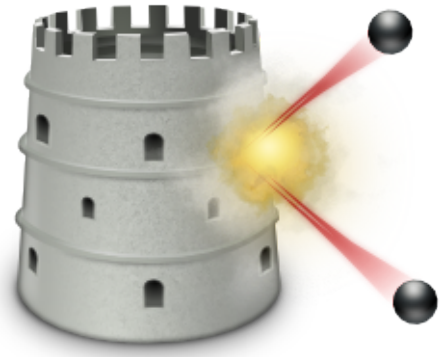
Managing Large Debian Repositories with Pulp (and less than 50GiB of RAM)

Quirin Pamp

04.02.2019

orcharhino, Foreman and Katello

Foreman



Katello

Pulp

- Pulp is a platform for software repository management.
- The pulp_deb plugin adds APT repository support.

Pulp

- Pulp is a platform for software repository management.
- The pulp_deb plugin adds APT repository support.

What about Pulp Debian support?

- Pulp Debian support has been around for a while.
- It was expanded by ATIX for use with Katello a little over a year ago.
- Works great for small to mid sized repositories.
- But, all was not well performance wise…

What about Pulp Debian support?

- Pulp Debian support has been around for a while.
- It was expanded by ATIX for use with Katello a little over a year ago.
- Works great for small to mid sized repositories.
- But, all was not well performance wise…

What about Pulp Debian support?

- Pulp Debian support has been around for a while.
- It was expanded by ATIX for use with Katello a little over a year ago.
- Works great for small to mid sized repositories.
- But, all was not well performance wise…

What about Pulp Debian support?

- Pulp Debian support has been around for a while.
- It was expanded by ATIX for use with Katello a little over a year ago.
- Works great for small to mid sized repositories.
- But, all was not well performance wise…

Several months ago...

- Our consultants wanted to synchronize all of Debian Stretch and Ubuntu Xenial for a Demo.
- and found that,
  - ...this generally takes about 5 hours.
  - ...only to fail with a "Cannot allocate memory" error.

Several months ago...

- Our consultants wanted to synchronize all of Debian Stretch and Ubuntu Xenial for a Demo.
- and found that,
  - ...this generally takes about 5 hours.
  - ...only to fail with a "Cannot allocate memory" error.

Several months ago...

- Our consultants wanted to synchronize all of Debian Stretch and Ubuntu Xenial for a Demo.
- and found that,
  - ...this generally takes about 5 hours.
  - ...only to fail with a "Cannot allocate memory" error.

What was going on?

- To answer this question we need to know things about the pulp_deb implementation:
  - Code is organized into several steps.
  - The implementation relies heavily on the python-debpkgr dependency.
  - Which in turn relies on deb822 from the python-debian library.

What was going on?

- To answer this question we need to know things about the pulp_deb implementation:
  - Code is organized into several steps.
  - The implementation relies heavily on the python-debpkgr dependency.
  - Which in turn relies on deb822 from the python-debian library.

What was going on?

- To answer this question we need to know things about the pulp_deb implementation:
  - Code is organized into several steps.
  - The implementation relies heavily on the python-debpkgr dependency.
  - Which in turn relies on deb822 from the python-debian library.

What was going on?

- To answer this question we need to know things about the pulp_deb implementation:
  - Code is organized into several steps.
  - The implementation relies heavily on the python-debpkgr dependency.
  - Which in turn relies on deb822 from the python-debian library.

What is python-debpkgr?

- python-debpkgr is mainly designed to take a pile of Debian packages and organize them into an APT repository.

How are Debian repositories structured?

```
/dists/stretch/Release
/dists/stretch/main/binary-amd64/Packages
/dists/stretch/contrib/binary-amd64/Packages
/dists/stretch/non-free/binary-amd64/Packages
/pool/
```

What was going on? (2)

- During a sync, we have the "MetadataStep".
- The "MetadataStep" is given a list of releases, components, and packages (with meta data) from the Mongo DB.
- It then performs some logic:
  - For every combination of architecture, component, and release, a list of packages is generated.
  - These lists contain the paths to the actual .deb package files on the disk.
  - Finally, each list is passed to a debpkgr call as an argument.

What was going on? (2)

- During a sync, we have the "MetadataStep".
- The "MetadataStep" is given a list of releases, components, and packages (with meta data) from the Mongo DB.
- It then performs some logic:
  - For every combination of architecture, component, and release, a list of packages is generated.
  - These lists contain the paths to the actual .deb package files on the disk.
  - Finally, each list is passed to a debpkgr call as an argument.

What was going on? (2)

- During a sync, we have the "MetadataStep".
- The "MetadataStep" is given a list of releases, components, and packages (with meta data) from the Mongo DB.
- It then performs some logic:
  - For every combination of architecture, component, and release, a list of packages is generated.
  - These lists contain the paths to the actual .deb package files on the disk.
  - Finally, each list is passed to a debpkgr call as an argument.

What was going on? (2)

- During a sync, we have the "MetadataStep".
- The "MetadataStep" is given a list of releases, components, and packages (with meta data) from the Mongo DB.
- It then performs some logic:
  - For every combination of architecture, component, and release, a list of packages is generated.
  - These lists contain the paths to the actual .deb package files on the disk.
  - Finally, each list is passed to a debpkgr call as an argument.

What was going on? (2)

- During a sync, we have the "MetadataStep".
- The "MetadataStep" is given a list of releases, components, and packages (with meta data) from the Mongo DB.
- It then performs some logic:
  - For every combination of architecture, component, and release, a list of packages is generated.
  - These lists contain the paths to the actual .deb package files on the disk.
  - Finally, each list is passed to a debpkgr call as an argument.

What was going on? (2)

- During a sync, we have the "MetadataStep".
- The "MetadataStep" is given a list of releases, components, and packages (with meta data) from the Mongo DB.
- It then performs some logic:
  - For every combination of architecture, component, and release, a list of packages is generated.
  - These lists contain the paths to the actual .deb package files on the disk.
  - Finally, each list is passed to a debpkgr call as an argument.

What does debpkgr do with the lists?

- Remember that debpkgr is mainly designed to take a pile of Debian packages and turn them into a repo.
- So, it does just that:
  - Each .deb file is accessed on the disk to extract the meta data debpkgr needs.
  - Because of the way the package lists overlap for different architectures, many of these .deb files will actually be parsed multiple times.

What does debpkgr do with the lists?

- Remember that debpkgr is mainly designed to take a pile of Debian packages and turn them into a repo.
- So, it does just that:
  - Each .deb file is accessed on the disk to extract the meta data debpkgr needs.
  - Because of the way the package lists overlap for different architectures, many of these .deb files will actually be parsed multiple times.

What does debpkgr do with the lists?

- Remember that debpkgr is mainly designed to take a pile of Debian packages and turn them into a repo.
- So, it does just that:
  - Each .deb file is accessed on the disk to extract the meta data debpkgr needs.
  - Because of the way the package lists overlap for different architectures, many of these .deb files will actually be parsed multiple times.

What did we do about it?

- First thought: Maybe theres a quick and dirty fix?
- Next possibility: Completely redesignt the way debpkgr works?
- Finally: Drop debpkgr (from the MetadataStep) and implement everything ourselves.

What did we do about it?

- First thought: Maybe theres a quick and dirty fix?
- Next possibility: Completely redesignt the way debpkgr works?
- Finally: Drop debpkgr (from the MetadataStep) and implement everything ourselves.

What did we do about it?

- First thought: Maybe theres a quick and dirty fix?
- Next possibility: Completely redesignt the way debpkgr works?
- Finally: Drop debpkgr (from the MetadataStep) and implement everything ourselves.

Basic Idea

- Exclusively use information from the Mongo DB to create the repository structure.

Needed functionality

- Remember that the old implementation already had to parse the meta data from the Mongo DB in order to generate the lists that were then passed to debpkgr.
- This essentially remained unchanged.
- We needed to create the desired directory structure ourselves.
- We needed to create the symlinks to the actual .deb files ourselves.
- We needed the ability to write Packages and Release files.

Needed functionality

- Remember that the old implementation already had to parse the meta data from the Mongo DB in order to generate the lists that were then passed to debpkgr.
- This essentially remained unchanged.
- We needed to create the desired directory structure ourselves.
- We needed to create the symlinks to the actual .deb files ourselves.
- We needed the ability to write Packages and Release files.

Needed functionality

- Remember that the old implementation already had to parse the meta data from the Mongo DB in order to generate the lists that were then passed to debpkgr.
- This essentially remained unchanged.
- We needed to create the desired directory structure ourselves.
- We needed to create the symlinks to the actual .deb files ourselves.
- We needed the ability to write Packages and Release files.

Needed functionality

- Remember that the old implementation already had to parse the meta data from the Mongo DB in order to generate the lists that were then passed to debpkgr.
- This essentially remained unchanged.
- We needed to create the desired directory structure ourselves.
- We needed to create the symlinks to the actual .deb files ourselves.
- We needed the ability to write Packages and Release files.

Needed functionality

- Remember that the old implementation already had to parse the meta data from the Mongo DB in order to generate the lists that were then passed to debpkgr.
- This essentially remained unchanged.
- We needed to create the desired directory structure ourselves.
- We needed to create the symlinks to the actual .deb files ourselves.
- We needed the ability to write Packages and Release files.

Stumbling Blocks

- debpkgr generates md5sum, sha1, and sha256 for metadata.
- The existing data base model only stored sha256 hashes.
- Actually using the meta data from the data base revealed a bug.
- User defined meta data fields/fields not stored in the existing data base model.

Stumbling Blocks

- debpkgr generates md5sum, sha1, and sha256 for metadata.
- The existing data base model only stored sha256 hashes.
- Actually using the meta data from the data base revealed a bug.
- User defined meta data fields/fields not stored in the existing data base model.

Stumbling Blocks

- debpkgr generates md5sum, sha1, and sha256 for metadata.
- The existing data base model only stored sha256 hashes.
- Actually using the meta data from the data base revealed a bug.
- User defined meta data fields/fields not stored in the existing data base model.

Stumbling Blocks

- debpkgr generates md5sum, sha1, and sha256 for metadata.
- The existing data base model only stored sha256 hashes.
- Actually using the meta data from the data base revealed a bug.
- User defined meta data fields/fields not stored in the existing data base model.

Results

- Two major pull requests:
  - https://github.com/pulp/pulp_deb/pull/61
  - https://github.com/pulp/pulp_deb/pull/57
- An end to our memory problems.
- Syncs for mid sized repositories (1500 packages) are more than twice as fast.
- Syncing Ubuntu Xenial (main, restricted, universe, multiverse) for amd64 (53837 Packages) took 3h36m on my test system.

Results

- Two major pull requests:
  - https://github.com/pulp/pulp_deb/pull/61
  - https://github.com/pulp/pulp_deb/pull/57
- An end to our memory problems.
- Syncs for mid sized repositories (1500 packages) are more than twice as fast.
- Syncing Ubuntu Xenial (main, restricted, universe, multiverse) for amd64 (53837 Packages) took 3h36m on my test system.

Results

- Two major pull requests:
  - https://github.com/pulp/pulp_deb/pull/61
  - https://github.com/pulp/pulp_deb/pull/57
- An end to our memory problems.
- Syncs for mid sized repositories (1500 packages) are more than twice as fast.
- Syncing Ubuntu Xenial (main, restricted, universe, multiverse) for amd64 (53837 Packages) took 3h36m on my test system.

Results

- Two major pull requests:
  - https://github.com/pulp/pulp_deb/pull/61
  - https://github.com/pulp/pulp_deb/pull/57
- An end to our memory problems.
- Syncs for mid sized repositories (1500 packages) are more than twice as fast.
- Syncing Ubuntu Xenial (main, restricted, universe, multiverse) for amd64 (53837 Packages) took 3h36m on my test system.

Lessons Learned

- Know your tools!
- Take some time to plan the architecture.
- Take some time to gain the needed domain knowledge.

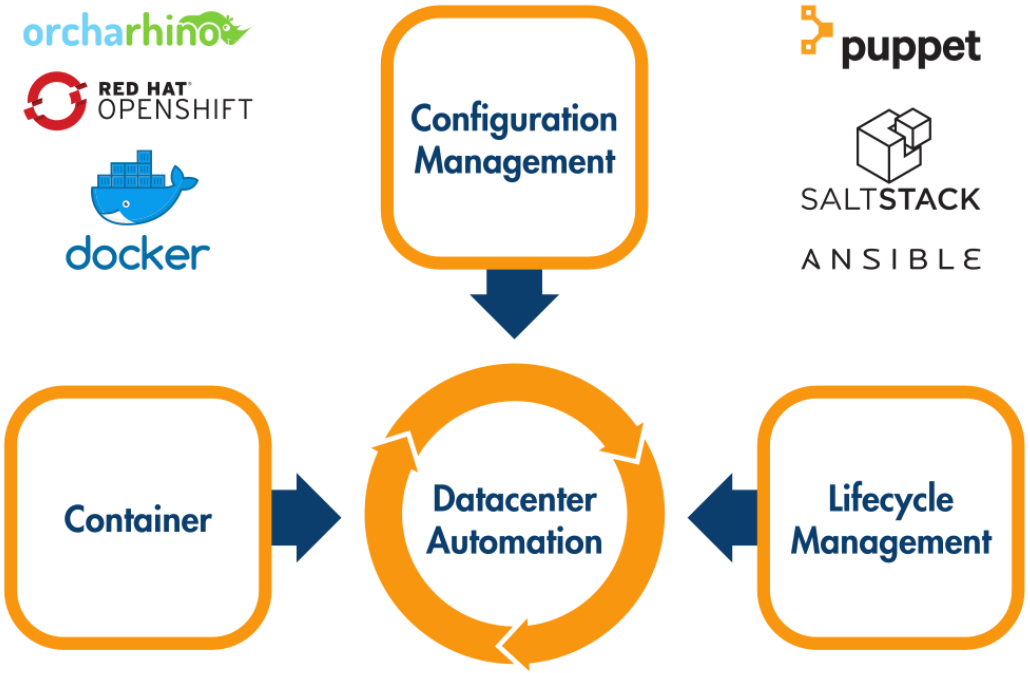
Lessons Learned

- Know your tools!
- Take some time to plan the architecture.
- Take some time to gain the needed domain knowledge.

Lessons Learned

- Know your tools!
- Take some time to plan the architecture.
- Take some time to gain the needed domain knowledge.

About Us



CONSULTING ENGINEERING SUPPORT

orcharhino

DEPLOY
RUN
CONTROL