In-Flight Mechanics
A Software Package Management Conversion Project

Philip J. Hollenback
Yahoo, Inc.

Overview

Over the course of most of 2009 I acted as the technical lead on a software package management conversion project in Yahoo Mail. I was part of a team of six people that ultimately reinstalled the software on most of the core servers and developed a completely new release management process. This paper documents the process we used, the problems we found, and the current state of the system as of mid-2010.

The core of Yahoo Mail is approximately 7000 servers in datacenters around the world. These servers are grouped into farms of roughly 15 machines and associated back-end mail storage. Each farm stores the mail for a few hundred thousand to a few million users and acts as the central access point for a user's Yahoo Mail experience.

Traditionally these servers have been managed with a proprietary Yahoo dependency-based packaging system called yinst. Yinst operates similarly to RedHat rpm or Debian dpkg (although with more integration with a central software repository, so perhaps the best analogy is with yum or apt-get).

In that system a release was just a list of packages to be installed on the hosts, with only minimal explicit considerations of dependencies. Any package dependencies not included in the initial install list would be automatically downloaded and installed as needed by yinst. The install process was based on an automated tool which would ssh to each host and launch yinst to request installation of the package list.

This mechanism served Yahoo Mail well for a number of years. As the size of our installation grew, however, several deficiencies became apparent. Over time the task of managing our installed software and settings became more and more difficult. For example, the old system was completely additive – it was simple to create and install a new package, but extremely difficult to remove a package once it had been installed in production. This eventually led to more than 1300 packages installed on each server. Yinst includes per-package settings management as well as package management, but the existing system included no easy way to tie settings and packages together in a coherent release. The idea of rollback to a previous system state was also not well supported.

To address these issues, a software tools group at Yahoo had been working for several years to develop a state-based package and setting management solution called Igor. Igor sits on top of yinst and enforces the exact package and setting list for a given host, through a versioned tagging mechanism. In essence this moves dependency checking from runtime to compile time (with compile time being the creation of the release). Igor had been in use for several years in other properties at Yahoo and appeared to work well. Thus in early 2009 my team was given the task of converting to Igor (igorizing) all the 7000 mail farm hosts. At the highest level this process included:

- modeling existing server environment in Igor
- converting to 'All Apps All Environments'
- converting all production servers to use Igor

Along the way we learned some useful lessons about planning and implementing large-scale software conversions.

All Apps All Environments

The core tenet of this conversion was a philosophy we call 'All Apps All Environments'. This means that all systems of a given type must have the same set of packages installed. Package behavior must be controlled by the use of yinst settings. We follow this principle to limit complexity and maximize reproducibility.

This lead to many challenges in the conversion process. In particular since yinst made it so easy to create

and publish software packages the natural tendency had been to encapsulate configuration data inside packages, and those packages would be installed on certain subsets of hosts. Since our goal was to install the same packages everywhere, we were forced to restructure many yinst packages to remove the configuration data from packages.

Why These Tools?

Why did we not use existing open source configuration management tools like Puppet or Cfengine? The main answer is that those decisions had largely been made by the company before this project was started. Igor and yinst were the company-recommended tools for configuration management and packaging. Using these tools meant that we could take advantage of a large repository of knowledge inside the company, instead of having to implement a solution from scratch.

Another key consideration for selecting software tools for this project was the number of servers involved. Remember we are managing about 7000 servers on multiple continents. Switching to a different set of tools would require extensive testing and planning to ensure that the tools would scale to that level.

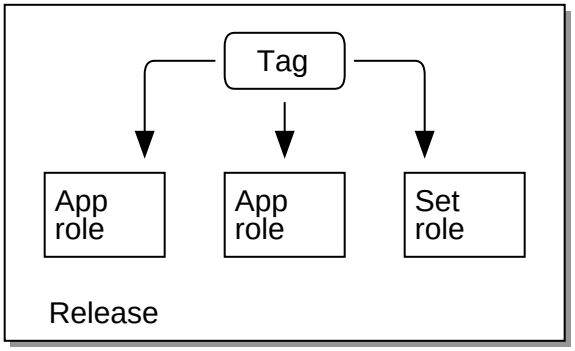
Finally, many of our existing software processes make deeply ingrained assumptions about the available tools. Changing to different tools would have greatly expanded the scope of the project, with little practical benefit.

Process

This conversion work went through several distinct steps. First we had to model the existing environment. Remember that the existing package management system did not allow exact specification of packages and settings on the individual servers, since it worked via dependencies which could be called in at any package install. For each unsatisfied dependency, yinst would check the distribution servers for the version of the package marked as `stable` and install that version. The stable flag can be moved to a new version of a package at any time by the package maintainers, making it impossible to completely lock down the packages installed on a set of machines over time.

Modeling

The first step in the process was to model the existing environment. We did this by splitting software packages into logical groups by development area – backend, middleware, ops and so on. Each development team then had to vet their package list, which was turned into an Igor 'application role'. An application role contains packages and settings for one component of a release. Once we had this list of packages and settings assembled in Igor we could then build a release, which consisted of a unique Igor tag pointing to particular revisions of each application role. Thus, one Igor tag precisely defined the collection of packages and settings in a release, and that was guaranteed to never change. There are also per-farm igor set roles which contain settings unique to each farm.



Note that in an igorized context, yinst no longer attempts to install dependencies - all packages and setting must be explicitly defined in the igor roles. Dependency information in yinst packages is only used for determining the order of package installs in the igorized context.

VM Installs

Following the modeling work we did server installs on virtual machines using the resulting package and setting lists. This was a several month iterative process, during which we discovered many hidden problems. One of the largest issues was that since we had previously used an additive software install process, packages were not rigorous in their dependency information. For example, many packages did not list Perl as a dependency, since Perl is such a low-level package that an additive install process can generally assume some other package installed it as a dependency.

In addition, yinst can auto-compute only a subset of dependencies orderings. If a package actually uses Perl after it is installed, yinst can detect that and make sure Perl is installed. However, this fails when the only use of Perl is in a package install script. Consider an install script which calls a perl library via a `Use` directive. yinst has no way to detect that this perl library must be installed before the install script runs. In the additive install environment, this is again not an issue as most perl libraries get installed on hosts very early in the install process. In the igorized environment, this leads to random unpredictable install failures, depending on what packages were added to or removed from a particular release.

Using virtual machines allowed us to quickly test many of these situations by doing repeated system in-

stalls. This allowed us over the course of several months to converge on a set of packages which could be installed as a group with no missing dependencies and result in what appeared to be a running system.

The Importance of QA

At this point in the process we had what appeared to be a working software release – it would install on a test server and respond to requests as expected. However, given the complexity of the software running on our mail servers this was not sufficient to guarantee a working release. Thus it was important that QA rigorously test the proposed release, which they did through running their large manual and automated test process against the release in QA. This step was absolutely critical because we needed as much certainty as possible that the release worked the same as existing software installs before we put it in production.

In Flight Mechanics

The massive scope of this project meant that a staggered rollout was the only option, as it would take multiple months to complete the deployment to all the machines. This presented the immediate complication that we would have to maintain parallel software releases in both the old and new system for the entire duration. This proved to be a huge administrative burden in situations like emergency software patches. The overhead of tracking which mail farms were and were not igorized was also substantial.

Initially we had hoped to complete the work in at most three months, based on the assumption that we could igorize a machine (set it to a known state by adding/removing packages and settings via igor) with our existing software installed. However we quickly realized this was impossible due to badly formed packages already installed on machines. In particular circular package dependencies and buggy package stop scripts made this completely impossible.

As a result we were forced to switch to recloning every machine to the base OS as the first step. This did have the advantage of setting every machine to the same OS release and known software state. However, it also extended the completion of the project by several months.

We also found while igorizing the first production farm that yinst settings were not consistent between the hosts on the farm. This could cause production-impacting outages as some of those settings were tied to external data sources which had been set on a per-host level. For example, half the hosts on a farm would communicate with (and were authorized for) one set of user database servers, while the other half were connected to a different set of user database servers.

This issue caused us to rethink our deployment model and realize the importance of auditing tools. The Mail Tools team took on the task of defining and developing our main auditing tool. This tool would log on to every host in a mail farm and collect the yinst package and setting list. Then it would compare these lists between all hosts to ensure they were the same. Any differences would be flagged in the output.

This tool was partially automated in that it could collect the data itself, but this still required human intervention to decide if the differences were meaningful or not. For example, had an extra package been installed on one host for testing? If so that discrepancy could be ignored as the system reclone would erase it. However if half of the machines were connected to one user database server and half to another, that was a critical difference that would have to be corrected before igorization could continue, or else an outage would result.

Because of this need for extensive auditing, we switched to a scout model, where one team member would run audits ahead of the rest of the team doing the igorization work. This way we always had a supply of audited farms ready for conversion as the project progressed.

A Long Slog

After the first few farms were igorized we had a defined process for attacking the problem. Thus the work became a long process of attacking farms every week until we were done. Our team of six people igorized the first farm in May 2009 and the last one in November of that year. This meant we had many months of parallel releases in both the old and new release systems.

Lessons Learned

Any large software project generates a long list of lessons learned and this was no exception. Here are some things I wish I'd known or thought about more when we began the process:

Software Tools Are Critical

We had assumed when we started that software packages and settings were at least consistent between servers on a given farm. This was not true in production, which forced us to develop an extensive farm auditing mechanism as detailed above. Our partial automation of this process was useful, but we should have automated it more fully and spent more time making the audit process more seamless. A result of the need for auditing was that

one person on the project could do little else besides auditing full time. If we had automated the auditing earlier in the process it would have sped things up considerably.

Don't Assume Developers Can Fix Packages

Most software developers have very little knowledge of (or interest in) how software installs work. This creates an immediate conflict because developers are expected to write their own software installation scripts. Our number one problem during this process (which continues to this day) is poorly assembled software packages and poorly written package installation and removal scripts. In retrospect we should have worked much harder to ensure these were fixed before we started the conversion. We should have much more clearly communicated to development groups about our standards for package quality. More best practices documents would have helped tremendously in this regard.

Configuration Packages Are Deadly

In a dependency-based package management system, packages which contain nothing but configuration information for other packages are acceptable (or at least tolerable). If you want to change the system configuration, simply write a new configuration package and push it to hosts. The design of the yinst package system made this particularly easy to do, since creating and distributing a new package is trivial.

This is completely untenable in a state-based package management system, because you are committing to creating and testing a complete release before rolling out to production. If you find a problem in a configuration package, your only choice is to create and test a new release with a new version of the package. There are alternatives such as pushing a special configuration package to some hosts, but then you are breaking the release model and the 'All Apps All Environments' philosophy.

In retrospect we should have demanded that all configuration packages be rewritten to expose their settings as yinst settings (which can be manipulated outside the context of a release) or to pull configuration from a central server. This has been the cause of many patches to our software releases and we continue to work with development groups to eliminate these packages.

Future Work

As of August 2010 we have assembled and pushed to production 25 igorized mail farm software releases. We have successfully proven the usefulness our state-based package management system. Outages due to software change have been reduced significantly, and we can now push software to production much more reliably. However, a tremendous amount of work remains to be done. Several highlights:

Continue Perfecting Rollback

If you specify the system state exactly, rollback should be a simple matter of moving the system back to a previous set of packages and settings. We have successfully rolled back releases in production, however the process is an extreme measure. To make this work we have had to invest significant amounts of time in package cleanup and rollback testing for each release.

One fundamental limitation with rollback of an entire release is that it rolls back all packages in a release, and that list invariably includes a multitude of fixes. It is very difficult to separate feature additions and bugfixes in a monolithic release.

Note that emergency rollback works very well in our system. If a release is deployed to a particular farm and that causes a problem, our pushmasters can quickly roll the farm back to the previous release to restore service. Our extensive QA of each release includes rollback and roll-forward testing to guarantee this always works.

Fully Implement Sync Monitoring

Once you have the state of all machines specified, you need to monitor all machines and validate that they are all in sync. If a machine is out of sync, either the host is not running the proper release or someone has manually changed it. Implementing a system to monitor this on all mail hosts has proven troublesome both because of the load it imposes on the state servers and because of the various ways systems can be out of sync. Sometimes it is necessary to temporarily change the packages or settings installed on a host, and this leads to low-level churn in the sync monitoring.

Continuous Integration

We are actively pursuing using continuous integration to build releases and to build other parts of our release system such as the templating mechanism we use to define per-farm settings. Our goal is to eventually be building and testing releases continuously with Hudson.

Conclusion

The purpose of this project was to explicitly define the state of Yahoo Mail as an entity, by exactly defining the state of all hosts. We have generally succeeded in this goal, as we now roll out state-based releases on a regular schedule, and we do rollbacks as necessary. How-

ever, much work remains to be done in areas such as monitoring and general package quality. In conclusion, state-based software package management does work on a large scale, but it requires a significant investment in planning and labor to be successful.

Acknowledgments

This project would not have been possible without the incredible dedication of everyone involved. In particular, thanks to the entire Release Management team: Jen Draper, Shajeeb Muhammad, Jerrod Kensil, Brian McNeff, Tisha Emmanuel and Prem Anand Ramnath.

A special thanks to Nic Harteau and the Mail Tools team, who created many of the tools that made this work possible. Nic also codified the 'All Apps All Environments' philosophy.