Задачи и инструменты автоматизации рабочего места майнтейнера операционной системы Linux

Abstract — **The popularity of the Linux operating system is growing at an incredible rate in the modern world of high technology both on the enterprise side (servers, supercomputers, etc.) and the consumer side (mobile phones, desktops, etc.) in many ways due to the open architecture model and open source code of the basic system components. Reliability, quality, and, finally, the success of these systems are primarily due to the right choice of developers (maintainers) and necessary tools.**

The development of any Linux distribution is, in fact, mostly in the right combination and modification of existing open source components according to the goals of this distribution. In spite of the differences in the design of the user interface and internal architecture of Linux distributions, all of them follow three general goals: provision of the latest and most stable versions of programs, provision of the wider variety of programs for all user groups and provision of as much as possible of the unique software primary developed for this distribution.

In order to achieve these goals, Linux maintainers have to solve various tasks and problems of different complexity. As a simpler problem for the beginner one can bring a problem with creating of installation packages for new software in the distribution or fixing bugs in existing software components. As a more complex problem one can bring a problem with adaptation of newer versions of existent components in the system that is significantly complicated by many forward and backward dependencies of system components and problems with backward compatibility of interfaces. As another complex problem one can bring a problem with quality assurance of package repositories which can reach enormous size in the tens of thousands of packages. In order to solve such problems there is a need to develop various automated tools.

The average Linux distribution from the top ten distributions like Ubuntu or Fedora is developed by hundreds of maintainers. So, in addition, the integral part of the work of a Linux maintainer is the continuous communication with other maintainers of dependent system components. Also, to minimize the difference between the code of the component in the distribution and the source code of this component in upstream the maintainer should communicate with the upstream authors of this component.

The stability of the next versions of the Linux distribution depends on the quality of the solution found for the problems met during the development. The spend time for the stabilization and frequency of the releases of the distribution and, finally, relevance of the software versions for customers depend on the spent time to solve such problems.

These and many other problems of a Linux maintainer and necessary automated tools to quickly solve such problems are discussed in this paper.

Keywords — ***Development of operating systems; Maintainers; QA tools; Linux***

Аннотация — **В современном мире высоких технологий операционная система Linux набирает все большую популярность в качестве решения для серверных станций, настольных и мобильных компьютеров во многом благодаря открытости своей архитектуры и исходных кодов базовых компонентов. Надежность, качество и, в конечном счете, успех этих систем обусловлен, прежде всего, правильным выбором разработчиков (майнтейнеров) и необходимых инструментов.**

Разработка любого дистрибутива Linux состоит, по сути, в нужной комбинации и доработке, согласно поставленным целям, уже существующих открытых компонентов майнтейнерами операционной системы. Несмотря на существенные различия в интерфейсе пользователя и системной архитектуре, большинство дистрибутивов Linux придерживается трех основных целей: предоставление наиболее новых и стабильных версий программ, предоставление их наибольшего разнообразия для различных групп пользователей, а также предоставление пользователю как можно большего количества фирменных разработок в программном обеспечении, отсутствующих в других дистрибутивах.

Для достижения этих целей майнтейнерам приходится решать множество задач различной сложности. К простейшим задачам начального уровня можно отнести такие задачи как создание установочных пакетов для новых компонентов операционной системы или исправление ошибок в уже существующих компонентах. К более сложным задачам относится, например, задача по адаптации новых версий уже присутствующих компонентов в системе, которая усложняется наличием множества прямых и обратных связей между компонентами и проблемами обратной совместимости их интерфейсов. Также к сложным задачам можно отнести контроль качества пакетных репозиториев, которые могут достигать огромных размеров в несколько десятков тысяч пакетов. Такие задачи уже требуют разработки специальных автоматизированных инструментов для их решения.

Средний дистрибутив Linux из числа наиболее популярных на сегодняшний день, такой как Ubuntu или Fedora, разрабатывается сотнями майнтейнеров. Поэтому, в дополнение к уже описанным задачам, неотъемлемой частью работы майнтейнера является постоянная коммуникация с другими майнтейнерами для разрешения вопросов с зависимыми компонентами. Кроме того, для уменьшения расхождения в будущем между кодом компонента в дистрибутиве и его первоначальным исходным кодом (в апстриме), необходимо также поддерживать коммуникации с авторами этого компонента.

От качества решения задач, возникающих при разработке, зависит стабильность будущих версий дистрибутива, от скорости – время стабилизации и частота выпуска релизов дистрибутива и, как следствие, актуальность дистрибутива для пользователей.

Эти и другие задачи майнтейнера операционной системы Linux, а также инструменты для их качественного и быстрого решения являются предметом рассмотрения данной работы.

Ключевые слова — Разработка операционных систем; Майнтейнеры; Контроль качества; Linux

I. ВВЕДЕНИЕ

На сегодняшний день существует более трехсот различных модификаций дистрибутивов Linux согласно ресурсу DistroWatch [1]. При этом количество наиболее крупных и известных команд разработчиков дистрибутивов Linux насчитывает всего около десятка: RedHat, Debian, Ubuntu, Fedora, OpenSUSE, Mandriva, Gentoo и др. – разработчики систем для настольных компьютеров и серверов, и MeeGo, Maemo, LiMo и Tizen – для мобильных компьютеров.

Разработка любого современного дистрибутива Linux заключается, по большому счету, в правильном объединении и доработке уже существующих свободных компонентов. Эту работу выполняют инженеры сопровождения – *майнтейнеры* операционной системы. Разработчиками же самих компонентов системы являются различные группы энтузиастов по всему миру, а также как крупные, так и небольшие коммерческие компании.

В качестве примера компонентов операционной системы Linux можно привести базу данных Berkeley DB от крупной компании Oracle, графическую библиотеку Qt от компании Nokia и множество других системных библиотек, таких как libxml2, glib, GTK+ и др., разрабатываемые сообществом. Официальный сайт или репозиторий того или иного компонента принято называть *апстримом* этого компонента.

Несмотря на существенные различия в графическом интерфейсе пользователя и системной архитектуре современных дистрибутивов Linux, можно выделить несколько главных целей присущих всем дистрибутивам:

- Предоставление *наиболее свежих* и *стабильных* версий свободных программ,
- Предоставление *наибольшего количества* различных свободных программ,
- Предоставление *уникальных* программ собственной разработки.

Для достижения этих целей майнтейнерам операционной системы необходимо решать множество различных задач от мониторинга новых версий компонентов в апстриме до создания и тестирования бинарных установочных пакетов. Эти задачи и применяемые инструменты для их решения подробно описаны в пункте 2.

Главной целью статьи является описание наиболее сложных задач майнтейнеров и методов для их решения, таких как анализ качества пакетной базы или анализ обратной совместимости новых версий пакетов. При этом для сохранения целостности картины описаны также элементарные задачи, такие как создание пакетов и использование пакетных менеджеров. Подготовленный читатель может смело пропустить эти пункты. Второстепенной целью статьи является ознакомление читателя с основными терминами, используемыми майнтейнерами в повседневной работе. Также данная статья может быть использована для подготовки начинающих майнтейнеров.

Одним из основных принципов построения дистрибутивов Linux является использование общих системных компонентов (разделяемых библиотек и системных утилит) для функционирования множества других как прикладных, так и системных программ. Это значительно усложняет работу майнтейнеров по обновлению компонентов дистрибутива, так как при обновлении одного из компонентов должны быть также обновлены соответствующим образом и все зависимости этого компонента. Зависимости компонента могут быть двух типов: прямые и обратные. *Прямые зависимости* – это список компонентов необходимых для работы данного компонента. *Обратные зависимости* – список компонентов, которым для работы необходим данный компонент. Начинающие майнтейнеры часто забывают про анализ и обновление обратных зависимостей при обновлении какого-либо компонента, обновляя только прямые зависимости. Это приводит к нарушению процесса сборки или некорректной работе компонентов, зависящих от данного компонента.

Таким образом, майнтейнеру необходимо разбираться не только в своих компонентах, но и во всех зависимых компонентах и взаимодействовать с другими майнтейнерами для своевременного внесения в их компоненты нужных изменений.

Сложность сопровождения компонента операционной системы напрямую зависит от количества прямых и обратных зависимостей этого компонента, а также от поддержки разработчиками этого компонента *обратной совместимости* его интерфейсов. Обратной совместимостью называют специальное свойство компонента, при котором возможна замена старой версии компонента на новую версию без необходимости изменения других компонентов, использующих данный компонент.

Компоненты операционной системы можно подразделить на несколько классов:

- *Ядро*, имеющее наибольшее число обратных зависимостей;
- *Системные библиотеки*, основанные на ядре и других библиотеках;
- *Системные утилиты*, основанные на системных библиотеках и других утилитах;
- *Прикладные приложения*, основанные на системных библиотеках и утилитах;

Чем больше обратных зависимостей у компонента, тем серьезнее могут быть последствия появления ошибок и тем опытнее должен быть майнтейнер. Ядро имеет наибольшее число обратных зависимостей, а прикладные приложения – наименьшее. Ошибки в прикладных приложениях выведут из строя только эти приложения и не затронут другие, в отличие от ядра, ошибка в котором может привести к неработоспособности всей системы.

II. Задачи и инструменты майнтейнера

В процессе разработки дистрибутивов Mandriva 2011 и ROSA 2012, а также анализа деятельности майнтейнеров других дистрибутивов Linux, нами были выявлены наиболее общие задачи, стоящие перед любым майнтейнером. При этом инструменты для решения этих задач применяются в большинстве случаев различные.

Далее представлен список наиболее общих задач майнтейнера операционной системы Linux и применяемых инструментов для их решения.

A. Создание установочных пакетов

Любой современный дистрибутив Linux имеет модульную архитектуру и состоит из огромного множества компонентов. Каждый из компонентов имеет соответствующий установочный пакет. С помощью этого пакета пользователь системы может добавить этот компонент себе в систему или удалить его.

Первоначально компоненты системы распространяются их разработчиками в виде пакетов с исходными кодами, которые необходимо нужным образом модифицировать, чтобы сделать совместимыми с другими компонентами дистрибутива или изменить функциональность, следуя поставленным целям дистрибутива.

Разные дистрибутивы используют разные форматы установочных пакетов. Различают бинарные и *source-based* дистрибутивы Linux.

В бинарных дистрибутивах Linux для установки компонентов используют заранее собранные бинарные пакеты, что значительно увеличивает скорость их установки. Самыми распространенными видами бинарных пакетов на сегодняшний день являются RPM и Deb. Формат пакетов RPM используют такие дистрибутивы как RedHat, Fedora, Mandriva, OpenSUSE, MeeGo и др. Формат пакетов Deb используют такие дистрибутивы как Ubuntu, Debian, Maemo, LiMo, Tizen и др.

В source-based дистрибутивах Linux для установки компонентов используются пакеты с исходными кодами. При этом сборка исходных кодов происходит непосредственно во время установки пакета пользователем. Тем не менее, это не освобождает майнтейнера от предварительной сборки пакетов, так как необходимо удостовериться в успешности их сборки заранее. К source-based относятся такие дистрибутивы как Arch и Gentoo.

Также существуют дистрибутивы с комбинированным подходом к установке пакетов, такие как Sabayon. В этом дистрибутиве применяются как бинарные установочные пакеты и соответствующий пакетный менеджер Entropy, так и source-based пакеты и соответствующий пакетный менеджер Portage.

Сборка любого установочного пакета осуществляется на основе трех составляющих:

- архив с исходным кодом
- набор патчей
- *файл спецификации*

В различных типах дистрибутивов применяются различные типы файлов спецификации. В дистрибутивах, основанных на RPM-пакетах, применяются *spec-файлы* [2]. В дистрибутивах же, основанных на Deb-пакетах, в качестве файлов спецификации используются *control-файлы* [3]. Форматы spec-файлов и control-файлов существенно различаются, но при этом являются описанием примерно одной и той же информации об установочном пакете:

- Название и версия
- Список прямых зависимостей
- Группа
- Архитектура
- Краткое и полное описание
- Имя майнтейнера
- Адрес домашней страницы
- И др.

Spec-файлы также содержат инструкции по сборке и установке пакета. В Deb-пакетах для этого есть специально выделенный *rules-файл* [4].

Сборка бинарного установочного пакета осуществляется с помощью системной утилиты `rpmbuild` для RPM-пакетов и `dpkg-buildpackage` для Deb-пакетов. Тестовая сборка пакета может быть произведена на локальном компьютере, но окончательная сборка производится только в специальных сборочных системах.

Для сборки пакета сборочная система создает чистое виртуальное окружение операционной системы с помощью специальной системной утилиты `chroot` [5] или виртуальной машины и устанавливает в него минимальный набор компонентов, необходимый для сборки этого пакета и определяемый майнтейнером в файле спецификации. Если в файле спецификации не будет прописана некоторая зависимость, необходимая для сборки исходных кодов пакета, то сборочная система прекратит процесс сборки пакета и вернет сообщение об ошибке. В таком случае необходимо дополнить файл спецификации пакета и перезапустить процесс сборки.

После сборки пакета сборочная система производит анализ соответствия содержимого пакета списку требований, заранее определенному майнтейнерами для всех пакетов дистрибутива. К таким требованиям относится, например, наличие имени майнтейнера в свойствах пакета или ограничение длины описания пакета. В RPM-дистрибутивах для этого используют инструмент `rpmlint`, а в Deb-дистрибутивах – инструмент `lintian`.

Если пакет не удовлетворяет одному из требований, то майнтейнеру будет сообщено об ошибке, а пакет уничтожен. Поэтому предварительный анализ локально собранных пакетов с помощью этих инструментов является обязательным перед сборкой пакета в сборочной системе, чтобы лишний раз не нагружать сборочные серверы.

Сборочные системы также осуществляют автоматическую загрузку успешно собранных пакетов в *репозитории* дистрибутива, а также могут предоставлять удобный графический интерфейс для просмотра информации о пакетах. Также неотъемлемой частью любой сборочной системы является возможность распределенной сборки пакетов. Поскольку количество пакетов в репозиториях среднего дистрибутива на сегодняшний день составляет около десяти тысяч, то произвести пересборку этих пакетов за разумное время можно только при параллельной сборке пакетов на множестве серверов.

В разных дистрибутивах Linux используют разные системы сборки. Например, в Fedora используют сборочную среду Koji [6], основанную на программе для создания виртуального окружения Mock [7], которая в свою очередь основана на chroot. В дистрибутиве ROSA в качестве сборочной системы используется система ABF [8], а для создания виртуального окружения пакета используется полноценная виртуальная машина QEMU [9].

B. Пакетные менеджеры

Для установки готовых пакетов, а также для их удаления, как разработчики, так и конечные пользователи системы используют специальные инструменты – *пакетные менеджеры*. Различают два вида пакетных менеджеров – *низкоуровневые* и *высокоуровневые*. Высокоуровневые пакетные менеджеры можно также разделить по типу интерфейса пользователя на *консольные* и *графические* менеджеры.

Низкоуровневые пакетные менеджеры предназначены для безопасного выполнения элементарных операций с одиночными пакетами, таких как установка, удаление, вывод информации о пакете и др. При использовании таких пакетных менеджеров для установки пакета также необходимо самостоятельно найти в репозиториях и установить все зависимые пакеты. Для установки зависимых пакетов также надо установить все их зависимости и так далее. Для автоматизации этой и других задач используют высокоуровневые пакетные менеджеры.

Высокоуровневые пакетные менеджеры осуществляют поиск пакетов по необходимым параметрам в репозиториях дистрибутива и их автоматическую установку с установкой всех зависимых пакетов. На основе таких пакетных менеджеров обычно создают графические оболочки для использования конечными пользователями системы. Майнтейнеры же обычно используют консольные высокоуровневые менеджеры заодно с другими консольными утилитами.

В дистрибутивах, основанных на RPM-пакетах, используют низкоуровневый пакетный менеджер RPM Package Manager [10]. К высокоуровневым пакетным менеджерам в таких системах можно отнести Yum [11], используемый в RedHat, Fedora, CentOS, OpenSUSE и MeeGo, и Urpmi [12], используемый в Mandriva и ROSA. В качестве графических оболочек для пакетных менеджеров используют PackageKit [13] в RedHat, Fedora и CentOS, Rpmdrake [14] в Mandriva и ROSA, и YaST [15] в OpenSUSE.

В дистрибутивах, основанных на Deb-пакетах, используют низкоуровневый пакетный менеджер Dpkg [16]. К высокоуровневым пакетным менеджерам в системах такого типа можно отнести APT [17] в Ubuntu, Debian, Maemo и Tizen. В качестве графических оболочек для пакетных менеджеров в Ubuntu и Debian используют PackageKit.

C. Мониторинг новых версий компонентов

Количество компонентов ведущих дистрибутивов Linux достигает нескольких десятков тысяч. По этой причине количество компонентов, за которыми должен следить и которые должен обновлять каждый майнтейнер, может достигать нескольких десятков, а иногда и сотен. При этом разные компоненты имеют разные расписания выпуска новых версий, а иногда и не имеют их вовсе. Таким образом, майнтейнеру надо либо каждый день просматривать сотни домашних страниц обслуживаемых компонентов либо использовать автоматизированные системы.

К таким системам относятся специальные системы мониторинга пакетов исходного кода, оповещающие майнтейнера о появлении новой версии компонента с помощью электронной почты или с помощью web-интерфейса. Эти системы могут сэкономить некоторое время майнтейнера по поиску новых версий компонентов и приступить к более сложным задачам. Кроме того, чем своевременнее будет обнаружена новая версия компонента, тем скорее будут начаты работы по ее интеграции и тестированию и тем скорее она будет включена в новые версии дистрибутива.

В качестве примера системы мониторинга можно привести online-сервис Upstream Tracker [18]. Этот сервис предназначен для мониторинга появления новых версий наиболее распространенных системных библиотек, используемых в различных дистрибутивах Linux. Он осуществляет ежедневный обход сайтов и репозиториев производителей библиотек на появление новых версий пакетов исходного кода.

Также есть специализированные системы мониторинга новых версий для компонентов конкретного дистрибутива. Например, в Debian используется система DEHS [19] (Debian External Health Status). В Mandriva для этого используется Global Updates Report [20], а в ROSA – Updates Tracker [21].

Иногда такие системы могут производить некоторый вид дополнительного анализа для новых версий пакетов с исходными кодами и тем самым облегчить дальнейшую работу майнтейнера по обновлению компонента. Например, система Upstream Tracker производит визуализацию изменений в новых версиях компонентов, а также анализ обратной совместимости их публичных интерфейсов. Такой вид анализа позволяет уменьшить риск внесения случайных изменений в интерфейсы

компонентов, нарушающих сборку или функционирование зависимых компонентов. Результаты анализа полезны как для майнтейнеров, так и для разработчиков компонентов. Разработчики компонентов могут проанализировать совместимость beta-версий новых релизов своих компонентов и предотвратить нарушение обратной совместимости в финальном релизе. Майнтейнеры же могут проверить обратную совместимость новых версий компонентов перед началом интеграции их в систему.

Отдельным классом систем мониторинга являются системы сравнения пакетных баз различных дистрибутивов (часто конкурирующих). В таких системах сравнение версии компонента осуществляется не с апстримом, а с другим дистрибутивом. К таким системам можно отнести MGA vs MDV Packages Report [22] для сравнения пакетных баз дистрибутивов Mageia и Mandriva, а также ROSA Updates Tracker для сравнения пакетных баз дистрибутивов ROSA, Mandriva и Mageia.

D. Анализ изменений в пакетах

Майнтейнеры часто сталкиваются с необходимостью сравнения двух версий одного и того же компонента. Например, при появлении новой версии компонента в апстриме майнтейнеру необходимо оценить изменения, произошедшие в пакете с исходными кодами, чтобы в дальнейшем безопасно для системы обновить соответствующий установочный пакет. Также довольно часто возникает необходимость анализа изменений в готовых установочных пакетах. Например, при рецензировании главным майнтейнером новых версий установочных пакетов, созданных его подчиненными или другими майнтейнерами.

При анализе изменений наибольшее внимание надо уделять обратной совместимости компонента, чтобы новая версия этого компонента сохранила совместимость с другими компонентами системы. Иначе может быть нарушена сборка или функционирование других компонентов, зависящих от данного компонента. Если обнаружена несовместимость, то необходимо либо попытаться устранить ее, либо внести соответствующие изменения во все компоненты системы, зависящие от данного компонента, чтобы они могли быть собраны без ошибок и функционировали нормально. Обратная совместимость бывает двух типов:

- *Бинарная совместимость* – это возможность запуска всех компонентов системы, зависящих от данного компонента, без необходимости их перекомпиляции
- *Совместимость на уровне исходных кодов* – это возможность перекомпиляции без ошибок всех компонентов, зависящих от данного компонента

Любой компонент системы можно разбить на элементарные составляющие – *интерфейсы*. Интерфейс компонента может быть следующих видов:

- Исполняемые файлы (бинарные или интерпретируемые)
- Системные библиотеки (разделяемые или статические)
- Модули интерпретируемых языков
- Файлы конфигурации
- Заголовочные файлы
- И др.

Каждый из перечисленных видов интерфейсов может быть разбит на более мелкие интерфейсы. Например, в заголовочных файлах можно выделить такие элементарные интерфейсы, как функции или классы.

Интерфейсы компонента можно разделить на *внешние интерфейсы* и *внутренние интерфейсы*. Внешние интерфейсы предназначены для использования другими компонентами или пользователями системы. Внутренние интерфейсы являются элементами реализации внешних интерфейсов и не предназначены для использования извне компонента. Разделение интерфейсов на внешние и внутренние производит майнтейнер компонента на основе анализа документации. Наибольшее внимание майнтейнер должен уделять именно изменениям во внешних интерфейсах компонентах, так как они могут повлиять и на другие компоненты системы.

Каждый компонент в системе может иметь довольно большое количество внешних интерфейсов. При этом в других компонентах системы может быть использована только часть этих интерфейсов. Такие интерфейсы будем называть *активными интерфейсами*. В первую очередь надо проверять изменения именно в таких интерфейсах.

Изменения в интерфейсах можно подразделить на три типа: добавление, изменение и удаление элементов интерфейса (например, добавление файла или удаление функции из библиотеки). Удаление или изменение существующего интерфейса обычно является более опасным для системы, чем добавление нового интерфейса. Но бывают и исключения. Например, добавление такого нового интерфейса как виртуальная функция в класс на языке C++ внутри системной библиотеки может привести к падению использующих эту библиотеку компонентов.

Простейшим способом анализа изменений является ручная распаковка пакетов компонента и сравнение полученных директорий с помощью системной утилиты `diff`. Пакеты с исходными кодами обычно представляют собой архивы TAR.GZ, TAR.BZ2 и др., которые могут быть распакованы при помощи соответствующих архиваторов Tar, GZip, BZip и др. Для распаковки RPM пакетов необходимо преобразовать этот пакет в CPIO-архив при помощи системной утилиты `rpm2cpio` и затем распаковать этот архив соответствующим архиватором. Пакеты формата Deb являются архивами формата Ar и могут быть распакованы одноименным архиватором.

Компоненты системы могут иметь порой десятки тысяч интерфейсов. Для анализа изменений в таких компонентах необходимо использовать автоматизированные инструменты. Одним из таких инструментов является инструмент PkgDiff [23]. Этот инструмент принимает на вход два пакета любого формата и производит визуальное сравнение их содержимого с предварительной классификацией найденных интерфейсов. Известными

альтернативами этого инструмента являются инструменты debdiff [24], urpmdiff [25] и tardiff [26].

Для анализа изменений в наиболее критичных компонентах системы, таких как системные библиотеки, разработан специализированный инструмент ABI Compliance Checker [27]. Этот инструмент производит визуализацию изменений и детальный анализ обратной совместимости в системных библиотеках, создавая удобный HTML-отчет для майнтейнера. В этом отчете есть отдельные вкладки с результатами анализа бинарной совместимости и совместимости на уровне исходных кодов. Таким отчетом также могут пользоваться и разработчики библиотек для проектирования обратно совместимых API и ABI интерфейсов.

Дополнительной задачей майнтейнера является мониторинг изменений в компоненте на ранней стадии его разработки. Для этого надо еще до выпуска финальной версии компонента проверять все его новые beta-версии на совместимость с компонентами дистрибутива. Идеальным вариантом является ежедневная сборка отслеживаемого компонента из последней версии исходных кодов, взятых из публичной системы контроля версий этого компонента, и анализ его совместимости. При обнаружении несовместимости надо сообщить об этом разработчикам компонента. Возможно, эта несовместимость будет исправлена в финальной версии компонента, что позволит обновить его без проблем. Обычно одной из целей разработчиков системных компонентов является совместимость с наибольшим числом дистрибутивов и приложений, что заставляет их исправлять проблемы совместимости довольно оперативно. Однако иногда внесение несовместимых изменений является необходимым и преднамеренным действием со стороны разработчиков. В таком случае майнтейнер должен оповестить разработчиков приложений, использующих данный компонент, об этих изменениях с целью ускорения адаптации приложений на новую версию компонента.

Задача мониторинга изменений на ранней стадии разработки частично автоматизирована в системе Upstream Tracker. Эта система производит ежедневную сборку последней версии компонента из исходных кодов, взятых из его системы контроля версий, и анализирует обратную совместимость с предыдущей стабильной версией этого компонента. Результаты анализа публикуются на странице этого компонента в системе, которая доступна для просмотра как заинтересованными майнтейнерами, так и разработчиками этого компонента.

E. Определение зависимостей пакетов

Зависимостью компонента называют некоторый интерфейс или компонент необходимый для успешной сборки исходных кодов или запуска данного компонента. Зависимости необходимые для сборки компонента будем называть *сборочными*. Зависимости необходимые для установки и запуска уже собранного компонента будем называть *установочными*. Наиболее распространенными сборочными зависимостями являются компилятор GCC и заголовочные файлы библиотек. Примерами установочных зависимостей являются интерпретаторы Perl, Python, а также системные библиотеки.

Изначально зависимости между компонентами возникают во время разработки этих компонентов в апстриме из-за использования вспомогательных средств разработки. При создании установочного пакета для этого компонента эти зависимости должны быть определены майнтейнерами в соответствующем файле спецификации. На уровне пакетов зависимости делятся на *требуемые* пакетом и *предоставляемые* пакетом зависимости. При установке пакета пакетный менеджер ищет требуемые этим пакетом зависимости среди зависимостей, предоставляемых другими пакетами.

Определение как сборочных, так и установочных зависимостей является итеративным процессом. Сначала производится анализ исходных кодов компонента на предмет использования сторонних интерфейсов. Наличие программ конфигурации в исходных кодах компонента может значительно упростить эту задачу. Например, во многих Linux-компонентах используют набор средств GNU Autoconf [28] для создания программы конфигурации `configure` для автоматической проверки наличия необходимых для сборки компонентов в системе и создания сборочных файлов (make-файлов). Такая программа конфигурации генерируется на основе файла `configure.ac` в исходных кодах компонента, который может быть проанализирован майнтейнером для определения зависимостей. Затем создается первоначальный список зависимостей и оформляется в файле спецификации. Затем файл спецификации передается в сборочную систему для сборки пакета в чистом окружении. В случае отсутствия какого-либо компонента или интерфейса в списке зависимостей, сборочная система сообщит о соответствующих ошибках сборки. Этот компонент или интерфейс должен быть добавлен в список зависимостей в файле спецификации, а процесс сборки перезапущен.

Определение некоторых установочных зависимостей, таких как системные библиотеки и модули интерпретируемых языков, производится автоматически сборочной системой на основе статического анализа исполняемых файлов пакета – бинарных и интерпретируемых программ (скриптов). Анализ бинарных программ производится с помощью системной утилиты `ldd`, которая определяет список требуемых имен библиотек на бинарном уровне, называемых *soname*. Определение необходимых модулей в интерпретируемых программах производится поиском специальных инструкций для загрузки модулей в коде этих программ, таких как `use` в языке Perl или `import` в языке Python. Определение остальных зависимостей возлагается на майнтейнера компонента.

Если пакет содержит библиотеку, то сборочная система автоматически добавляет soname этой библиотеки в список предоставляемых этим пакетом зависимостей. При установке приложений осуществляется поиск и установка пакета с библиотекой предоставляющего нужный soname. При обновлении системной библиотеки майнтейнер должен следить за изменением soname этой библиотеки. Если в интерфейсах этой библиотеки произошли обратно несовместимые изменения, то soname должен быть изменен. Иначе при разрешении зависимостей для установки собранных ранее компонентов будет

установлен несовместимый пакет. Иногда авторы библиотек сами меняют soname при внесении несовместимых изменений в интерфейс библиотеки. Однако майнтейнер не должен полагаться на авторов библиотеки и всегда проверять внесенные изменения. Обычно soname библиотеки состоит из ее имени и версии бинарного интерфейса. Изменение soname заключается в изменении составляющей версии. Soname библиотеки записывается в процессе ее сборки с помощью дополнительной опции `-soname` для компоновщика.

В силу того, что определение большинства зависимостей возлагается на майнтейнера, имена зависимостей в различных дистрибутивах различаются. Например, зависимость от компилятора G++ в некоторых дистрибутивах называется `g++`, а в других `gcc-c++`. В результате этого пакеты в разных дистрибутивах обычно несовместимы между собой.

F. Контроль качества репозиториев

Обычно установка Linux-системы осуществляется с помощью специального CD или DVD носителя информации. При этом устанавливаются только самые необходимые для работы большинства пользователей пакеты. Остальные пакеты располагаются майнтейнерами в *репозиториях*.

Пакеты в репозиториях Linux-систем обычно делятся на несколько групп. Например, в Mandriva Linux есть четыре репозитория: main – официально поддерживаемые майнтейнерами пакеты, contrib – пакеты, собранные сообществом для дистрибутива, non-free – пакеты с несвободными лицензиями и restricted – пакеты, распространение которых запрещено в некоторых странах. Другие дистрибутивы придерживаются похожего разбиения пакетов в репозиториях.

Подключение репозиториев и установка необходимых пакетов осуществляется при помощи пакетных менеджеров. Однако не всегда установка пакета бывает успешной. Причиной этому может послужить, например, отсутствие в репозитории некоторых нужных зависимостей пакета или несовместимость некоторых пакетов. Такие проблемы могут возникать из-за несогласованной работы майнтейнеров различных пакетов, перемещения пакетов между репозиториями и по другим причинам.

Для того чтобы гарантировать успешную установку любого пакета из репозитория майнтейнеры должны регулярно проверять его качество. Главными критериями качества репозитория являются:

- Отсутствие *кольцевых зависимостей*, т.е. имеющих вид A->B->C->A
- *Замкнутость* по зависимостям, т.е. присутствие всех необходимых зависимостей для всех пакетов
- Отсутствие *конфликтов по файлам*, т.е. отсутствие одинаковых файлов в разных невзаимозаменяемых пакетах

Для выявления проблем репозиториев в большинстве дистрибутивов Linux есть специализированные инструменты. Например, в дистрибутивах Mandriva и ROSA есть набор инструментов Urpm-tools [29]. Для выявления кольцевых зависимостей в этих дистрибутивах применяется инструмент Urpm-repograph. Основным назначением данного инструмента является построение графа зависимостей репозитория, но он также может быть использован для выявления циклов в графе с помощью специальной опции `-loops`. Для проверки замкнутости репозиториев используется инструмент Urpm-reposclosure. Этот инструмент производит статический анализ зависимостей пакетов и выявляет отсутствие требуемых зависимостей одних пакетов среди предоставляемых зависимостей других пакетов. В дистрибутивах, основанных на пакетном менеджере Yum, таких как Fedora или OpenSUSE, для выявления проблем репозиториев применяется набор инструментов Yum-utils [30].

После исправления ошибок новые версии сломанных пакетов должны быть загружены в соответствующие репозитории, а инструменты контроля качества перезапущены на обновленных репозиториях.

G. Соблюдение стандартов

Дистрибутивы Linux состоят из огромного числа системных компонентов. При этом разные дистрибутивы могут быть основаны на разных версиях компонентов, включающих различающиеся наборы интерфейсов для приложений. В результате этого, приложения, созданные для одного дистрибутива, могут не работать на других дистрибутивах из-за различия в их интерфейсах.

Для улучшения совместимости дистрибутивов Linux и обеспечения переносимости прикладных программ создан стандарт LSB (Linux Standard Base) [31]. Этот стандарт опирается на такие проверенные временем стандарты как POSIX [32], SUS [33], SVID [34] и др., дополняя и расширяя их. Целью стандарта является определение базового набора необходимых интерфейсов системы и их свойств, который должен иметь каждый LSB-совместимый дистрибутив Linux. Если приложение использует для работы только этот набор интерфейсов, то оно является LSB-совместимым. Все LSB-совместимые приложения могут быть запущены на всех LSB-совместимых дистрибутивах Linux.

Стандарт LSB сопровожден необходимым набором инструментов для проверки дистрибутивов Linux и приложений на LSB-совместимость. Для проверки дистрибутивов Linux имеется инструмент LSB Distribution Checker [35], а для проверки приложений – LSB Application Checker [36]. Для навигации по стандарту создан портал LSB Navigator [37]. В нем можно просмотреть информацию обо всех библиотеках и интерфейсах, которые могут содержать LSB-совместимые приложения и должны содержать LSB-совместимые дистрибутивы.

Многие ведущие дистрибутивы Linux, такие как RedHat, Ubuntu, SUSE, Mandriva и др., уже получили сертификаты LSB-совместимых дистрибутивов. Чтобы улучшить совместимость разрабатываемого дистрибутива Linux с этими дистрибутивами и тем самым обеспечить

лучшую переносимость приложений, майнтейнерам рекомендуется также сертифицировать свой дистрибутив. Для этого нужно сначала убедиться в LSB-совместимости при помощи инструмента LSB Distribution Checker и исправить все выявленные проблемы совместимости, а затем подать заявку на сертификацию в Linux Foundation [38].

Во время работы над выпуском очередной версии дистрибутива Linux в системные компоненты может быть внесено множество изменений. В результате этих изменений LSB-совместимость может быть потеряна. По этой причине повторной сертификации подлежит каждая новая версия дистрибутива.

III. Дальнейшие исследования

Дальнейшая работа по автоматизации рабочего места майнтейнера дистрибутива Linux будет направлена главным образом на доработку таких описанных ранее инструментов как PkgDiff и ABI Compliance Checker, предназначенных для автоматического выявления несовместимых изменений в дистрибутивах, нарушающих работу системных компонентов и приложений. Это позволит быстрее обновлять системные компоненты и приложения и выпускать более качественные и стабильные версии дистрибутива.

IV. Заключение

В работе рассмотрены основные цели и задачи майнтейнеров дистрибутивов Linux и приведены примеры инструментов применяемых для решения этих задач в различных дистрибутивах. Затронуты как простейшие задачи, такие как создание пакетов, так и более сложные связанные с обновлением системных компонентов имеющих обратные зависимости и анализом изменений в них.

Примечания и сноски

[1] DistroWatch, http://distrowatch.com/

[2] RPM Spec-File, http://www.rpm.org/max-rpm/s1-rpm-build-creating-spec-file.html

[3] Debian Control File, http://www.debian.org/doc/debian-policy/ch-controlfields.html

[4] Debian Rules File, http://www.debian.org/doc/debian-policy/ch-source.html

[5] chroot, http://www.gnu.org/software/coreutils/manual/html_node/chroot-invocation.html

[6] Fedora Koji, http://fedoraproject.org/wiki/Koji

[7] Mock, http://fedoraproject.org/wiki/Projects/Mock

[8] ABF, https://abf.rosalinux.ru/

[9] QEMU, http://wiki.qemu.org/

[10] RPM Package Manager, http://rpm.org/

[11] Yum, http://yum.baseurl.org/

[12] Urpmi, http://wiki.mandriva.com/en/Tools/urpmi

[13] PackageKit, http://www.packagekit.org/

[14] Rpmdrake, http://en.wikipedia.org/wiki/Rpmdrake

[15] YaST, http://en.opensuse.org/Portal:YaST

[16] Debian package management system, http://www.debian.org/doc/manuals/debian-faq/ch-pkg_basics.en.html

[17] Advanced Package Tool, http://wiki.debian.org/Apt/

[18] Linux Upstream Tracker, http://upstream-tracker.org/

[19] Debian External Health Status, http://wiki.debian.org/DEHS

[20] Mandriva Updates Global Report, http://youri.zarb.org/demo/mandriva/updates.html

[21] ROSA Updates Tracker, http://upstream-tracker.org/updates/rosa/2012/

[22] MGA vs MDV Packages Report, http://mib.pianetalinux.org/mga-mdv.html

[23] PkgDiff, http://pkgdiff.github.com/pkgdiff/

[24] debdiff, http://manpages.ubuntu.com/manpages/precise/en/man1/debdiff.1.html

[25] urpmdiff, http://stuff.onse.fi/man?program=urpmdiff

[26] tardiff, http://tardiff.coolprojects.org/

[27] ABI Compliance Checker, http://ispras.linuxbase.org/index.php/ABI_compliance_checker

[28] GNU Autoconf, http://www.gnu.org/software/autoconf/

[29] Urpm-tools, http://wiki.rosalab.ru/en/index.php/Urpm-tools

[30] Yum-utils, http://yum.baseurl.org/wiki/YumUtils

[31] Linux Standard Bse, http://linuxbase.org

[32] Portable Operating System Interface for Unix, http://pubs.opengroup.org/onlinepubs/9699919799/

[33] Single UNIX Specification, http://www.unix.org/what_is_unix/single_unix_specification.html

[34] System V Interface Definition, http://www.sco.com/developers/devspecs/

[35] Linux Distribution Checker, http://ispras.linuxbase.org/index.php/LSB_Distribution_Checker_Getting_Started

[36] Linux Application Checker, http://www.linuxfoundation.org/collaborate/workgroups/lsb/all-about-linux-application-checker

[37] LSB Navigator, http://www.linuxbase.org/navigator/

[38] LSB Certification Management System, https://www.linuxbase.org/lsb-cert/status.php