Deployme: Tellme's Package Management and Deployment System

Kyle Oppenheim & Patrick McCormick – Tellme Networks

ABSTRACT

Many administrators use a central software repository because managing distributed packages is difficult. Deployme is our solution to manage the package update lifecycle across a large number of independently configured hosts.

Deployme is highly flexible and has been extended to handle many different types of packages. Deployme packages include standard UNIX tools, local applications, web site content, and voice site content. Most packages require fast, frequent deployment.

Deployme has a web-based user interface that allows less technical users to deploy on their own. Deployme also restarts appropriate server processes, a feature which was much more difficult than we expected.

We discuss other lessons learned from the first implementation and planned improvements for the future.

Introduction

Using small, independent software packages is a common approach to managing software. Many tools have been developed to address package management including Depot [Manheimer1990] and LUDE [Dagenais1993].

The package philosophy behind these tools is easily extended to rapidly changing content. However, we found these systems to be incomplete because they only cover one or two stages of the package update lifecycle.

Similar to the software distribution cycle outlined in [Furlani1996], we have divided the package update lifecycle into four steps:

1. Create the package.
2. Distribute the package to designated hosts.
3. Install and activate the package on each remote host.
4. Delete old, unused packages from remote hosts.

Deployme automates package creation and eliminates the need for a release engineer to manually package data. (As you may have guessed, "Deployme" is one of many tools derived from Tellme's corporate moniker.) Deployme exports source data from CVS [Cederqvist1993] and then builds executables and other generated files.

The system also handles installation, activation, and removal of packages, completing the package lifecycle. Deployme automates most of the traditional release engineer role.

The original design was intended to manage and deploy web site and voice site content. We have since found the system to be general enough to be used for local applications, common tool management, and other types of content.

With Deployme, our producers and developers manage their own releases. A straightforward web user interface allows the development team to update the Tellme service without the intervention of our network operations team.

Motivation

Our web-based content management system before Deployme was difficult to maintain, but was easier to use than we expected. This success led us to make a simple user interface a major Deployme goal.

As applications passed quality assurance, content developers checked their changes into a CVS repository. Once all applications had reached a known good state, the tree could be released.

To perform a release, a developer would go to a web page with a single button on it. The button started a shell script that checked out the CVS content tree and copied the data to the production server.

Platform releases were similar. Again, we built a web page with a single button that kicked off a shell script. The script performed a CVS pull, ran the build, and copied the release to the production server. As more servers arrived, we added them to the list that received completed builds.

The benefit of the release scripts was that they took much of the manual work out of releases and freed up developers to do more constructive tasks. The scripts also significantly reduced developer stress, as platform releases became an everyday occurrence instead of a frightening event. Finally, they allowed any developer to be appointed release engineer, because the only process involved was filling out the form on the web page.

The script proved increasingly difficult to modify as we increased the number of build targets and machines. It also could not support features such as rollback. If we wanted to push the same build to a new machine using the script, we had to wait while the script recompiled the platform from scratch. It became obvious that our inefficient distribution script could not scale.

At this point, we seriously considered having our production servers attach to a storage area network (SAN) instead of distributing software packages to local storage. We decided against this for several reasons.

Distributing packages to local disks still provides high availability, but it is cheaper and more flexible. A centralized file repository does not span across cities and continents without significant cost in time, hardware, and complexity. Also, most centralized solutions would lock us to one vendor. Our failures are restricted to individual machines, not an entire storage array. Even with high-availability components, SANs still act as a single point of failure.

So, to replace the unscalable release script, we wrote a new script to perform releases. This script was the first implementation of our distributed package architecture. However, it only performed distribution and activation of packages. It had no web interface, requiring a network operations engineer to serve as gatekeeper for all releases of content, platform, tools, and configuration files.

Updates to the production system became very infrequent. Developers plaintively asked, "Why can't we bring back the button?"

Goals

Deployme's mission is to provide a central system for tracking the entire lifecycle of software packages. We established the following goals to judge the success of the project:

- *Support a wide audience*. Users who are less inclined to system administration tasks should be able to create and deploy packages. A human gatekeeper will impede updates.
- *Robustness*. With a wide audience using the system, the system must never activate a package without verifying that it was distributed successfully. Detailed audit logs and notification must be included to inform our Network Operations team of system changes.
- *Augment the development process*. Packaging should not be a complex task, but instead it should be seamlessly integrated into existing processes.
- *Flexible destinations*. Packages should be able to be deployed to various destinations such as development systems, testing systems, and production systems.
- *Efficient use of network bandwidth*. When large groups of servers are located at remote data centers, Deployme should only push bits over the slow link once and distribute where there is good bandwidth.
- *Quick pushes*. The system should allow, but not enforce, staging of a package for testing. If a "hot fix" (e.g., a critical bug fix or a press release for the web site) must be pushed immediately, it should be possible to bypass staging servers.
- *Seamless activation*. End users (customers) should never be aware of a new package being pushed (e.g., Web site links should never be broken). If a server binary is being pushed, the server process should be gracefully restarted.
- *Rollback*. It should be possible to return to a known good state quickly and robustly.
- *Scalability*. Deployme needs to be able to handle hundreds of modules, hundreds of servers, and thousands of packages.

Non-Goals

We intentionally chose not to address several issues in the first implementation.

- *No local package management*. Enough robust single-server package managers have been developed in the past that we should not invent a new one.
- *No dependencies*. Our aggressive schedule would not leave us enough time to develop dependency tracking properly. We felt we could achieve all of the goals above without dependency tracking.
- *No fine-grained operations control*. While we chose to integrate certain operations functionality into Deployme, we decided against turning Deployme into a "control panel" for controlling remote processes.

Previous Work

There is much previously published work on package management and deployment tools. Many tools, such as Depot [Manheimer1990], take advantage of transparent remote network file system access such as NFS [Sandberg1985] or AFS [Howard1988]. A system that contains a centralized database containing server configuration information is presented in [Finke1997]. Microsoft has an architectural discussion of their corporate website publishing tool in [Moore1999].

Various methods for remote execution and server process restarting have been presented in tools such as Igor [Pierce1996] and Synctree [Lockard1998].

As stated in the previous section, Deployme does not address local package management directly. Instead, it calls routines to "activate" and "deactivate" packages on a machine. In our environment, we use GNU Stow [Glickstein1996] for this purpose.

Design and Implementation

The biggest problem with the old content release process was that all content was released at once.

Every tree push was potentially catastrophic. If any one application had some code checked in that had not been fully tested, we would have to roll back the entire tree. We learned that a big tree of content is more difficult to test than a big tree of code.

We decided to rearrange the entire content system in order to make it possible to release one application at a time using Deployme.

Deployme is written entirely in Perl5 [Wall2000]. We chose Perl because of our familiarity with it, its straightforward database integration, and our desire to attempt a complex, structured project in the Perl language.

Deployme has a simple three-tier architecture: the user interface, the rules logic, and the database.

User Interface

The Deployme interface is a set of web pages that display a list of available products and a list of potential target servers. There are also information pages that show which packages are active on a given server.

Each product is called a ''module'', after the CVS term. A product could be a collection of web pages, or the source required to build an executable.

Each release of a module is called a ''tag'', again after the CVS term. A tag corresponds to a physical package on disk. The package is assembled by requesting that the version control system write out the tagged version of the module.

The central assumption of Deployme is that the contents of a tag never change. This is stricter than the version control sense of ''tag'', since most version control systems allow users to move a tag forward or backward on a given file's timeline.

Because of this assumption, Deployme's tag namespace is a superset of the version control tag namespace. For example, the CVS tag ''engine-1'' can be tagged in Deployme as ''engine-1.DEBUG.i386.solaris.5_6'' or ''engine-1.RELEASE.sparc.solaris.5_7''. The source code is the same, but the resulting package is compiled with the appropriate build flags and architecture.

By assuming that packages never change, we can skip crosschecking between servers to establish the version of a package. This also means that the slightest modification requires a new tag.

To start a Deployme job, the user first picks a set of tags. The user is restricted to only selecting one tag for each module, because Deployme does not allow two versions of the same module to be active at the same time. Then, the user picks a set of servers. Deployme checks to ensure that the request does not break any rules, and then executes a job fulfilling the request.

To rollback a module, the user selects a previous tag and follows the same procedure.

Rules Logic

The middle tier comprises the rules logic that creates and executes a deployment job.

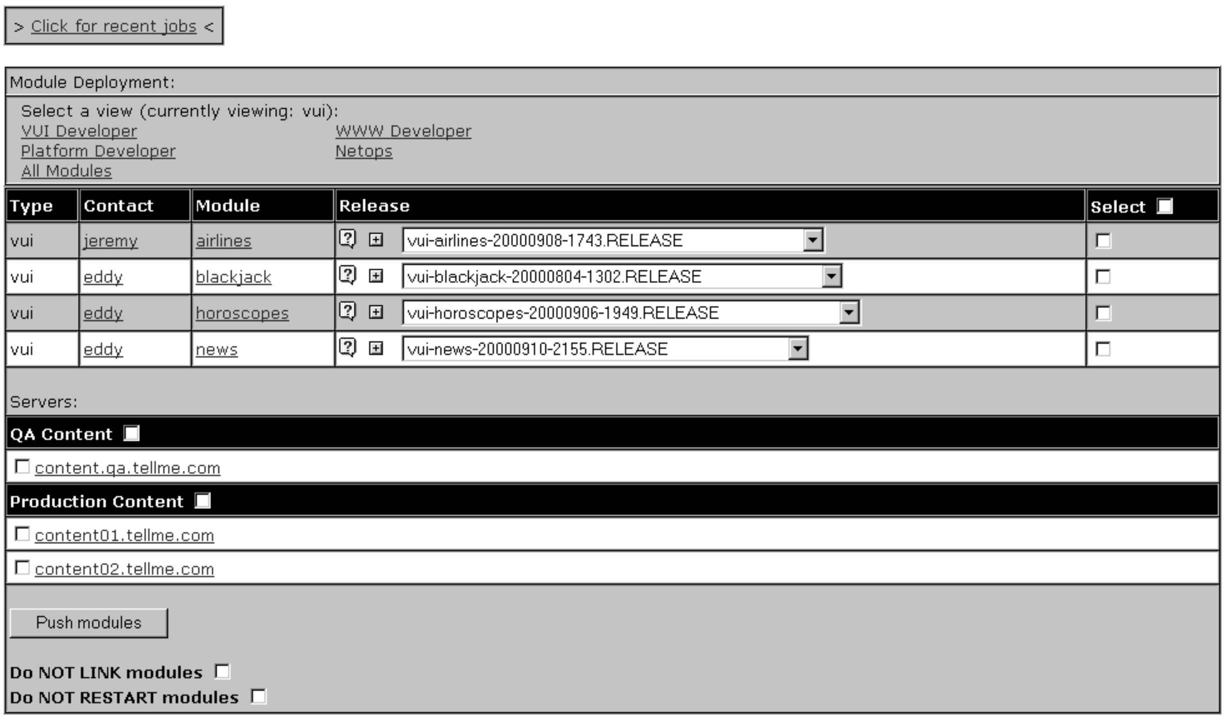

Figure 1: The Deployme web interface.

Deployme's logic falls into two categories: job creation logic and job execution logic. The former uses a set of rules to determine what tasks should be done. The latter uses the database state tables to carry out those tasks.

Job creation expands the list of packages and destinations into a series of tasks. This step is triggered when the user submits a request form in the web UI. See Table 1.

In order to figure out which tasks are necessary, the job creation algorithm relies on each module's type; see Table 2.

Module Type	Detail
platform	Source code that needs to be compiled.
www	Website content.
vui	Content for our voice-driven service.
tool	Packages that need no compilation; vendor binaries, tools, and configuration files fall into this category.

Table 2: Deployme module types.

The rules are generally very simple at this stage. For example, platform modules need a BUILD step; the others do not.

For the PUSH step, Deployme first finds out if the package is already on the destination server. If the package is already present, we skip the PUSH step, because we assume that packages never change.

This optimization makes rollback extremely fast. No data needs to be copied to the server; instead, we progress immediately to the LINK step that simply reactivates the old package.

If a PUSH is necessary, the job creation algorithm determines what route the package should take to the destination. In some cases, a single PUSH step is all that is required to copy the package from the Deployme master package repository to the remote location. However, if the final destination is at a remote facility, Deployme will instead add PUSH1 and PUSH2 steps.

The PUSH1 step sends the package to a machine at the remote facility which is designated as a gatekeeper. This machine serves as a cache of the master repository. The PUSH2 step asks the gatekeeper machine to copy the package from the cache to another machine at the same facility over the high-speed local network.

Since most of our machines are in remote locations connected by low-bandwidth connections, this logic saves hours of file transfer time.

The Database

Using a database on the backend makes it incredibly easy to provide a rich web UI. Also, a database allows operators to create highly refined reports using SQL queries. Deployme requires that the backend support SQL and the Perl Database Interface (DBI). We selected the freely available MySQL RDBMS [MySQL2000] as its cost and speed were attractive. Also, the current version of Deployme does not require any advanced database support such as transactions.

The database schema has tables that represent the visible items in the UI: modules, tags, and servers. There are additional tables, like "servergroups", that aggregate rows for a prettier display. Deployme can skip tasks that have already been done by previous jobs. The "state" table tells us which packages are active on which servers, and the "history" table holds jobs in all stages of execution.

One problem with using a database to record machine state is that the database can become unsynchronized with the real world. For instance, if a given server is replaced or if an operator performs a manual package upgrade, the database will become stale. This issue was not much of a problem for content pushes, but as we extended Deployme to include platform and tool pushes to hundreds of servers, the inconsistencies began to pile up.

We attack this problem through sanity checks and a reconciliation script. The sanity checks are performed both during job creation and execution. For

Lifecycle Stage	Deployme Task	Detail
Create package	PULL	Export package from source control.
	BUILD	Build executable or content from sources or templates.
Distribute package	PUSH	Push package directly to destination host.
	PUSH1	Push package to intermediate host.
	PUSH2	Push package from intermediate host to destination host.
Install and activate package	LINK	
Remove obsolete packages	CLEAN	

Table 1: Deployme tasks.

example, if you attempt to push a SPARC package to an Intel machine, we stop the job before it starts.

The reconciliation script scans the packages on remote machines and compiles a list of active versus inactive packages. This list is compared against the database. The check script then updates the database to match the actual state of the server.

The many problems we encountered with database consistency sparked a debate about whether the database should contain any state information at all. Another approach would be to make all job execution decisions based on the actual state of the server instead of a cached version. However, we decided against local state in this version of Deployme due to the time it takes to inventory a single server and our desire to preserve the database's excellent reporting capability. We will revisit this issue in version 2.

Job Execution

Most of Deployme's code is in the job execution system.

After job creation is complete, all tasks are filed in the database's history table. The web UI then kicks off a background process to execute the new job.

The job execution system is modular. We split Deployme into a set of Perl packages designed to handle different cases for each task.

The first tasks are generally PULL tasks. These tasks use CVS to retrieve the appropriate tag for the given module. Once the tag is pulled into a package, the state table is updated.

BUILD tasks are currently only performed for platform builds; the build task executes "make" and verifies build completion. The build module finds a build server in the server table and then uses ssh [Ylonen1996] to remotely start and monitor the job. All build servers mount the master package repository via NFS and write the build results to the package, thus completing package formation.

PUSH tasks are performed using either the rsync utility [Tridgell2000] over ssh, or the rsync utility with a listening rsync daemon. Configuration information in the database indicates which transport to use. Other transports can be added easily in the future.

For PUSH2 tasks, we open an ssh connection to the intermediate server, and then initiate an rsync over ssh to the final server.

We choose the appropriate code module for the LINK task based on the module type that is being pushed.

Linking "www" and "vui" modules types is relatively simple. Package activation often boils down to maintaining a few symlinks that point to the current packages. For our voice site, we also update the list of available keywords appropriately.

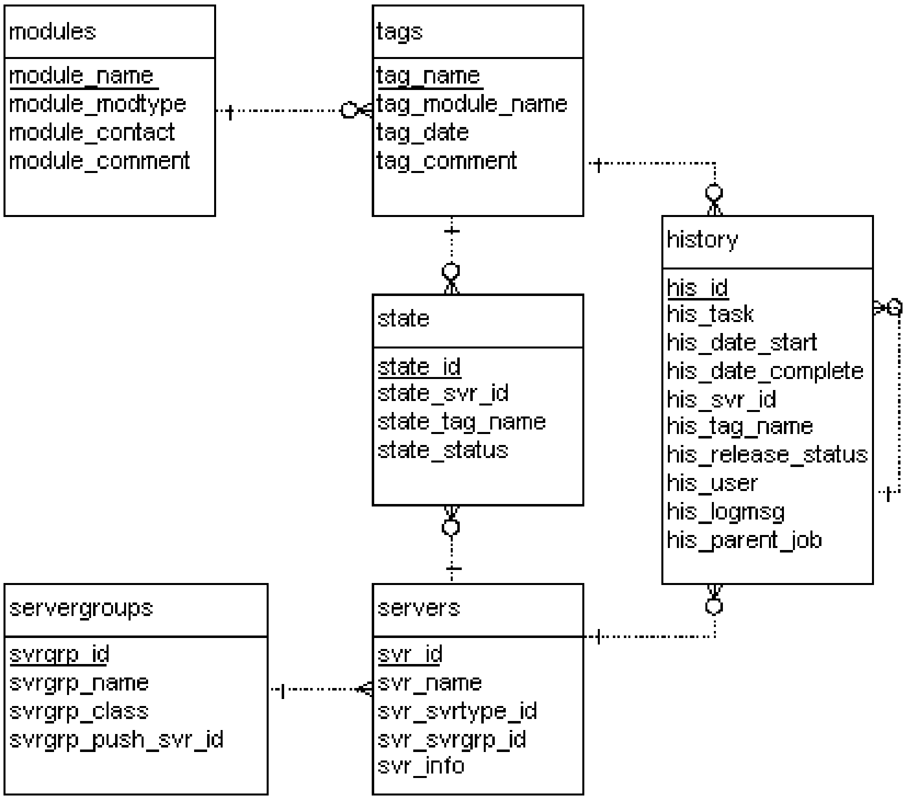

Figure 2: Database schema.

The ‘‘platform’’ and ‘‘tool’’ module types have the most complicated LINK step. We adopt the policy that any software that is running should be restarted when linked. For this reason, the LINK step needs to figure out which processes should be shut down before the link and restarted afterward. In some cases we also need to wait for processes to come back up.

As producers and developers are free to do their own releases, the extent of our release engineering responsibilities boils down to when to run CLEAN jobs. Cleanup has its own job creation logic that determines which packages are eligible for cleanup. For each module we specify how many old packages to keep on local disks and the minimum age of a package before it is eligible for deletion. Keeping some old packages allows us to do fast rollbacks since the package is already at the destination – just a quick re-link is required. Using a minimum age is an attempt to make sure we have a good package to roll back to, and not just a series of bad packages that were released in quick succession. The cleanup logic examines the content of each server and creates a list of packages to delete with the additional constraint that no active packages can be removed. A cleanup job is created that is executed similarly to a deployment job.

The Problem With Restart

The decision to add process restarts to Deployme proved to be much more difficult than we first realized.

We first tried to create some general rules about what kinds of processes to restart on a machine, and only included certain critical processes in the rules. This group of special cases was small and we wrote them into the code base. As Deployme expanded, however, it quickly began to encompass many more programs than we had planned.

This solution became a small dependency tree. This violates one of our Non-Goals discussed above. Because the dependency tree was restricted to platform LINK tasks, and because it remains rather small, we felt this transgression was minor. It turned out to be a continuing headache as we used Deployme for more modules and more servers.

From a conceptual viewpoint, ‘‘restart’’ does not fit into the package lifecycle. We rolled it into the ‘‘Link and activation of package’’ step, which increases the possibilities for failure in the LINK task.

There are significant ordering issues involved with restarting processes. While each task only relates to a given server/tag pair, the LINK step must aggregate all tasks for a given server together and then determine in what order to shut down the dependent processes. Order is determined using a small table in the database (not shown in the schema) that contains ranked module-process dependencies.

Here is an example that shows how this can become complicated. Assume we have a job pushing new telephony firmware along with new monitoring software. The firmware requires that the telephony driver and the telephony server be shut down, where the telephony driver has higher priority. The monitoring software requires that the monitoring process be shut down. For maximum efficiency, the aggregator creates a job that will shut down first the monitoring software, then the telephony server, and then the telephony firmware. All links are done, and then the software is restarted in reverse order. We also add a special-case delay for the telephony firmware since it requires a half-minute to reconfigure after the load is complete.

The ordering problem shows that it is not enough to just implement dependencies between modules and processes. We need to explicitly represent dependencies between processes and other processes.

We were correct to not spend too much effort tackling the dependency problem early on. Our implementation would have likely been incorrect. Having completed the first implementation of Deployme, we now understand what kinds of dependencies should be tracked.

Parallel Remote Execution

Another area of complexity is the method for triggering jobs on remote servers.

The remote part of the platform link stage is too complicated to use individual ssh statements, so instead we created the ‘‘linkworker’’. The linkworker is a small Perl script with no dependencies that can be used to perform link, unlink, start, and stop operations on a remote host.

Even if a link only takes a minute to perform, doing links serially across many servers takes a very long time. Serial processing severely inhibits

Detail for job #31819 [view log]

Task #	Task	Started	Completed	User	Module	Tag	Server	Info	Status
31820	PULL	2000-08-04 13:08:10	2000-08-04 13:08:10	eddy	blackjack	vui-blackjack-20000804-1302.RELEASE		Packages pulled from CVS.	SUCCESS
31821	PUSH	2000-08-04 13:08:10	2000-08-04 13:08:32	eddy	blackjack	vui-blackjack-20000804-1302.RELEASE	content.tellme.com	Package pushed	SUCCESS
31822	LINK	2000-08-04 13:08:32	2000-08-04 13:08:33	eddy	blackjack	vui-blackjack-20000804-1302.RELEASE	content.tellme.com	Linking package	INPROGRESS

Figure 3: Deployme status.

scalability. To help meet our scalability goal, we created a "parallel_mommy" routine that parallelizes all links.

When a linkworker is started, the mommy passes a "flag file" argument. As the linkworker progresses through the jobs, it records what it has done in a flag file, and dumps logging output to a log file. Once the linkworker terminates, it prints "SUCCESS" or "FAILURE" in the flag file.

The mommy routine starts all linkworkers in the background and then polls the servers to check on the flag file status. We poll instead of keeping our session open in case a transient network outage drops our network connection. As each server reports in, the mommy updates the job status and merges the remote log into the primary log.

The linkworker reports to Deployme which processes were actually shut down. After the link, Deployme only restarts those processes that were previously running.

Error Handling

A standard in Deployme is that user error should not result in a failed job. Instead, Deployme attempts to catch all user errors in the job creation step. This means that most of our users do not need to understand how Deployme works. These users just look for green boxes on the status page.

Errors are handled using Perl's structured exception handling. This system catches all expected and unexpected errors and gracefully prints an error message to the web page or log file before aborting.

All job execution actions are logged in text files that are stored for later review. The history table in the database tracks all jobs and stores the exception message if a job fails. Email is used for immediate notification of success or failure.

Automatic Rollback On Failure

Originally, part of our error handling strategy was to make individual tasks transactional. If we detected any problem with a task, we would set the state of the machine back to what it was before the task started. We found that automatic rollback on failure worked better in theory than in practice.

This policy only applied to platform and tool LINK tasks. Content LINK tasks rarely fail, and do not require any servers to be restarted. Failed PULL and BUILD tasks usually result from bad tags, and do not affect remote servers in any case. Failed PUSH tasks also do not affect remote server state because the PUSH logic copies the package to a temporary directory and renames it to the actual package name once the copy is complete. This prevents half-completed packages from appearing on remote hosts.

Initially, automatic LINK rollback seemed easy. The task is already separated into link, unlink, start, and stop commands. Also, the parallel_mommy routine (discussed above) knows what tasks were completed before the task failed. The obvious rollback is to undo whatever was done and call it quits. If the rollback fails, exit immediately. If we are executing tasks in parallel, wait until they all report in and then initiate rollback for each one that failed, serially.

The reasons for automatic rollback on failure were straightforward. We did not want to suffer downtime if a job failed, so we needed the machine restored to a functional state. Also, we felt that LINK rollback was necessary to prevent database inconsistency. We were afraid that individual failures over a period of time could make the database progressively more unreliable.

We implemented the automatic rollback system, and it turned out to be such an operational failure that we tore the code out after several weeks.

The problem was that all of the reasons cited above were wrong.

Automatic rollback on failure became a major inconvenience for the network operations team. Instead of a job failing immediately so they could go in and fix it, they would be forced to wait while processes were restarted. The serial nature of the rollback made this maddening when multiple servers suffered similar failures.

Besides the delay, it turned out that operations did not want the machine restored to its original state. Due to our network architecture, it is not a problem to have a single machine out of service. After a successful rollback, the first thing that operations did was shut down the server that rollback had restarted.

Automatic rollback did not make the database more consistent. For many failed tasks, the database was inconsistent before the job began. Rolling back just went from one inconsistent state to the original inconsistent state.

We learned from this experiment. In our case, a fail-fast system results in shorter outages than a system that tries to fix problems on its own.

The Minimal Downtime Paradox

We found that some of our goals were not exactly what our Operations team wanted. All of the link scripts meet our "Seamless Activation" goal to get the servers back up and running as soon as possible. However, in regular maintenance scenarios that is not the usual Operations procedure. They often will take servers down for an extended time to perform other maintenance along with a software push. Also, some did not agree with the policy of "that which is on disk must be running."

As with rollback, we saw the paradox that restarting server processes automatically can result in longer outages, since operations will just have to stop and start the server anyway for their own maintenance.

In response, we added a feature to prevent server restarts from occurring during a LINK job.

War Stories

Deployme does not have any UI for performing reporting, but its database is enormous. The tables contain sufficient information to reconstruct every software upgrade Tellme has ever made since Deployme was introduced.

In several cases, we have used the database to ferret out information such as ''When did this package reach the production servers?'' and ''Which modules were upgraded since yesterday?''

Deployme makes it so easy and fast to move to a previous package that our Operations team is rarely alerted when a bad package is released. Instead, the content producer quickly reverts to a good package and keeps the outage brief.

In many ways, Deployme has reduced the number of war stories we have to tell. The only major content-related outages since Deployme was instituted resulted from individuals going around the Deployme process and editing files directly on production servers. (Old habits are hard to break.) There really is no reason to do such a thing in the Deployme world.

Future Work

Despite Deployme's merits, the current implementation is creaking loudly. The code base, approximately 10,000 lines of code, has become difficult for us to maintain.

The maintenance problem centers around the conditionals sprinkled throughout the code to check for a particular set of module types. For example, the ''execute_link'' subroutine performs a large conditional to figure out which Perl module to call. Another example is that the web UI is littered with module type checks to determine what options to offer when creating a tag.

Our solution for this is a concept called ''services'' which was introduced in [Finke1997], but our usage is slightly different. Instead of a module type, each module will be associated with a list of target services where that module can be deployed. Servers will provide services instead of having a single server type. The services table will contain a reference to the Perl object that will provide the service-specific code, isolated from the Deployme core. Additionally, we will add a START and STOP task to the list of deployment tasks so service restarts are distinct from the LINK task.

One application of services is to allow Deployme to push to virtual web servers. Deployme only allows web content to go to a single location on a remote server. With services, we can push to the ''developer'' service on a web box separately from the ''corpweb'' service.

Our preliminary services plug-in implementation works very well. Because of this, we are moving all of the Deployme logic into objects descended from an abstract base class. For instance, CVS access will be abstracted under an SCS, or source control system, class. In this particular case, there has been some discussion about moving to Perforce [Perforce2000] from CVS, and we want to be able to accommodate such a switch with little pain.

All job execution steps and web UI are also being moved into subclasses. Once this is complete, the Deployme logic will be completely separated from the Tellme site logic.

Another problem with Deployme is that it has no concept of a physical location shared between two machines. We investigated changing the database schema to know the difference between physical and logical locations, but the result was extremely convoluted and hard to use. Instead, we plan to provide a mapping table that designates a single server as the master for a shared mount point. At job creation, we will map any requests for deployment to a specific server/service pair to the master server if specified in the mapping. Since the job creation logic removes duplicate steps, we can be certain that we will deploy to a shared location only once.

We plan to change the underlying database from MySQL to PostgreSQL [Lockhart2000]. PostgreSQL supports true referential integrity, which is necessary for the more complex Deployme2 schema. Also, PostgreSQL has transaction support, which makes error handling easier.

Security is another area where much work remains. We want to build a granular system that limits what a user can do. Deciding how fine to slice access control is a tough question. To start, we will probably create ''roles'' which have different capabilities.

We need a way to quickly copy a package to multiple servers on a subnet. We have some prototype multicast file transfer programs written, but we have not yet decided how to incorporate them into Deployme. In addition to transferring bits via multicast, we could also blast commands to many hosts at once.

Further enhancements include integrating Deployme into the system we use to initialize brand new machines. Also, we would like to add a ''last-known-good'' feature so that it is easy for operators to know what package is safe to roll back to.

Status

As of this writing, we are working on Deployme 2. We intend to integrate many of the conceptual changes cited in Future Work. Most of the technology upgrades, such as multicast, are not a priority right now. This is because we are more interested in adding the ability to add technology than the technology itself.

Conclusions

We originally wanted to find an off-the-shelf tool for accomplishing the goals stated above. However, none of the tools we reviewed provided an end-to-end solution for package lifecycle management.

Deployme has greatly exceeded our expectations. While it began as a simple content manager, it has expanded to become the upgrade center for our entire server environment. It currently tracks a few hundred servers, and we believe it can track thousands as we improve the underlying technology.

The best thing Deployme has done for Tellme is something we never considered when writing up the goals. It has reduced our stress level.

Stress is not something normally considered when writing tools. Usually we focus on scalability, speed, ease of use, and other immediately evident goals. When all of these immediate goals are met and the tool is firmly inserted into your process, that is when you see the secondary benefits.

As Deployme began to encompass more teams within the company, our collective confidence grew and usage skyrocketed. Each day, Deployme processes 40 to 60 jobs encompassing hundreds of individual tasks.

Our development schedules are not held hostage to a release team. Instead, Deployme has engendered a release democracy, where even the newest employee is empowered to take over a module and start sending new packages through QA and up to production.

The best accomplishment of all is our most frequently asked question, "That's all I need to do?"

Author Information

Kyle Oppenheim is a software engineer at Tellme Networks. He graduated from Carnegie Mellon University with B.S. and M.S. degrees in Electrical and Computer Engineering. He is currently playing the role of Release Engineer, but is intently trying to automate his job away. Reach him electronically at kyleo@tellme.com .

Patrick McCormick is a software engineer at Tellme Networks. He graduated from the Massachusetts Institute of Technology with B.S. and M.Eng. degrees in Electrical Engineering and Computer Science. His current professional interests include voice recognition, computer telephony, and software deployment. His email address is patrick@tellme.com .

References

[Cederqvist1993] P. Cederqvist, et al., "Version Management with CVS," http://www.loria.fr/˜molli/cvs/doc/cvs_toc.html , 1993.

[Dagenais1993] M. Dagenais, S. Boucher, R. Gerin-Lajoie, P. Laplante, P. Mailhot, "LUDE: A Distributed Software Library." *Proceedings of the Seventh Large Installation Systems Administrators Conference,* Monterey, CA, November 1-5, 1993, pp. 25-32.

[Finke1997] Finke, Jon, "Automation of Site Configuration Management," *Proceedings of the Eleventh Large Installation Systems Administrators Conference,* San Diego, October 26-31, 1997, pp. 155-168.

[Furlani1996] J. Furlani, P. Osel, "Abstract Yourself with Modules," *Proceedings of the Tenth Large Installation Systems Administrator's Conference*, Chicago, September 29-October 4, 1996.

[Glickstein1996] B. Glickstein, "Managing the Installation of Software Packages," http://www.gnu.org/software/stow/manual.html , 1996.

[Howard1988] Howard, John H, "An Overview of the Andrew File System," *Proceedings of the USENIX Winter Technical Conference,* Dallas, pp. 23-26, February 1988.

[Lockard1998] J. Lockard, J. Larke, "Synctree for Single Point Installation, Upgrades, and OS Patches," *Proceedings of the Twelfth Large Installation Systems Administrator's Conference,* Boston, December 6-11, pp. 261-270, 1998.

[Lockhart2000] Lockhart, Thomas, ed., "PostgreSQL User's Guide," http://www.postgresql.org/docs/user/index.html , 2000.

[Manheimer1990] K. Manheimer, B. Warsaw, S. Clark, W. Rowe, "The Depot: A Framework for Sharing Installation Across Organizational and UNIX Platform Boundaries," *Proceedings of the Fourth Large Installation Systems Administrator's Conference,* Colorado Springs, CO, October 18-19, pp. 37-46, 1990.

[Moore1999] Moore, Michael, "Publishing to the Web in a Distributed Environment," http://www.microsoft.com/backstage/bkst_column_10.htm , July 1999.

[MySQL2000] MySQL AB, "MySQL Documentation," http://www.mysql.com/documentation/index.html , 2000.

[Perforce2000] Perforce Software, "Perforce Documentation," http://www.perforce.com/perforce/technical.html , 2000.

[Pierce1996] C. Pierce, "The Igor System Administration Tool," *Proceedings of the Tenth Large Installation Systems Administrator's Conference,* Chicago, pp. 9-18. September 29-October 4, 1996, pp. 9-18.

[Sandberg1985] R. Sandberg, D. Goldberg, S. Kleiman, D.Walsh, B. Lyon, "Design and Implementation of the Sun Network Filesystem." *USENIX Conference Proceedings,* USENIX Association, Berkeley, CA, pp. 119-30, Summer 1985.

[Tridgell2000] A. Tridgell, "Efficient Algorithms for Sorting and Synchronization," http://linuxcare.com.au/tridge/phd_thesis.pdf, 2000.

[Wall2000] L. Wall, "Programming Perl," 3rd ed, O'Reilly & Associates, Sebastopol, CA, 2000.

[Ylonen1996] T. Ylonen, "SSH – Secure Login Connections over the Internet," *Proceedings of the Sixth USENIX UNIX Security Symposium,* San Jose, CA, pp. 214, 37-42, July 22-25, 1996.