Diplomarbeit

TECHNISCHE UNIVERSITÄT CHEMNITZ

TECHNISCHE UNIVERSITÄT CHEMNITZ

Fakultät für Informatik

Professur Rechnernetze und verteilte Systeme

Diplomarbeit

Qualitätssicherungs- und Build-System für das Fedora Projekt

Enrico Scholz

Chemnitz, den 11. März 2005

Betreuer: Prof. Dr. U. Hübner

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

Typische Software für GNU/Linux wird auf zwei Arten zur Verfügung gestellt: als Quellpaket vom Entwickler oder vorgefertigt von Distributoren. Quellpakete sind häufig allgemein gehalten und bedürfen manueller Übersetzung unter Beachtung verschiedenster Abhängigkeiten, der Erstellung von Konfigurationsdaten, der Einrichtung von Nutzern und anderen Anpassungen an das lokale System. Ebenfalls muß die Aktualisierung durch neuere Versionen und die vollständige Deinstallation manuell gelöst werden. Je nach Komplexität der Software können diese Schritte ein ressourcenhungriger Prozeß sein, der detaillierte Kenntnisse und hohen Zeitaufwand erfordert.

Distributoren, das heißt Organisationen, Unternehmen oder auch Einzelpersonen, verallgemeinern diesen Prozeß und führen die Schritte ganz oder teilweise durch, so daß der Endanwender ein fertiges Paket mit minimalem Aufwand installieren, konfigurieren, aktualisieren und entfernen kann. Damit verbunden sind Mechanismen zur Verteilung der Software.

Durch diese Verallgemeinerung entsteht auf Seiten des Distributors zusätzliche Komplexität, so daß die Wahrscheinlichkeit für Fehler im Verpackungsprozeß steigt und zusätzliche Zeitressourcen benötigt werden.

1.1. Distributionskonzepte

Im Laufe der Zeit wurden einige Versuche unternommen, obige Aufgabe zu lösen. Nachfolgend sollen die für dieses Projekt wichtigsten Konzepte dargestellt und analysiert werden.

Red Hat Linux (RHL) und Red Hat Enterprise Linux (RHEL) sind Basisdistributionen, deren Pakete ausschließlich durch geschulte Red Hat Mitarbeiter erstellt werden. Es finden beziehungsweise fanden (für RHL) ausgiebige Testphasen zur Qualitätssicherung (QA) statt. Ebenfalls werden Ressourcen zur langfristigen Aktualisierung bei Sicherheitslücken eingesetzt. Dieser Ressourcenverbrauch führte dazu, daß die kostenlose RHL Distribution eingestellt und durch Fedora Core (siehe unten) ersetzt wurde.

Aufgrund der begrenzten Anzahl an Red Hat Mitarbeitern war bzw. ist – wie in Abschnitt A.1 ersichtlich – die Anzahl der Pakete begrenzt.

Fedora Core ist ebenfalls eine Basisdistribution und der Nachfolger von Red Hat Linux. Hauptunterschied ist das Einsparen von Ressourcen durch eine wesentliche Verringerung der Lebensdauer auf 6–9 Monate. Ansonsten erfolgt die Paketentwicklung weiterhin durch Red Hat Mitarbeiter und die Paketanzahl bleibt begrenzt.

Red Hat Powertools war ein Zusatz zum damaligen Red Hat Linux und wurde nach RHL 7.2 eingestellt. Die Betreuung erfolgte durch Red Hat Mitarbeiter, allerdings mit geringerem Aufwand für die Qualitätssicherung.

Red Hat Contrib war ein unorganisierter Bereich auf dem Red Hat Server, wo beliebige Software im RPM Format sowohl als Quell- als auch Binärpaket bereitgestellt werden konnte. Es fand keinerlei organisierte Qualitätssicherung statt, so daß die wenigsten Pakete wirklich brauchbar waren. Ebenfalls war keine Kontrolle über den Inhalt möglich, was insbesondere bezüglich *Digital Millennium Copyright Act* (DMCA), Patent-, Lizenz- und Copyrightverletzungen die Gefahr kostspieliger Klagen birgt.

Es war unklar, unter welchen Umgebungen die Pakete gebaut wurden, so daß unerfüllbare Abhängigkeiten auftreten konnten oder das Quellpaket auf anderen System nicht übersetzbar war. Es konnte ebenfalls nicht garantiert werden, daß die Binärpakete vom jeweils bereitgestellten Quellpaket abstammen.

Ein weiteres Problem eröffnet sich in Fragen der Sicherheit: die Paketautoren blieben anonym und könnten Schadcode in das Paket einfügen. Der Endnutzer müßte deshalb das Paket selbst überprüfen, was zum einen der Zielsetzung von Softwarepaketen widerspricht, als auch bei Binärpaketen nahezu unmöglich ist.

rhcontrib@bero.org war Mitte 2001 eine Initiative des damaligen Red Hat Mitarbeiters Bernhard Rosenkränzer, um einige Defizite von Red Hat Contrib zu beheben. So wurden ausschließlich Quellpakete angenommen und dann vom System zu Binärpaketen gebaut. Dies garantierte, daß das Paket unter einer definierten Umgebung erzeugbar ist und die Binärpakete wirklich vom bereitgestellten Quellpaket abstammen.

So wurde auch das unbemerkte Einbringen von Schadcode durch den Paketautor etwas erschwert, da dies in den Quellpaketen nachvollziehbar ist.

Allerdings blieb auch hier der Autor anonym, und es existierten Defizite in der Infrastruktur, wie zum Beispiel Bau der Pakete in leicht brechbaren chroot-Umgebungen [23, Seite 14f] oder ungenügende Plattenkapazitäten.

Nachdem Bernhard Rosenkränzer Red Hat verließ, wurde dieses Projekt Anfang 2002 abgebrochen.

fedora.us wurde Ende 2002 von Warren Togami mit dem Ziel ins Leben gerufen, ein hochqualitatives Repository aufzubauen, welches für jedermann offen ist. Hauptaugenmerk lag auf einer strikten Qualitätskontrolle, die gewährleisten sollte, daß vom Paketautor kein bösartiger Code eingebracht werden kann. Der Hauptunterschied zu den vorherigen Konzepten bestand darin, daß diese Kontrolle vor dem Veröffentlichen der Pakete stattfindet.

Die Anonymität der beteiligten Personen wurde reduziert, indem kryptographische Signaturen für das Bereitstellen der Quellpakete oder für Testberichte benötigt wurden.

Zum Paketbau wurden virtuelle Server[1] eingesetzt, welche resistent gegen chroot- und andere Attacken sind.

Allerdings war zum Betrieb des Systems, d.h. dem Interpretieren von Testberichten, dem Überprüfen von Signaturen, dem Bau und Signieren der Binärpakete, sowie dem Annoncieren neuer Pakete viel manuelle Arbeit nötig, was für alle Beteiligten einen hohen Zeitaufwand bedeutete. Außerdem gelang es nicht, genügend Tester zu gewinnen, so daß viele bereitgestellte Pakete nie oder nur nach einer langen Zeit ins Repository aufgenommen wurden. Versuche, durch Ausnahmeregeln oder besondere (manuelle) Verfahren die Durchlaufzeiten zu verringern, führten zu einer Komplizierung und wurden oft nicht akzeptiert, so daß teilweise der gegenteilige Effekt eintrat.

Aus diesen Gründen fand im September 2003 der Zusammenschluß mit Red Hat zum Fedora Projekt statt, in der Hoffnung, von fedora.us Erreichtes mit Hilfe der Ressourcen von Red Hat fortsetzen und verbessern zu können.

1.2. Zielsetzung

Das Ziel dieser Arbeit ist die Vereinfachung und Automatisierung von Abläufen, welche zur Erzeugung von Software für das Fedora Projekt führen. Das Resultat sollen qualitativ hochwertige Pakete sein, die nach kurzer Zeit für den Nutzer verfügbar sind. Ausgehend von den bei fedora.us gemachten Erfahrungen soll ein System mit folgenden Eigenschaften entworfen werden:

- Die Schritte – beginnend beim Einbringen neuer Pakete, über den Paketbau, möglichen Qualitätssicherungsmaßnahmen bis hin zum finalen Veröffentlichen – laufen weitestgehend automatisiert ab.
- Der Verursacher einer Aktion muß sicher identifiziert und seine Autorisierung überprüft werden. Da hier Software vollständig automatisiert gebaut, veröffentlicht und weltweit auf tausenden von Rechnern installiert wird, existiert eine große Motivation für Angreifer, diese Überprüfung zu umgehen. Deshalb ist dieser Punkt sehr paranoid umzusetzen.
- Für die manuelle Qualitätskontrolle sind später noch genauer zu bestimmende Hilfsmittel anzubieten; für automatisierte Tests sind Schnittstellen zu schaffen.
- Die Qualitätssicherungsmaßnahmen finden vor dem Veröffentlichen statt
- Das System muß ausschließlich durch Software des Fedora Projekts realisierbar sein. Dies schließt unfreie Software aus.

[1] `http://linux-vserver.org`

2. Qualitätssicherungsmaßnahmen

Nachfolgend soll untersucht werden, was alles einer Qualitätssicherung unterworfen werden soll und welche Maßnahmen sinnvoll und durchführbar sind. Dieses Kapitel gibt außerdem einen Überblick über die Erstellung von RPM-Paketen.

Grundsätzlich ist bei dieser Betrachtung zwischen der verpackten Software und den Paketdaten zu differenzieren. Beide unterscheiden sich sehr stark in ihrer Komplexität, so daß verschiedene Methoden der Qualitätssicherung angewendet werden müssen. Der Begriff „Komplexität“ beinhaltet in diesem Zusammenhang die Anzahl der Codezeilen, die verwendete Programmiersprache, die genutzten Sprachmittel, implementierte Algorithmen u.ä..

2.1. Software

Die zu verpackende Software stammt von unabhängigen Autoren, besitzt meist eine hohe Komplexität und ist deshalb nicht vollständig bewertbar; das heißt, nicht jede Programmzeile kann auf Fehler hin untersucht werden. Teilweise können vom Autor bereitgestellte Testverfahren angewendet werden, wie zum Beispiel das Ausführen von „`make check`“ beim Bau der Pakete. Diese Verfahren testen aber selten den gesamten Funktionsumfang und insbesondere bei Programmen mit graphischer Nutzeroberfläche fehlen sie gänzlich.

Deswegen können durch vorliegendes System die in [14] genannten Qualitätsmerkmale „Funktionalität“, „Zuverlässigkeit“, „Benutzbarkeit“, „Effizienz“ und „Änderbarkeit“ nicht durch technische Maßnahmen überprüft werden. Stattdessen muß sich sowohl auf die Qualitätssicherung beim Programmautor verlassen werden, als auch manuelle Tests durchgeführt werden.

Die Einhaltung rechtlicher Rahmenbedingungen wie patentrechtliche Verträglichkeit oder des *Digital Millennium Copyright Act* (DMCA) kann ebenfalls nur manuell überprüft werden.

Einzig Teilaspekte der „Übertragbarkeit“ lassen sich technisch verifizieren, indem die Software in unterschiedlichen Umgebungen gebaut wird.

2.2. Paketdaten

Vorliegendes System erlaubt die Bereitstellung der Paketdaten auf zwei Arten: entweder in Form einer SRPM Datei (Endung `.src.rpm`) oder als einzelne Dateien, die sich mittels „`rpmbuild -bs ...`“ in eine SRPM Datei umwandeln lassen. Nachfolgende Betrachtungen

werden sich am `qiv`-Paket (vergleiche [25]) orientieren, dessen Inhalt in den Abbildungen 2.1, 2.2 und 2.3 dargestellt ist.

```
$ rpm -qlpv qiv-2.0-0.fdr.1.2.src.rpm
-rw-r--r--  1 machbuilmachbuil    76561 Mai 22  2004 qiv-2.0-src.tgz
-rw-r--r--  1 machbuilmachbuil      472 Jun 12  2004 qiv-optflags.patch
-rw-r--r--  1 machbuilmachbuil     1353 Jun 28  2004 qiv.spec
```

Abbildung 2.1.: Inhalt einer SRPM Datei

Die verpackte Software steht typischerweise in Form eines komprimierten Tar-Archivs (hier: `qiv-2.0-src.tgz`) bereit. Zur Qualitätssicherung sollte die Übereinstimmung dieses Archivs mit dem vom Softwareautor bereitgestellten geprüft werden. So wird garantiert, daß der Paketautor keinen bösartigen Code auf diese Weise einbringt.

Die Überprüfung ansich kann durch zeichenweisen Vergleich der Pakete mittels geeigneter Programme (zum Beispiel `cmp` oder `diff`), durch Vergleich der Prüfsummen oder mittels kryptographischer Signatur erfolgen. Welches Verfahren angewendet wird, hängt von den vom Softwareautor bereitgestellten Daten ab; bevorzugt werden, sollte die kryptographische Signatur.

Obwohl der Vergleich automatisch durchgeführt werden könnte, indem die Quelle der Vergleichsdaten aus der `.spec` Datei extrahiert, von dort das tar-Paket heruntergeladen und mit dem vorliegenden verglichen wird, sollte dieser Test manuell erfolgen. Ansonsten besteht die Gefahr, durch geschickt konstruierte URLs ein speziell präpariertes Pakete für den Vergleich zu nutzen.

Änderungen seitens des Paketautors können durch wie in Abbildung 2.2 dargestellte Patche oder als eigenständige Datei erfolgen. Ziel ist das Vermeiden von Unregelmäßigkeiten beim Kompilieren der Software, das kurzfristige Beheben von Fehlern oder Anpassungen an die Zielumgebung.

Der in diesen Änderungen enthaltene Code wird unter Umständen mit Administratorrechten auf dem Endnutzersystem ausgeführt und sollte deshalb gründlich nach Sicherheitslücken oder bösartigem Code durchsucht werden. Anders als die zu verpackende Software, besitzen die Änderungen aber nur einen geringen Umfang und können deshalb vollständig begutachtet werden.

Die Spezifikationsdatei enthält Regeln zum Bau des Paketes und ist in [1, Teil 2] beschrieben. Grundsätzlich sind in einer `.spec` Datei die unten angegebenen Daten enthalten (vergleiche Abbildung 2.3). Abschnitt A.2 stellt einige Zahlen über ihren durchschnittlich zu erwartenden Umfang bereit.

- Informative Daten wie eine kurze und eine ausführliche Beschreibungen der Paketfunktionalität, die Einordnung in eine Paketgruppe, Angaben über die Lizenzierung,

```
--- Makefile.orig       2004-05-22 10:21:47.000000000 +0300
+++ Makefile    2004-06-12 10:37:51.086331940 +0300
@@ -58,7 +58,7 @@
 ######################################################################

 CC        = gcc
-CFLAGS    = -O2 -Wall -fomit-frame-pointer -finline-functions \
+CFLAGS    = $(RPM_OPT_FLAGS) -Wall -fomit-frame-pointer -finline-functions \
             -fcaller-saves -ffast-math -fno-strength-reduce \
             -fthread-jumps #-march=pentium #-DSTAT_MACROS_BROKEN
```

Abbildung 2.2.: `qiv-optflags.patch` Datei

die Homepage der verpackten Software und die Historie der Änderungen (*changelog*). Typische QA-Maßnahmen sind die Überprüfung der Rechtschreibung und des Einhaltens von Regeln wie Groß- oder Kleinschreibung des ersten Wortes in der Zusammenfassung, Abschluß der Zusammenfassung mit einem Punkt, Angabe der jeweiligen Versionsnummer bei Änderungseinträgen oder die ausschließliche Verwendung vordefinierter Paketgruppen[1].

Viele dieser Tests lassen sich automatisieren, aber zumindest Rechtschreib- und Plausibilitätsprüfungen sollten manuell erfolgen.

- Versionsinformationen, welche die Version der verpackten Software („`Version:`“) und die Version der `.spec` Datei („`Release:`“) beinhalten. Weiterhin existiert ein `Epoch:` Feld, welches aber nur dazu dienen sollte, vorherige Versionierungsfehler zu korrigieren.

  Die Korrektheit dieser drei Felder ist von hoher Bedeutung, da beim automatischen Paketupdate ältere Pakete von neueren eindeutig unterschieden werden müssen.

  Insbesondere bei alphanumerischen Versionsnummern wie `1.0a` ist es ohne Kontext nicht klar, ob diese Version älter als `1.0` ist (zum Beispiel, wenn das `a` für „alpha“ steht) oder den nächsten Entwicklungsschritt kennzeichnet. `rpm` sieht leider keine Möglichkeit vor, den erstgenannten Fall allein mittels des dafür vorgesehenen `Version:` Feldes zu lösen. Stattdessen wird das `Release:` Feld dafür mißbraucht, was Regeln wie [9, Abschnitt C-3] notwendig macht.

  Grobe Fehler in den Versionsinformationen können teilweise durch automatische Tests entdeckt werden; zur Überprüfung der Korrektheit ist jedoch eine manuelle Begutachtung nötig.

- Auflistung der Abhängigkeiten für den Paketbau („`BuildRequires:`“) in Form von Paket- oder Dateinamen. Durch Angabe dieser Abhängigkeiten soll der Buildprozeß

[1] siehe `/usr/share/doc/rpm-*/GROUPS`

```
Name:           qiv
Version:        2.0
Release:        0.fdr.1.1
Epoch:          0
Summary:        Quick Image Viewer

Group:          Applications/Multimedia
License:        GPL
URL:            http://www.klografx.net/qiv/
Source0:        http://www.klografx.net/qiv/download/qiv-2.0-src.tgz
Patch0:         %{name}-optflags.patch
BuildRoot:      %{_tmppath}/%{name}-%{version}-%{release}-root-%(%{__id_u} -n)

BuildRequires:  gtk+-devel, imlib-devel

%description
qiv is a very small and pretty fast gdk/Imlib image viewer.


%prep
%setup -q -n %{name}-%{version}
%patch0 -p0


%build
make %{?_smp_mflags}


%install
rm -rf $RPM_BUILD_ROOT
install -Dpm 755 qiv $RPM_BUILD_ROOT%{_bindir}/qiv
install -Dpm 644 qiv.1 $RPM_BUILD_ROOT%{_mandir}/man1/qiv.1


%clean
rm -rf $RPM_BUILD_ROOT


%files
%defattr(-,root,root,-)
%doc README README.CHANGES README.COPYING README.TODO qiv-command.example
%{_bindir}/qiv
%{_mandir}/man1/qiv.1*


%changelog
* Sat Jun 12 2004 Ville Skyttä <ville.skytta at iki.fi> - 0:2.0-0.fdr.1
- Update to 2.0.

* Tue Jan 27 2004 Ville Skyttä <ville.skytta at iki.fi> - 0:1.9-0.fdr.1
- Update to 1.9.

...
```

Abbildung 2.3.: `qiv.spec` Datei

und damit die Ergebnisse reproduzierbar sein. Die angegebene Abhängigkeitenmenge sollte so beschaffen sein, daß sich ausschließlich durch Wegfall beliebiger Abhängigkeiten das Ergebnis ändert. Das heißt, daß

$$f(A) \text{ definiert} \tag{2.1}$$

$$\forall X \subset A: \quad f(X) \neq f(A) \tag{2.2}$$

$$\forall Y \subseteq P: \quad f(A \cup Y) = f(A) \tag{2.3}$$

gelten soll. A kennzeichnet hierbei die durch die Abhängigkeiten beschriebene Paketmenge, P die Menge sämtlicher verfügbarer Pakete und $f(X)$ das Resultat der Paketbaue mit der installierten Paketmenge X. Die Relation $f(X) = f(Y)$ ist genau dann erfüllt, wenn beide Operationen identische Pakete erzeugen, d.h., die selben Dateien und Installationsskripte entstehen.

Da der Paketbau $f(X)$ ein zeitintensiver Prozeß ist und $|P| > 1000$ gilt, kann Gleichung 2.3 praktisch nicht überprüft werden. Relation 2.2 ist bei größeren Paketen und Abhängigkeitsmengen ebenfalls nicht überprüfbar. Zum Beispiel benötigt kdelibs-3.3.2 mit $|A| = 30$ ca. $1.5h$ für $f(A)$ auf einem Pentium 4 2.6 GHz.

Stattdessen kann $f(A) = f(P)$ getestet werden, das heißt, ob das Paket mit den angegebenen Abhängigkeiten überhaupt baubar ist und ob zusätzliche Pakete das Ergebnis ändern würden. Dieser Test kann automatisch erfolgen; überflüssige Abhängigkeiten müssen durch manuelle Begutachtung entdeckt werden.

- Übersetzungs- und Installationsanweisungen sind (meist) Shellskripte, in denen gegebenenfalls rpm-spezifische Makros expandiert werden. Im einfachsten Fall wird nur die bei GNU Paketen übliche „`./configure && make && make install`“ Sequenz mit angepaßten Pfaden durchgeführt; bei komplizierteren Paketen sind aber auch umfangreiche Skripte möglich. So besitzt beispielsweise das glibc-2.3.4-3 Paket 550 Codezeilen in den `%build` und `%install` Abschnitten. Typisch sind jedoch < 50 Zeilen, wie Abschnitt A.2 zeigt.

  Aus Sicherheitsgründen müssen diese Skripte als normaler Nutzer und nicht nur als Systemadministrator durchführbar sein, was leicht überprüfbar ist, indem das Paket als normaler Nutzer gebaut wird.

  Hauptaugenmerk bei der Qualitätssicherung liegt auf dem Verwenden allgemeiner Makros anstatt fester Pfade, so daß das Paket leichter in andere Umgebungen übertragbar ist. Weiterhin sollten eventuelle Änderungszeiten (*Timestamps*) der Dateien erhalten bleiben, die Übersetzung auf Mehrprozessormaschinen möglichst parallel erfolgen, die `%build` und `%install` Abschnitte mehrfach hintereinander ausführbar sein (sogenannte *short-circuit-builds*), sowie die Skripte gute Wartungseigenschaften aufweisen.

  Bis auf das Testen erfolgreicher *short-circuit-builds* und das Durchsuchen nach festen Pfaden sind obige Punkte nur manuell verifizierbar. Aufgrund des relativ geringen Um-

fanges der Skripte ist diese Qualitätssicherung vollständig durch Begutachtung jeder einzelnen Zeile möglich.

- Auflistung der zu verpackenden Dateien und Verzeichnisse in „`%files`“ Abschnitten. Dateien können mit `%config` so markiert werden, daß lokale Änderungen beim Paketupdate erhalten bleiben. Außerdem kann der Eigentümer und die Dateirechte mittels „`%attr`“ festgelegt, und Dokumentationsdateien gekennzeichnet werden.

  Wie in [21] dargestellt, ist darauf zu achten, daß zwar alle neu hinzugefügten, aber keine allgemeinen Verzeichnisse wie `/usr` registriert werden. Um das Paket leichter in andere Umgebungen übertragbar zu machen bzw. um die Spezifikationsdatei nahezu unverändert mit neuere Softwareversionen zu verwenden, sollten die Dateien möglichst allgemein mittels Platzhaltern (*wildcards*) angegeben werden.

  Fehler in der Datei-Attributierung werden teilweise automatisch von `rpmlint`[2] erkannt, und fehlende bzw. überflüssige Verzeichnisse werden durch Installationstests sichtbar. Manuelle Qualitätssicherung braucht nur zur Überprüfung der Wartbarkeit und der korrekten Zuordnung bei mehreren Unterpaketen erfolgen.

- Explizite Abhängigkeiten für das Binärpaket werden mittels „`Requires:`“ Angaben vereinbart. Durch zusätzliche Attribute wie „`preun`“ läßt sich der genaue Zeitpunkt festlegen, an dem die jeweilige Abhängigkeit erfüllt sein muß. So können eventuelle Zyklen im Abhängigkeitsgraphen durchbrochen und den Installationsskripten (siehe unten) die nötigen Pakete bereitgestellt werden.

  Da technisch-bedingte Abhängigkeiten wie dynamische Bibliotheken oder Perl-Module während des Buildprozesses automatisch registriert werden, stellen explizit angegebene „`Requires:`“ die Ausnahme dar und sind durch automatisierte Verfahren nur sehr begrenzt überprüfbar.

- Installationsskripte und *Trigger* [3] werden zur Installationszeit des Binärpaketes auf dem Endnutzersystem ausgeführt. Typische Aufgaben sind das Hinzufügen neuer Nutzer, das Registrieren von Bootskripten oder die Aktualisierung temporärer Daten.

  Diese Skripte besitzen nur einen geringen Umfang und sind deshalb vollständig begutachtbar. Hauptaugenmerk liegt auf der Aufnahme genutzter Programme in die jeweilige „`Requires(...):`“ Anweisung, der Verwendung generischer Makros anstatt fester Pfade und guter Wartbarkeit. Da diese Skripte mit Administratorrechten ausgeführt werden, sollten sie insbesondere auf typische Angriffspunkte wie der unsicheren Erzeugung temporärer Dateien getestet werden.

[2] `http://people.mandrakesoft.com/~flepied/projects/rpmlint/`

[3] *Trigger* sind Skripte, welche bei Installation oder Deinstallation anderer Pakete ausgeführt werden; siehe `/usr/share/doc/rpm-*/triggers`

2.3. Zusammenfassung

Es wurde gezeigt, daß einige Maßnahmen zur Qualitätssicherung durch automatisierte Tests durchführbar sind. Ein bereits existierendes Werkzeug hierfür ist „rpmlint“, welches grobe und häufig vorkommende Fehler entdeckt. Fehlerklassen wie ungenügende Abhängigkeiten, inkorrekte Übersetzungsanweisungen, nicht-existierende Einträge in den Dateilisten werden durch Fehlschlagen der Builds erkannt. Für anderen, sind neue Mechanismen zu schaffen.

Für notwendige manuelle Tests sollten die jeweiligen Daten in gut lesbarer Form präsentiert werden. Zur Erleichterung dieser Arbeit sollte es auch möglich sein, die Änderungen bezüglich der letzten akzeptierten Paketversion hervorzuheben.

3. Architektur

3.1. Nutzeroberfläche und -interaktion

Kriterien für die Wahl der Nutzeroberfläche sind:

Verfügbarkeit, d.h., welche Voraussetzungen zum Betrieb der Oberfläche erforderlich sind. Idealerweise sollte die Nutzerinteraktion mit den Bordmitteln aktueller Linuxdistributionen möglich sein und keine Installation zusätzlicher Software benötigen.

Nutzerfreundlichkeit, was Eingabehilfen wie Eingabevervollständigung und frühzeitiges Hinweisen auf Fehler beinhaltet. Weiterhin fallen Aspekte wie Farb- und Schriftgestaltung sowie Barrierefreiheit unter diesen Punkt.

Die Einarbeitungszeit trägt ebenfalls zur Bewertung der Nutzerfreundlichkeit bei; sollte jedoch nicht überbewertet werden, da die Nutzergruppe des Systems hauptsächlich Entwickler sind, die bereit sind, die Handhabung komplexer Systeme zu erlernen.

Offline-Fähigkeit, d.h., ob Aufgaben auch ohne Internetzugang erledigt und beispielsweise als Batchjob ausgeführt werden können. Bei Zunahme von Breitbandverbindungen mit Volumen- statt Zeittarifen kann dieser Punkt an Bedeutung verlieren.

Als Realisierungen für die Nutzerinteraktion kommen nachfolgende Techniken in Betracht:

HTML Oberflächen bieten eine hohe Verfügbarkeit über vorhandene Browser. Nachteile liegen teilweise bei der Nutzerfreundlichkeit, da einige Dinge mit HTML nur schwer lösbar sind bzw. verpönte Techniken wie JavaScript benötigen. Offline-Fähigkeit ist ebenfalls nicht vorhanden.

Diese Art der Benutzerführung wird vom Klientel des Systems erwartet; Beispiele für ähnlich Systeme sind Bugzilla[1] oder Mantis[2].

Programme/Oberflächen auf Endnutzersystemen benötigen meist eine gesonderte Installation und besitzen deshalb eine geringe Verfügbarkeit. Allerdings können sie sehr nutzerfreundlich und mit Offline-Fähigkeiten gestaltet werden.

Ein Beispiel für ein verwandtes Programm ist „bug-buddy“[3].

[1] `http://www.bugzilla.org/`
[2] `http://www.mantisbt.org/`
[3] `http://directory.fsf.org/All_Packages_in_Directory/bugbuddy.html`

Steuerungs-E-Mail müssen in einem bestimmten Format vorliegen. Die Verfügbarkeit ist sehr groß, da durch jedes Mailprogramm verschickbar, und Offline-Fähigkeit ist vorhanden. Vorteile bei der Nutzerfreundlichkeit liegen darin, daß kryptographische Signaturen zur Integritätssicherung und Authentifizierung zu den Standardaufgaben vieler Mailprogrammen gehören. Allerdings liegen die Antwortzeiten im Minutenbereich[4] was neben fehlender Visualisierung eventueller Abfrageergebnisse zu großen Abstrichen bei der Nutzerfreundlichkeit führt.

Beispiele für ähnliche Systeme sind Majordomo[5] oder das Debian Bugtracking System[6]

SCM-Nachrichten, d.h. spezielle Nachrichten, die vom Entwickler beim Einbringen neuer Daten ins Quellcodemanagement-System erzeugt werden. Diese müssen wie obengenannte Steuerungs-E-Mails speziell formatiert sein und können ebenso kryptographische Signaturen enthalten. Dadurch, daß zum Erstellen dieser Nachrichten kein externes Programm wie Browser oder Mailclient benötigt wird, gliedert sich diese Methode gut in den Entwicklungsprozeß der RPM-Pakete ein.

Benachrichtigungs-E-Mail, Mailinglisten und RSS-Feeds sind reine Ausgabeverfahren und dienen zur Benachrichtigung über durchgeführte Aktionen, was u.a. einen Kontrolleffekt hervorruft. Sie bieten hohe Verfügbarkeit (E-Mail und Mailinglisten über herkömmliche Mailprogramme, Mailinglist-Archive und RSS-Feeds über Browser) und geringe Einarbeitungszeit. Offline-Fähigkeit ist bei E-Mail und Mailinglisten ebenfalls möglich.

E-Mail-Benachrichtigungen sind Standard in den meisten existierenden Bugtracking-Systemen; Änderungen am Quellcode werden durch Versionsverwaltungssysteme oft auf Mailinglisten verbreitet, und das existierende fedora.us Repository annonciert neue Pakete in RSS-Feeds.

Da ein HTML-Interface erwartet wird, wurde nur dieses in vorliegender Arbeit entworfen und auf die anderen Eingabemethoden wegen Zeitmangel verzichtet. Für das Bereitstellen der Endprodukte (Binärpakete, Buildlogs, Repository-Metadaten) kommen HTTP-Server zum Einsatz, deren Konfiguration jedoch außerhalb dieser Arbeit liegt.

3.2. Datenspeicherung und -Zugriff

Ausgehend von der Art der zu Daten sind unterschiedliche Arten der Speicherung zu nutzen.

[4] Obwohl die Antwortmails schon nach einigen Sekunden im Postfach liegen, dürfte das Abholen von E-Mails im Sekundentakt für Probleme mit dem Administrator sorgen. Erweiterungen wie IDLE für IMAP [18], welches sofortige Benachrichtigung bei neuer Mail ermöglicht, sind bei den meisten Mailservern nicht aktiviert.

[5] `http://www.greatcircle.com/majordomo/`

[6] `http://www.debian.org/Bugs/Reporting`

Allgemeine QA- und Builddaten: Die in Kapitel 4 genannten Objekte wie Projekt- und Repositorykataloge, Begutachtungen, Richtlinien und historische Performancedaten werden in einer SQL Datenbank gehalten. Dadurch sind leistungsfähige und performante Abfragen möglich, die bei anderen Datenspeichermethoden (Text-/Binärdateien, LDAP) manuell – und damit fehlerträchtig – implementiert werden müßten.

Kriterien für die Auswahl des Datenbanksystems sind Performance, Unterstützung von Transaktionen, Standardtreue (SQL92/99), Schnittstellen für die gewählten Programmiersprachen, sowie persönliche Erfahrung. Außerdem können nicht-freie Lösungen wie Oracle aufgrund der Natur des Projekts von vornherein ausgeschlossen werden. Ebensowenig kommen rein klientenseitige Datenbanken wie Berkeley DB oder SQLite[7] in Betracht, da sich die Anwendung über mehrere Rechner verteilen kann.

Ausgehend davon, wurde sich für PostgreSQL[8] entschieden: es ist nahe dem SQL99 Standard, unterstützt Transaktionen, bietet Schnittstellen für C, C++, Python, TCL, Ruby, kommt gut mit größeren Tabellen zurecht, paßt in die bereits existierende Infrastruktur (Bugzilla mit PostgreSQL Datenbank) hinein, und es existieren vielfältige Erfahrungen damit. MySQL[9] hingegen hat seinen eigenen, inkompatiblen SQL Syntax und ist bei komplexen Abfragen, Schreibaktionen sowie bei Nutzung der InnoDB und Berkeley DB Backends (welche für Transaktionen benötigt werden) weniger performant als PostgreSQL. Für Firebird[10] existieren zu wenig Erfahrungswerte, um es in Betracht zu ziehen.

Authentifizierungsdaten der Nutzer: Je nach Art der Authentifizierung können bereits existierende Mechanismen genutzt oder eigene implementiert werden. Beispiele für existierende Mechanismen sind das Nutzerlogin über *Pluggable Authentication Module* (PAM), die Weitergabe von GSSAPI/Kerberos Token [17, 19], Nutzung der HTTP-Authentifizierung [10, Abschnitt 11], sowie klientenseitige SSL/TLS Zertifikate. In diesen Fällen erfolgt die Speicherung transparent für das System und der Zugriff auf die nötigen Informationen (Nutzername, Kerberos-Principal, TLS-DN) über spezielle Interfaces.

Die im System verwendete `simple` Authentifizierungsmethode (vergleiche Abschnitt 6.1), speichert die Authentifizierungsdaten (Login und Paßwort) in der Datenbank und greift mittels SQL darauf zu.

Autorisationsdaten der Nutzer: Obwohl dafür eine Datenspeicherung in einer LDAP Datenbank möglich wäre, werden diese Daten zusammen mit den oben genannten QA-

[7] `http://www.sqlite.org`
[8] `http://www.postgresql.org/`
[9] `http://www.mysql.com`
[10] `http://firebird.sourceforge.net/`

Daten in einer SQL Datenbank gehalten. Dies hat den Vorteil, daß Abfragen kombiniert und damit Overhead vermieden werden kann. Ein Nachteil dieser kombinierten Datenhaltung ist allerdings die reduzierte Flexibilität, d.h., daß diese Autorisationsdaten nur über Umwege für andere Anwendungsgebiete (z.B. das Fedora *Content-Management-System* (CMS) oder das Entwicklungs-*Sourcecode-Management-System* (SCM)) nutzbar sind.

Die zu verpackende Software: Unter diesen Begriff fallen die SRPM Pakete, welche für einige Zwischenschritt auch extrahiert vorliegen müssen, sowie schon gebaute, aber noch nicht veröffentlichte Binärpakete. Diese Daten liegen temporär in der Größenordnung von einigen Tagen vor, und es fallen relativ große Datenmengen an; zum Beispiel benötigen die Quellpakete von openoffice.org-1.1.2 oder kde-i18n-3.3.2 einzeln jeweils ca. 200 MB.

Prinzipiell könnten diese Daten in der SQL Datenbank gespeichert werden – die meisten Pakete bleiben weit unter diesen Größen[11] und PostgreSQL 7.4.6 erlaubt Schreib- und Leseraten von ca. 5 MB/s bei hinreichend großem Hauptspeicher auf einem Pentium 4 1.6 GHz. Allerdings addiert sich der Speicherbedarf bei parallelem Zugriff, so daß die Forderung nach hinreichend großem Hauptspeicher eventuell nicht mehr erfüllt wird, das System mit *Swapping* beginnt und sich die Übertragungsraten extrem verschlechtern. Außerdem würden Anwendungen und Hilfstools speziell für diesen Zweck geschriebenen Wrapper benötigen, um auf diese Daten zuzugreifen.

Eine andere Möglichkeit sind HTTP oder FTP Server, auf die mit den gängigen Protokollen zugegriffen wird. Aber auch hier entsteht der Nachteil, daß URLs von viele Hilfstools nicht verstanden und gesonderte Wrapper benötigt werden.

Deshalb verwendet vorliegendes System ein Netzwerkfilesystem, auf welches jede Teilanwendung oder Hilfstool transparent mittels des *Virtual File System Layer* (VFS) des Betriebssystems zugreifen kann.

Endprodukte: Darunter fallen die zu veröffentlichenden Binär- und Quellpakete, Buildlogs sowie die Repository-Metadaten. Diese Daten werden entsprechend der existierenden Infrastruktur im Filesystem gespeichert. Das Einbringen kann entweder direkt über den VFS-Layer oder mittels geeigneter Protokolle (HTTP-PUT, WebDAV, FTP-PUT) erfolgen.

Vorherige Paketversionen: Welche für QA-Aufgaben verwendet werden können. Die Art der Datenspeicherung und des -zugriffs hängt vom jeweiligen System ab; Details sind in Abschnitt 5.4 zu finden.

[11] ein „`ls -l | LANG=C awk '{ z++; a+=$5; printf "%8.2f\n", a/z; }'`“ Scriptlet auf dem fedora.us stable Branch liefert 1.3 MB als durchschnittliche Paketgröße

Temporäre Statusdaten werden entsprechend ihrer Art an unterschiedlichen Orten gespeichert. Vorgänge wie das einmalige Anmeldung und die spätere Anmeldung mittels eines eindeutigen Identifikators, oder das zeitverzögerte Signieren vorbereiteter Texte erfordern die Implementierung einiger Teilanwendungen als *stateful server*.

Kurzlebige Daten, die nur in einer einzelnen Teilanwendung existieren, werden aus Einfachheitsgründen nur als Variablenwerte gespeichert und gehen nach einem Neustart verloren. Falls sich dies im laufenden Betrieb als Problem herausstellt, kann dem mittels Speicherung in der Datenbank oder in Dateien vorgebeugt werden.

Die Speicherung temporärer Daten, die über mehrere Teilanwendungen hinweg oder über längere Zeit (mehrere Tage) benötigt werden, erfolgt in der Datenbank. So wird sichergestellt, daß diese Daten bei Neustarts nicht verlorengehen und eine konsistente Sicht darauf existiert.

3.3. Anwendung

[29, Kapitel 16] nennt ein großes monolithisches Programm oder mehrere kleinere, miteinander kommunizierende Anwendungen als mögliche Implementierungsform komplexer Systeme.

Der Vorteil des monolithischen Programmes liegt bei der Vereinfachung der inneren Kommunikation. Da alle Programmteile auf der selben Maschine und im selben Speicher ausgeführt werden, können dafür normale Variablen eingesetzt werden. Nachteile liegen allerdings in den Bereichen der Skalierbarkeit, der Ausfallsicherheit und des Zugriffsschutzes.

Bei verteilten Teilanwendungen muß hingegen die innere Kommunikation genauer spezifiziert werden. Dafür können wichtige Programmteile redundant auf getrennten Maschinen ausgeführt werden, was zur Erhöhung der Ausfallsicherheit und auch der Skalierbarkeit beiträgt. Ebenso können Teilanwendungen, die ohne äußere Kommunikation auskommen oder potentiell gefährliche Aktionen ausführen, firewallgeschützt zu betrieben. Dadurch können Angriffe auf IP-Ebene verhindert werden.

Aufgrund dessen, wurde das QA- und Buildsystem als verteiltes System entworfen, dessen Komponenten in Kapitel 5 näher beschrieben werden.

3.4. Programmiersprachen

Als Kriterien für die Wahl der Programmiersprache werden die Entwicklungsgeschwindigkeit, Wartbarkeit, verfügbare Bibliotheken und die Performance der Programmabarbeitung nachfolgend betrachtet.

C ist eine imperative Programmiersprache. Darin geschriebene Programme werden direkt in den Maschinencode des ausführenden Rechners übersetzt und sind – wenn von erfah-

renen Entwicklern geschrieben – in der Regel performanter und ressourcenschonender als vergleichbare Programme in Skriptsprachen.

Da sich mit C Programme erzeugen lassen, die nur geringe Abhängigkeiten (mit eventuellen Sicherheitslöchern) haben, bietet sich diese Sprache für SUID-Wrapper an, d.h. Programmen, die bei Ausführung eine andere Nutzeridentifikation zugeordnet bekommen. Solche Wrapper sind beispielsweise nützlich, um Geheimnisse des Systems wie Datenbankpaßworte zu wahren, wenn externe Programme für Aufgaben wie WWW-Download oder CVS-Checkout eingesetzt werden.

Für größere, insbesondere netzwerkbezogene Anwendungen ist C weniger geeignet. Da hier das Speichermanagement, d.h. das Anfordern und Freigeben von Speicher, sowie Überprüfung auf Bereichsüberschreitungen, manuell und ohne Unterstützung seitens der Programmiersprache durchzuführen ist, kommen sicherheitskritische Fehler wie *Bufferoverflows* sehr häufig vor. Ein Blick auf gängige Sicherheitslisten zeigt, daß solche Fehler auch in von erfahrenen Programmierern geschriebenen Projekten auftreten.

C++ ermöglicht sowohl prozedurale, objektorientierte, als auch generische Programmierung. C++ bietet Unterstützung für Basisalgorithmen und -strukturen (Listen, assoziative Felder, ...); für darüberhinausgehende Anforderungen und insbesondere für XML-RPC müssen externe Bibliotheken mit schlechter Qualität oder großen Abhängigkeiten herangezogen werden. Deshalb wäre ein nicht-geringfügiger Aufwand zur Implementierung dieser Funktionalität einzuplanen.

Durch Mechanismen wie *Stack-Unwinding*, d.h. dem Löschen von auf dem *Stack* befindlicher Objekte, und damit verbundener Entwurfsmuster (*Guards*, *Auto-Pointer*) können fehlerhafte Programmierung bezüglich des Speichermanagements vermieden werden. Klassen, die den Pufferzugriff kapseln, bieten Schutz vor *Bufferoverflows*, und *Exceptions* helfen bei der Fehlerbehandlung.

C++ bietet aufgrund strikter Typ- und Syntaxprüfung sowie durch Dokumentationshilfen wie Doxygen[12] eine sehr gute Wartbarkeit. Diese wird allenfalls durch die hohe Komplexität von C++ reduziert, welche erfahrene Entwickler zur effizienten und sauberen Programmierung erfordert.

PHP 4[13] ist eine Programmiersprache speziell für die Erstellung dynamischer HTML Inhalte. Es verfügt über Basisfunktionalität für objektorientierte Programmierung, die allerdings Features wie *Namespaces* oder unterschiedliche Sichtbarkeit von Methoden oder Attributen missen läßt. Dies kann negative Einflüsse auf die Wartbarkeit ausüben. Demgegenüber steht eine umfangreiche Standardbibliothek, die zum Beispiel Unterstützung für XML-RPC beinhaltet.

[12] `http://www.stack.nl/~dimitri/doxygen/index.html`

Wie alle Skriptsprachen, so bietet auch PHP ein eingebautes Speichermanagement und Schutz vor *Bufferoverflows*, so daß es auch von unerfahrenen PHP-Programmierern angewendet werden kann.

Wenn es zur Erstellung dynamischer HTML Seiten verwendet wird, ist insbesondere darauf zu achten, daß von Nutzern eingegebene Informationen nicht ungefiltert dargestellt werden. Bei Nichtbeachtung werden sonst *cross-side-scripting* (XSS) Attacken [24] durch Einfügen geeigneten JavaScripts ermöglicht.

Neben der manuellen Bearbeitung der jeweiligen Informationen mittels `htmlencode()`, wird in der vorliegenden HTML Nutzeroberfläche die *Smarty Template Engine*[14] verwendet, welche Variablenwerte automatisch in eine geeignete Form kodiert. Smarty erlaubt außerdem der Trennung von PHP-Code und HTML-Oberflächenbeschreibung, was zur Erhöhung der Wartbarkeit beiträgt.

Python stellt eine Skriptsprache mit Unterstützung objektorientierter Programmierung dar. Nach Erfahrung des Autors erlaubt sie aufgrund der umfangreichen Standardbibliothek und der dynamischer Typisierung die schnelle Entwicklung von Code und ist deshalb sehr gut für das Prototyping von Anwendungen geeignet.

Wie PHP so bietet auch Python ein automatisches Speichermanagement und Schutz vor *Bufferoverflows*. Nachteile von Python liegen – wie bei den meisten Skriptsprachen – in der Wartbarkeit: durch dynamische Typisierung und späte Variablen- und Methodenbindung werden Schreibfehler oder falsche Variablennutzung versteckt und treten erst beim Ausführen des entsprechenden Codes in Erscheinung. Dies macht sehr umfangreiche Testsoftware erforderlich, welche möglichst jede Codezeile einmal ausführt.

3.5. Kommunikation

Zur Kommunikation zwischen Klient und Anwendung, sowie zwischen den einzelnen Teilanwendungen sind verschiedenste Kommunikationswege möglich. Nachfolgend sollen diese unter den Gesichtspunkten der Sprachunterstützung, Tauglichkeit in restriktiven Netzwerkumgebungen (Blockade direkter Verbindungen durch Firewall, nicht-routebare Adressen [20]), Performance sowie der Integritäts- und Vertraulichkeitssicherung untersucht werden.

Wie später noch deutlich wird, spielt aber die Performance der Datenübertragung nur eine untergeordnete Rolle.

Pipes oder auch FIFOs sind unidirektionale Kommunikationswege und können nur für Applikationen, die auf der selben Maschine laufen, verwendet werden. Neben der üb-

[14] `http://smarty.php.net/`

lichen Interprozeßkommunikation mit Kommando- und Datenpfaden, sind die Synchronisation bei der Prozeßerzeugung sowie Ein- und Ausgabeumleitungen typische Anwendungsgebiete für Pipes.

Zur Kommunikation zwischen Eltern- und Kindprozessen werden üblicherweise mittels des `pipe(2)` Syscalls erstellte anonyme Pipes verwendet. Darüberhinaus existieren sogenannte *named pipes*, d.h. von `mkfifo(3)` bzw. `mknod(2)` erzeugte Dateien, die zur Datenübertragung zwischen unabhängigen Prozessen genutzt werden können.

Da die Kommunikation ausschließlich auf einer Maschine stattfindet, sind Fragen bezüglich der Integritäts- und Vertraulichkeitssicherung irrelevant.

Die Socket API, d.h. die direkte Nutzung der `socket(2)`, `bind(2)`, `connect(2)`, `recv(2)`, `send(2)` u.ä. Funktionen (siehe [28]), erlaubt eine sehr effiziente und damit schnelle Datenübertragung. Allerdings muß das Serialisieren der Daten manuell erfolgen, was insbesondere bei unterschiedlichen Programmiersprachen, Zeichensätzen oder Rechnerarchitekturen (*Big-* vs. *Little-Endian*) an den Verbindungsenden zu Problemen führen kann.

Als standardisierte Repräsentationsform der Daten könnte XDR [27] verwendet werden; allerdings erlaubt dieser Standard nur einfache Datentypen (Zahlen, Zeichenketten), gleichmäßige Listen und Strukturen. Unterstützung für assoziative Arrays oder gemischte Listen muß manuell programmiert werden.

Wesentlich leistungsfähiger, aber auch komplexer ist ASN.1 [15]. Diese Datenkodierung wird beispielsweise zur Kommunikation mit LDAP Servern oder bei der Identifikation via SPNEGO [2] verwendet; direkte Unterstützung in den Standardbibliotheken gängiger Programmiersprachen fehlt aber.

Für integritäts- und vertraulichkeitssichernde Maßnahmen müßte manuell gesorgt werden; hier bietet sich die Verwendung der OpenSSL Bibliothek[15] an, welche anerkannte und erprobte Verschlüsselungsverfahren bereitstellt.

Direkte Socketnutzung ist bei firewallgeschützten Umgebungen problematisch; ein Ausweg könnte *Tunneling* (z.B. durch HTTP Proxies) sein, was zusätzlichen Implementierungsaufwand hervorruft. Dieser Aufwand erhöht sich bei Verwendung der obengenannten OpenSSL Bibliothek.

Aus diesen Gründen kommen Sockets zur Kommunikation zwischen Klient und Anwendung nicht in Betracht. Insbesondere bei einfachen Daten sind sie jedoch eine überlegenswerte Option zur Kommunikation zwischen Teilanwendungen.

SUN RPC ermöglicht C Programmen, Prozeduren auf entfernten Rechnern auf transparente Art und Weise aufzurufen. Dazu werden die fraglichen Funktionen und Datenstrukturen in der *RPC Language* [26, Abschnitt 11] verfaßt und mittels des `rpcgen` Pro-

[15] `http://www.openssl.org/`

gramms zu *Stubs* für Server- und Klientseite transformiert. Die eigentliche Kommunikation zwischen Klient und Server geschieht über eine RPC Laufzeitumgebung, welche Teil des Basissystems auf den meisten Linuxmaschinen ist.

SUN RPC unterstützt Authentifizierungsmechanismen und mittels RPCSEC_GSS [7] kann Vertraulichkeit und Integrität der übertragenen Daten gewährleistet werden.

Aufgrund des direkten Verbindungsaufbaus ist SUN RPC bei firewallgeschützten Umgebungen und der hier benötigten Klient–Server Kommunikation nicht anwendbar.

Da es für andere Programmiersprachen als C oder C++ nicht verfügbar ist, scheidet es ebenfalls zur Kommunikation zwischen den Teilanwendungen aus.

XML-RPC kodiert Funktionsaufrufe, deren Parameter und Rückgabewerte in XML Datenströmen. Als Datentyp sind primitive Typen wie Integer, Gleitkommazahlen und Zeichenketten möglich. Ebenfalls werden Listen und assoziative Felder unterstützt, wobei bei letzterem nur Zeichenketten als Schlüsselwerte möglich sind. XML ermöglicht Spezialzeichen wie Umlaute oder graphische Symbole durch „`&...;`“ *Entities* oder durch Vereinbarung des Zeichensatzes im XML Kopf.

Die Datenübertragung erfolgt in einer HTTP Anforderung gekapselt, so daß mittels HTTP Proxies der Einsatz in firewallgeschützten Umgebungen möglich ist. Vertraulichkeit und Integrität der Kommunikation kann durch HTTPS garantiert werden, d.h. dem Tunneln von HTTP durch eine SSL Verbindung.

Für die verwendeten Skriptsprachen „Python“ und „PHP“ existiert Unterstützung in den Standardbibliotheken. Der Einsatz auf Klientenseite erfolgt für Python vollkommen transparent; es muß nur darauf geachtet werden, daß keine nicht-unterstützten Datentypen wie „`None`“ oder assoziative Felder mit Nicht-Zeichenketten als Schlüsselwert verwendet werden.

Ein Nachteil von XML-RPC liegt bei der Performance: XML Ströme enthalten Klartextdarstellungen für alle Start- und Endmarkierungen (z.B. „`<string>...</string>`“), was Redundanz und damit unnötigen Ballast bei der Datenübertragung bedeutet. Ebenso fehlen Informationen über die benötigten Speichermengen (z.B. Länge der Zeichenketten, Elementanzahl in Listen), so daß keine effiziente Speicherverwaltung in den XML Parsern möglich ist.

Da aber die Übertragungsgeschwindigkeit der in Frage kommenden Daten nebensächlich ist, wiegen die Standardisierung von XML-RPC und die gute Sprachunterstützung die Nachteile auf. Deshalb wird dieses Protokoll sowohl für die Kommunikation mit externen Klienten, als auch innerhalb des Systems verwendet.

4. Objekte

4.1. Projekte (*projects*)

Dies ist die Software, welche von unabhängigen Autoren geschrieben wurde und letztendlich als RPM-Paket vorliegen soll.

Um der Einzigartigkeit in den Repository-Klassen gerecht zu werden (siehe unten), werden die Projekte in Zweigen (Branches) erfaßt, welche den jeweiligen Entwicklungsstand kennzeichnen. So kann zum Beispiel die mit "alpha" gekennzeichnete Entwicklungsversion in einem Repository mit entspannten Qualitätsrichtlinien geführt werden. Währendessen befindet sich die normale Programmversion in einem Repository mit strikteren Richtlinien.

Das Hinzufügen eines Projektes muß durch ranghohe Personen autorisiert werden, um die Aufnahme rechtlich fragwürdiger oder für das jeweilige Repository unpassender Software zu verhindern.

4.2. Pakete

Dies sind einzelne Projektversionen, die ins RPM-Format umgewandelt wurden. Dabei ist zwischen Quell- und Binärpaketen zu unterscheiden. Die Quellpakete sind architekturunabhängig und enthalten meist den Quelltext des jeweiligen Projektes, Patches, zusätzliche Daten und ein .spec-File als Beschreibung für die Umwandlung in ein Binärpaket.

Das System bildet aus einem Quellpaket ein oder mehrere architekturspezifische Binärpakete, welche mittels des rpm-Paketmanagers oder Frontends wie apt, yum oder smartpm installiert werden können.

Zum Beispiel ist `rpm-4.3.3-1.src.rpm` ein typischer Name für Quellpakete, während die Namen `rpm-4.3.3-1.i386.rpm` und `rpm-build-4.3.3-1.i386.rpm` für die daraus erzeugten Binärpakete möglich sind.

Pakete werden als Quellpaket ins System gebracht, müssen den Qualitätssicherungsprozeß durchlaufen, um dann als Binärpaket öffentlich zur Verfügung gestellt zu werden.

4.3. Repositories (*repositories*)

Damit wird eine Sammlung von Paketen bezeichnet, welche gewissen Ansprüchen genügt. Im Einzelnen können dies Qualitätsrichtlinien (*Policies*) sein, oder Festlegungen, welche andere Repositories zur Auflösung von Paketabhängigkeiten herangezogen werden können.

Ein Beispiel für Richtlinien ist: „ein Paket muß von mindestens drei Leuten begutachtet werden, bevor es in das *Repository* übernommen wird“. Abhängigkeiten können wie: „Pakete aus dem ‘unstable’ Repository können ‘stable’ Pakete benötigen“ lauten.

In der hier vorliegenden Implementierung werden *Repositories* in Klassen gruppiert, in denen ein Projektzweig nicht mehr als einmal vorkommen darf. So kann zum Beispiel die “fedora.us” Klasse mit den Repositories “stable”, “testing” und “unstable” existieren.

4.4. Architektur (*architectures*)

Dies ist ein Tripel aus Repository, Version des Fedora Core Systems und CPU-Architektur und bezeichnet das Ziel des fertigen RPM-Paketes. Ein Beispiel ist (‘fedora.us, stable’, ‘Fedora Core 1’, ‘Intel i386’) oder kurz ‘i386-fc1-fedus-stable’.

Der Grund für die Aufnahme der CPU-Architektur in dieses Tripel ist offensichtlich: Pakete mit Maschinencode für z.B. ARM CPUs laufen nicht auf MIPS Maschinen, oder Intel Pentium 4 (i686) kennt Befehle, die nicht auf Intel i386 zur Verfügung stehen.

Da das Core System grundlegende Projekte wie die C-Library (glibc) beinhaltet, ist es eine Basisabhängigkeit für alle Repositories. Dabei ist die Version entscheidend: es kann *nicht* davon ausgegangen werden, daß Repositories für Core-Version *a* mit Core-Version *b* zusammenarbeiten. Diese Inkompatibilität wird durch unterschiedliche Libraryversionen und weggefallene Projekte verursacht.

4.5. Personen (*people*)

Als „Personen“ werden sowohl die Autoren der Pakete bezeichnet, als auch die Begutachter im Qualitätssicherungsprozeß. Als Identifikationsmerkmal wurde im vorliegenden System die E-Mail-Adresse gewählt, da diese eindeutig einer Person zuordenbar ist, was z.B. beim natürlichen Namen nicht gewährleistet ist.

Personen können Gruppen angehören, denen Zugriff auf besonders geschützte Information wie Patches für Sicherheitslücken gewährt wird. Obwohl die Entwicklung im Fedora Projekt für jeden frei zugänglich sein sollte, sind solche Beschränkungen aufgrund von Stillschweigeabkommen (*Nondisclosure Agreements*, NDAs) auf Mailinglisten wie *vendor-sec* nötig.

4.6. Ränge (*ranks*)

Personen besitzen repositoryabhängige Ränge. Mit steigendem Rang wachsen zum einen die Zugriffsmöglichkeiten auf bestimmte Informationen, als auch die Rechte bei der Qualitätssicherungsarbeit. Beispielsweise können Personen mit hohem Rang neue Repositories erzeugen, deren Richtlinien ändern und Daten anderer Personen einsehen. Ein hoher Rang macht

es auch einfacher, ein Paket als qualitativ in Ordnung zu markieren und zu veröffentlichen.

Ränge werden im System als Zahl gespeichert: je höher die Zahl, desto höher der Rang. Zur besseren Visualisierung werden militärische Bezeichnung wie "*Private*", "*Lieutenant*" oder "*General*" verwendet. Administratoren mit repositoryübergreifenden Privilegien wird der Rang "*Ares*" oder "*Mars*" zugeordnet.

Das Rangsystem ist nötig, da die Mitarbeit im Fedora Projekt für jeden offen, seine Fähigkeiten und Vertrauenswürdigkeit anfangs jedoch unbekannt sind. Nach längerer guter Mitarbeit kann durch einen Beförderungsprozeß der Rang angepaßt werden. Die Grundidee ist, daß das Erreichen hoher Ränge viel Arbeit erfordert und niemand diese durch leichtfertige Aktionen aufs Spiel setzt. Mehrfach-Identitäten für die selbe Person, die als „Wegwerf-Accounts" für Angriffe mißbraucht werden können, machen deshalb auch wenig Sinn.

Technisch wird der Beförderungsprozeß so gelöst, daß bei Erzeugung eines Repositories von den Administratoren eine Person mit dem *General*-Rang ausgestattet wird, welche weiteren Personen entsprechende Ränge verleiht. Nun kann jede Person eine andere mit niedrigerem Rang zur Beförderung oder Degradierung vorschlagen. Je nach Repositoryrichtlinie hat dies aber keine sofortige Wirkung, sondern benötigt weitere Befürworter.

4.7. Kryptographische Schlüssel

Obwohl sich die Nutzer mittels Paßwort anzumelden haben, ist diese Art der Authentifizierung für viele Aufgaben nicht ausreichend. So könnten Schwachstellen im Browser, im Frontend oder in den Anmeldemechanismen zum Abschicken manipulierter Anträge ausgenutzt werden. Typische Lücken sind die „Fehlinterpretation" aktiver Inhalte (JavaScript, ActiveX; siehe Beispiele unter [3]), *cross-side-scripting* (XSS) Attacken [24], schwache Paßworte oder ungenügende externe Authentifizierungsmechanismen (z.B. sequentiell aufsteigendes und damit vorhersagbares Login-Cookie für Bugzilla [5]).

Das Ergebnis solcher Attacken können unberechtigte Rang- und damit Privilegienerhöhung durch gefälschte Beförderungsanträge, oder das automatische Bauen und Veröffentlichen boshafter Pakete sein.

Um dem vorzubeugen, sind derartige Anträge kryptographisch mit einem vertrauenswürdigen Schlüssel zu unterzeichnen. Neben höherer Sicherheit ist die exakte Protokollierung rechtlich eventuell problematischer Vorgänge (z.B. dem Einbringen von Paketen) ein Vorteil des Signierens derartiger Aktionen.

Grundsätzlich sind *Public Key Infrastrukturen* (PKI) mit hierarchische Strukturen und das *Web-of-Trust* zu unterscheiden. Eine PKI-Implementierung könnte so aussehen, daß Red Hat eine *Certification Authority* (CA) betreibt und jeder Person ein X.509v3 Zertifikat [13] ausstellt. Dies hätte jedoch einen relativ hohen Verwaltungsaufwand zur Folge, da strenge Sicherheitsstandards eingehalten werden müssen (das Kompromittieren des CA Zertifikats würde sämtliche bereits ausgestellte Zertifikate ungültig machen) und *Certificate Revocation Lists* (CRLs) gepflegt und verteilt werden müssen.

Stattdessen verwendet das vorliegende System ein auf OpenPGP [6] aufbauendes *Web-of-Trust*. Hierzu erstellt sich jede Person einen eigenes Schlüsselpaar. Für geringe Ränge sind keine zusätzlichen Bestätigungen nötig; bei höheren Rängen sind eine oder mehrere Echtheitsbestätigungen aus einem definierten *Web-of-Trust* erforderlich. Sämtliche damit verbundene Aufgaben lassen sich mit Bordmitteln aktueller Linux Distributionen erledigen: die Zertifikate sind auf bereits existierenden *Keyservern*[1] abholbar, der OpenPGP Standard beinhaltet Widerrufszertifikate, GnuPG[2] – eine der verbreitetsten OpenPGP Implementierungen – versteht diese Widerrufszertifikate und unterstützt eine Vertrauensdatenbank (*Trust-DB*), mit der sich genanntes *Web-of-Trust* implementieren läßt.

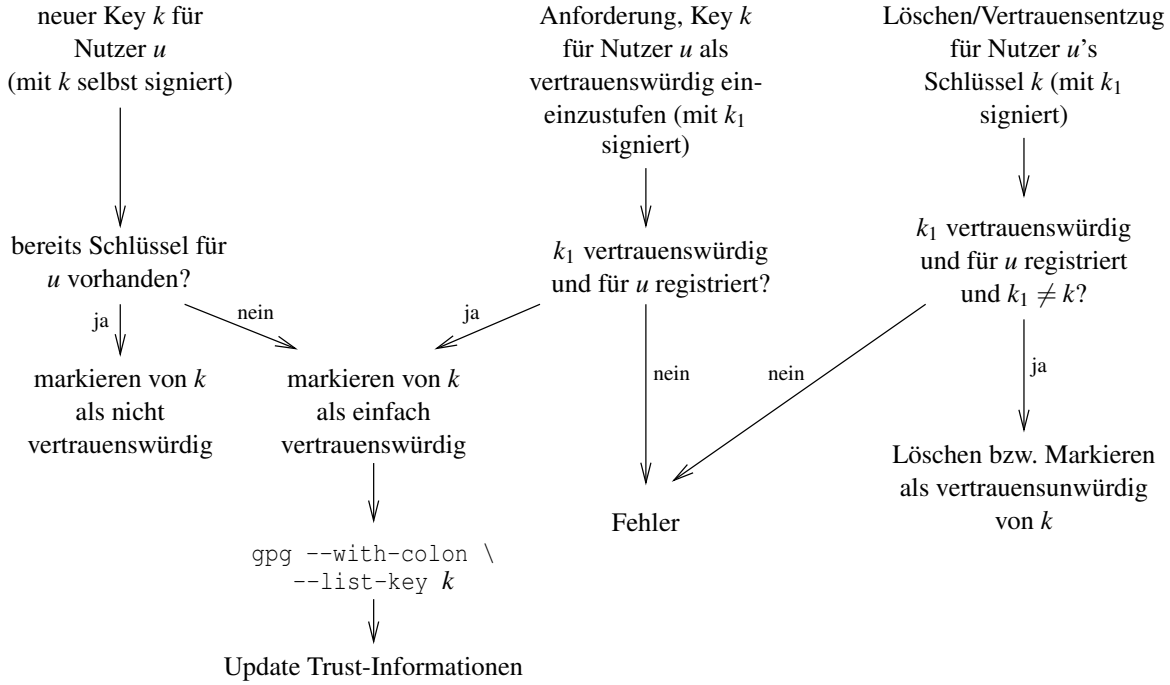

Abbildung 4.1.: Bearbeitung von GPG-Schlüsseln

Abbildung 4.1 stellt das verwendete Verfahren zum Bearbeiten der Schlüssel dar, welches gewährleistet, daß:

- jede Person entweder noch nie einen Schlüssel hinterlegt hat oder mindestens einen einfach vertrauenswürdigen Schlüssel besitzt; das Löschen des einzigen vertrauenswürdigen Schlüssels ist nicht möglich
- ein Angreifer keinen neuen Schlüssel ohne Kenntnis eines vorhandenen hinzufügen

[1] z.B. SKS Keyserver
[2] http://www.gnupg.org

kann, wenn schon Schlüssel vorhanden sind. Das selbe gilt für das Löschen eines existierenden Schlüssels.

- das Hinzufügen eines ersten, automatisch vertrauenswürdigen Schlüssels durch einen Angreifer relativ harmlos ist. Der Account wäre zwar für den rechtmäßigen Eigentümer wertlos, ihm kann aber kein höherer Rang zugeordnet sein, da dafür signierte Aktionen nötig gewesen wären.

4.8. Tickets (*tickets*)

Als „Ticket“ werden Anträge zur Aufnahme von Paketen ins System bezeichnet. Ein solches Ticket beinhaltet:

- die Bezeichnung des Projektzweiges (z.B. „Projekt ‘rpm’, Branch ‘alpha’“)
- die Versions- und Releasenummer des Projektzweiges. Die Versionsnummer ist meist die Version des Projektes, während die Revision die Version des Paketes kennzeichnet. Bei Verwendung nicht-numerischer Projektversionsnummern, kann diese Unterscheidung aber auch verwischen.
- den Ort, wo das Quellpaket zu finden ist. Dies kann eine HTTP oder FTP URL, aber auch eine Subversion- oder CVS-URL sein. In den letzten beiden Fällen wird nicht das Quellpaket ansich, sondern die darin enthaltenen Dateien an angegebener Stelle erwartet.
- Angaben, für welche Architekturen das Paket gedacht ist.
- ein Datum, ab wann das neue Paket frühstens veröffentlicht werden darf. Solche Verzögerungen müssen aufgrund der schon erwähnten Stillschweigeabkommen möglich sein.
- ein Flag, ob das Paket begutachtet werden muß. Bei hohem Rang des Ticketautors, kann ein solcher Zwang standardmäßig entfallen, jedoch bei großen Änderungen gewünscht sein.
- ein Flag, ob Pakete, welche das durch das Ticket beschriebene Paket als Abhängigkeit besitzen, neu gebaut werden sollen. Normalerweise ist dies nicht nötig, kann aber bei Änderung des *Application Program Interfaces* (*API*), des *Application Binary Interfaces* (*ABI*) oder der Verzeichnisstruktur nötig sein.
- Informationen über Schwerpunkt und Komplexität des neuen Paketes im Vergleich zur nächstfrüheren Version. Diese Daten sind für die Begutachter für eventuelle Aufwandsabschätzungen gedacht und werden vom System ignoriert.

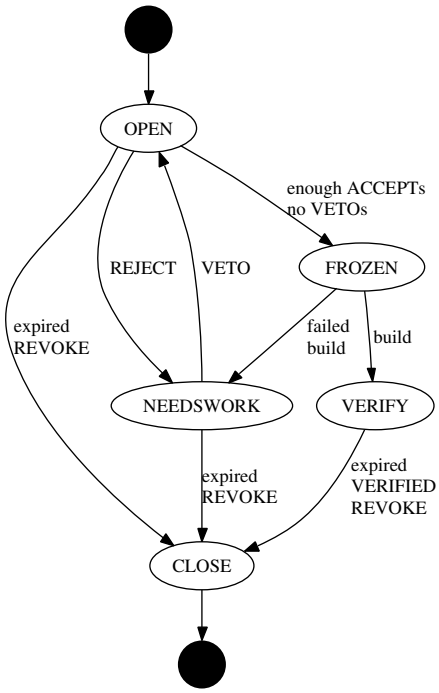

Abbildung 4.2.: Ticketlebenslauf

Jedes Ticket durchläuft pro Architektur den in 4.2 angegebene Lebenszyklus. Statusübergänge können durch Alterungsprozesse (*expiration*), Aktionen des Buildsystems, Automatismen und Begutachtungen anderer Personen hervorgerufen werden. Der Einfluß bzw. die Parameter der jeweiligen Komponenten wird durch architektur- und zeitabhängige Richtlinien bestimmt.

4.9. Begutachtungen

Die Begutachtungen, welchen den im letzten Abschnitt genannten, architekturabhängigen Lebenszyklus beeinflussen, lassen sich in die unten aufgeführten Arten einteilen.

Zustimmungen (*ACCEPTs*) welche das Ticket als den Ansprüchen genügend markiert. Da diese Art von Bewertung den Bau von Binärpaketen zur Folge haben kann, muß sie kryptographisch signiert werden. Eine solche Bewertung kann auch nur für einzelne Architekturen des Tickets ausgesprochen werden.

Ablehnungen (*REJECTs*) können wie die Zustimmungen nur für einzelne Architekturen ausgesprochen werden und markieren das beschriebene Paket als fehlerhaft. Das Ticket geht daraufhin in den `NEEDSWORK` Status über und wird nach einer gewissen Zeit automatisch geschlossen.

Das QA- und Buildsystem meldet Fehlersituation wie falsche URL im Ticket oder fehlgeschlagende Builds ebenfalls als *REJECT*. Hochrangige Personen können dagegen Einspruch einlegen, nachdem temporäre Fehler (Defekte in der Netzwerkverbindung, nicht genügend Plattenplatz u.ä.) beseitigt wurden.

Bestätigungen (*VERIFYs*), um ein vom Buildsystem erstelltes Binärpaket letztendlich freizugeben. Dieser Schritt kann normalerweise[3] nur vom Paketautor oder einer hochrangigen Person durchgeführt werden.

Widerrufe (*REVOCATIONs*), um Tickets für alle oder einzelne Architekturen zu entfernen. Dies kann bei nicht mittelfristig behebbaren Fehlern oder beim Bereitstellen einer neueren Paketversion nötig sein. Normalerweise kann diese Aktion nur vom Paketautor oder hochrangigen Personen durchgeführt werden.

Einsprüche (*VETOs*) gegen das Ticket ansich dienen dazu, um einen eventuell automatische Übergang von `OPEN` zu `FROZEN` zu stoppen, ohne formal eine Ablehnung auszudrücken. Dies kann bei unklaren Sachverhalten erwünscht sein.

Ein solches Veto ist nur für eine bestimmte Zeit gültig, deren Dauer sich nach dem Rang der einsprucheinlegenden Person richtet. Nach Ablauf des Vetos wird das Ticket so behandelt, als ob das Veto nie existiert hätte.

[3] „normalerweise“ kann hier als „bei Verwendung der Standardrichtlinie“ verstanden werden

Einsprüche (*VETOs*) gegen Zustimmungen dienen auf der einen Seite dazu, eigene Zustimmungen zu widerrufen, als auch fremde zeitweilig ungültig zu machen, so daß sie nicht in die Kalkulation des `OPEN` zu `FROZEN` Übergangs einfließen.

Solche Einsprüche sind – je nach Richtlinie – meist nur gegen Zustimmungen rangniederer Begutachter möglich.

Einsprüche (*VETOs*) gegen Ablehnungen dienen dazu, um ein Ticket vom `NEEDSWORK` wieder in den `OPEN` Status zu versetzen. Dieser Übergang erfolgt jedoch erst, wenn dies mit allen REJECTs geschah.

Anders als die bisher genannten Vetos gilt hier keine zeitliche Begrenzung: wenn Einspruch gegen ein REJECT eingelegt wurde, wird es sofort und für immer ungültig. Aus diesem Grund muß ein solches Veto stets von einer ranghöheren Person erfolgen.

Einsprüche (*VETOs*) gegen Einsprüche verursachen ein sofortiges Auslaufen der Gültigkeitsdauer der jeweiligen Vetos. Auch hier gilt, daß ein solcher Einspruch nur von ranghöheren Personen ausgesprochen werden kann.

Im vorliegenden System ist die Schachtelung solcher Einsprüche nicht möglich; d.h. Vetos gegen Vetos, welche Vetos aufheben sind verboten. Stattdessen müßte ein neues Veto gegen das ursprüngliche Objekt (Ticket, Ablehnung oder Zustimmungen) eingelegt werden.

Weiterhin sind beliebig geschachtelte Kommentare möglich, die aber nur zur Diskussion dienen und keinen Einfluß auf den Ticketlebenslauf ausüben.

Aufgrund der Architekturabhängigkeit kann keine strikte Eingabeüberprüfung der Bewertungen stattfinden. Während Begutachter *X* zum Beispiel in Architektur *a* einen höheren Rang als Begutachter *Y* besitzt, kann dies in Architektur *b* umgekehrt sein. Deshalb wären Vetos von *X* auf Begutachtungen von *Y* in *a* gültig, jedoch nicht in *b*. Da der Bewertungsprozeß ein langwieriges Verfahren sein kann, können momentan unmögliche Aktionen durch Beförderungen plötzlich ermöglicht werden.

4.10. Richtlinien (*policies*)

In den vorangegangen Abschnitten wurde deutlich, daß Richtlinien eine große Rolle bei der Bewertung von Aktionen oder Eingabedaten besitzen. Viele – meist global gültige – Richtlinien sind starr im Code oder in statischen Konfigurationsdateien verankert und eine Änderung würde einen Neustart der entsprechenden Server erzwingen.

Um unterschiedliche QA-Richtlinien in den verschiedenen Entwicklungsphasen (zum Beispiel Beta-Test → nur noch kritische Fehlerbehebung, *String-Freeze* → kein Ändern der übersetzbaren Zeichenketten, ...) zu ermöglichen, wurde die Regeln verallgemeinert, und ihre Parameter werden zeitabhängig in der Datenbank gespeichert.

```
{
    ...
    # params for initial submissions
    'autobuild_new_rank'      :   RANK_COLONEL,
    'autobuild_new_time'      :     8*24*3600,

    # params for initial submissions
    'autobuild_update_rank'   :   RANK_LIEUTENANT,
    'autobuild_update_time'   :     4*24*3600,

    # which people are required to do an OPEN -> FROZEN transition
    'build_new_rule'      :   R('2*PRIVATE + MAJOR, CORPORAL + MAJOR, '
                                '2*LIEUTENANT'),
    'build_update_rule'   :   R('PRIVATE + LIEUTENANT, MAJOR'),
    ...
}
```

Abbildung 4.3.: Beispiel: Parameter einer Ticket-Richtlinie

Als Beispiel sei der in Abbildung 4.3 gezeigte Ausschnitt der Richtline für neue Tickets genannt. Mit diesen Parametern würde – wenn das Repository noch keine frühere Version des Paketes enthält – ein Person vom Rang „*Colonel* “ oder höher, und 8 Tage „Ruhezeit“ benötigt, um das Paket automatisch zu akzeptieren. Alternativ würden die Stimmen von zwei „*Lieutenants* “, oder einem „*Corporal* “ und einem „*Major* “ ausreichen, um das Paket sofort zu bauen.

Diese Parameter lassen sich über Einträge in der Datenbank beeinflussen, indem entsprechende Attribut–Wert Paare in eine zeitabhängige Parameter-Tabelle eingetragen werden.

5. Komponenten

Ausgehend von dem in Abbildung 5.1 dargestellten Paketfluß werden nachfolgend die jeweiligen Aktivitäten und Komponenten des Systems näher vorgestellt. Nicht eingezeichnet sind der „notifierd“ Daemon, welcher die unterschiedlichsten Benachrichtigungen beim Vollenden bzw. beim Fehlschlag gewisser Operation erzeugt, sowie der „stated“ Daemon, welcher im QA Prozeß eine Rolle spielt.

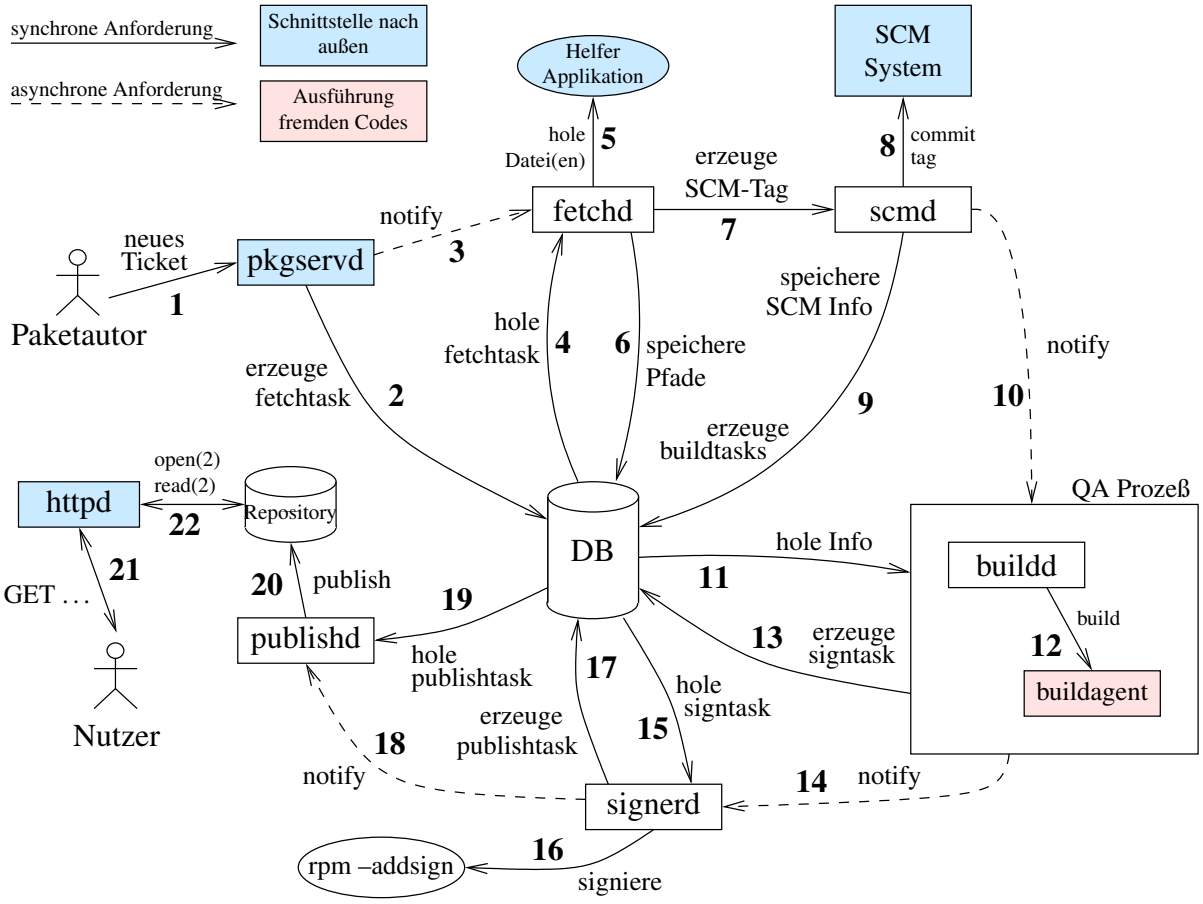

Abbildung 5.1.: Allgemeiner Paketfluß

Sofern eine Komponente als XML-RPC Server entworfen ist, werden folgende allgemeine

Befehle unterstützt:

`ping()`: Dies dient zur Überprüfung der Erreichbarkeit, und die einzig gültige Antwort ist „`pong`“.

`getNodeID()`: Dieser Befehl gibt einen Identifikator zurück, der eine Komponente eineindeutig im System identifiziert. Dieser Identifikator wird in der jeweiligen Konfigurationsdatei fest eingetragen und bleibt über Systemneustarts konstant.

5.1. Die Datenbank

Die Datenbank stellt eine zentrale Komponente dar. Abschnitt B.1 zeigt, daß beim verwendeten PostgreSQL sehr viel Zeit für den Verbindungsaufbau benötigt wird. Da vom Nutzer gewünschte Aktionen oftmals in mehrere kleine XML-RPC Kommandos zerlegt werden müssen, würde allein der Verbindungsaufbau mit der Datenbank Antwortzeiten im Sekundenbereich hervorrufen.

Beim Programmstart des „pkgservd“ Daemon, dem Kernstück der Nutzerinteraktion, wird ein Pool von Verbindungen zur Datenbank angelegt, welche jeweils für eine voreingestellte Anzahl von XML-RPC Kommandos genutzt werden. Obwohl geringe Antwortzeiten praktisch nur für diese Komponente relevant sind, werden die dort entwickelten Mechanismen zum Ansprechen der Datenbank auch auf die anderen Python-basierenden Server angewendet.

Zum Ansprechen der Datenbank wird eine leichte Abänderung das bei PostgreSQL mitgelieferten `pgdb` Modul genutzt. Diese Änderungen beinhalten das Erhöhen der Thread-Sicherheit sowie das Verfügbarmachen des von einigen SQL-Operationen zurückgelieferten `oid` Attributs.

5.2. Der “pkgservd” Daemon

Dieser Daemon ist das Kernstück der Nutzerinteraktion und ist als XML-RPC Server implementiert. Aufgrund des in Abschnitt 6.1 dargestellten Authentifizierungverfahrens und des kryptographischen Signierens einiger Anforderungen (siehe Abschnitt 6.3) ist dieser Daemon als *stateful* Server entworfen. Das heißt, daß einige, für nachfolgende Prozesse benötigte Daten im Server gehalten und nicht in jeder Anforderung neu übergeben werden müssen. Dies ist hauptsächlich aus Sicherheitsgründen nötig; bessere Performance oder Erleichterungen im Protokollentwurf waren nicht ausschlaggebend bzw. existieren nicht.

Innerhalb des gesamten Systems können mehrere „pkgservd“ Daemons mit unterschiedlicher Konfiguration nebeneinander existieren. So können zum Beispiel „gefährliche“ Befehle wie `resetSystem`, welches die gesamte Datenbank löscht und neu initialisiert, standardmäßig deaktiviert und nur auf einem mittels Firewall abgeschirmten Server verfügbar gemacht

werden. Ein weiteres Szenario sind getrennte Server für direkten XML-RPC Zugriff und für Zugriff über eine HTML Nutzeroberfläche.

Aufgaben:

- Verwaltung der in Kapitel 4 beschriebenen Datenbankobjekte unter Beachtung der Autorisierung des angemeldeten Nutzers. Abschnitt 6.2 zeigt, wie dies mittels Richtlinien erreicht wird.
- Annahme neuer Tickets und Begutachtungen; der Schritt „neues Ticket“ in Abbildung 5.1 wird beispielsweise durch die in Abbildung 5.2 dargestellte Kommunikation gelöst.
- Nutzerverwaltung, d.h. das Anlegen neuer Nutzer und die Beförderung existierender
- Ausgabe von QA Zwischenberichten wie Buildlogs, Differenzen zu vorhergehenden Versionen, Metriken u.ä.. Da diese Informationen Aufschluß über geheime Daten liefern können, ist auch hier eine Überprüfung der Autorisierung nötig.

5.3. Der “fetchd” Daemon

Der „fetchd“ Daemon lädt externe Daten, deren Standort vom Paketautor während der Ticketerstellung bekanntgegeben wurde, kanonifiziert diese und stellt sie in ein von allen Komponenten gemeinsam nutzbares Netzwerkfilesystem. Innerhalb des Systems sind mehrere „fetchd“ Server gleichzeitig möglich.

Die Reihenfolge der zu herunterladenden Daten wird allein durch den „fetchd“ Daemon bestimmt. Die vom „pkgservd“ gesendeten asynchronen Benachrichtigungen informieren nur darüber, daß neue Aufgaben bereitstehen, aber nicht welche.

Als Standort der Daten werden Quellcodeverwaltungssysteme, sowie HTTP/FTP Server unterstützt; der eigentliche Download geschieht mittels einer Hilfsanwendung. Als Eingabe werden bei den Quellcodeverwaltungssystemen die entpackten Dateien erwartet und bei HTTP/FTP ein SRPM-Quellpaket, welches von der Hilfsanwendung extrahiert wird.

Um die Auswirkungen eventueller Sicherheitslöcher in den genutzten Hilfsanwendungen möglichst gering zu halten, laufen diese Anwendungen unter einem eigenen UNIX Nutzeraccount. So können Konfigurationsdateien mit Paßwörtern für die Datenbank mittels restriktiven Dateirechten vor Ausspähen geschützt werden.

Implementierungsdetails: Der „fetchd“ Daemon ist als *stateless* Server konzipiert und nutzt die in Abschnitt 6.4 vorgestellten Mechanismen zur Verarbeitung der vom„pkgservd“ Daemon in die Datenbank gestellten *fetchtasks* Aufgaben.

```
C: request.init(<session-id>, "ticket.add",
                { "project" : "test", "branch" : "default"},
                { "uri"         : { "type"     :  "cvs",
                                    "cvsroot"  :  ":_predefined:fedora",
                                    "module"   :  "rpms/test",
                                    "revision" :  "test-0_1-1" },
                  "description" : { "version"  :  "0.1",
                                    "release"  :  "1",
                                    ... },
                  "repository"  : [ "i386-fc3-stable",
                                    "x86_64-fc3-stabe" ]})

S: return { "id"      : 1,
            "message" : "Request 20050130235012-P1_1261-0

   Project:  test (branch default)
   Version:  0.1-1
   ..." }

C: request.execute(<session-id>, 1, "-----BEGIN PGP SIGNED MESSAGE-----
   Hash: SHA1

   Request 20050130235012-P1_1261-0

   Project:  test (branch default)
   Version:  0.1-1
   ...
   -----BEGIN PGP SIGNATURE-----
   Version: GnuPG v1.2.6 (GNU/Linux)

   iQEVAwUBQW/wLmH8pResm0gjAQIzvwf/W+jqPPGsUSYQsJ1YHnPORztb9yYHRwmI
   ...
   -----END PGP SIGNATURE-----")

S: return 42
```

(C . . . Klient, S . . . „pkgservd“ Daemon)

Abbildung 5.2.: Beispiel: Einbringen eines neuen Tickets

Die momentan unterstützten Quellcodeverwaltungssysteme sind Subversion (SVN) und CVS, wobei für den Download die entsprechenden Kommandozeilentools verwendet werden; für das HTTP und FTP Protokoll wird `curl` genutzt. Das Starten des jeweiligen Programmes wird durch die in Abbildung 5.3 aufgeführte `_createFetchProcess()` Methode erledigt, welche dafür einen in C geschriebenen SUID Wrapper aufruft.

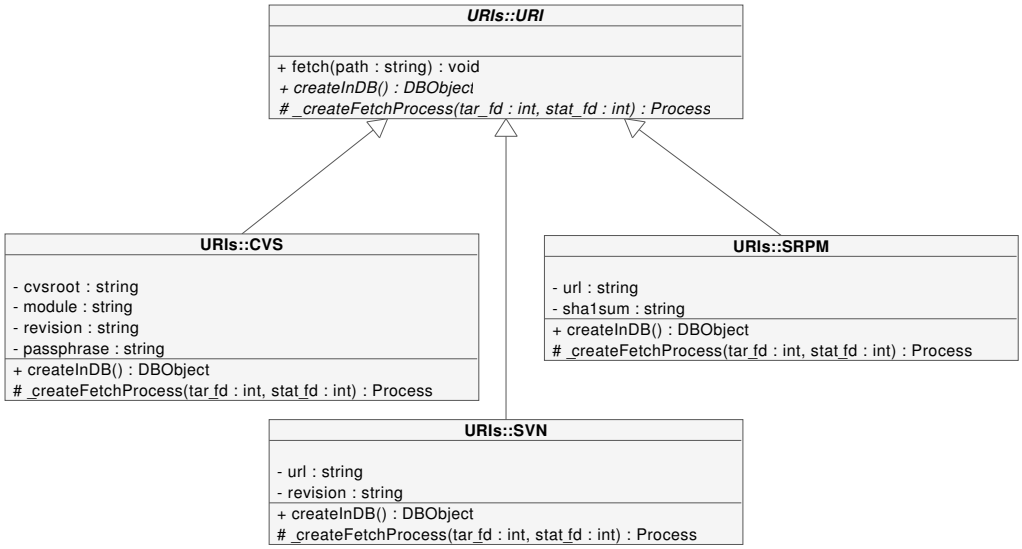

Abbildung 5.3.: Klassendiagramm: URI Hierarchie

Von dieser Methode wird erwartet, daß auf dem `tar_fd` Dateideskriptor ein Tar-Stream zur Verfügung gestellt wird, der die heruntergeladenen Daten enthält. Dadurch benötigt das potentiell gefährliche Helferprogramm keinen Zugriff auf das Netzwerkfilesystem.

Als weiterer Output werden auf dem `stat_fd` Filedeskriptor statistische Daten wie Anzahl der heruntergeladenen Bytes oder der benötigter Plattenplatz erwartet. Diese Daten werden zeilenweise als „*Schlüssel Wert*“ Folge übergeben. In zukünftigen Programmversionen können diese von der `TaskFactory::sortByPriority()` Methode (siehe Abbildung 6.9) zur Optimierung der Downloadreihenfolge herangezogen werden.

5.4. Das Quellcoderverwaltungssystem

Das Quellcodeverwaltungssystem (*sourcecode-management*, SCM) dient zur Verwaltung der in den SRPM-Paketen enthaltenen Dateien. Jede eingeschickte Version dieser Pakete wird gekennzeichnet, so daß Änderungen in neuen Versionen leicht bestimmbar sind und damit QA-Maßnahmen erleichtert werden. Außerdem kann externen Parteien die Pflege even-

tuell veränderter Pakete erleichtert werden, indem nur die Änderungen der neuen Pakete eingepflegt werden, anstatt die lokalen Änderungen erneut vorzunehmen.

Ausgehend von den Anforderungen des Systems ergeben sich folgende Kriterien für die Auswahl der konkreten SCM-Implementierung:

Lizenzierung: die Art des Projektes erlaubt es nicht, proprietäre Software zu verwenden, welche nur in Binärform verfügbar ist

Netzwerkunterstützung: die Kommunikation mit externen Klienten sollte möglichst firewallfreundlich geschehen. Deswegen ist ein HTTP-basierendes Protokoll erwünscht. Zum Wahrung NDA-geschützter Informationen sollte diese Kommunikation möglichst verschlüsselt erfolgen.

Feinkörniger Zugriffsschutz: es muß möglich sein, bestimmte Bereiche und Paketrevisionen nur für bestimmte Personen oder Gruppen zugänglich zu machen, da sich dort NDA-geschütztes Material befindet.

Effiziente Unterstützung großer Binärdaten: die in den SRPM-Paketen enthaltenen Dateien (insbesondere die Tar-Archive) liegen in Binärform vor und besitzen Größen im zweistelligen Megabyte-Bereich. Ihre Verwaltung durch das SCM-System sollte möglichst effizient erfolgen.

Dieser Punkte kann allerdings an Bedeutung verlieren, wenn Binärdaten nicht im SCM-System, sondern in einem besonderen Zwischenspeicher gehalten wird.

Ein solcher Zwischenspeicher wird vom aktuellen Entwicklungs-SCM-System von Fedora Extras genutzt; dazu wird statt den Tar-Archiven eine `sources` Datei in SCM-System aufgenommen, welche die Dateinamen und die zugehörige MD5-Summen beinhaltet. Aus den Informationen wird daraufhin eine HTTP-URL zusammengesetzt, unter welcher sich die tatsächliche Binärdatei befindet.

Vorteil dieser Methode ist, daß die Binärdaten als normale Dateien existieren und alte bzw. ungenutzte Versionen leicht identifiziert und entfernt werden können. Die Speicherung im SCM-System hat hingegen kaum Vorteile: durch das Komprimieren der Tar-Archive werden Ähnlichkeiten zwischen aufeinanderfolgenden Versionen komplett zerstört, so daß keine Platzeinsparung stattfinden kann. Der Zugriff und die Speicherung kann auch nicht schneller erfolgen, als dies bei einem auf der Platte vorliegendem Archiv der Fall ist.

Für andere Bereiche wichtige Aspekte wie Unterstützung von Datei- und Verzeichnisumbenennung, dem Löschen dieser Objekte oder das Verfügbarsein graphischer Nutzeroberflächen sind für dieses System weniger relevant, da dieses SCM-System existierende Pakete nur archiviert, aber nicht zu deren Entwicklung dient.

Ausgehend von [11] kommen folgende SCM-Systeme in die engere Wahl:

CVS: dies ist wohl das am meisten genutzte SCM-System für freie Software. Den oben gestellten Anforderung wird es allerdings nicht gerecht: Binärdaten werden nur ineffektiv gespeichert, Zugriffsschutz kann nur auf Datei-, nicht aber auf Revisionsebene eingerichtet werden und HTTP-basierender Transport ist nicht möglich.

GNU Arch: für dieses System existieren tla[1] und bazaar[2] als Implementierung. Das Ziel von GNU Arch ist die Erleichterung verteilter Entwicklung. Dies bedeutet, daß jeder in einem lokalen *Branch* eines Projekts entwickeln kann, ohne daß ihm Schreibzugriff auf das Hauptrepository gewährt werden muß. Ein Effekt von GNU Arch sollte sein, das bei der Linux Kernel-Entwicklung genutzte, proprietäre Bitkeeper abzulösen.

Die Datenspeicherung erfolgt bei GNU Arch auf beliebigen FTP oder HTTP Servern, so daß es eine sehr gute Netzwerkunterstützung bietet. Da das Erzeugen von *Branches* durch Erzeugen neuer Dateien und Verzeichnisse stattfindet, kann ein feinkörniger Zugriffsschutz durch Anwendung bekannter Konfigurationsoptionen (zum Beispiel `.htaccess`) erfolgen.

Subversion (SVN): dieses System wurde mit dem Ziel entwickelt, die Schwächen von CVS zu beseitigen, aber dennoch von CVS bekannte Verfahren beizubehalten und damit den Umstieg zu erleichtern. So ist SVN fast kommandozeilenkompatibel mit CVS; erlaubt aber Umbenennen von Dateien unter Beibehaltung der Änderungshistorie, geht effizient mit Binärdaten um und die Kommunikation findet entsprechend [4] entweder mittels eines eigenen Protokolls oder über WebDAV/DeltaV, einer HTTP-Erweiterung statt. In letzterem Fall wird SVN als ein Apache Modul (`mod_dav_svn`) betrieben, und es kann ein feinkörniger Zugriffsschutz durch das `mod_authz_svn` Modul erreicht werden.

Ein Nachteil von SVN war, daß die Daten in einer Berkeley DB4-Datenbank gespeichert wurden. Im Falle von Programm- oder Rechnerabstürzen bestand somit immer die Gefahr von Datenbankkorruption. Seit Version 1.1 steht das „fsfs“ Backend zur Verfügung, welches die Daten direkt im Dateisystem speichert.

Da eine Sprachanbindung an Python und positive Erfahrungen außerhalb dieses Projekts existieren, wurde SVN letztendlich als SCM-System ausgewählt.

5.5. Der “scmd“ Daemon

Dieser Server stellt eine XML-RPC basierende Schnittstelle zwischen den restlichen Komponenten und dem SCM-System bereit. Im Moment dient es nur zur Aufnahme eines Projektes, der Projektzweige und der durch Tickets beschriebenen Dateien ins SCM-System. Da diese Operationen in relativ kurzer Zeit durchführbar sind, wurde diese Komponente als *stateless* Server entworfen.

[1] `http://www.gnu.org/software/gnu-arch/`
[2] `http://bazaar.canonical.com/`

Als zukünftige Erweiterungen sind Funktionen für das Zeigen von Unterschieden zwischen zwei Paketversionen denkbar. Es bietet sich an, dafür externe Programme wie `viewcvs`[3] zu verwenden, da dort Features wie Syntaxhighlighting bereits enthalten sind. Bei einer Implementierung basierend auf den low-level Funktionen des SCM-Systems (zum Beispiel „`svn diff`") hätte die Repräsentation dieser Daten einen erheblichen Entwicklungsaufwand zur Folge.

Aufgrund des erforderlichen komplexen Zugriffsschutzes ist es nicht ratsam, die externen Programme dem Nutzer direkt zur Verfügung zu stellen. Die Gefahr, daß Zugriff auf geschützte Bereiche durch geschickte Manipulation der Anfragepfade erlangt werden kann, ist dafür zu groß. Stattdessen kann die Nutzerautorisation durch den „pkgservd" Daemon überprüft und bei Erfolg die Anfrage an den „scmd" Daemon gestellt werden.

Implementierungsdetails: Zur Ansteuerung an das eigentliche SCM-System stehen zwei grundsätzliche Möglichkeiten zur Wahl: die Nutzung externer Programme oder die Einbindung entsprechender Bibliotheken. Externe Programme haben den Vorteil, daß eventuelle Programmierfehler nur geringen Einfluß auf „scmd" hätten: bei Speicherzugriffsfehlern, die den Absturz des Programms zur Folge haben, wäre nur das Programm betroffen. Ein weiterer Vorteil sind das einfache Einrichten von Zugriffsrechte auf Betriebssystemebene durch SUID Wrapper.

Als problematisch erweist sich bei externen Programmen hingegen die Interpretation des Ausgabe oder das Ausführen ungewöhnlicher Aktionen. Beispielsweise wird zur *Tag*-Erzeugung die beim „`svn commit`" vergebene Revisionsnummer benötigt. Die dort verwendete Ausgabe ist jedoch völlig unspezifiziert und nur bedingt maschinenlesbar. Deshalb nutzt der „scmd" Daemon die von Subversion bereitgestellte Python-API Bibliothek.

5.6. Der "stated" Daemon

Diese Komponente stellt – wie Abbildung 5.4 zeigt – einen zentrale Schaltstelle im QA-Prozeß dar. Beim Eintreffen neuer Bewertungen, was durch „notify" Befehle seitens des „pkgservd" oder „buildd" geschehen kann, werden offene Tickets analysiert und entsprechend Abbildung 4.2 in einen neuen Status überführt.

5.7. Der Build-Agent

Der eigentliche Paketbau wird von Build-Agents durchgeführt, welche – wie in [22] dargestellt – folgende zwei Anforderungen zu bewältigen haben:

- Das Bauen von Binärpaketen

[3] `http://viewcvs.sourceforge.net`

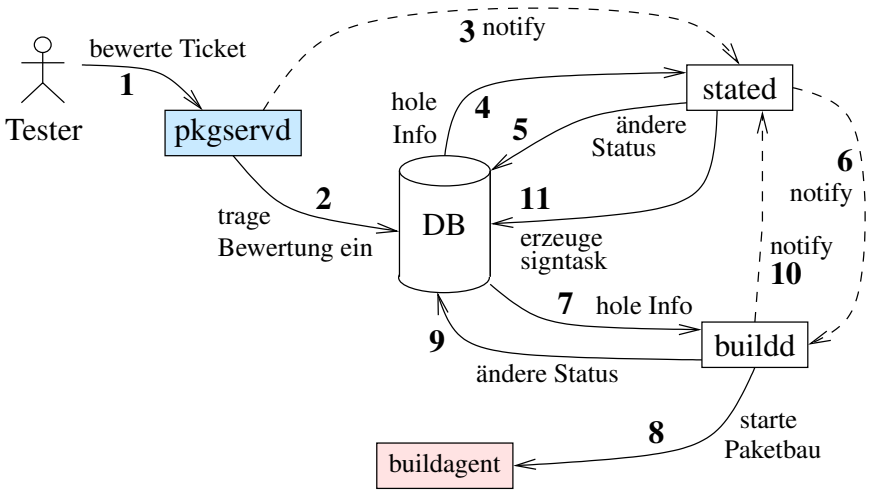

Abbildung 5.4.: Ablauf des QA-Prozesses

- Das Standhalten gegenüber Angriffen

Anders als die übrigen Komponenten, welche auf der selben physikalischen Maschine laufen können, führt der Build-Agent architekturspezifische Programme aus, und es wird deshalb mindestens ein Rechner pro CPU-Familie benötigt. Virtualisierungslösungen wie qemu[4] zur Emulation anderer CPUs sind zwar prinzipiell möglich, erhöhen den, bei vielen Paketen ohnehin schon großen Zeitbedarf für den Paketbau aber entscheidend.

5.7.1. Paketbau

Entsprechend den in Abschnitt 2.2 gemachten Annahmen muß es möglich sein, in einem Repository mit der Paketmenge P für ein Quellpaket mit einer Abhängigkeitsmenge A sowohl den Paketbau $f(A)$ als auch $f(P)$ durchzuführen und beide Ergebnisse zu vergleichen. Dabei eröffnen sich folgende Probleme:

- A entspricht nur selten bzw. nie genau den durch „`BuildRequires:`“ beschriebenen Paketen. Stattdessen besitzen diese wiederum Abhängigkeiten, so daß Abhängigkeitenauflöser eingesetzt werden müssen.

  In RPM-Umgebungen sind solche Programme „apt“[5], „smart“[6], „yum“[7] und in begrenztem Maße „rpm“ selbst.

[4] `http://fabrice.bellard.free.fr/qemu`
[5] `http://moin.conectiva.com.br/AptRpm`
[6] `http://smartpm.org/`
[7] `http://linux.duke.edu/yum/`

- In den „`BuildRequires:`“ können sowohl echte als auch virtuelle Paketnamen, sowie Dateinamen angegeben sein. Letztgenannte Dateinamen können durch den unter Fedora verfügbaren „apt“ und „yum“nicht aufgelöst werden, so daß diese Programme dafür nicht direkt anwendbar sind.

  Mittels des „`rpm -qf datei`“ Befehls ist es zwar möglich, den benötigten Paketnamen zu bestimmen. Praktisch besteht aber das Problem, daß dafür ein Eintrag in der rpm-Datenbank erforderlich ist, der bei nicht-installierten Paketen fehlt.

- Insbesondere dann, wenn virtuelle Paketnamen involviert sind, besteht die Möglichkeit, daß Abhängigkeiten mehrdeutig sind. Als Beispiel kann die Abhängigkeit „`smtpdaemon`“ dienen, welche sowohl vom `sendmail` als auch vom `exim` oder `postfix` Paket erfüllt wird.

Praktisch bietet es sich an, den Paketbau in einer Umgebung durchzuführen, die unabhängig vom Basissystem des genutzten Rechners ist. Das direkte Nutzen des Basissystems hat den Nachteil, daß Builds für instabile oder sehr alte Fedora Core Versionen die dort verwendeten Basispakete wie glibc oder python installieren würden, mit denen der Build-Agent ansich aber eventuell nicht lauffähig ist.

5.7.2. Angriffe während des Paketbaus

Ein Teil des Paketbaus $f(A)$ besteht darin, die Paketmenge A auf dem Build-Agenten zu installieren. Diese Installation erfolgt mit Administratorrechten und muß in A enthaltene Installationsskripte mit diesen Rechten ausführen. Obwohl die Qualitätskontrolle solche Skripte auf offensichtliche Angriffe überprüfen sollte, können komplexere Attacken nie ausgeschlossen werden, da nicht nur Standardprogramme, sondern auch verpackte Software in den Skripts ausgeführt werden kann. Da die Quellen nicht vollständig überprüft werden können, sind so Angriffe seitens des Autors der verpackten Software möglich.

Eine weitere Attacke stellt die Installation präparierter Programme an durch `$PATH` bevorzugten Stellen statt. So würde beispielsweise ein `/bin/make` statt `/usr/bin/make` bei einem `make`-Aufruf genutzt und könnte Schadfunktionen ausführen.

Außerdem können Hintergrundprozesse gestartet werden, welche auf nachfolgende Builds warten und dort Prozesse oder Dateien beeinflussen. Eine solche Beeinflussung könnte beispielsweise das Einbringen bosärtigen Codes an den Beginn jeder `main()` Routine sein. Neben der Manipulation fremder Builds besteht auch die Gefahr des Ausspähens von NDA-geschützten Informationen.

Natürlich sind einige dieser Angriffe auch beim Nutzer während der regulären Paketinstallation möglich. Das hier vorliegende Buildsystem ist davon jedoch besonders betroffen, da durch Veränderung des Toolsets (Compiler, Basis-Werkzeuge, ...) oder mittels Hintergrundprozesse in jedes später, in der selben Umgebung gebaute Paket bösartiger Code eingeschleust werden kann.

Während des Paketbaus ansich wird ebenfalls Code unkontrollierbar ausgeführt. Obwohl dies als nicht-privilegierter Nutzer geschieht, können aber Hintergrundprozesse gestartet werden, die den nächsten, unter der selben Nutzer-ID laufenden Paketbau wie oben beschrieben, beeinflussen. Ebenso könnten Rechner des internen Netzwerkes angegriffen und bei Außenanbindung unerwünschte Programme (Warez-Server, DDOS Klient) gestartet werden.

Schlußfolgerungen:

1. Die Build-Agents müssen durch eine restriktive Firewall abgeschirmt werden.
2. Es muß sichergestellt werden, daß ein nachfolgender Paketbau in einer reinen Umgebung abläuft. Dazu gibt es zwei Möglichkeiten:
   a) die Umgebung wird jedesmal neu erstellt; je nach verwendeter Technik ist dies ein zeitaufwendiger Prozeß.
   b) die Umgebung wird nach jedem Paketbau auf Unregelmäßigkeiten hin untersucht. Beispielsweise könnten Dateilisten und Checksummen vor und nach dem Bau genommen und vergleichen werden. Nachteil dieser Methode ist, daß Ausnahmen zu beachten sind, auf denen wiederum Angriffe aufbauen können.

   Beispielsweise wird `/etc/ld.so.cache` von vielen Installationsskripten modifiziert, ebenso, wie die rpm-Datenbank vom Installationsprozeß selbst. Da diese Dateien normalerweise nur vom Administrator geändert werden können, besteht die Gefahr, daß in den genutzten Parsern die Eingabeüberprüfung nicht besonders ernst genommen wurde und sich Exploits konstruieren lassen, die außerhalb von Build-Umgebungen nicht relevant sind.

   Da sich nach Meinung des Autors Punkt 2b nie mit hundertprozentiger Sicherheit implementieren läßt, sollte Punkt 2a zur Anwendung kommen und die Build-Umgebung jedesmal neu erzeugt werden.
3. Prozesse eines Paketbaus dürfen keinen Zugriff auf Prozesse oder Dateien haben, die nicht zum aktuellen Paketbau gehören. Dies beinhaltet sowohl vergangene, zukünftige als auch eventuell parallel ablaufende Operationen.
4. Direkter Hardwarezugriff muß unmöglich sein; beispielsweise könnten Änderungen der Lüftersteuerung zur physikalischen Hardwarezerstörung führen, ebenso wie sehr häufiges Hoch- und Herunterfahren des Plattenmotors.

5.7.3. Technologien

Zur Implementierung der Build-Agents kommen nachfolgende Technologien in Betracht:

Physikalische Rechner, die für jeden Paketbau neu initialisiert werden. Diese Initialisierung kann durch Kickstart oder Plattenimages erfolgen. Der Zeitaufwand beträgt aber allein schon wegen des dafür notwendigen, mehrmaligen Bootvorganges mehrere Minuten, und es besteht die Gefahr direkten Hardwarezugriffs.

chroot-Umgebungen, d.h., mittels des chroot(2)-Syscalls abgeschirmte Ausführungsumgebungen. Eine solche Abschirmung kann allerdings mit einfachsten Mitteln[8] durchbrochen werden und würde sämtliche oben genannten Angriffsformen erlauben. Nichtsdestotrotz, ist diese Art des Paketbaus sehr populär; ein Programm, welches die nötigen Aufgaben wie Abhängigkeitenauflösung erledigt, ist das von Thomas Vander Stichele geschriebene Tool „mach“[9].

Eventuell läßt sich Sicherheit durch SELinux[10] erhöhen; der Betrieb von SELinux in chroot-Umgebungen gestaltet sich aber schon allein wegen der gewählten Schnittstelle zum Linux-Kernel als problematisch, weil diese ein proc-Filesystem voraussetzt, welches in chroot-Umgebungen nicht unbedingt existieren muß.

Außerdem haben jüngste Ereignisse[11] gezeigt, daß der Linux-Kernel sehr verwundbar gegenüber sogenannten *local-root-exploits* ist, mit denen der durch SELinux gegebenen Schutz durchbrechen werden kann.

Virtualisierungstechnologien, d.h. Methoden, welche einen oder mehrere virtuelle Rechner auf einem physikalischen Rechner emulieren. Das gegenwärtige, für fedora.us eingesetzte Buildsystem nutzt beispielsweise Linux-Vserver[12] in Verbindung mit einem angepaßten „mach“.

Zumindest bei User-Mode-Linux (UML), Linux-Vservern und optimierten Varianten von QEMU[13] besteht aber ebenfalls eine Anfälligkeit gegenüber ausnutzbaren Fehlern im Linux-Kernel. Inwieweit andere Technologien wie Xen[14] davon betroffen sind, muß noch untersucht werden; genauso wie die Möglichkeiten, diese für Paketbuilds einzusetzen.

5.8. Der “signerd” Daemon

Dieser Server implementiert ebenfalls das in Abschnitt 6.4 gezeigte Verfahren und signiert die in der `signtasks`-Tabelle aufgeführten Objekte. Momentan sind dafür nur die vom

[8] zum Beispiel mittels einer „`chroot("/bin"); chroot("../..");`“ Sequenz
[9] `http://thomas.apestaart.org/projects/mach`
[10] `http://www.nsa.gov/selinux/`
[11] vergleiche Links in `https://rhn.redhat.com/errata/RHSA-2005-092.html`
[12] `http://www.linux-vserver.org`
[13] `http://fabrice.bellard.free.fr/qemu`
[14] `http://www.sourceforge.net/projects/xen/`

„buildd“ erzeugten Quell- und Binärpakete vorgesehen; das Signieren erfolgt dafür durch ein externes `rpm --addsign` Kommando.

Als zukünftige Erweiterung ist das Signieren der vom „notifierd“ erzeugten E-Mails denkbar. Aus Sicherheitsgründen sollte dieser Daemon in einer abgeschirmten Umgebung auf einem eigenen physikalischen Rechner laufen.

5.9. Der “notifierd” Daemon

Obwohl in Abbildung 5.1 nicht eingezeichnet, stellt diese Komponente einen zentralen Baustein dar, welcher von den meisten anderen Servern genutzt wird. Sein Zweck besteht in der Kommunikation mit der Außenwelt; im Detail sind seine Aufgaben:

- Erzeugen von Benachrichtigungen beim Hinzufügen, dem Ändern oder dem Löschen von Datenbank-Objekten.
- Weiterleiten von QA-Bewertungen, Meldungen des Buildsystems oder Informationen über Statusübergänge der Tickets an die jeweiligen Personen
- Einige Aufgaben der Nutzerverwaltung wie das Erzeugen von Bestätigungsmail bei der Nutzererstellung oder Erinnerungsmails bei vergessenen Paßworten.

Als klassisches Ausgabemedium steht E-Mail zur Verfügung, aber auch RSS-Feeds können für einige Anwendungsfälle gewünscht sein. Beim Erstellen von E-Mails ist insbesondere auf Standardkonformität zu achten: E-Mail-Header dürfen ausschließlich Zeichen aus dem US-ASCII Alphabet, aber keine Umlaute enthalten [16, Abschnitt 2.3.1], und beim Entwurf des Textkörpers müssen gegebenenfalls entsprechende MIME-Header erzeugt werden [12]. Glücklicherweise wird die dafür benötigte Funktionalität vollständig vom Python `email`-Modul zur Verfügung gestellt.

Durch Vergabe einheitlicher `Message-ID:`-Header und dem Hinzufügen entsprechender `References:`- bzw. `X-In-Reply-To:`-Header läßt sich der Begutachtungsprozeß in vielen Mailprogrammen als Baumstruktur visualisieren.

Anmerkungen: Die momentane Implementierung verwendet zum Erzeugen der Nachrichten eine hierarchische Repräsentation des Textes mittels der in `Util::message` definierten Klassen. In der Praxis erweist sich dieses Modell als zu starr, da Änderungen des Textes eine Code-Änderung und den Neustart des Servers benötigen. Es sollte deshalb auf eine Technik mit externer Speicherung der Nachrichtengerüste umgestiegen werden. Bei der Realisierung ist insbesondere darauf zu achten, daß Textnachrichten nach 70–75 Spalten umgebrochen werden.

Zukünftige Programmversionen sollten es außerdem ermöglichen, Nachrichten kryptographisch mittels PGP/MIME [8] zu signieren. Für diese Funktionalität könnte der „signerd“

Daemon eingesetzt werden. Insbesondere bei den Nutzerverwaltungsaufgaben würden Signaturen sogenannte *Phishing*-Attacken erschweren.

Die Behandlung NDA-geschützter Informationen ist ein weiterer Problembereich, der in zukünftigen Programmversionen adressiert werden muß. Üblicherweise wird über neue Tickets auf öffentlichen Mailinglisten informiert, die im „notifierd“ Daemon als zusätzliche Empfänger konfiguriert sind. Bei geschützten Informationen muß dieser Mechanismus ausgeschaltet werden, und die E-Mail darf nur an autorisierte Personen geschickt werden. Idealerweise sollte dies verschlüsselt erfolgen.

6. Subsysteme

6.1. Authentifizierung

Wie in Abbildung 6.1 demonstriert, erfolgt das Anmelden ans System in zwei Schritten:

1. Erzeugung einer neuen Session, welche durch eine eineindeutige, nicht vorhersagbare Zeichenkette identifiziert wird.

2. Authentifizierung dieser Session gegenüber dem System.

Nachfolgende Operationen nutzen stets den unter Schritt 1 erzeugten Sitzungs-Identifikator, und die Anmeldedaten sind nicht mehr neu zu übertragen. Dieses Verfahren erhöht auf der einen Seite die Sicherheit, da die Anmeldeinformationen nur einmal auf Klientenseite genutzt werden und danach verworfen werden können. Außerdem wird dadurch das mehrfache Ausführen der eventuell ressourcenhungrigen oder komplexen Authentifizierung vermieden.

```
## Schritt 1: Session-Erzeugung
session  = server.newSession()
  # --> session = '0a27c6735d6e4d8f6826de94cb7b1edf'

## Schritt 2: Anmeldung
server.authenticate.simple(session, '_ruth','foo')

## Ausfuehrung privilegierter Operationen
server.resetSystem(session)
```

Abbildung 6.1.: Beispiel: Anmeldung ans System

Es ist offensichtlich, daß Sorgfalt bei der Übertragung der in Schritt 2 genutzten Daten und des Session-Identifikators anzuwenden ist, was nur durch Verschlüsselung der Verbindung erreicht wird. Bei Wahl des Verfahrens ist darauf zu achten, daß gleichzeitig integritätssichernde Maßnahmen wie *message authentication codes* (MAC) Anwendung finden.

Für das in Schritt 2 genutzte Authentifizierungsverfahren sind folgende Szenarios möglich:

- Der Nutzer meldet sich mit einem Login–Paßwort Paar an, welches vom System verwaltet wird. Dazu steht die `authenticate.simple` XML-RPC Funktion zur Verfügung, welche eine einfach Datenbank-Abfrage durchführt. Wie der Name schon andeutet, ist dies eine sehr einfach zu implementierende Methode.

Als Erweiterung in zukünftigen Programmversionen, kann zum Schutz der gespeicherten Paßwörter die entsprechende Datenbanktabelle nur für bestimmte Nutzer verfügbar gemacht werden, so daß „pkgservd“ oder andere Komponenten keinen direkten Zugriff darauf hätten. Die eigentliche Authentifizierung oder auch Paßwortänderungen müßten dann durch ein separates, mittels SUID-Wrapper gestartetem Programm stattfinden, welches die Anmeldeinformationen (Login und Paßwort) über eine Pipe erhält. Da die Anmeldung nur einmal pro Sitzung geschieht, sollten die Auswirkungen auf die Performance nicht bemerkbar sein.

- Der Nutzer meldet sich mit einem Login–Paßwort Paar an, welches von einem externen System wie Bugzilla verwaltet wird. Der Vorteil dieses Verfahrens ist, daß die selben Logindaten für miteinander verwandte Systeme gültig sind. Außerdem kann etwas Codierungsaufwand gespart werden, indem für Aufgaben wie Paßwortänderung das bereits existierende System von Bugzilla genutzt wird.

  Auch hier gilt, daß zur Erhöhung der Sicherheit ein externes Programm die eigentliche Authentifizierung durchführen sollte.

- Die Anmeldung am XML-RPC Server erfolgt mittels bekannter HTTP Authentifizierungsmethoden wie

  - HTTP-Authentifizierung [10, Abschnitt 11]
  - klientseitigen SSL/TLS Zertifikaten
  - GSSAPI/SPNEGO [2], was aber nur in abgeschlossenen Netzen sinnvoll ist; die unter Linux verfügbaren Kerberos5 Implementierungen erlauben keinen Einsatz in Netzwerkumgebungen mit *Network Address Translation* (NAT) oder restriktiven Paketfiltern.

  Vorteil der letztgenannten Methoden ist, daß die Anmeldung paßwortlos geschieht.

- Der Nutzer meldet sich an einer vertrauenswürdigen Nutzeroberfläche an, welche den Nutzernamen ans System weiterleitet. Die Anmeldung an der HTML-Nutzeroberfläche kann mit den im vorigen Punkt genannten Verfahren realisiert werden.

  Weiterhin ist denkbar, daß speziell präparierte Nachrichten aus vertrauenswürdigen Entwicklungs-Quellcodemanagement-Systemen Informationen über den Urheber enthalten und die Aktionen unter dessen Account ausgeführt werden. Als Problem erweist sich, daß mit diesem Verfahren das in Abschnitt 6.3 beschriebene Signieren nicht funktioniert.

6.2. Grundlegende Operationen auf Datenbankobjekten

Wie in Abschnitt 5.2 beschrieben, dient der „pkgservd“ Daemon zur Verwaltung einiger Datenbankobjekte. Die vier grundlegenden Operationen für solche Objekte lassen sich verall-

gemeinern und sind in der abstrakten `Dispatcher::DbDispatcher` Klasse implementiert. Dieser Abschnitt gibt einen Überblick darüber.

Gemeinsam ist allen, daß für jeden verwendbaren Objekttyp Subklassen für den entsprechenden Dispatcher, das Datenbank-Objekt und die Richtlinien implementiert werden müssen.

Die Objekterzeugung (*create*) ist in Abbildung 6.2 dargestellt. Die `washCreateData()` Methode des Datenbankobjekts dient dazu, die in der XML-RPC Anfrage übermittelten Daten in eine einheitliche Form zu bringen. Typischerweise werden alle nicht-modifizierbaren Attribute (zum Beispiel automatisch erzeugte Sequenz-Nummern) dabei entfernt.

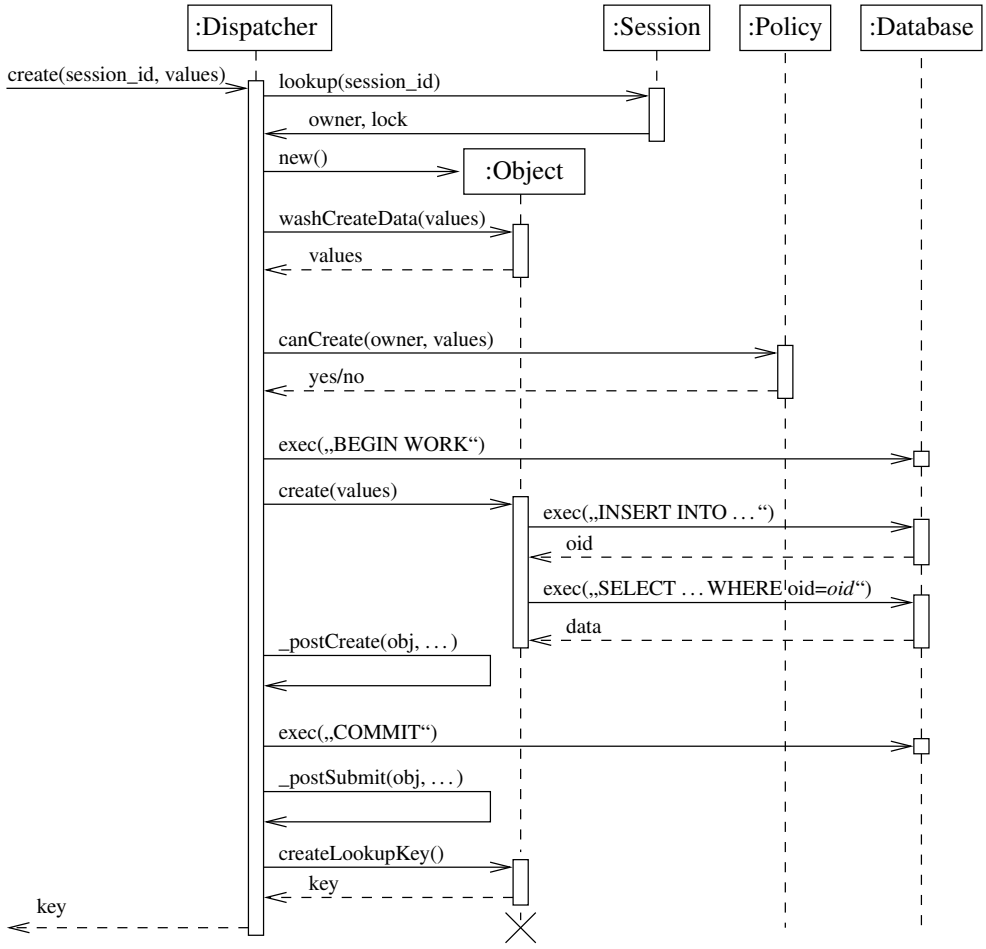

Abbildung 6.2.: Sequenzdiagramm: Dispatcher::create

Die so gereinigten Daten werden einem Richtlinientest unterzogen, das heißt, es wird

getestet, ob der Nutzer autorisiert ist, ein Objekt mit diesen Daten zu erzeugen. Im Falle des Verbots wird eine `PolicyError` Exception ausgelöst.

Bei Erfolg wird eine Datenbanktransaktion begonnen, das Objekt in die Datenbank eingetragen und (inklusive implizierter Default-Werte) wieder ausgelesen. Dieses Eintragen nutzt die Besonderheit von PostgreSQL, daß bei einer `INSERT` SQL-Operation der Identifikator der neu erzeugten Zeile zurückgeliefert wird. So kann das neue Objekt beim `SELECT` einfach adressiert werden.

Einige Datenbankobjekte stehen mit anderen in einer $1 : n$ mit $n \geq 1$ Beziehung, das heißt, daß während der Objekterzeugung die Erzeugung anderer Objekte möglich sein muß. Ein Beispiel sind Nutzer und ihre Authentifizierungsdaten: das Anlegen ausschließlich des Nutzers hätte den Effekt, daß der Nutzer zwar in der Datenbank existiert, sich danach aber nicht anmelden kann, weil die Authentifizierungsdaten noch nicht existieren. Vorheriges Eintragen dieser Daten ist nicht möglich, da sie stets mit genau einem Nutzer über Fremdschlüssel (`FOREIGN KEYS`) verbunden sein müssen.

Aus diesem Grund wird durch eine überladbare `_postCreate()` Methode die Möglichkeit bereitgestellt, objektspezifische Zusatzeinträge in der Datenbank vorzunehmen. Dieser Methode werden eventuelle Zusatzparameter des XML-RPC `create()` Befehls weitergegeben, welche zum Beispiel die Authentifizierungsmethode und das Paßwort beinhalten.

Wenn beim Ausführen dieser Schritte ein Fehler auftritt, wird eine Exception ausgelöst und die Datenbanktransaktion rückgängig gemacht (`ROLLBACK`). Ansonsten wird die Transaktion beendet und die Daten eingetragen, so daß sie für andere Komponenten sichtbar sind. Deshalb wird auch erst jetzt eine überladbare `_postSubmit()` Methode aufgerufen, welche andere Komponenten über die vorgenommenen Änderungen informiert und entsprechende Aktionen auslöst.

Der Rückgabewert der `create`-Operation ist ein Schlüssel, mit dem das erzeugte Objekt eindeutig identifiziert werden kann.

Die Objektsuche (*lookup*) nutzt in einem ersten Schritt die in Abbildung 6.3 dargestellte `_lookupObject()` Methode. Diese bringt den Suchschlüssel in eine kanonifizierte Form, was nötig ist, da XML-RPC die Übertragung von Werten wie `None` oder assoziative Felder mit Nicht-Zeichenketten als Schlüssel nicht unterstützt.

Die so erhaltenen Schlüssel werden dahingehend untersucht, ob der aktuelle Nutzer sie zur Objektsuche verwenden darf. Zum Beispiel wäre es unvorteilhaft, wenn die Suche nach Authentifizierungsobjekten mittels Paßwort-Schlüssel erlaubt wäre. Das Anzeigen der Daten könnte später zwar verhindert werden; durch Unterscheidung der möglichen Fehlermeldungen („Objekt nicht gefunden“ versus „Richtlinienfehler“) könnte aber überprüft werden, ob ein Paßwort verwendet wird und dieses später gegen alle Nutzer getestet werden. Durch Verbieten gewisser Anfragen sind solche Rückschlüsse nicht möglich.

Nun wird die Datenbank nach dem gewünschten Objekt durchsucht, die Daten geladen und überprüft, ob sie für den aktuellen Nutzer sichtbar sind. Im Erfolgsfalle wird daraufhin –

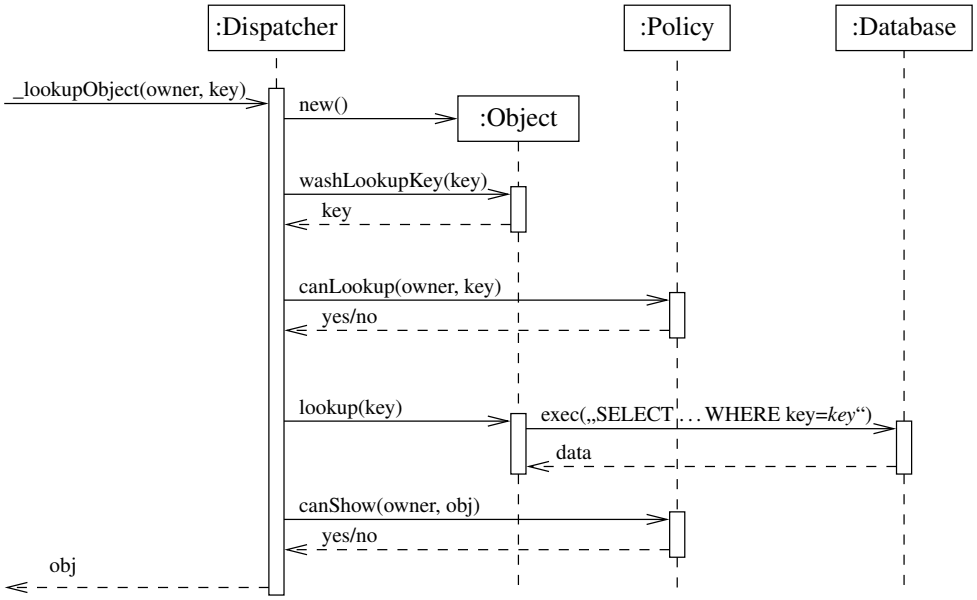

Abbildung 6.3.: Sequenzdiagramm: Dispatcher::_lookupObject()

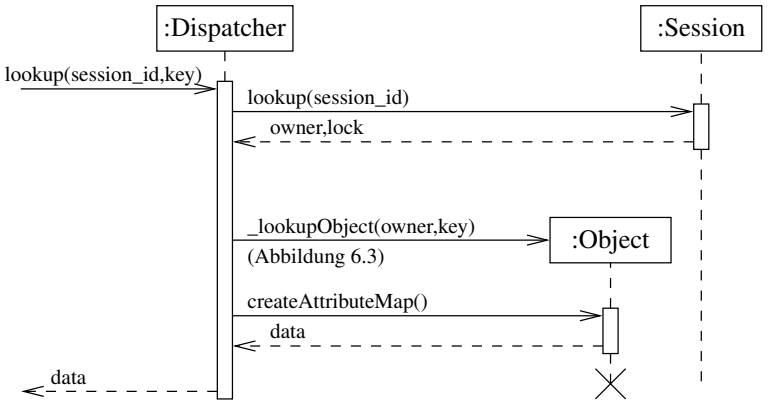

Abbildung 6.4.: Sequenzdiagramm: Dispatcher::lookup()

wie in Abbildung 6.4 dargestellt – ein assoziatives Feld mit den Objektdaten zurückgeliefert. Je nach Objekt und Richtlinie können daraus aber noch einige Attribute entfernt werden.

Die Objektänderung (*update*) nutzt in einem ersten Schritt ebenfalls die oben schon erwähnte `_lookupObject()` Methode zur Bestimmung des betroffenen Objektes. Danach werden – ähnlich der `create`-Operation – die neuen Daten in eine kanonische Form gebracht, unveränderte Werte ausgefiltert und auf Richtlinienverträglichkeit hin untersucht.

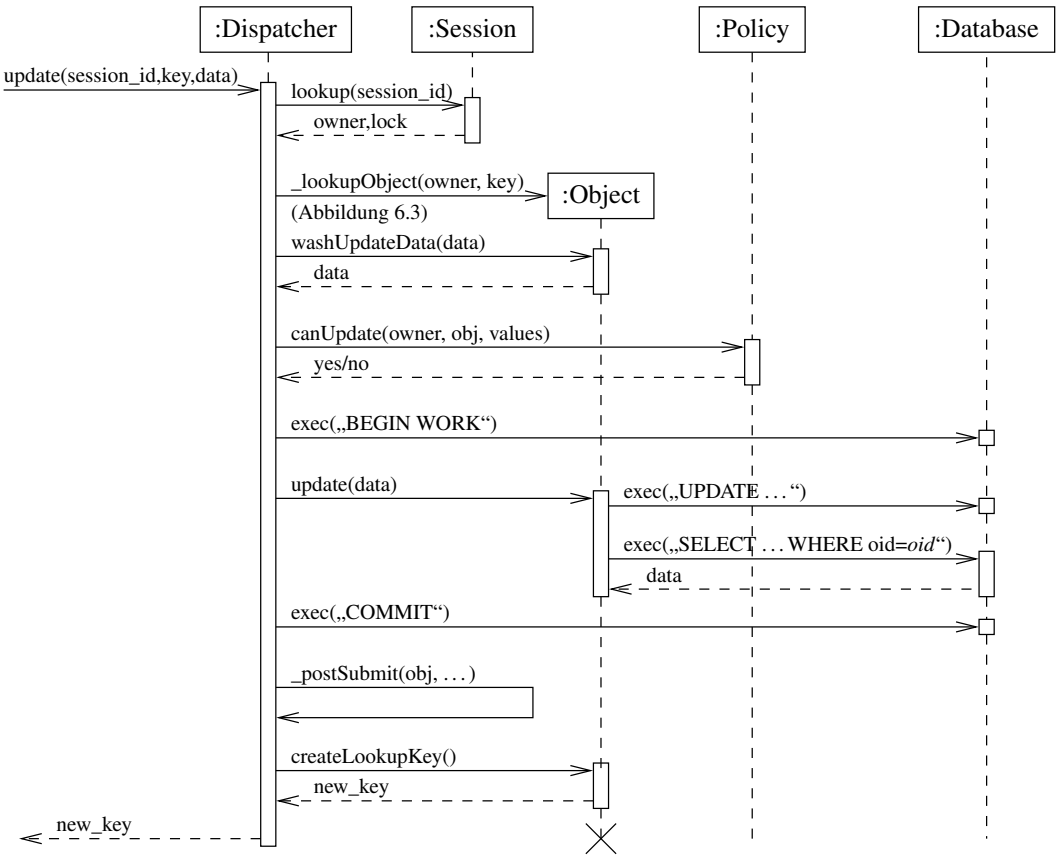

Abbildung 6.5.: Sequenzdiagramm: Dispatcher::update()

Danach erfolgt das Erzeugen einer neuen Datenbanktransaktion, das Ausführen der eigentlichen `UPDATE`-Operation und die Aktualisierung der Daten. Da `_lookupObject()` neben den eigentlichen Daten auch den unveränderlichen Identifikator des Datensatzes zurückliefert (`oid`), ist die Aktualisierung ohne größeren Aufwand durchführbar.

Nun wird die Transaktion beendet, was die Daten global sichtbar macht, eventuell andere Komponenten mittels `_postSubmit()` über die Änderung informiert und ein Schlüssel

zurückgeliefert, der das neue Objekt eindeutig identifiziert.

Die Objektentfernung (*delete*) verläuft – wie in Abbildung 6.6 dargestellt – mit angepaßten Richtlinienchecks und SQL-Befehlen analog der Objektänderung. An dieser Stelle ist anzumerken, daß diese Operation meist Administratoren vorbehalten ist, da die meisten Objekte durch Fremdschlüssel mit anderen verbunden sind und ein Entfernen eine Kettenreaktion zur Folge hätte.

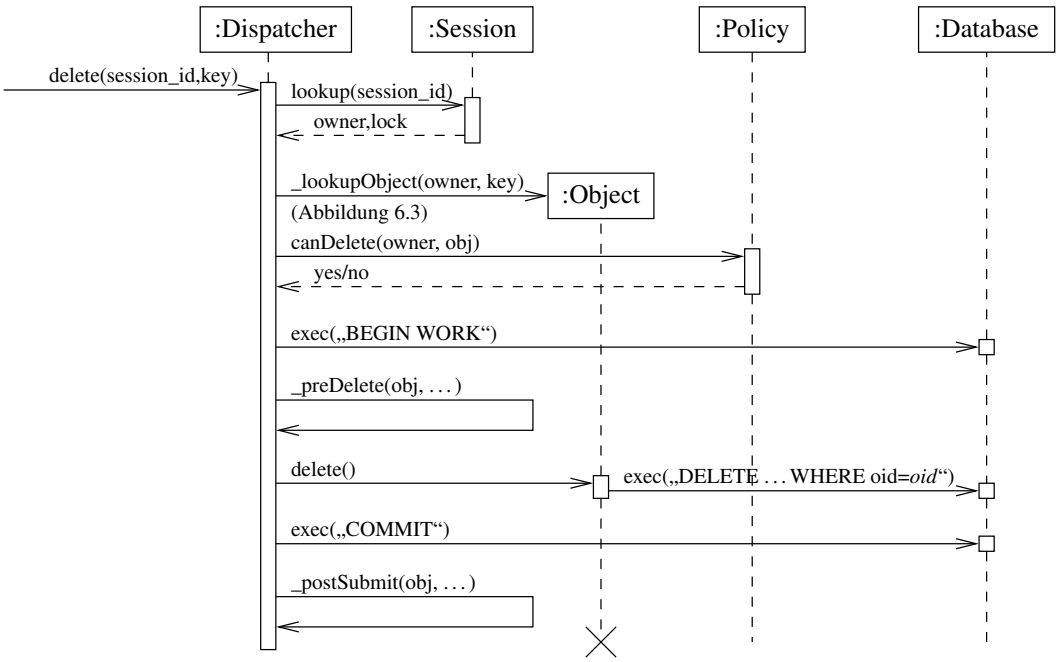

Abbildung 6.6.: Sequenzdiagramm: Dispatcher::delete()

6.3. Mehrstufige Anfragen

Zur sicheren Authentifizierung und späterer Nachvollziehbarkeit sind einige Anforderungen kryptographisch zu signieren. Dabei sind Replay-Attacken zu vermeiden, das heißt, es muß für einen Angreifer unmöglich sein, eine bereits signierte Anforderung nochmals ins System zu bringen und Aktionen damit hervorzurufen. Aus diesem Grund wurde das in Abbildung 6.7 dargestellte, zweistufige Verfahren implementiert.

Konstruktion einer eindeutigen Nachricht: Im ersten Schritt wird – ausgehend von der Art der Anfrage – eine einzigartige Nachricht von der „`Request::getMessage()`“ Metho-

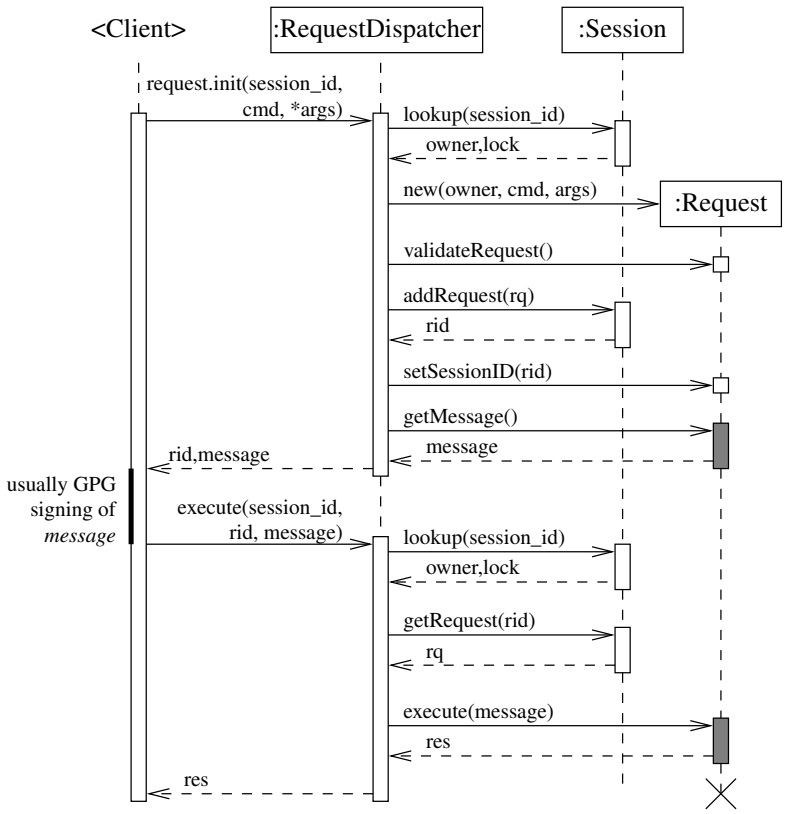

Abbildung 6.7.: Sequenzdiagramm: Dispatcher::Request

de generiert; ein Beispiel dafür wurde in Abbildung 5.2 angedeutet. Die Einzigartigkeit wird garantiert, indem in der Nachricht ein Bezeichner wie „`20050130235012-P1_1261-0`“ aufgenommen wird, der besteht aus:

- Datum und Zeit mit Sekunden-Auflösung
- dem Identifikator der Komponente, welcher genau einmal im System vorkommt
- der Prozeß-ID (PID)
- einem sich ständig erhöhenden Zähler

Mehrdeutigkeiten können bei diesem Verfahren nur auftreten, wenn:

- das Datum- und Zeitfeld überläuft, was nach 10000 Jahren eventuell passiert, oder
- die Uhrzeit inkorrekt ist, oder
- innerhalb einer Sekunde die selbe PID mehrfalls auftritt und aufeinanderfolgend gestarteten Instanzen der selben Komponente zugeordnet wird.

Letztgenannter Fall kann unter Umständen eintreten, wenn schon sehr viele andere Prozesse existieren und deshalb nur noch wenige PIDs verfügbar sind. Deshalb wird beim Programmstart eine Pause von 2 Sekunden eingelegt.

Eine alternative Möglichkeit zur Konstruktion der Identifikatoren könnte über Zähler in der Datenbank realisiert werden. Dies würde die Länge der Zeichenkette erheblich reduzieren und wäre auch bei falscher Uhrzeit zuverlässig; hätte aber etwas mehr Implementierungsaufwand zur Folge. Als Problem kann sich bei diesem Verfahren das Sperren der entsprechenden Tabelle erweisen, welches bei der Änderungsoperation geschieht und erst beim Beenden einer Transaktion aufgehoben wird.

Signieren auf Klientenseite: Die so erzeugte Nachricht wird zusammen mit einer sitzungsspezifischen Anforderungsnummer dem Klient übermittelt, welcher in den meisten Anwendungsfällen eine GPG Signatur mittels „`gpg --clearsign`“ hinzufügt und den so erzeugten Text mit dem „`request.execute()`“ XML-RPC Befehl zurücksendet.

Die vorliegende HTML-Oberfläche implementiert das Signieren mit Hilfe einer simplen Copy&Paste-Operation, d.h., der Anwender hat den ursprünglichen Text mit der Maus zu markieren, in die Standardeingabe des `gpg` Kommandos zu kopieren, wiederum die Ausgabe zu markieren und in ein Textfeld des HTML-Formulars zu kopieren. Als Problem können sich – vom Nutzer unbemerkte – Änderungen der Zeilenenden (UNIX `\n` wird zu DOS `\r\n` oder umgekehrt) erweisen, welche die Signatur ungültig werden lassen. Tests mit den praktisch relevanten Browsern (Firefox, Mozilla, konqueror, Opera, w3m, elinks, links) und Terminals (xterm, gnome-terminal, Linux Konsole) haben diesen Fehler aber nicht provozieren können.

Nichtsdestotrotz, sollten spätere Versionen der HTML-Oberfläche eine Möglichkeit bieten, die Nachricht als Datei herunterzuladen und später die Signatur wieder hochzuladen.

Ausführen der Anforderung Gewöhnlich geschehen in der `Request::execute()` Methode folgende Aktionen:

1. Die Gültigkeit der Anfrage und die Autorisierung des Nutzers wird erneut überprüft, da während des klientseitigen Signierens sich Änderungen an der Datenbank ergeben können.
2. Es wird die Gültigkeit der Signatur mittels eines externen `gpg --verify` Kommandos überprüft. Normalerweise muß die Signatur von einem mindestens einfach vertrauenswürdigen Schlüssel (vergleiche Abbildung 4.1) des jeweiligen Nutzers erzeugt worden sein.
3. Der signierte Text wird mit dem ursprünglich übermittelten verglichen
4. Es wird eine neue Datenbanktransaktion gestartet und darin die jeweilige Aktion ausgeführt.
5. Es werden nachfolgende Aktionen wie zum Beispiel die Benachrichtigung des „stated“ oder das Verschicken von Nachrichten durch „notified“ ausgelöst.

Falls eine der ersten Aktionen fehlschlägt, wird eine *Exception* ausgelöst und die gesamte Anforderung verworfen.

6.4. Asynchrone Aufgabenverarbeitung

Da einige Aufgaben eine sehr große Zeit für ihre Vollendung benötigen, ist es bei ihrer Erstellung nicht möglich, auf das Ergebnis zu warten. Stattdessen werden die Details der Aufgabe in einer von `_claimed` abgeleiteten Tabelle (vergleiche Abbildung 6.8) gespeichert, der Status auf „PENDING“ gesetzt und der verantwortliche Server mittels einer XML-RPC „notify“ Nachricht aufgeweckt.

Diese Nachricht generiert ein `Condition::notifyAll()` Ereignis, welche alle in Wartestellung befindlichen Arbeits-Threads aktiviert (vergleiche Abbildung 6.9). Neben genannter „notify“-Nachricht kann auch eine periodisch auftretende Zeitüberschreitung diesen Effekt hervorrufen. Anschließend wird eine Liste der anstehenden Aufgaben, d.h. denen, deren Status auf „PENDING“ gesetzt ist, aus der Datenbank abgefragt und entsprechend ihrer Priorität sortiert und gefiltert.

Die Priorität wird von verschiedenen Aspekten beeinflußt. So kann die Art des Auftrages (zum Beispiel finaler Paketbau versus Testbuild) darauf Einfluß haben. Oder Builds, die voraussichtlich nur geringe Zeit benötigen, werden längeren vorgezogen. Damit letztgenannte Entscheidungen möglich sind, müssen statistische Daten von vorherigen Aktionen erhoben werden.

Zu diesem Zeitpunkt kann es passieren, daß mehrere Threads oder Komponenten die selbe Liste von Aufgaben vorliegen haben. Um Kollisionen beim Bearbeiten zu vermeiden wird

```
create table _claimed (
        claimed_by      text NULL,
        claimed_time    timestamp with time zone,

        status          text NOT NULL CHECK (status IN
                        ('NEW',      -- can not be claimed yet...
                        'PENDING',   -- can be claimed
                        'RUNNING',   -- claimed and running
                        'STOPPED',   -- claimed, but stopped
                        'READY',     -- finished successfully
                        'FAILED',    -- action failed, see 'reason'
                                     -- for details
                        'SKIPPED'    -- skipped and assumed as succeeded,
                                     -- see 'reason' for details
                        ))
                        DEFAULT 'NEW',
        reason          text NULL );

create table buildTasks (
        id              serial not null primary key,
        ...
        type            text NOT NULL CHECK (type IN
                        ('min', 'full', 'final')),
        ...
) inherits (_claimed,_statistics);
```

Abbildung 6.8.: SQL: Erzeugung der “_claimed” und “buildTasks“ Tabelle

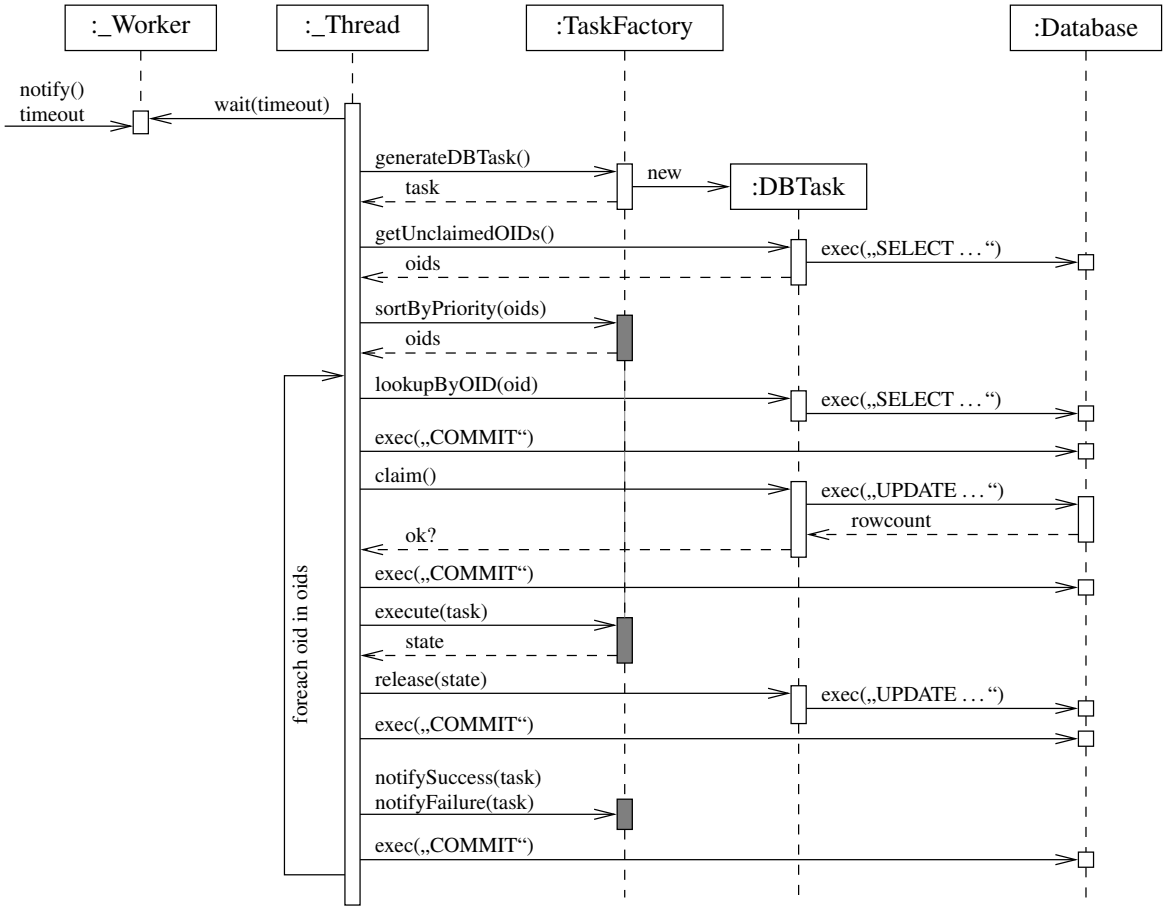

Abbildung 6.9.: Sequenzdiagramm: Task-Behandlung

nun das in PostgreSQL implementierte *Locking* von Tabellenzeilen aktiv ausgenutzt. Jeder Thread führt Kommandos wie

```
BEGIN WORK
UPDATE ... SET claimed_by=<ID>, status='PENDING'   \
  WHERE claimed_by IS NULL AND oid=<oid>
COMMIT
```

aus, wobei das Resultat des `UPDATE` Befehl die Anzahl der geänderten Zeilen ist. Offensichtlich kann nicht mehr als ein Thread an dieser Stelle „1“ erhalten. Genau dieser Thread führt nun die weitere Bearbeitung des Tasks durch; alle anderen wechseln zum nächsten.

Die Bearbeitung wird nun entsprechend der Art der Komponente in der `execute()` Methode fortgeführt und bringt den Task in einen neuen Status. Nach Beenden der Transaktion, welche eventuelle Änderungen in der Datenbank für alle anderen Komponenten sichtbar einträgt, werden zum Beispiel andere Komponenten über den geänderten Status informiert.

A. Zahlen zum Fedora Projekt

A.1. Paketanzahl

Durch „`ls *.rpm | wc -l`“ auf dem jeweiligen Verzeichnis ergeben sich die in untenstehender Tabelle aufgeführten Zahlen. Die dynamischen Werte für den Entwicklungszweig, fedora.us und für freshmeat spiegeln den Stand vom 18.02.2005 wider. Für fedora.us wurden die „stable“, „unstable“ und „testing“ Zweige zusammengefaßt.

Distribution	Quellpakete	Binärpakete (i386)
Red Hat Linux 7.3	814	1438
Red Hat Linux 8.0	838	1472
Red Hat Linux 9	839	1402
Fedora Core 1	875	1466
Fedora Core 2	947	1619
Fedora Core 3	970	1652
Entwicklungszweig	1063	1869
fedora.us, FC2	465	635
Projekte auf freshmeat	36244	

A.2. Paketkomplexität

Durch Zählung[1] und Zusammenfassung[2] der jeweiligen Abschnitte der in den Quellpakete enthaltenen Spezifikationsdateien ergeben sich die in Tabelle A.1 und Tabelle A.2 dargestellten Zeilenanzahlen. Der unter „total“ aufgeführte Wert beinhaltet sämtliche Zeilen, während bei „bereinigt“ die Leer- und Kommentarzeilen nicht mitgezählt wurden.

Die code-enthaltenden Abschnitte wurden dahingehend unterteilt, ob sie beim Bau oder bei der Installation des Paketes ausgeführt werden. Der in den Tabellen angegebene Durchschnitt bezieht sich in den zusammengefaßten Werten auf die Gesamtzahl der Pakete, während in den unterteilten Werten der Durchschnitt pro Skript angegeben wurde.

Diskrepanzen zwischen tatsächlichen und ermittelten Werten lassen sich damit erklären, daß eventuelle Makroexpansionen durch das verwendete Skript nicht durchgeführt wurden.

[1] vergleiche `test/srpm-metric`
[2] vergleiche `test/srpm-totals`

Insgesamt sind davon aber nur eine sehr geringe Anzahl von Paketen, so daß die ermittelten Werte eine hohe Genauigkeit aufweisen.

		Zeilenanzahl		⌀ Zeilenanzahl	
Abschnitt	Anzahl	total	bereinigt	total	bereinigt
Gesamt	1063	299061	225141	281.3	211.8
Ohne `%changelog`	1063	131773	102709	124.0	96.6
`%changelog`	1062	167288	122432	157.5	115.3
Metadaten	1063	48544	39025	45.7	36.7
Dateilisten	1986	22799	19735	11.5	9.9
Build-Skripte		43302	30679	40.7	28.9
`%prep`	1052	11786	8384	11.2	8.0
`%build`	1019	10716	7749	10.5	7.6
`%install`	1058	20636	14420	19.5	13.6
`%check`	18	164	126	9.1	7.0
Installations-Skripte		9871	7251	9.3	6.8
`%pre/post*`	1677	9320	6804	5.6	4.1
`%trigger*`	1019	551	447	7.9	6.4

Tabelle A.1.: Paketkomplexität, Fedora Core Entwicklungszweig

Abschnitt	Anzahl	Zeilenanzahl total	Zeilenanzahl bereinigt	⌀ Zeilenanzahl total	⌀ Zeilenanzahl bereinigt
Gesamt	465	49721	36783	106.9	79.1
Ohne `%changelog`	465	35700	26013	76.8	55.9
`%changelog`	464	14021	10770	30.2	23.2
Metadaten	465	14467	11306	31.1	24.3
Dateilisten	633	5873	4729	9.3	7.5
Build-Skripte		10632	6784	22.9	14.6
`%prep`	465	2509	1512	5.4	3.3
`%build`	457	2663	1625	5.8	3.6
`%install`	465	5086	3465	10.9	7.5
`%check`	93	374	182	4.0	2.0
Installations-Skripte		1798	1163	3.9	2.5
`%pre/post*`	416	1656	1050	4.0	2.5
`%trigger*`	34	142	113	4.2	3.3

Tabelle A.2.: Paketkomplexität, fedora.us

B. Datenbank-Benchmarks

B.1. Verbindungsaufbau

Tabelle B.1 vergleicht die Zeiten, die für unterschiedliche Verfahren des Verbindungsaufbaus mit einer PostgreSQL-Datenbank benötigt werden.

Anzahl	Zeiten in *ms* für				
	$\oslash f()$ (`trust`)	$\oslash f()$ (`trust` + SSL)	$\oslash f()$ (`krb5` + SSL)	$\oslash(init + g())$ (`trust`)	$\oslash(init + g())$ (`trust` + SSL)
1	239.82	235.78	352.36	232.84	229.61
2	233.05	229.62	353.49	119.08	115.82
3	235.17	229.27	367.52	79.10	77.85
4	233.94	229.36	354.92	59.55	58.47
5	233.27	229.42	340.56	47.54	47.46
6	233.78	228.60	343.15	40.13	39.38
7	233.15	231.07	348.46	34.40	34.01
8	234.83	229.66	347.79	30.01	30.24
10	233.96	230.29	336.49	24.55	24.38
12	234.08	230.16	330.75	20.32	20.33
14	245.95	229.13	323.31	17.41	17.70
16	233.80	229.26	325.00	15.40	15.62
32	233.76	229.14	335.64	8.09	8.36
64	248.34	240.07	337.95	4.34	4.67
128	243.78	239.47	341.99	2.55	2.87
256	234.30	287.27	352.16	1.62	2.11

Tabelle B.1.: Datenbank-`connect()`-Zeiten

Dazu wurden die in Abbildung B.1 dargestellten Funktionen[1] $f()$ und $g()$ mehrfach ausgeführt mit dem Unterschied, daß $f()$ für jede Operation eine neue Verbindung zur Datenbank aufbaut, während $g()$ eine bereits existierende nutzt. Für den Verbindungsaufbau wurden rein IP basierende Authentifizierungsmechanismen (`trust`) und Kerberos5 basierende getestet; die Datenübertragung fand sowohl mit SSL geschützt, als auch ohne statt.

Serverseitig wurde `postgresql-7.4.7-1.FC2.2` (Fedora Core 2, Kernel 2.4.29) auf einem Pentium III mit 600 MHz und 384 MiB RAM eingesetzt; auf der Seite des Clients

[1] vergleiche `test/db-connect.py`

postgresql-7.4.7-1.FC3.2 (Fedora Core 3, Kernel 2.6.10) auf einem Pentium 4 1.3 GHz mit 384 MiB RAM. Die Verbindung erfolgte über 100 Mb/s Ethernet. In den Meßergebnissen auftauchende Ungenauigkeiten lassen sich damit begründen, daß auf dem Server einige andere Anwendungen parallel zum Test liefen.

```
def f():
    db = pgdb.connect(host=dbhost,
                      database=database, password='X')
    c  = db.cursor()
    c.execute("SELECT 42")


glob_db=None
def init():
    global glob_db
    glob_db = pgdb.connect(host=dbhost,
                           database=database, password='X')

def g():
    c  = glob_db.cursor()
    c.execute("SELECT 42")
```

Abbildung B.1.: Verwendete Funktionen des db_connect() Benchmarks

Es ist erkennbar, daß:

- die bei $g()$ eingesetzten persistenten Verbindungen eine deutliche Performanceverbesserung zur Folge haben, wenn mehrere Operationen ausgeführt werden
- der Verbindungsaufbau ansich relativ teuer ist (230*ms* für trust-Authentifikation) verglichen mit dem Ausführen einzelner Kommandos (1.5*ms*)
- auf Serverseite keine Optimierung bezüglich aufeinanderfolgenden Verbindungsaufbaus stattfindet.
- je komplexer die Authentifikationsmethode ist, desto mehr Zeit wird für den Verbindungsaufbau benötigt.
- die SSL-Verschlüsselung bei diesem Test, der nur sehr wenige Daten überträgt, einen sehr geringen Einfluß auf das Ergebnis ausübt.

C. Beschreibung des CD-Inhalts

Nachfolgend eine kurze Beschreibung des auf der CD befindlichen Tar-Archiv und der darin verpackten Dateien bzw. Verzeichnisse. Aus diesem Archiv lassen sich mittels `rpmbuild -ba fedqa-*.tar.bz2` Binärpakete erzeugen, die auf Rechnern mit Fedora Core 3 und konfiguriertem Fedora Extras Repository installierbar sind. Als zusätzliches Paket wird Smarty benötigt, welches sich ebenfalls auf der CD befindet.

bin/ Dieses Verzeichnis enthält ein generisches Start-Skript

contrib/ Dieses Verzeichnis enthält einige Hilfsdateien, die für die Erzeugung der RPM-Pakete benötigt werden.

etc/ Dieses Verzeichnis enthält die Konfigurationsdateien

helper/ Dieses Verzeichnis enthält den generischen, in C geschriebenen SUID-Wrapper

python/ Dieses Verzeichnis enthält die Python-Module. `Util/pgdb_thread.py` wurde dem postgresql-python Paket entnommen und lokal angepaßt.

python/testsuite/ Dieses Verzeichnis enthält Regressionstests für die einzelnen Komponenten

scripts/ Dieses Verzeichnis enthält verschiedene bash-Skripte, welche unter anderem für den Download genutzt werden.

sql/ Dieses Verzeichnis enthält das SQL Skript zum Erzeugen der Datenbank

src/ Dieses Verzeichnis enthält verschiedene, in C geschriebene Hilfsprogramme. `scramble.c` wurde dem CVS Quellpaket unverändert entnommen.

web/ Dieses Verzeichnis enthält die PHP-Skripte, die HTML-Oberflächenbeschreibung und die für den Betrieb der HTML Oberfläche nötigen Apache httpd Konfiguration.

Glossar

$f(X)$ Diese Funktion bezeichnet den Paketbau, wenn die Paketmenge X installiert ist. f selbst wird durch den Inhalt der Spezifikationsdatei definiert; das Resultat sind ein oder mehrere Binärpakete.

NDA *Nondisclosure Agreement* (Stillschweigeabkommen); bei freier Software häufig anzutreffen bei Informationen über Sicherheitslücken, die erst nach einer gewissen Zeit veröffentlicht werden dürfen

PID *process identification*; Prozeßidentifikator

QA *Quality Assurance*, Qualitätssicherung

RPM ursprünglich: *Red Hat Package Manager*, nun: *RPM Package Manager*; ein Format zum Verpacken von Software. Ein RPM-Paket enthält Informationen über Abhängigkeiten, Installations- und Deinstallationsskripte, einige Metadaten und die Dateien ansich. Eine Sonderform, nämlich SRPM (*Source-RPM*; häufig auf `.src.rpm` endend), enthält die Quelldateien und Informationen, wie daraus ein fertiges Binärpaket erstellt werden soll.

SCM *Sourcecode Managementsystem* (Quellcodeverwaltungssystem)

SRPM *Source*-RPM; vergleiche RPM

SUID Bit ein spezielles Bit in den Rechtemaske von Dateien. Wenn dieses gesetzt ist, wird das Programm unter der Nutzer-ID des Eigentümers ausgeführt anstatt unter der, des aufrufenden Nutzers.

Literaturverzeichnis

[1] BAILEY, EDWARD C., PAUL NASRAT, MATTHIAS SAOU und VILLE SKYTTÄ: *Maximum RPM – Taking the RPM Package Manager to the Limit*, 2000. http://www.rpm.org/max-rpm-snapshot/.

[2] BAIZE, E. und D. PINKAS: *RFC 2478: The Simple and Protected GSS-API Negotiation Mechanism*, Dezember 1998. Status: PROPOSED STANDARD.

[3] BELGIUM, SCANIT: *Browser Security Test*. http://bcheck.scanit.be/bcheck/alltests.php.

[4] BEN COLLINS-SUSSMAN, BRIAN W. FITZPATRICK, C. MICHAEL PILAT: *Version Control with Subversion – For Subversion 1.1*. TBA, 2004.

[5] BUGZILLA, MOZILLA: *SECURITY: predictable sessionid (Use a token instead of logincookie)*. Bugzilla, Januar 2002. https://bugzilla.mozilla.org/show_bug.cgi?id=119524.

[6] CALLAS, J., L. DONNERHACKE, H. FINNEY und R. THAYER: *RFC 2440: OpenPGP Message Format*, November 1998. Status: PROPOSED STANDARD.

[7] EISLER, M., A. CHIU und L. LING: *RFC 2203: RPCSEC_GSS Protocol Specification*, September 1997. Status: PROPOSED STANDARD.

[8] ELKINS, M., D. DEL TORTO, R. LEVIEN und T. ROESSLER: *RFC 3156: MIME Security with OpenPGP*, August 2001. Status: STANDARDS TRACK.

[9] FEDORA WIKI: *Package Naming Guidelines*. Wiki. http://fedoraproject.org/wiki/PackageNamingGuidelines.

[10] FIELDING, R., J. GETTYS, J. MOGUL, H. FRYSTYK und T. BERNERS-LEE: *RFC 2068: Hypertext Transfer Protocol — HTTP/1.1*, Januar 1997. Status: PROPOSED STANDARD.

[11] FISH, SHLOMI: *Better SCM Initiative*. http://better-scm.berlios.de.

[12] FREED, N. und N. BORENSTEIN: *RFC 2045: Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*, November 1996. Obsoletes RFC1521, RFC1522, RFC1590. Updated by RFC2184, RFC2231. Status: DRAFT STANDARD.

[13] HOUSLEY, R., W. POLK, W. FORD und D.SOLO: *RFC 3280: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*, April 2002. Status: STANDARDS TRACK.

[14] ISO: *ISO/IEC 9126-Information Technology, Software Product Evaluation, Quality, Characteristics and Guidelines for their Use*, 1991.

[15] ITU-T, INTERNATIONAL TELECOMMUNICATION UNION TELECOMMUNICATION STANDARDIZATION SECTOR: *Recommendation X.680 (12/97) — Abstract Syntax Notation One (ASN.1): Specification of Basic Notation*. ITU, Geneva, Dezember 1997.

[16] KLENSIN, J.: *RFC 2821: Simple Mail Transfer Protocol*, April 2001. Status: STANDARDS TRACK.

[17] KOHL, J. und C. NEUMAN: *RFC 1510: The Kerberos Network Authentication Service (V5)*, September 1993. Status: PROPOSED STANDARD.

[18] LEIBA, B.: *RFC 2177: IMAP4 IDLE command*, Juni 1997. Status: PROPOSED STANDARD.

[19] LINN, J.: *RFC 1964: The Kerberos Version 5 GSS-API Mechanism*, Juni 1996. Status: PROPOSED STANDARD.

[20] REKHTER, Y., B. MOSKOWITZ, D. KARRENBERG, G. J. DE GROOT und E. LEAR: *RFC 1918: Address Allocation for Private Internets*, Februar 1996. Status: BEST CURRENT PRACTICE.

[21] SCHOLZ, ENRICO: *[db4-java] Does not work when installed with an umask of 077 and messes filesystem*. Bugzilla, September 2002. `https://bugzilla.redhat.com/bugzilla/show_bug.cgi?id=74046`.

[22] SCHOLZ, ENRICO: *The fedora.us buildsystem*. Draft, fedora.us, Januar 2004.

[23] SCHOLZ, ENRICO: *Using Linux VServer*. In: *LinuxTag 2004*, Juni 2004.

[24] SHIFLETT, CHRIS: *Foiling Cross-Site Attacks*. Technischer Bericht, `http://shiflett.org/articles/foiling-cross-site-attacks`, 2003.

[25] SKYTTÄ, VILLE: *qiv Paket*. `http://www.fedora.us/tempspecs/stable/qiv.spec`, `http://download.fedora.us/fedora/fedora/2/i386/SRPMS.stable/qiv-2.0-0.fdr.1.2.src.rpm`.

[26] SRINIVASAN, R.: *RFC 1831: RPC: Remote Procedure Call Protocol Specification Version 2*, August 1995. Status: PROPOSED STANDARD.

[27] SRINIVASAN, R.: *RFC 1832: XDR: External Data Representation Standard*, August 1995. Status: DRAFT STANDARD.

[28] STEVENS, RICHARD: *UNIX Network Programming (Volume 1: Networking APIs: Sockets and XTI) – 2nd edition*. Prentice-Hall, Inc., 1998.

[29] STEVENS, RICHARD: *UNIX Network Programming (Volume 2: Interprocess Communications) – 2nd edition*. Prentice-Hall, Inc., 1999.

Selbstständigkeitserklärung

Hiermit erkläre ich, daß ich die vorliegende Arbeit selbstständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendete Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Chemnitz, den 11. März 2005

Scholz Enrico