn't do any good to run a backup job on a database server that is still running its maintenance. In fact, doing so can be devastating. At best, both jobs conflict with each other and slow each other down, causing them to run longer than needed. At worst, one job corrupts the other; the backup job backs up a database that is only partially repaired, or the repair job attempts to repair a database that is locked by the backup job.

You might be tempted simply to combine both the repair

and backup functions into the same script. That approach isn't as flexible, however, and it simply results in large scripts that do a lot of different things and no one knows why.

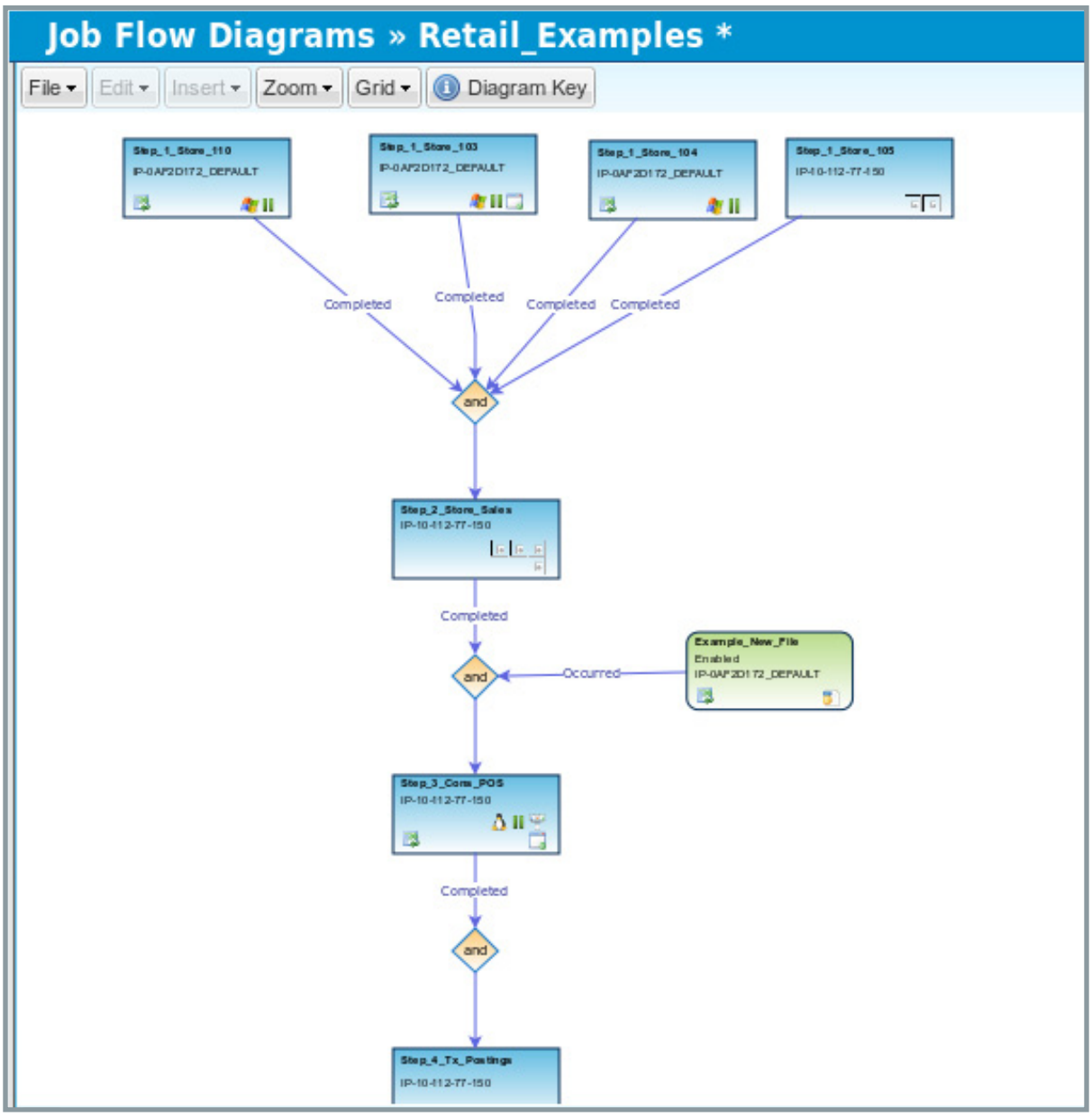

FIGURE 2. Stakeholders need to be able to visualize job dependencies so they can make more informed change requests.

All too often sysadmins will work around the issue of job dependencies by using clever, though perhaps sloppy, job scheduling. For instance, you could schedule the database maintenance right at 8:00pm. Then, knowing that the maintenance typically takes only an hour to run, you could consider running the backup job at 10:00pm, giving yourself an hour of leeway in case the maintenance job takes longer than normal (and you might want to have a sysadmin be on call while they run in case there's a problem).

Visualization: So over time, you may have managed to build up a complex hierarchy of jobs (and the servers on which they run) that automate much of the day-to-day operations of your enterprise. And everything probably works perfectly—until the day someone asks you how it all works.

Usually, this someone is another system administrator—maybe a new, or junior, system administrator—or a manager who wants to understand better how it all "fits together". Printing out the crontab files for all of your servers obviously isn't going to provide anyone with the big picture. You might be able to explain when the various jobs on a given server run, but that doesn't even begin to explain *why* they run when they do. Perhaps you have some job dependency issues, like I discussed earlier, and you've had to schedule jobs with that in mind. Essentially, is job A scheduled after job B because it's convenient, or is it because job B actually depends on job A completing first? Just looking at when a job is scheduled doesn't capture that distinction.

You also might find yourself in the situation where the web development team members want to understand how their stuff works, but they really don't care how the DBAs' stuff works. To put a perverse twist on things, the DBAs may ask to have their backup jobs scheduled earlier in the day, but they don't understand that their backups can't run until the accounting department completes its processing.

Nobody knows how it all works but you, and you don't want to spend the rest of your life in meetings trying to explain it all. A true enterprise job scheduler would make it easy to visualize the "big picture".

Change Management: Most system administrators are extremely busy. Scheduling, documenting and monitoring jobs often should be delegated to other staff, if possible. Perhaps the web developers want to be able to manage the jobs that run on their servers, while the DBAs certainly don't want web developers breaking things on their servers. Maybe the help-desk staff members want to be able to see what jobs are scheduled and determine if they ran correctly; maybe they don't need the ability to change job schedules. Managers always are curious, and rightly so, but no one wants them doing anything but looking at what's scheduled and where.

The other side of the equation is knowing who made what changes and when. Change logging and accountability is important. If something is found to be broken, it's often helpful to know exactly what was changed. Also, if changes are made incorrectly, this might reveal an opportunity for additional training for the

employee who made the change.

Management by Exception: Most system administrators are used to checking their email in the morning and seeing several messages triggered by the various jobs that ran the night before, and usually, perusing those messages is part of their morning work flow. At first, they actually may read every line of every email message. Over time, they get used to not finding anything untoward in those messages, so they begin to skim over them, looking for log entries that "stick out". Logs are important, but no one actually has time to read them all, unless they're important, but you can't tell if they're important unless you read them—all of them.

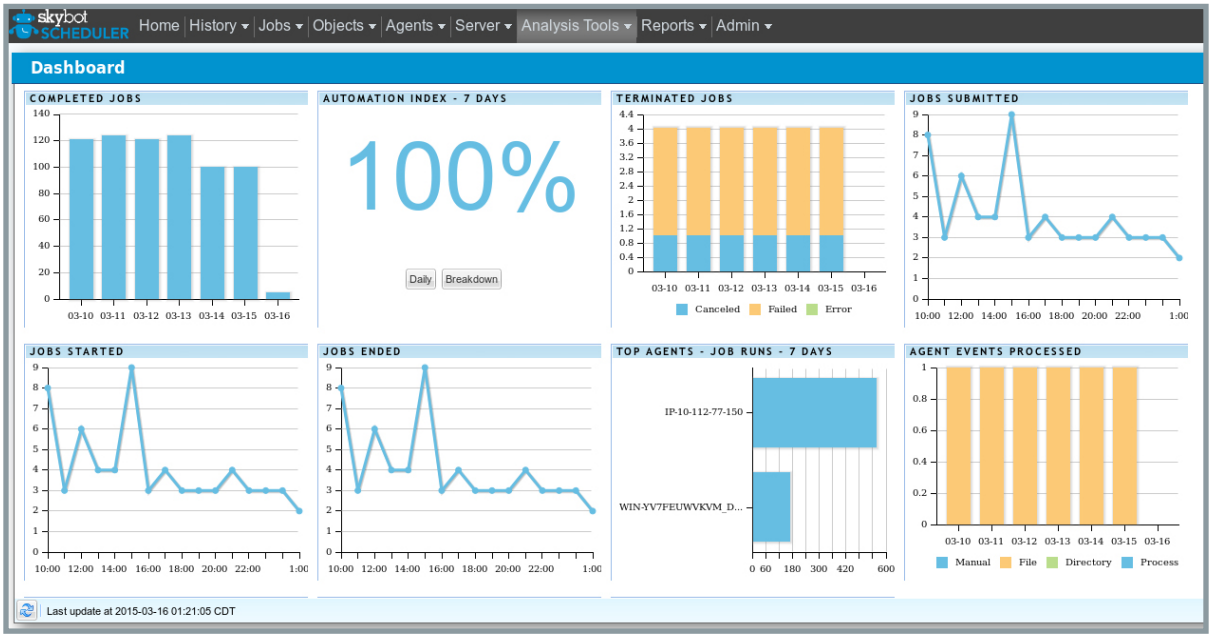

FIGURE 3. Instead of being inundated with job status reports, sysadmins should have a convenient dashboard that summarizes the current state of the enterprise.

As odd as it sounds, it would be so much easier if system administrators received only the bad news. This is where “management by exception” comes into play.

As odd as it sounds, it would be so much easier if system administrators received only the bad news. This is where “management by exception” comes into play. Sysadmins never need to react when things run as expected, so why even get the message? In this case, “no news is good news”. When there actually is a problem, it should be indicated unambiguously by a clear error indication. A job scheduling tool that is worthy of running an entire enterprise should be smart enough to tell administrators only when something goes wrong instead of drowning them in a flood of messages.

Flexibility: The cron scheduling system is pretty good at scheduling simple recurring jobs. It’s easy to schedule a job to run at midnight every night. Scheduling a job to run every 15 minutes, but only during the business week is something you wouldn’t spend more than ten minutes doing. Let’s face it, cron is good at what it does and is easy to use, but its scheduling capabilities are limited. For example, sometimes you actually want to run a job on the very last day of the month, instead of merely running it very early on the first day of the month.

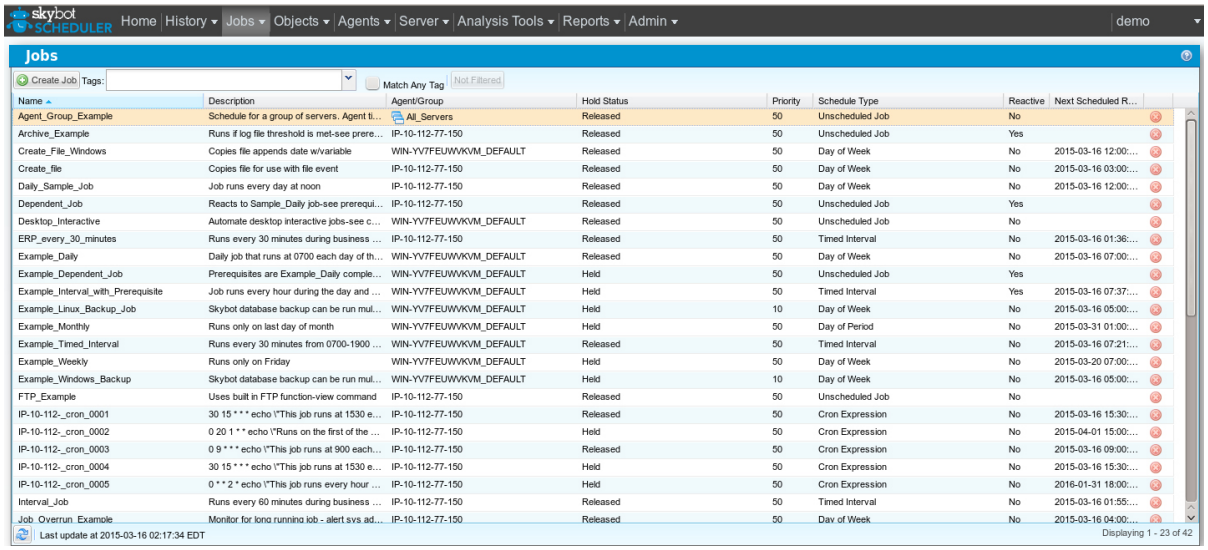

FIGURE 4. An enterprise-at-a-glance view, complete with job descriptions, helps to show how everything fits together.

Enterprise job scheduling isn't just about making sure jobs run at particular times. Sometimes you want a job to run based on an external trigger. For example, a network intrusion detection system might create a log file when it detects suspicious activity. Although many network intrusion detection systems can run scripts in response to various triggers, it might make sense to consolidate that type of job under the control of the scheduling system.

On the other hand, you might want a job to run when a particular host is no longer reachable via ICMP. Once again, your network management tools may be able to perform the same function, but an enterprise job scheduler may be able to do it in a more flexible fashion.

This is just a matter of being able to use the tool that does the job best, and it seems reasonable to expect that

a job scheduling tool can manage various jobs no matter what triggered them to run.

Now that I've discussed a few compelling reasons why you might consider migrating to an enterprise-class scheduling system, let's take a look at how to do it.

Part II, Implementation

Planning: If your organization is growing enough to need an enterprise scheduling system, it's also growing enough that such a migration isn't going to be trivial. You're going to need to do some planning in advance.

Having a detailed inventory of all of your servers is a great way to start. You'll want to enumerate carefully each job that runs on each server, and you can save some time by making a few blanket statements like "all servers need to be backed up". You also may discover you have servers that aren't running jobs they should be, such as log analysis and rotation.

You also could approach this task from the other direction and make a careful list of all of the processes that run and then identify the server, or servers, on which they run. The benefit of this approach is that various departments in your enterprise can make their own separate lists. The IT department would, of course, have to perform some quality assurance checks on each list. For example, the accounting department may neglect to put backups on its list because it might not consider backups to be "accounting".

No matter how you approach it, you still should end up at the same destination—with a solid list of everything your servers do. You even may find a few discrepancies. Consider that a bonus.

Now comes the tricky part. You need to take a close look at each of the tasks you've identified, and for each task, ask yourself these questions:

- On which servers should this task run? Is there a common category of servers that need to run this, such as web servers, database servers and file servers?
- Are there any prerequisites that need to be in place before this job can run? What are they?
- Have tests for any prerequisites been embedded in the job?
- Could these tests be factored out and treated like any other tool?
- Is there any way this job could fail?
- Are you checking to see that the job didn't fail?
- What do you want to do if it does fail? Try again? Send an email?
- Who needs to know if this job fails?
- Does anyone particularly need to know that it ran successfully?
- Does this job block any other jobs from running?

- Do any other jobs require this one to complete before they can start?

- Who should be able to manage this job? For instance, web developers shouldn't be managing database backups, but DBAs might want to be able to schedule database maintenance.

This is also where you might start thinking of optimizations you could make. For example, instead of looking for a new inventory transaction file every ten minutes, wouldn't it be nice to have a job start as soon as the transaction file is created? Try to think about how you would like to have a job run if you weren't constrained by the known limitations of cron.

At this point, you should have a complete picture of all of the processes in your enterprise. You should know how many servers you have and what should run on each of them. You probably also will realize that you never want to have to go through this exercise again. When you're done, you won't have to.

Budgeting: Now it's time to set (and get) a budget. First, you need to identify the software you intend to migrate to. For budgeting purposes, you'll need to know all the costs. Does the package require a central management console? Does it require an agent on each server, and if so, how much does each agent cost? Do you need to factor in an additional server?

Time is the other side of the equation. How much time is it going to take to install the software? Can you automate

the installation of any agents that may be required? Since, as I'll discuss shortly, you don't have to do it all at once, how many hours each week can you devote to this effort? How much of it can be delegated to lower-level staff members or other departments entirely?

Once you know what software you intend to use, how much it will cost and how long it will take to deploy, you can set some realistic goals for the project. The following are some suggestions (in no particular order):

- Improve error detection and recovery.
- Delegate select operations to lower-level staff members.
- Delegate responsibility (and cost?) to other departments.
- Improve change management.
- Reduce on-call staffing needs.
- Improve operations visibility.
- Improve just-in-time order and inventory processing.

Funding: When you approach your managers to get funding for this project, they may ask why they should spend money to fix something that isn't broken. After all, it's been running fine as is for some time. Be prepared to make the argument that the new system will reduce IT's level of effort in managing daily operations. These

If you have change management software, such as Puppet or CFEngine, this software deployment should be almost effortless.

reductions could result in reduced staffing requirements or increased time to devote to more pressing operational issues. You could suggest that other departments might be willing to contribute staffing or funding in order to make the project successful, and make sure to set goals that would benefit those other departments.

Installation: Next, it's time to install the hardware and software. You'll probably find that you need a central management server and that each server in your enterprise needs some sort of software agent installed on it. If you have change management software, such as Puppet or CFEngine, this software deployment should be almost effortless. If you don't, this is the perfect time to add such a tool to your software toolbox.

Rather than jumping in and starting your enterprise-wide deployment, perhaps a different approach is in order. The temptation is to go ahead and start deploying. After all, you've already done the planning, right?

By going a bit slowly though, you actually can save some effort; here's how. Deploy a few innocuous jobs to a few internal servers. Let the IT department become familiar with how the job scheduling system works. Finally, have senior staff train junior staff in the scheduling system's day-to-day

operations. Let the system mature under IT's watch before you move on. Most lower-level staff members likely are anxious to learn a new skill, especially if it allows them to carve a new niche for themselves.

And, it gets better. Eventually, your lower-level staff members can cross-train employees from other departments. Finally, other departments, such as web development, DBAs and accounting, ultimately can become responsible for managing their own jobs. If things run as expected, you'll never hear from them. If they don't, they'll let you know, and that's how you want things to operate. That's the final goal.

In the meantime, you have to make a decision. When you did your planning, you either took a server-centric approach or a process-centric approach. That approach allowed you to enumerate your servers and to understand fully which jobs ran where.

Just as there were two different approaches to planning, there are two different approaches to deployment: process-centric and department-centric. In process-centric deployment, you migrate a particular process at once, no matter which servers it runs on or which departments it affects. For example, you might migrate the backups for all of your servers first. Then, you would choose another process to deploy next, trying to do the low-risk processes first.

On the other hand, you might consider a department-centric approach if you discover resistance to the project by one or more departments. Perhaps the accounting department is more risk-averse than you had anticipated. If that's the case, try to identify a more sympathetic

department to migrate first and use them as an example to show the other departments that you can perform the migration smoothly and that there are benefits to be had by getting on board.

Importing: Importing your existing crontabs should be a fairly automated process. With Skybot Scheduler from HelpSystems, for example, you simply run an import command, and then you have the opportunity to edit each imported job. At this point, you should be answering some of the questions I listed earlier. For example, is there logic embedded in the script that is simply meant to check for prerequisites? If so, perhaps factoring them out and using the job scheduler's dependency management functionality makes more sense. That way, all of those dependencies will be well documented and error messages can be provided when those assertions fail. Given the ease with which job

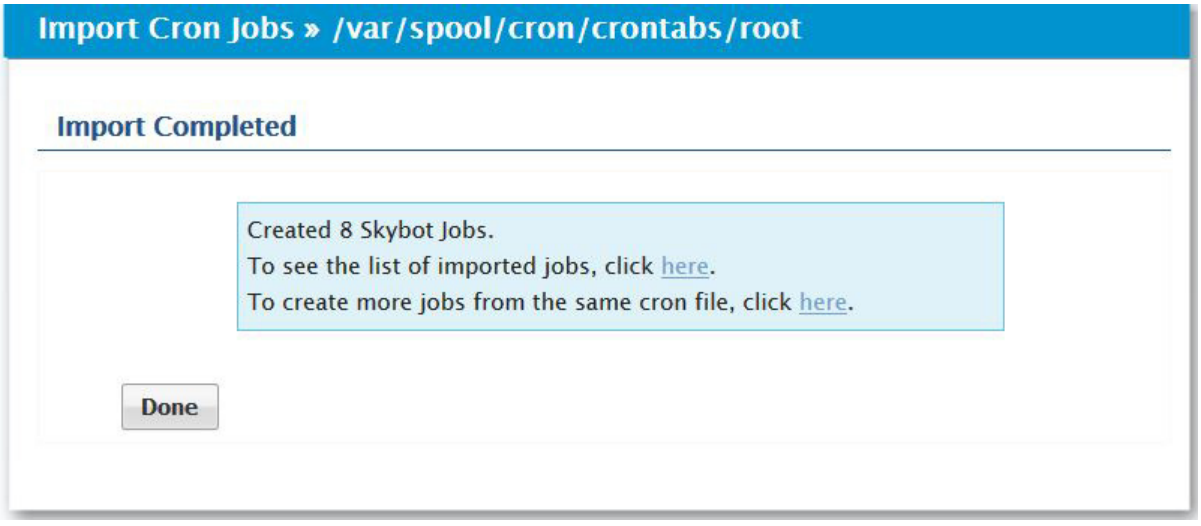

FIGURE 5. Importing existing crontabs should be simple and error-free.

dependencies can be created, perhaps you will find that other dependencies make sense but never were checked because it was too much effort. This is also an opportune time to eliminate the "sloppy scheduling" I discussed

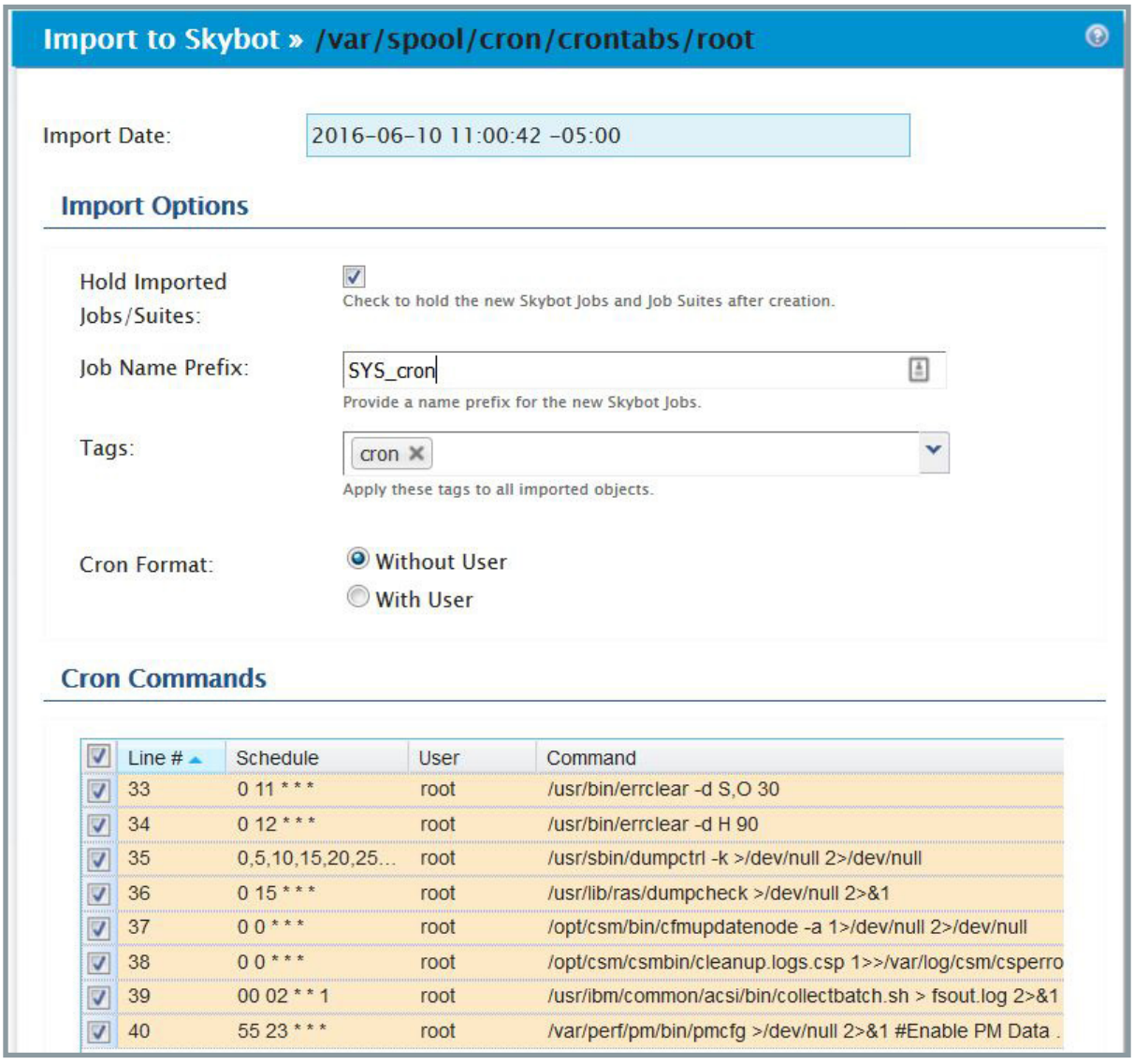

FIGURE 6. Imported jobs can be held pending further modification and validation. You don't have to activate the whole crontab at once.

earlier. Simply create job dependencies where one job can run only after one or more other jobs complete.

As you finish migrating each process or department to the new scheduler, try to delegate the task of monitoring that job (and the jobs to come) to a less senior member of your staff or to the other department's employees. Finally, decide who needs to be notified when a job fails. The point of all this, of course, is to free up time so that senior staff members can take care of more pressing matters.

Conclusion

No one makes sweeping operational changes like the ones I've discussed here without a good reason. In all likelihood, your scheduling infrastructure isn't broken, but that doesn't mean it can't be improved. These days, system administrators' lives are pretty hectic; they're always busy and usually under-funded. In this ebook, I've demonstrated that a true enterprise-class job scheduling system can lead to a more efficiently run operation. I've also described how an enterprise job scheduler can reduce staffing costs by reducing on-call time, delegating operations to lower-level staff and delegating day-to-day operations directly to various stakeholders. I hope I've given you a thought-provoking framework from which you can build your own implementation plan.■