International Journal of Applied Engineering Research
ISSN 0973-4562 Volume 10, Number 12 (2015) pp. 30833-30844

http://www.ripublication.com

Dynamic Dependency Version Control System

Adivandhya B. R. and Alind Jain

Department of Computer Science
Amrita School of Engineering, Bangalore, India
adivandhya@yahoo.co.in
Department of Electrical and Electronics,
Birla Institute of Technology and Science Pilani. Hyderabad, India
alindjain2@gmail.com

Abstract

The growth in Open Source technology and its widespread adoption has drawn attention to issues related to the compatibility and management of dependencies. Open Source software is different from traditional software as the core software that is to be installed has many external dependencies (independent of the project itself) that are downloaded and installed from external repository servers at the time of installation. There is an implicit problem with this method as the updated versions of these packages which are downloaded by default from the external repositories during installations may not be compatible with the previous versions of the package, hence causing installation failures. While much of the previous work has focused on ensuring the compatibility of dependencies during the packaging phase of the core software, there is insufficient emphasis on ensuring the compatibility of dependencies at the time of installation. In this paper, we propose a system which simplifies the management of core software dependencies, and provides repository level version control. The proposed system mitigates the installation failures of the core software and ensures stable updation to the dependencies through the algorithms proposed in the paper. Our contribution provides a unique method of automating the process of installation, regression, updation and versioning of the repositories.

Keywords— Software Packages, Dependency resolution, Version Control, Regression analysis, OpenStack

I. INTRODUCTION

The concept of open source has grown tremendously in its popularity and impact over the last decade. More and more enterprises are adopting open source technologies. It is helping them to keep their cost low, develop applications faster and provide better quality components. The biggest advantage of using open source technologies is the flexibility and control that comes with them. Since the code is made available publicly, it provides more control to the customers and developers [1]. This has also led to the creation of a huge contributor base and people are constantly contributing to improve the source code and introduce new features.

However, there are some downsides to open source. Since the contribution to a single project is highly diverse with many contributors, the scope for compatibility issues is highly substantial[2]. This could happen because of many reasons. In some cases, the dependency resolution for the core software is incorrectly listed or sometimes when a customer uses one open source technology on top of another open source technology, the compatibility between the two is lost due to upgrades to the dependencies. Compatibility has been a major issue since the inception of open source and still poses a serious problem to consumers. In this paper, a solution is devised to mitigate the problem of package version mismatch in a repository.

In this paper, we propose a Dynamic Dependency Version Control System (DDVCS) which simplifies the management of core software installations by providing an automated method for the version selection of the core software dependencies, and implementation of conditional updates to the existing local repository with version control. In our system, a stable local repository of the packages is maintained which contains curated, sanitized packages with repository level version control. Newer versions of the packages are added to the local repository only after performing regression tests on the installation of the target software which is dependent on these packages.

In this paper, we propose a Dynamic Dependency Version Control System (DDVCS) which simplifies the management of core software installations by providing an automated method for the version selection of the core software dependencies, and implementation of conditional updates to the existing local repository with version control. In our system, a stable local repository of the packages is maintained which contains curated, sanitized packages with repository level version control. Newer versions of the packages are added to the local repository only after performing regression tests on the installation of the target software which is dependent on these packages.

The first section of the paper introduces the topic and gives a brief idea of the domain area. The second section of the paper discusses previous work in this area. The third section of the paper provides an insight into the architecture of the system. In the fourth section, the system design has been explained in detail. Then, we perform a comparative study to illustrate the advantages of using the DDVCS system. In the final sections of the paper, we discuss the importance of the proposed system and the scope for future work in this area.

II. RELATED WORK

The concept of dependency version management system has been an emerging area of interest in the open source community. Prior work related to the proposed system has been discussed below.

Fabio Mancinelli, Jaap Boender, Roberto di Cosmo and Jer´ ome Vouillon in [3] establishes a framework for Managing the Complexity of Large Free and Open Source Package Based Software Distributions which can be used by a distribution editor to assess the quality of its distribution and to track problems concerning the underlying package repository. Contributors to the Apache Maven project [4] developed repository managers to speed up the builds by installing local cache from a central repository.

Jesse Ambrose, Thomas M. Rothwein and Klaus W. Strobel in [5] developed a suite of graphical interface to configure, modify and manage software modules. This tool represents graphically the relationship between modules and its dependencies. David B. Leblang, et al., in [6] proposes a Dynamic rule-based version control system to control versions of objects, by maintaining a storage device for storing plurality of versions of a set of objects. It also proposes a processor which evaluates version selection rules by performing one or more queries on each target object. Jonathan A.

Forbes, et al., in [7] invented a computerized method to manage the installation, execution, and uninstallation of software packages on a computer using a distribution unit containing components for a software package and a manifest file. Pietro Abate, et al., in [8] propose a modular package manager which uses modular architecture which is easily adaptable to new platforms and allows pluggable dependency solvers. Umar Manzoor and Samia Nefti in [9] developed a Silent Unattended Installation Package Manager that automates the process of silent unattended installations and un-installations and requires the minimal possible level of interaction with the user.

The above mentioned work have made significant progress in improving the way how software packages are handled and have made it easier for engineers to handle complex open source software installations. But the proposed system is unique in the way it automates the process whole of installation, regression and up gradation and versioning of the repositories which have been stabilized. In the next section of the paper, we describe the proposed system in more detail and illustrate how it could be used to ensure stability.

III. SYSTEM ARCHITECTURE

The DDVCS system Architecture has been illustrated in figure(1). The architecture has been described below.

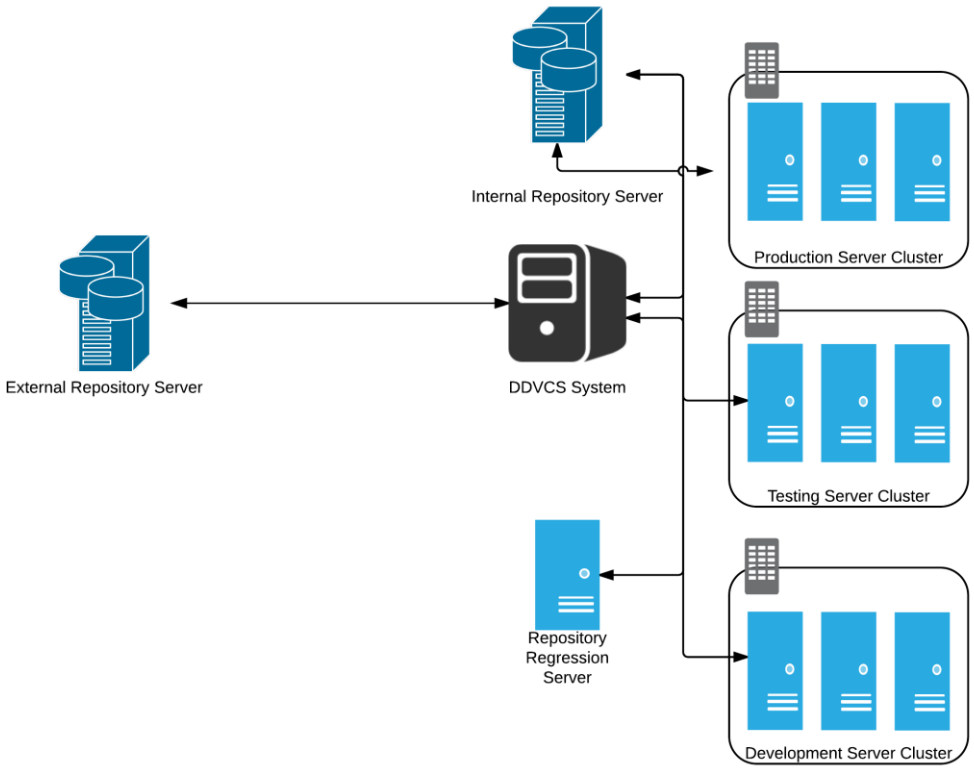

Fig. 1. System Architecure

A. ***External Repository Server***

The External Repository Server serves as the source for housing the software repositories and packages. This system is usually controlled by third-party vendors or the open-source communities. The native package management software on the user machines would refer to this repository server to download and install packages.

B. ***Internal Repository Server***

The Internal Repository Server serves as the storage unit to store version-controlled, tested and stable custom repositories. The Internal Repository Server is used by the DDVCS system as a storage system. The repositories in this system are exposed to the internal server clusters either as an http web service or as an ftp service.

C. ***Repository Regression Server***

The Repository Regression Server is used as a testing server that is used to validate the core software (which has dependencies) with the updated versions of the packages using regression tests. The Repository Regression Server works in conjunction with the DDVCS system to provide reports about the success or failure of the core software after using the upgrades in the packages.

D. *Internal Server Cluster*

This includes the Development Cluster, testing cluster and the production cluster. These clusters are a group of machines that are used by the developers, testers and the end users at different points in the project life-cycle.

The requirements of these clusters could be varied as different clusters would require different versions of the repository containing different versions of the packages. For example, the development cluster would want to refer to the repository containing the latest versions of the packages that have been qualified by the Repository Regression Server. On the other hand, the testing cluster would want to use the version of the repository that the development cluster had used prior to the code transfer to the test cluster. The production cluster would want to use the version of the repository that the testing cluster used as it would ensure security against any untested changes in the repository. The DDVCS system could accommodate all these requirements through its dynamic versioning system that ensures stability and provides more flexibility.

E. *DDVCS (Dynamic Dependency Version Control Central Server)*

The DDVCS server forms the core the system design. It runs the main programs needed to check for package updates on the External Repository Server, Orchestrate the Regression testing on the Repository regression server and maintain the versioning of the Internal Repository server. A more In-Depth discuss-ion of the DDVCS system would be done in the next section of the paper.

IV. DDVCS SYSTEM DESIGN

The DDVCS System Design has been illustrated in figure 2. The System design has been explained below.

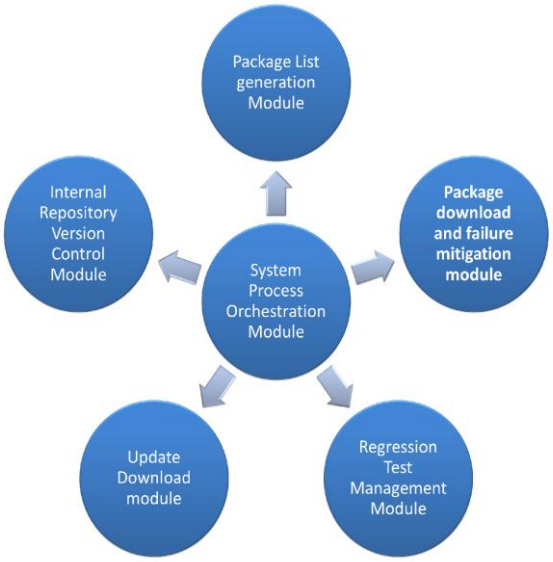

Fig. 2. System Design

A. ***Package List Generation Module***

This module is used to query the remote server for a list of packages that are installed on the remote system. To do so, the module communicates to the remote server using secure Shell (SSH) protocol. It then requests for the required list using the remote server's native package management system.

Once the list is obtained, the list is then moved to DDVS server for processing. After the list is moved to DDVCS server, this module processes the list for format errors that may have occurred due to the remote system's default settings. The package list generation steps are illustrated below.

1) Connect to remote server(s) with installed package list.
2) Send request to retrieve the package list and store the output in a file.
3) Transfer the file to the execution server (DDVCS server).
4) Process the file to correct format errors to prevent parsing errors in further stages of the algorithm.

B. ***Package Download and Failure Mitigation Module***

This module is used to parse the package list file and transform it into a consistent form and then iteratively communicate with the remote server to download the requested version of the packages. The downloaded packages are stored temporarily in the remote server. The module also mitigates download failures by capturing the packages that could not download in the first pass and retries the download using a hold time which increases with the number of retries.

The package download algorithm is illustrated below.

1) Read package-list file and parse it into the required format.
2) For package in the package-list do:
a) Send request to the download the package with explicit version specification from the external repository server.
b) Monitor package download, if download fails, add the package name and version to the failed_packages list.
3) Set retry_no = 0
4) If {(failed_packages = empty) or (retry_no = 3)}, Goto step 10
5) Increment retry_no by 1
*6) Set hold-off time = AvgICMPRespTime * retry_no*
7) For package in failed_packages do:
a) Sleep(hold_off time)
b) Send request to the download the package with explicit version specification from the external repository server to pre-defined path on the remote machine.
c) Monitor package download, if download succeeds, remove the package name and version from the failed_packages list.
8) Go to step 5
9) Copy the downloaded packages to the internal Repository server.
10) Create repository metadata index using the native package management tool on the platform.
11) Send the download report to System Process Orchestration Module.

Where, **AvgICMPRespTime** (Equation 1) is the average round trip time taken by an ICMP packet to be sent from the DDVCS server to the external repository server with acknowledgement from the external repository server. It is calculated as the average response time of n=10 ping requests to the external repository server.

$$AvgICMPRespTime = \frac{\sum_{i=1}^{n}(resp.time\ for\ ping_i)}{n}$$

Equation 1. Average ICMP Response Time Calculation

C. ***Package update inspector module***

This module continually monitors for updates to the packages present in the internal repository at user defined intervals. If an update is found in the packages, then the updated package list is obtained and then the System Orchestration Module is alerted.

The package update checking algorithm is illustrated below:
1) Connect to remote server(s) with installed packages list.
2) For package in package-list do:
a) Send request to external repository server to fetch the latest version of the package.
b) Compare the local version of the package with the latest version.
i) If local_version = latest_version,
Pass (the local package is already at the latest version, so nothing to do)
ii) If local_version < latest_version,
Add the package name with the latest version to the package_updates.
c) Send the package_updates list module to the System Process Orchestration Module.

D. ***Download Updates Module***

This module works in conjunction with the package update inspector module to download the packages which needs version updation. The downloaded packages are then merged with the existing packages in a separate isolated temporary repository which is then used in Regression Tests of the core software. This module leverages many features of the package download module and downloads the packages using the package_updates list and stores them in an ephemeral location before the merging operation begins.

The steps involved in this module are illustrated below:
1) Run the Package download algorithm using the package_updates list.
2) Transfer the downloaded packages to the internal repository server for further processing
3) Copy the latest version of the repository before checking for updates to the temporary folder.
4) Replace the older versions of the packages found in the package_updates list with their new versions.
5) Recreate repository metadata index with the updated repository in the temporary folder.

E. ***Regression Test Management Module***

This module controls the regression testing of the core software on the Repository regression server. Once, the regression testing is completed, and then the results of the regression tests (success/failure) are reported back to the DDVCS module.

F. ***Internal repository Versioning Control Module***

On being triggered by the System Process Orchestration Module, the new version creation process and the updation of the packages in the repository holding the latest packages is performed by this module. The new version creation process serves to store the updated state of the repository while the latest branch updation ensures that the machines referring to the latest local repository end- point will always get the latest stable updated version of the packages.

The structure of the versioning is illustrated below.

1) ***Numbered immutable repository versions:*** *These are the versions that carry immutable snapshots of the stable versions of the repository which are generated every time a new latest stable version is created.*

2) ***Latest Stable Version:*** *This version holds the latest stable packages in the repository.*

3) ***Temporary Version:*** *These are versions that are used during the testing phase of the process. These are deleted once the testing is completed.*

In the repository life cycle, when new updates are downloaded, the new repository generated from the updates is created as a Temporary version. Upon successful completion of the regression tests, the temporary repository is then tagged as the latest stable version and is moved into to the folder holding the latest repository. Also, a copy of this repository is held as a numbered immutable repository version to hold the current snapshot of the repository. These states are maintained by the Versioning Control Module.

G. ***System Process Orchestration Module***

This module controls the flow of tasks by monitoring the progress of various tasks, initiating the task flow based on the reports from the previous tasks and co-ordinates the transfer of files between various machines in the system. The system process orchestration module also serves as a single pane of window for the administration and tracking of the system. The work flow of the module is described below.

The Orchestration Module has two separate sub-work-flows which are followed based on the lifecycle phase of the core software. The two groups are illustrated below:

1) Initial Local repository creation from global repository

This is the preliminary phase of the core software installation and management. In this phase the local repository is created for the first time. In doing so, the DDVCS communicates with the system with the core software installed using the external repository.

Then the following steps are followed in order to complete the creation of the local repository.

a) Generate and retrieve package list.
b) Download the packages that are present in the list.
c) Create local repository on the internal repository Server.
2) Package update monitoring and version control

For monitoring updates to the packages and version control of the repositories, the following steps are followed:

a) Polling the External Repository for Updates using the Package update inspector module.
b) If update is found, initiate Download Updatesmodule.
c) Once the Update download is complete, trigger the Regression Test Management Module.
d) If regression Tests succeed, then proceed to Versioning of the newly constructed repository using the internal repository Versioning Control Module.

V. COMPARATIVE STUDY

In order to illustrate the results and elicit the advantages of this system, we consider the usage of DDVCS for supporting the installations of OpenStack deployment in data center and lab environments.

OpenStack is a free and open-source cloud computing software platform. OpenStack software controls large pools of compute, storage, and networking resources throughout a datacenter, managed through a dashboard or via the OpenStack API [10]. OpenStack works with popular enterprise and open source technologies making it ideal for heterogeneous infrastructure.

Being one of the world's largest open source projects, OpenStack has numerous dependencies most of which are from the open source community projects which are independent to OpenStack itself. However, being dependent on numerous external dependencies makes OpenStack susceptible to compatibility issues when the dependencies are updated.

In our study, we consider two OpenStack deployments, the first deployment runs on directly provided external repository packages, this setup is meant to be a baseline reference in our study. In the second setup, we consider an OpenStack deploy-ment with the DDVCS system to control and manage the versions of the dependencies.

Table 1 Package Updation and Incompatibilities

Criteria	Period (days)						
	0 – 10	11 – 20	21 – 30	31 – 40	41 – 50	51 – 60	Total
No. Package Updation	37	54	41	38	44	32	246
No. Incompatible Packages	9	0	11	0	8	10	38

The package update inspector module was configured to be triggered once every 10 days. The OpenStack setup is deployed on Redhat Linux 7 machines using the RHEL OSP Installer support. During the study, OpenStack was triggered for installation on both the setups periodically once every two days for a period of 60 days with the prior mentioned configurations. Hence during the period of study, a total of 30 reinstall triggers and 6 version upgrade checking triggers of repository were generated. The OpenStack software deployment had over 2330 external package dependencies.

A. ***Observations***

Over 60 day period 246 updates arrived across various packages that OpenStack was dependent on, giving an average rate of about 4 updates per day. Out of this 38 updates were found to be incompatible over 60 day period. The incompatibility was detected from the logs during the installation of OpenStack software.

The failures were also traced to the respective packages that were responsible for the installation failure. Table 1 shows the measurements of the number of packages updated on each 10 day interval and the number of incompatible updates among them. Further it was observed that whenever such a failure occurred, recovering the system to compatible updates was consuming on the average about 1 day. With the exception of the 31 -40 day interval, every other interval consisted of updation that was incompatible to OpenStack software.

But even considering the best case scenario that four incompatible updates that happened during each day the average time spent in recovery was 9 days in a period of 60 days, or about 15% productivity loss. However, considering a the worst case scenario, if the 38 incompatible updates had occurred on different days, it would have resulted losing 38 days in 60 days or >60% productivity loss! It is clear that there can be significant loss of productivity without addressing this problem without the use of our algorithm.

Also, during the period of study, no alterations were made to the Redhat systems and a constant installation and setup procedure was followed for consistency. In table 2, the number of installations on the respective systems is shown.

Table 2 No. Installation Failure on each system

Criteria	**Period (days)**						
	0 – 10	11 – 20	21 – 30	31 – 40	41 – 50	51 – 60	Total
No. of failures on system 1	2	0	2	0	1	1	6
No. of failures on system 2	0	0	0	0	0	0	0
V No.*	1	2	2	3	3	3	N/A

** V No. : The version number of the latest repository created by DDVCS.*

It can be observed that system 2 clearly outperforms system 1 in terms of the number of installation failures. Over a 60 day period, while system 1 could fail 38 times, system 2 on the other hand is not affected even a single time by these changes.

This is the direct outcome of the DDVCS system of testing and qualification of the updated packages before they are allowed to enter into the repository system.

The new version creation by the DDVCS system is shown as V No. in table 2. It can be observed the new version creation has a direct co-relation with the number of failed packages, i.e., the Regression tests on the Regression test server would fail if incompatibilities were present in the updated packages. This would result in the DDVCS server preventing new version creation due to the regression test failure. This can be observed in the V# data. The version creation happens only during the intervals where the no. of package incompatibility is 0.

The results of this study show that the DDVCS system has more stability and non-susceptible to external once the initial internal repository creation is completed as part of the initialization process.

VI. FUTURE SCOPE

The proposed system provides many features to address some of key pain point faced in the open stack community. However, there are a few regions in this area that could be improved. The proposed system is designed to manage the package dependencies of the core software. More work needs to done to accommodate the scenario in which the version of the core software is itself updated.

In this case, there are new challenges that are present. The new updated core software may have different set of dependencies from the previous versions. In this case, the package management system needs to be reinitialized to resolve the new dependencies and download them into the repository. Automating this process could be one of the future tasks in this system.

VII. CONCLUSION

Repository management systems have evolved a lot over the last decade. However, Dependency version management has not been managed in an automated way to accommodate incompatibilities to software in the open source domain which are often dependent on many packages and these dependencies are resolved at the time of installation of the package. In this paper, we have introduced the concept of Dependency version management system from the open source perspective.

Later, we have discussed the system architecture and design of the proposed system through which we have illustrated the functioning of the system and how it could be used to prevent susceptibility to changes in the dependencies. Also, the proposed system enhances the ability to enforce consistency in the versions of the packages in the repository that maybe in use by an organization.

Acknowledgment

The authors would like to thank Amrita Vishwa Vidyapeetham, Ettimadai, Birla Institute of Technology and Science, Pilani, and Cisco Systems for supporting this work.

References

[1] Fadi P. Deek, James A. M. McHugh, "Open Source: Technology and Policy" Cambridge University Press,2008, pp. 1-11.

[2] James W. Paulson, Giancarlo Succi, and, Armin Eberlein "An Empirical Study of Open-Source and Closed-Source Software Products", IEEE Transactions On Software Engineering, vol. 30, no. 4, April 2004.

[3] Fabio Mancinelli, Jaap Boender, Roberto di Cosmo, Jer´ ome Vouillon, "Managing the Complexity of Large Free and Open Source Package-Based Software Distributions", 21st IEEE International Conference on Automated Software Engineering 2006

[4] Apache Maven Project(2015). Available: http://maven.apache.org".

[5] Jesse Ambrose, Thomas M. Rothwein, Klaus W. Strobel, "Development tool, method, and system for client server applications", US Patent 6553563 B2, Apr 22, 2003

[6] David B. Leblang, Larry W. Allen, Robert P. Chase, Jr., Bryan P. Douros, David E. Jabs, Gordon D. McLean, Jr., Debra A. Minard, "Dynamic rule-based version control system", US Patent 5649200 A, Jul 15, 1997

[7] Jonathan A. Forbes, Jeremy D. Stone, Srivatsan Parthasarathy, Michael J. Toutonghi, Michael V. Sliger, "Software package management", US Patent 6381742 B2, Apr 30, 2002

[8] Pietro Abate, Roberto DiCosmo, Ralf Treinen, Stefano Zacchiroli, "MPM: A Modular Package Manager", Proceedings of the 14th international ACM Sigsoft symposium on Component based software engineering, 2011

[9] Umar Manzoor, Samia Nefti, "Silent Unattended Installation Package Manager – SUIPM", Computational Intelligence for Modelling Control & Automation, 2008 International Conference on, 2008

[10] WikiMedia(2015),OpenStack[Online].Available: http://en.wikipedia.org/wiki/OpenStack