VerifierCloud: Implementierung eines Web-Service zur Software-Verifikation

Bachelorarbeit

an der Fakultät für Informatik und Mathematik
der Universität Passau

Prüfer:

Prof. Dr. Dirk Beyer

Sebastian Ott
Sommersemester 2014

Zusammenfassung

Softwareverifikation benötigt im Allgemeinen eine sehr große Menge an Rechenressourcen, die zum Beispiel durch ein Cloud-System wie die VerifierCloud bereitgestellt werden können. Um diese Ressourcen auch externen Gruppen zugänglich zu machen, wird aufbauend auf die VerifierCloud ein Web-Service zur Software-Verifikation implementiert, der das Opensource-Verifikationswerkzeug CPAchecker als Backend verwendet. Es wird eine REST-Schnittstelle bereitgestellt, die in die bestehende Infrastruktur integriert wird. Eine experimentelle Evaluation der Web-Schnittelle wird deren Leistungsfähigkeit und Verwendbarkeit für cloudbasierte Software-Verifikation zeigen.

Inhaltsverzeichnis

Abbildungsverzeichnis **5**

Tabellenverzeichnis **6**

1 Einleitung **7**

2 Grundlagen **8**
2.1 VerifierCloud 8
2.2 REST 9
2.3 JAX-RS API 10
2.4 Jersey 11

3 Implementierung des Web-Clients **12**
3.1 Notwendige Anpassungen an der VerifierCloud 12
3.1.1 Kommunikation zwischen Master und Client 12
3.1.2 Verarbeitung großer Dateien 13
3.1.3 Weitere Änderungen 14
3.2 Initialisierung der Web-Anwendung 14
3.3 Anbindung an Versionsverwaltung 15
3.4 Separate Dateiinhalt-Speicherung 17
3.5 REST-Schnittstelle 19
3.5.1 Resource-Klasse 19
3.5.2 Bean-Klassen 19
3.6 WebClientAPI 20
3.7 Generierung des Hilfetextes 20
3.8 Generierung der Master-Informationsseite 22
3.9 Kompilieren des CPAcheckers 22
3.9.1 Kompilieren in der VerifierCloud 22
3.9.2 Vorausberechnung der CPAchecker-Revisionen 23
3.10 Ausführen des Verifikationswerkzeuges 24
3.11 Bereitstellen der Run-Zustände und der Resultate 24
3.12 Bereitstellen des gebauten Verifikationswerkzeuges 25
3.13 Sicherheitskonzept 25
3.14 Konfiguration 26
3.15 Ausnahmebehandlung 28

4 Qualitätssicherung **30**
4.1 JUnit-Tests . . . 30
4.2 Automatischer Systemtest . . . 31

5 Experimentelle Evaluation **32**
5.1 Lasttest . . . 32
5.2 Vergleich mit anderen Clients . . . 33
5.2.1 CPAchecker-Client . . . 37
5.2.2 Benchmark-Client . . . 37

6 Beschreibung der Web-Schnittstelle **41**
6.1 Hilfetext . . . 41
6.2 Einreichen von Verifikationsaufgaben . . . 42
6.3 Einreichen von Verifikationsaufgaben mit höherer Priorität . . . 42
6.4 Abfrage des Run-Zustandes . . . 44
6.5 Erhalten der Resultate . . . 45
6.6 Herunterladen des CPAcheckers . . . 46
6.7 Master-Informationsseite . . . 47
6.8 Update-Trigger für den Git-Klon . . . 47

7 Benutzer des Web-Clients **48**

8 Fazit **49**

Literatur **50**

Erklärung zur Bachelorarbeit **51**

Abbildungsverzeichnis

2.1 Architektur der VerfierCloud 8

3.1 Diagramm der `SvnRevision`-Klassen 17
3.2 Diagramm der `ClientFileStorage`-Klassen 18
3.3 Diagramm der Bean-Klassen 21

5.1 Antwortzeit über die Anzahl der parallelen Anfragen 34
5.2 Antwortzeit von Zustandsanfragen über die Anzahl der parallelen Anfragen 35
5.3 HTTP-Anfragen pro Minute 36
5.4 SV-Comp14: Aufsummierte Anzahl der analysierter Ergebnisse über die Zeit 39
5.5 BDD-Integrationstest: Aufsummierte Anzahl der analysierter Ergebnisse über die Zeit 40

Tabellenverzeichnis

2.1 HTTP-Methoden 10

6.1 Übersicht der REST-Ressourcen des Web-Clients 41
6.2 Vorkonfigurierte Beschränkungen 42
6.3 Parameter für Aufgabeneinreichung 43
6.4 Fehlercodes bei Aufgabeneinreichung 44
6.5 Run-Zustände 44
6.6 Fehlercodes bei Abfrage eines Run-Zustandes 44
6.7 Fehlercodes bei Abfrage eines Run-Ergebnisses 46
6.8 Parameter für Verifikationswerkzeug-Download 46
6.9 Fehlercodes bei Verifikationswerkzeug-Download 46
6.10 Fehlercodes bei Abruf der Master-Informationsseite 47

1 Einleitung

Der Einsatz von Softwareverifikation in realen Projekten erfordert große Mengen an Rechenressourcen. Dabei sind typischerweise zahlreiche unabhängige Verifikationsaufgaben durchzuführen, die sich daher einfach parallelisieren lassen. Cloudbasierte Softwareverifikation ist deswegen seit einigen Jahren Gegenstand der Forschung. Ein Beispiel dafür ist das Spiel Code Hunt[1], das auf einem Windows-Azure Service[2] basiert, der Pex[3] als Backend verwendet [7]. CPAchecker wurde erfolgreich erweitert, so dass er in der GoogleAppEngine ausgeführt werden kann[4], unterliegt dort aber einigen Restriktionen [1].
Ein anderes Projekt zur cloudbasierten Softwareverifikation ist die VerifierCloud, die zahlreiche Rechner an der Universität Passau und einen Cluster am Hasso Plattner Institut in Potsdam zum Lösen von Verifikationsaufgaben nutzt, wenn diese Rechner gerade nicht anderweitig verwendet werden. Zu den meisten Zeiten verfügt das System über umfangreiche freie Kapazitäten. Benutzer der VerifierCloud müssen Zugang zum lokalen Netzwerk haben. Die Nutzung der enormen durch die VerifierCloud bereitgestellten Ressourcen kann für externe Personen durch ein Webinterface, das vom Internet aus erreichbar ist, ermöglicht werden. Diese Arbeit beschäftigt sich mit der Entwicklung einer solchen Web-Schnittstelle zur Softwareverifikation unter Verwendung der VerifierCloud.
Diese Abschlussarbeit umfasst neun Kapitel, von denen das erste diese Einleitung ist, gefolgt von Kapitel 2, das Grundlagen zur VerifierCloud und dem verwendeten Framework erklärt. Kapitel 3 beschreibt ausführlich die Implementierung des Web-Clients. Kapitel 4 befasst sich mit vorgenommenen Qualitätssicherungsmaßnahmen. Kapitel 5 erläutert die experimentelle Evaluation des Web-Clients. In Kapitel 6 wird die Web-Schnittstelle des Web-Clients beschrieben und die bisherigen Benutzer des Web-Client werden in Kapitel 7 kurz vorgestellt. Die Arbeit schließt in Kapitel 8 mit einer Zusammenfassung der Ergebnisse und einem Ausblick auf mögliche weitere Entwicklungen.

[1] `https://www.codehunt.com/`
[2] `http://api.codehunt.com/`
[3] `http://research.microsoft.com/en-us/projects/pex/`
[4] `http://trunk.cpachecker.appspot.com/`

2 Grundlagen

Dieses Kapitel erläutert die wichtigsten Konzepte und Programme bzw. Bibliotheken, die durch den Web-Client verwendet werden.

2.1 VerifierCloud

Die VerifierCloud ist ein ein verteiltes System zur parallelen Berechnung von mehreren Verifikationsaufgaben durch ein Verifikationswerkzeug auf vielen Rechnern. Sie ist in der Sprache Java[1] implementiert und speziell auf die Anforderungen des Opensource-Verifikationswerkzeugs CPAchecker[2] ausgerichtet. Die VerifierCloud gliedert sich, wie in Abbildung 2.1 dargestellt, in die drei Grundkomponenten Worker, Master und Client.

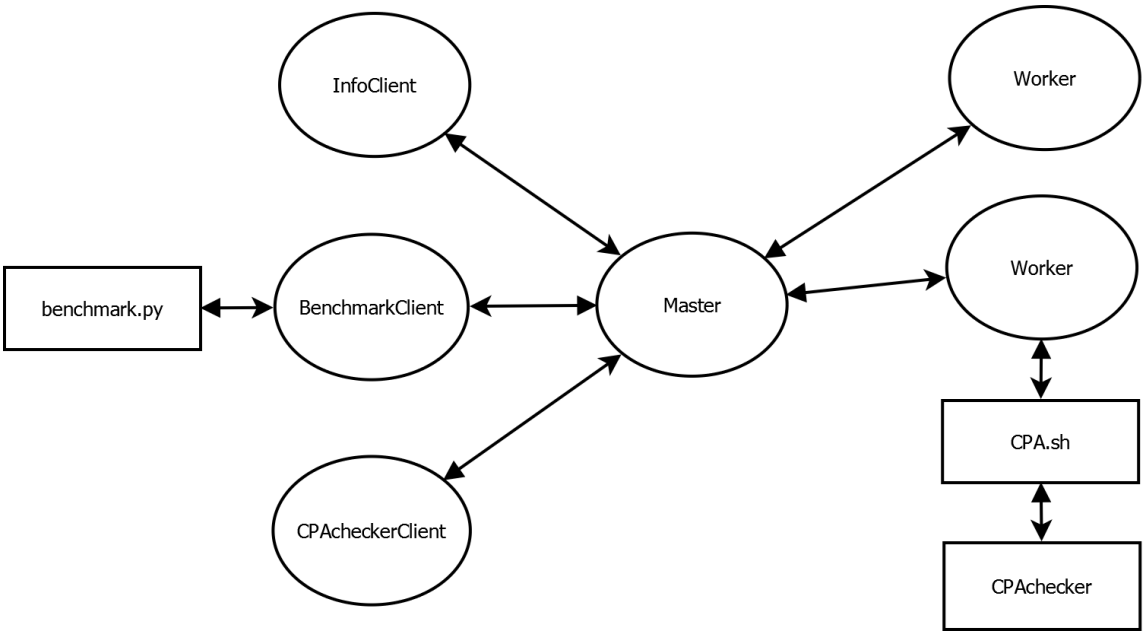

Abbildung 2.1: Architektur der VerfierCloud

[1] Version 7
[2] `http://cpachecker.sosy-lab.org/`

Die Aufgabe des Workers ist das Ausführen der sogenannten Runs, von denen jeder einen frei wählbaren Kommandozeilenaufruf darstellt, der das Verifikationswerkzeug startet. Zuvor stellt der Worker die im Run enthaltene Dateihierarchie, die alle für die Ausführung benötigten Dateien beinhaltet, im Dateisystem des Worker-Rechners her. Für jeden Run werden Speicher, Zeit und Zahl der CPU-Kerne, die der zu startende Prozess verwenden kann, begrenzt, wodurch auch mehrere Runs auf einem Rechner parallel bearbeitet werden können, ohne dass es zu gegenseitigen Störungen kommt. Das Resultat eines Runs ist die Ausgabe des gestarteten Prozesses, alle neu erstellten Dateien, deren Inhalte und der Exitcode. Auf jedem als Worker verwendeten Rechner läuft eine Instanz des Workerprozesses.
Der Master stellt die zentrale Verwaltungseinheit der VerifierCloud dar. Er startet die Worker, weist den Workern Runs zur Ausführung zu und nimmt Runs von den Clients entgegen. Zur Kommunikation zwischen den Komponenten werden Nachrichten über TCP-Verbindungen verschickt. Es gibt einen Master je VerfierCloud-Instanz und dieser hat die Rolle eines Servers.
Nutzer der VerifierCloud erzeugen mit Hilfe von einem Client Runs, der sie dann an den Master sendet und die Ergebnisse vom Master entgegennimmt. Dabei wird für jeden Run die Kommandozeile, die Begrenzung von Zeit, Speicher und CPU-Kernen sowie die benötigten Dateien in Form einer Dateihierarchie festgelegt. Es gibt verschiedene Client-Varianten: Den Interactive-Client, ein einfacher, interaktiver Kommandozeilenclient, den CPAchecker-Client zum Verifizieren von Dateien aus einem Ordner mit denselben Optionen durch den CPAchecker und den Benchmark-Client, der ganze Benchmarks ausführen kann. Er dient als Schnittstelle der VerifierCloud zum Benchmark-Skript (*benchmark.py*) des CPAcheckers. Am Worker wird der CPAchecker jeweils über das *CPA.sh*-Skrippt gestartet. Außerdem gibt es noch den Info-Client, der einen Text oder eine HTML-Seite zum Zustand des Masters bzw. der Cloud generieren kann.
Alle Dateiinhalte werden über Hashwerte referenziert und am Master und den Workern dauerhaft vorgehalten, so dass der selbe Dateiinhalt nur einmal zu jedem Rechner transportiert werden muss.

2.2 REST

REST ist die Abkürzung für *Representational State Transfer* und bezeichnet ein von Roy Thomas Fielding entwickeltes Architekturparadigma für ressourcenorientierte und repräsentationsbasierte Schnittstellen [4, 5]. Eingesetzt wird es vor allem im Web unter Verwendung von HTTP, ist aber nicht darauf beschränkt. Im Folgenden wird REST in Verbindung mit dem Protokoll HTTP behandelt. Durch Operationen, typischerweise HTTP-Methoden, kann mit REST-Anwendungen interagiert werden. Tabelle 2.1 listet die wichtigsten HTTP-

Methoden[3] auf.

Methode	Beschreibung
GET	frag Ressource an
POST	erzeugt neue Ressource als Sub-Ressource der angegebenen Ressource, Name der neuen Ressource wird normalerweise zurückgeben
PUT	aktualisiert oder erzeugt Ressource
DELETE	löscht Ressource

Tabelle 2.1: Beschreibung der HTTP-Methoden

Eine Ressource wird über eine eindeutige URI identifiziert. Ressourcen können dabei in verschiedenen Repräsentationen übertragen werden, die vom Client bei Anfragen mittels HTTP-Header ausgewählt werden können.
Eine wichtige Eigenschaft ist Zustandslosigkeit auf Server- und Clientseite. Das bedeutet, dass zum einen alle für die Bearbeitung einer Anfrage benötigten Informationen in dieser enthalten sein müssen und zum anderen mehrere Anfragen der selben Ressource das gleiche Ergebnis liefern, sofern sie nicht explizit geändert wurden. Zustandslosigkeit erleichtert die Anfragebearbeitung und ermöglicht einfaches Caching. Reale Anwendungen sind meistens nicht vollkommen zustandslos, da dies nur schwer umzusetzen ist.

2.3 JAX-RS API

Java API for RESTful Web Services[4] ist die Spezifikation [6] einer Java-API für REST-basierte Web-Applikationen. JSR 311 bzieht sich auf Version 1.0 und 1.1, JSR 339 auf Version 2.0, die für den Web-Client verwendet wird.
Die Web-Applikation stellt Ressourcen-Methoden zur Beantwortung von Anfragen bereit, wobei durch Annotationen festgelegt wird, welche Anfragen die jeweilige Methode beantworten kann. Dabei können Pfad, konsumierter und produzierter Content-Type sowie die HTTP-Methode festgelegt werden. Die Anfrage wird automatisch in die Java-Typen der Parameter von Methoden und Konstruktoren transformiert. Möglicherweise sind dafür Konvertierungsmethoden oder -klassen zu erstellen [2].

[3] `http://tools.ietf.org/html/rfc2616#section-9`
[4] `https://jax-rs-spec.java.net/`

2.4 Jersey

Die Referenzimplementierung von JAX-RS ist Jersey[5], das im Web-Client in Version 2.12 zum Einsatz kommt. Außerdem werden die Erweiterungen für Bean-Validation[6], Multipart[7] und das Test-Framwork[8] verwendet.
Manche Funktionen des Kernpaketes und der Erweiterungen müssen genauso wie die eigene Web-Applikation registriert werden, bevor sie genutzt werden können. Anschließend kümmert sich das von Jersey verwendete Injection-Framework darum, dass die benötigten Instanzen bereitgestellt werden. Standardmäßig wird für jede Anfrage ein neues Objekt der Ressourcenklasse der Web-Applikation, die die Anfragen entgegennimmt, erzeugt, wodurch während der Bearbeitung paralleler Anfragen unerwünschte Seiteneffekte vermieden werden und anfragespezifische Informationen im Konstruktor übergeben werden können.
`org.glassfish.jersey.servlet.ServletContainer` muss in der *web.xml*-Datei der Anwendung als Servlet-Klasse angegeben werden. Die `Application`-Klasse der Web-Anwendung wird als Einstiegspunkt durch den Initialisierungsparameter des Servlets mit Namen `javax.ws.rs.Application` und dem Name der `Application`-Klasse der Web-Anwendung als Wert festgelegt.

[5] `https://jersey.java.net/`
[6] `https://jersey.java.net/documentation/latest/bean-validation.html`
[7] `https://jersey.java.net/documentation/latest/media.html#multipart`
[8] `https://jersey.java.net/documentation/latest/test-framework.html`

3 Implementierung des Web-Clients

Der Web-Client stellt als ein weiterer Client der VerfierCloud eine Web-Schnittstelle zur Software-Verifikation bereit. Als Verifikations-Werkzeug kommt dabei CPAchecker zum Einsatz. Es werden viele Komponenten der bisherigen Clients weiterverwendet bzw. erweitert.
Diese Arbeit bezieht sich auf Revision 4062 des SVN-Repositories[1] des VerifierCloud-Projektes und Revision 13737 des SVN-Repositories[2] des CPAchecker-Projektes. In diesem Kapitel wird die Implementierung des Web-Clients beschrieben. Dabei werden zuerst die notwendigen Änderungen thematisiert und anschließend der Web-Client selbst.

3.1 Notwendige Anpassungen an der VerifierCloud

Das Verhalten des Web-Clients unterscheidet sich in einigen entscheidenden Aspekten von dem der schon existierenden Clients. Sie arbeiten sequenziell und sind nicht als dauerhaft laufende Anwendungen konzipiert worden. Im Gegensatz dazu bearbeitet der Web-Client als dauerhaft laufende Web-Anwendung viele Anfragen gleichzeitig. Die deswegen notwendigen Änderungen an bisher vorhandenen Komponenten werden in den folgenden Abschnitten erläutert.

3.1.1 Kommunikation zwischen Master und Client

Bisher waren die Clients bei den meisten Nachrichtentypen nur in der Lage eine Anfrage gleichzeitig zu stellen, weil diese gespeichert und das Ergebnis der Anfrage in ein `Future` geschrieben wurde, das die Rückgabe an die Methode war, die die Nachricht veranlasste. Das `Future` wurde in der Klasse `ActiveRequest` verwaltet. Sie konnte immer nur eine Anfrage zur gleichen Zeit verarbeiten und war als Singelton konzipiert. Mit Hilfe von zusätzlichen Abhängigkeiten, waren einige Nachrichten von dieser Einschränkung befreit, in dem sie nicht von `ActiveRequest` Gebrauch machten, wodurch die Kommunikationsschnittstellen jedoch unübersichtlich gestaltet waren.

[1] `https://svn.sosy-lab.org/software/verifiercloud/`
[2] `https://svn.sosy-lab.org/software/cpachecker/`

Die Möglichkeit der parallelen Kommunikation mit dem Master ist für den Web-Client unerlässlich. Dafür wurde die Klasse in `ActiveRequests` umbenannt und so erweitert, dass sie eine beliebige Menge von Anfragen und deren Antworten gleichzeitig verwalten kann. Alle an den Master geschickten Nachrichten enthalten eine eindeutige ID, die vom Master in die Antwort eingefügt wird, damit sie am Client wieder der ursprünglichen Anfrage zugeordnet werden kann. Deswegen enthält die abstrakte Klasse `ClientToMasterCommand` ein Feld `requestId`, das mit einem kryptographisch sicheren Zufallswert initialisiert wird.
In diesem Zusammenhang wurden alle vom Client verschickten Nachrichten auf dieses neue System umgestellt, wodurch die Ergebnisse aller Anfragen als `Future`-Objekte bereit gestellt werden, in die `ActiveRequests` die Ergebnisse schreibt, sobald es sie vom Master erhält. Auf diese Weise weiß der Client auch bei Anfragen die kein unmittelbares Ergebnis zurück liefern, ob Anfragen erfolgreich vom Master bearbeitet wurden, da in diesen Fällen ein `Success`-Objekt zurückgegeben wird. Außerdem wurden komplizierte Abhängigkeiten zwischen der `DefaultMasterConnection` und der `DefaultClientApi` aufgelöst.
Besonders hervorzuheben ist das Verhalten beim Senden einer neuen `RunCollection`. Der Master schickt erst eine Erfolgsmeldung, wenn alle Dateien am Master vorhanden sind und die `RunCollection` an den `Scheduler` übergeben wurde. Der Web-Client wartet bis zu diesem Zeitpunkt mit einer Antwort an den Benutzer, damit dieser sicher sein kann, dass die eingereichte Verifikationsaufgabe auch tatsächlich bearbeitet wird. Im Falle eines Fehler kann dem Benutzer so auch eine Fehlermeldung übermittelt werden.

3.1.2 Verarbeitung großer Dateien

Der Web-Client und die durch den Web-Client erstellten Runs erzeugen potentiell erheblich größere Dateien als die VerifierCloud bislang üblicher Weise verarbeitet hat. Zum einen sind die Zip-Archive des gebauten CPAcheckers ca. 50 MB groß und zum anderen sind sich Benutzer des Web-Clients der Problematik sehr großer Log- und anderer Ausgabedateien nicht bewusst, weswegen jene solche Dateien nicht wie normale Benutzer der VerifierCloud vermeiden werden.
Die bisher genutzten `FileContent`-Implementierungen halten Dateiinhalte vollständig im Speicher. Dieses Verhalten kann bei großen Dateien trotz genutzter Kompression zum Absturz der VerfierCloud-Komponenten wegen Speichermangels führen. Deshalb wurde für große Dateien eine schon vorhandene Implementierung aktiviert, die den Dateiinhalt nie vollständig im Speicher sondern im Dateisystem hält. Um nun auch einen speichereffizienten Zugriff zu gewährleisten, wurde das Interface `FileContent` um die Methode `getContentAsByteSource()` erweitert. Die von dieser Methode zurückgegebene `ByteSource` ermöglicht das Öffnen eines `InputStream`s, mit dessen Hilfe auf den Dateiinhalt zugegriffen werden

kann.
Der Web-Client nutzt dies bei HTTP-Antworten, deren Inhalt eine Datei ist, indem die entsprechende Methode ein `Response` mit einem `StreamingOutput` zurückgibt, das den Dateiinhalt in den Ausgangsstream kopiert. Dadurch können große Dateien ohne Schwierigkeiten über den Web-Client zum Benutzer geschickt werden.

3.1.3 Weitere Änderungen

Neben den bereits aufgezeigten Änderungen waren noch einige kleinere Anpassungen an bereits vorhanden Komponenten notwendig.
Die `DefaultMasterConnection` ist in die Lage versetzt worden mehrere Versuche durchzuführen, die Verbindung vom Client zum Master aufzubauen, damit der Web-Client, wenn der Master beim Start nicht erreichbar ist, sich nicht für jeden Verbindungsversuch vollständig neu initialisieren muss.
Jersey kann Parameter aus Anfragen automatisch in Java-Datentypen konvertieren. Dies funktioniert auch für eigene Klassen, wenn eine statische `fromString(String)`-Methode vorhanden ist. Da dies einfacher ist als separate Konvertierungsklassen zu schreiben und die Methoden, die die eigentliche Konvertierung vornehmen schon existieren, wurden die Klassen `MemoryUnit` und `TimeInterval` unter Verwendung der bereits für die Konfiguration vorhandene Konvertierungsmethoden ergänzt.

3.2 Initialisierung der Web-Anwendung

Das Hochfahren des Web-Clients unterscheidet sich deutlich von anderen Komponenten der VerifierCloud, die als normale Java-Applikationen gestartet werden und dann den initialen Objekt-Graphen mit Hilfe des Injection-Framworks *Guice*[3] herstellen. Der Web-Client hingegen läuft als Jersey-Web-Anwendung auf einem Java-Anwendungsserver, wie zum Beispiel *Apache Tomcat*[4].
Der verwendete Java-Anwendungsserver lädt den Web-Client. Dabei wird die *web.xml*-Datei gelesen, die den Pfad, unter der die Anwendung erreichbar sein soll, die zu verwendete `Servlet`-Klasse und deren Initialisierungsparameter enthält. Es handelt sich um den Servlet-Container von Jersey, dem als Parameter der Name der Klasse `WebClientResourceConfig` mitgegeben wird. Jersey instanziiert ein `WebClientResourceConfig`-Objekt und ruft dessen `initialize()`-Methode auf, da sie mit `@PostConstruct` annotiert ist.
`WebClientResourceConfig` erweitert `AbstractWebClientResourceConfig`, da es noch eine

[3] `https://github.com/google/guice`
[4] `http://tomcat.apache.org/`

Dummy-Klasse für Testzwecke gibt, die ebenfalls von `AbstractWebClientResourceConfig` erbt. Sie stellt aus Sicht von Jersey den Einstiegspunkt der Web-Anwendung dar. Die Initialisierungsmethode von `AbstractWebClientResourceConfig` konfiguriert einige Eigenschaften von Jersey, bevor sie die Multipart-Unterstützung von Jersey, Untersützung für komprimierte HTTP-Inhalte sowie rekursiv alle Klassen des Pakets `org.sosy_lab.verifiercloud.client.applications.webclient` registriert, so dass alle darin enthaltenen Klassen und Schnittstellen nach relevanten Annotationen durchsucht und gegebenenfalls entsprechend registriert werden. Dies umfasst unter anderem die `WebClientResource`-Klasse, von der je Anfrage ein Objekt erstellt wird, das diese beantwortet, und Provider-Klassen, die benötigte Objekte zur Verfügung stellen. Zuletzt wird das von den Subklassen durch eine abstrakte Methode bereitgestellte `WebClientAPI`-Objekt registriert, indem es in einem `WebClientBinder` gekapselt wird. Auf diese Weise wird es bei *HK2*[5], dem Injection-Framework von Jersey registriert. So steht jeder `WebClientResource`-Instanz automatisch die `WebClientAPI` zur Verfügung.
Falls noch nicht zuvor geschehen, registriert `WebClientResourceConfig` den `DefaultWebAppReloader`, den es als statisches Feld enthält, damit er auch beim Neuladen des Web-Clients erhalten beleibt. Darüber hinaus implementiert es `getWebClientAPI()`. Sie erzeugt mit Hilfe von Guice ein `DefaultWebClientAPI`-Objekt samt dem davon abhängigen Objekt-Graphen. Dies geschieht auf gleiche Weise, wie bei allen anderen Komponenten der VerifierCloud, weswegen alle vorhandenen Schnittstellen und das Konfigurationssystem von der `DefaultWebClientAPI` verwendet werden können. `WebClientAPIModule`, das `ClientModule` erweitert, liest in der `initialize()`-Methode die Konfiguration ein, erzeugt die konfigurierten `Logger` und setzt einige Optionen, wie `FileContent`-Objekte deserialisiert werden sollen. Die `configure()`-Methode definiert die zu verwendeten Implementierungen und setzt die Konfigurationsoptionen, so dass anschließend beides bei der Erzeugung des Objekt-Graphen berücksichtigt wird.
In der *web.xml*-Datei ist eingestellt, dass die Applikation beim Start direkt geladen werden soll, damit die Konfiguration geladen, die Verbindung zur VerfierCloud aufgebaut wird und Vorberechnungen gestartet werden. Auf diese Weise werden Fehler schon beim Start erkannt und die Beantwortung der ersten Anfrage wird nicht verzögert.

3.3 Anbindung an Versionsverwaltung

Der Web-Client verwendet einen lokalen Git[6]-Klon des SVN-Repositories des Verifikationswerkzeuges. Der Zugriff auf den Git-Klon erfolgt mit dem `DefaultGitManager` als Im-

[5] `https://hk2.java.net/2.3.0/`
[6] `http://git-scm.com/`

plementierung der `GitManager` Schnittstelle. Dabei kommt JGit[7] in Version 3.4.1 mit der Erweiterung für Java 7 als Bibliothek zum Einsatz. Außerdem muss Git mit git-svn mindestens in Version 2.0 oder neuer installiert sein. Das SVN-Repository muss als Quelle der SVN-Informationen für den Git-Klon konfiguriert sein. Der Trunk wird intern wie eine Branch mit dem Namen *trunk* verarbeitet.
Vor allen Zugriffen wird der Git-Klon aktualisiert. Standardmäßig geschieht das mit dem Befehl *git svn fetch*. Während der Initialisierung des `DefaultGitManager`s wird dies nebenläufig gestartet, weil dieser Vorgang sehr lange dauern kann und Tomcat den Startvorgang mit einem Fehler beendet, wenn dieser nicht innerhalb von 45 Sekunden abgeschlossen werden konnte. Der Aktualisierungsvorgang kann von außen getriggert werden, damit der Git-Klon immer aktuell ist, wodurch Anfragen schneller bearbeitet werden können. Es gibt die Möglichkeit, die automatische Aktualisierung zu deaktivieren, wenn der Trigger bei jeder neuen Version im SVN-Repository ausgeführt wird, wodurch sich die für die Anfragebearbeitung benötigte Zeit weiter verkürzt.
Der `GitManager` kann eine beliebige existierende SVN-Revision auschecken und den Pfad dazu zurückgeben. Dabei ist wichtig, dass der Aufrufer solange auf das `GitManager`-Objekt synchronisiert, wie er auf die ausgecheckten Dateien zugreift, damit nicht zwischenzeitlich eine andere Revision ausgecheckt wird. Der `DefaultGitManager` führt immer einen *Reset* im Modus *hard* unter Verwendung von JGit durch durch, wofür der Hash der der SVN-Revision entsprechenden Git-Revision benötigt wird. *git svn find-rev –before* mit der SVN-Revisionsnummer und dem Namen des Branches oder Tags aus Sicht des Git-Klons als Parameter liefert den gesuchten Hash-Wert. Die Option *before* bewirkt, dass von der gegebenen Revisionsnummer rückwärts gesucht wird, falls für die Revision keine Änderung in dem Branch oder Tag vorgenommen wurde. Anderenfalls wäre die Ausgabe in einem solchen Fall leer. Es wird auch überprüft, dass die Revisionsnummer nicht neuer als die neuste Revision in diesem Branch oder Tag ist.
Svn-Revisionen werden durch `SvnRevision`-Objekte repräsentiert. Abbildung 3.1 zeigt die dafür verwendeten Klassen und Schnittstellen. *HEAD*-Revisionen sind nicht dafür geeignet in Caches eingefügt zu werden, weil die dahinterstehende Revisionen über die Zeit veränderlich ist. Deshalb gibt es die Schnittstelle `ImmutableSvnRevision` mit je einer implementierenden Klasse für Tags und Branches. Eine weitere Funktion des `GitManager`s ist es, *HEAD*-Revisionen zu einer Revisionsnummer aufzulösen. Dafür wird *git svn find-rev* mit dem Branch- oder Tagnamen ausgeführt. Die letzte Zeile der Ausgabe ist dann die neuste Revisionsnummer des Branchs oder Tags. `SvnRevision`-Implementierungen nutzen dies um eine `ImmutableSvnRevision` zu erzeugen, die dem aktuellen Zeitpunkt entspricht.

[7] `http://www.eclipse.org/jgit/`

`SvnRevisionPattern` wird von der Konfiguration genutzt um mehr als eine Revision auf einmal zur Verwendung im Web-Client freizugeben. Es stellt jeweils eine `SvnRevision` bereit, die vorberechnet werden soll.

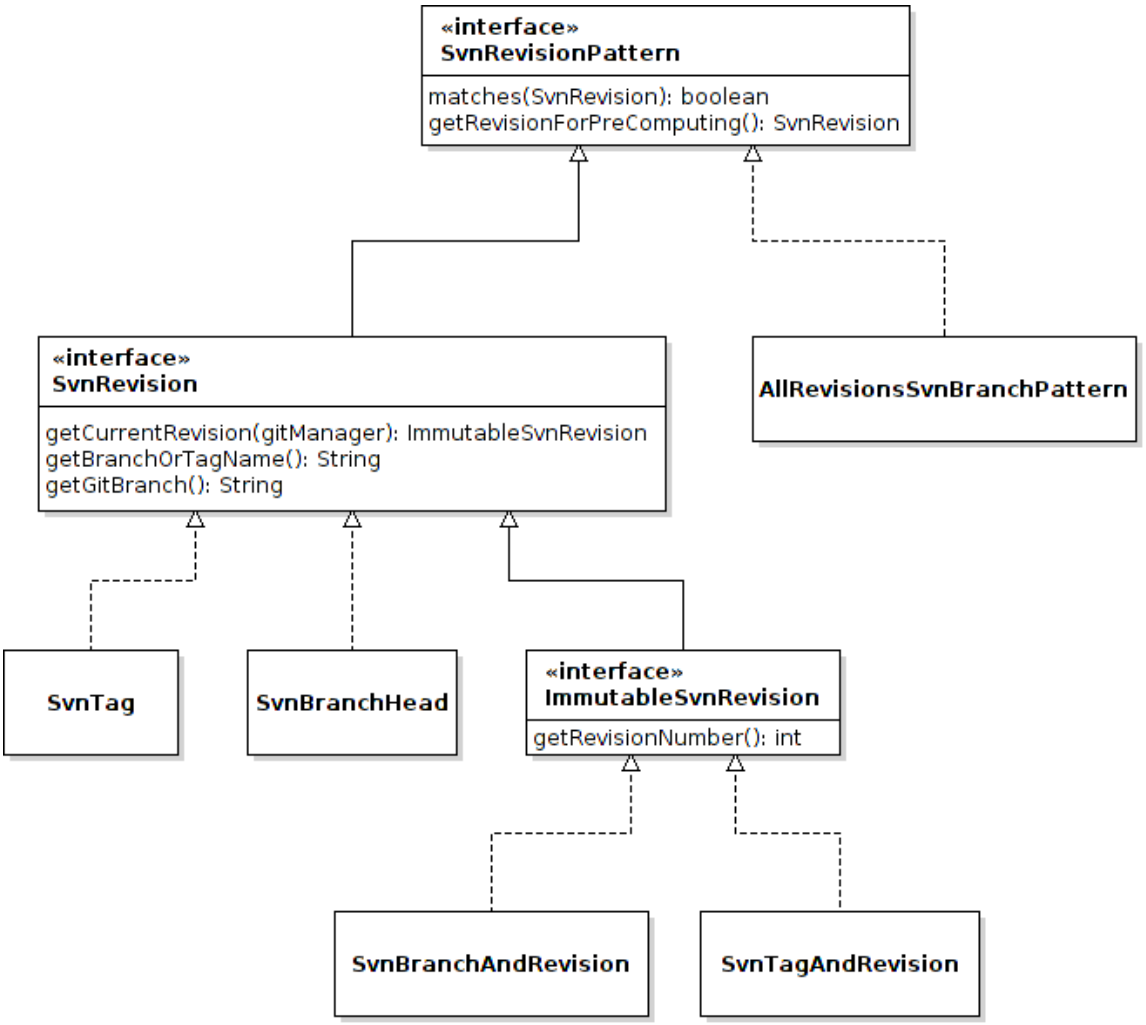

Abbildung 3.1: Klassendiagramm der `SvnRevision`-Klassen und -Schnittstellen

3.4 Separate Dateiinhalt-Speicherung

Die `ClientFileStorage`-Schnittstelle und deren Implementierungen erfüllen nicht alle Anforderungen, die durch den Web-Client gestellt werden. Abbildung 3.2 zeigt die Beziehung der verschiedenen `ClienFileStorage`-Implementierungen.

Die Schnittstelle `WebClientFileStorage` erweitert `ClientFileStorage` um die Möglichkeit auch `FileContent`-Objekte einzufügen und die Methode `cleanup()`, die beim Been-

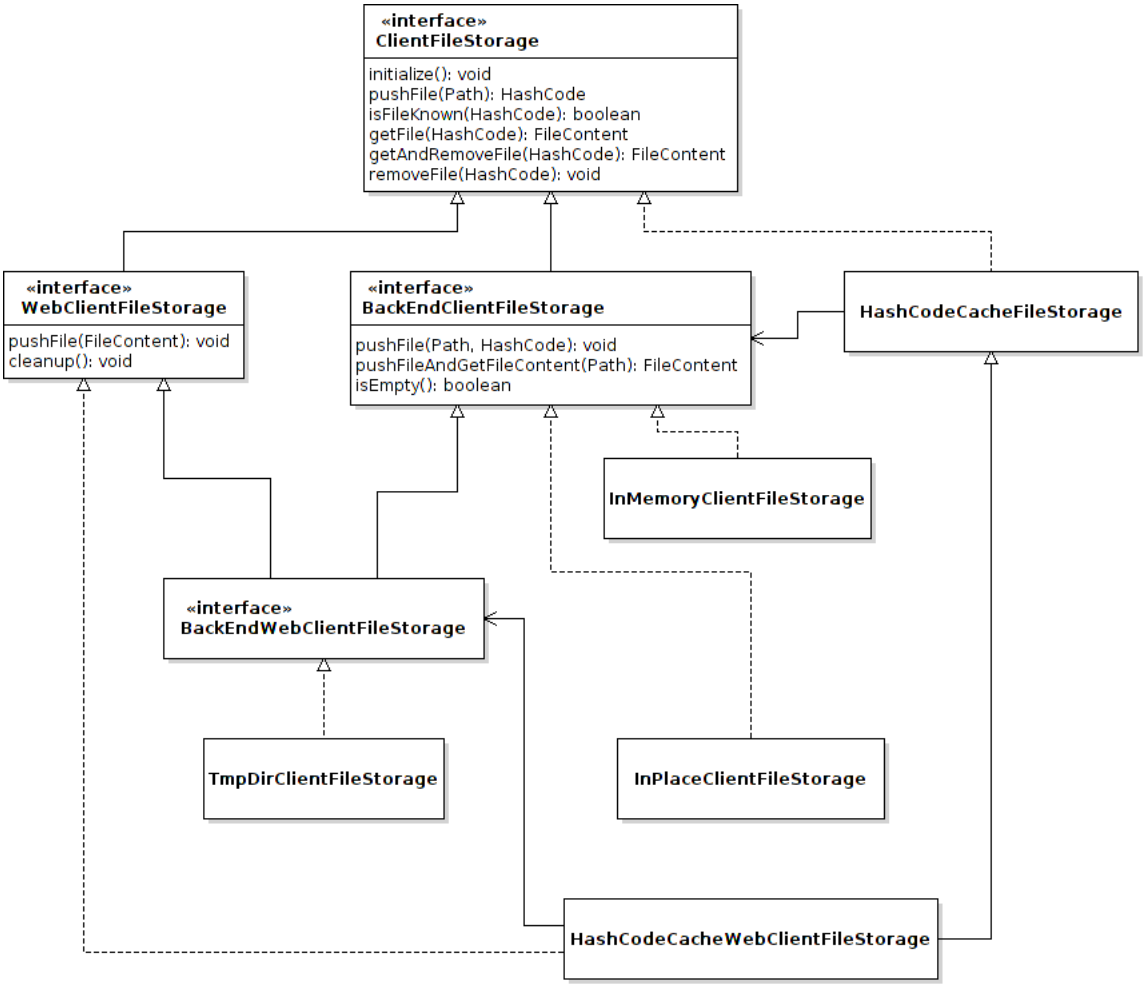

Abbildung 3.2: Klassendiagramm der `ClientFileStorage`-Klassen und -Schnittstellen

den einer Web-Client-Instanz Aufräumoperationen ausführt, wie das Beenden von internen Threads und Löschen von noch vorhandenen temporären Dateien. `BackEndWebClientFileStorage` ist die Vereinigung der Schnittstellen `WebClientFileStorage` und `BackEndClientFileStorage`.

Der Web-Client muss in der Lage sein, den CPAchecker innerhalb kurzer Zeit in verschiedenen Revisionen bereitzustellen, weshalb die im lokalen Git-Klon ausgecheckte Revision nur möglichst kurz verwendet werden sollte. Die bisherige `ClientFileStorage`-Implementierung liest die Dateien direkt vom ursprünglich eingefügten Pfad, wenn der Master sie anfordert. Der Web-Client kann nicht warten, bis alle Dateien durch den Master angefragt bzw. als nicht benötigt markiert wurden, weil dies potentiell unendlich lange dauern kann. Deswegen wurde der `TmpClientFileStorage` als Implementierung der `BackEndWebClientFileStorage`-Schnittstelle erstellt. Eingefügte Dateien werden in einen

temporären Ordner kopiert, so dass sie unabhängig vom Ursprungspfad bereitstehen.
Bisher wird vom Benchmark-Client eine `ClientFileStorage`-Implementierung verwendet, die die Hashwerte von Dateien persistent speichert, wodurch bei der nächsten Verwendung des Clients diese nicht neu berechnet werden müssen. Außerdem verwendet die Implementierung einen Lese-Cache. Dies bewirkt in der Praxis ein erheblich schnelleres Erstellen der Runs. Zum Speichern des Dateiinhalte wird ein `BackEndFileStorage` verwendet. `HashCodeCacheWebClientFileStorage` erweitert `HashCodeCacheFileStorage` und implementiert `WebClientFileStorage`. Zum Speichern der Dateiinhalte verwendet er einen `BackEndWebClientFileStorage`.
Auf diese Weise werden die Vorteile der bisherigen Implementierung genutzt und durch die neuen Klassen und Schnittstellen entsprechend der zusätzlichen Anforderungen ergänzt.

3.5 REST-Schnittstelle

Die REST-Schnittstelle verwendet Jersey, das HTTP-Anfragen vom HTTP Server entgegennimmt und dann die entsprechende Methoden von `WebClientResource` aufruft.

3.5.1 Resource-Klasse

Die Klasse `WebClientResource` enthält für jede zulässige Anfrage eine Methode, die diese beantwortet. Dafür sind Pfad, HTTP-Methode, konsumierter und gegebenenfalls produzierter Mediatype durch Annotationen definiert. Für jede Anfrage wird ein neues `WebClientResource`-Objekt erstellt, dem per Konstruktor die `WebClientAPI` und die HTTP-Anfrage, aus der die Client-Adresse extrahiert wird, mitgegeben werden. Alle Methoden loggen die Anfrage mit Parametern und Client-Adresse, bevor die entsprechende Methode der `WebClientAPI` aufgerufen wird.
Die ID des Runs wird bei Abfrage des Status oder des Ergebnisses als Teil des Pfades übergeben. Die Methoden zum Herunterladen des CPAcheckers und zum Einreichen von Verifikationsaufgaben nehmen die Parameter als Bean-Parameter entgegen, weswegen sie mit `@BeanParam` annotiert sind. Dabei werden die Parameter den Konstruktoren der verwendeten Bean-Klassen übergeben. Letztere werden dann der Methode in `WebClientResource` als Parameter übergeben.

3.5.2 Bean-Klassen

Die POST-Parameter bei der Einreichung von Verifikationsaufgaben werden in Bean-Klassen entgegengenommen und aufbereitet. Dabei wird der Ressource-Methode jeweils eine Implementierung der Schnittstellen `RunConfiguration` und `ArgumentFilesBean` übergeben. Es

gibt jeweils eine Implementierung für *application/x-www-form-urlencoded* und *multipart/form-data*. `AbstractRunConfiguration` verwendet `LimitationsBean` und `SvnRevisionBean`, die die übergebenen Parameter zu einem `Limitations-` beziehungsweise `SvnRevision-`Objekt zusammenfassen. `QuerySvnRevisionBean` nimmt Prarameter aus einer URL entgegen, was für Anfragen zum Herunterladen des Verifikationstools verwendet wird. `DefaultStartRunInformation` enthält ein `ArgumentFilesBean-` und ein `RunConfiguration` Objekt sowie einen booleschen Wert, der angibt, ob es ein Run mit Premium-Priorität ist. Das Klassendiagramm 3.3 zeigt den Zusammenhang der zur Parameterentgegennahme verwendeten Klassen.

3.6 WebClientAPI

Die `WebClientAPI` stellt Methoden zur Bearbeitung der Anfragen und zum Zugriff auf Klassen, die durch Guice zur Verfügung gestellt werden, bereit. Das sind der `Logger`, der `HelpTextGenerator` und Konfigurationsoptionen.
`DefaultWebClientAPI` implementiert `WebClientAPI` und `ConnectionEventHandler`, damit beim Verlust der Verbindung zum Master eine Neuinitialisierung der Web-Anwendung angestoßen werden kann. Die `initialize()`-Methode startet den Vorgang zum Erstellen der Verbindung zum Master, der alle zwei Minuten neue Verbindungs- und Authentifizierungsversuche startet, bis diese erfolgreich verlaufen. Wenn keine Verbindung zum Master besteht, werden alle Anfragen, die eine solche benötigen, mittels einer `ServiceUnavailableException` abgewiesen.
In den folgenden Abschnitten wird auf die Bearbeitung der einzelnen Anfragen näher eingegangen. Die `DefaultWeb-ClientAPI` verwendet zahlreiche Klassen, die die meiste Funktionalität des Web-Clients bereitstellen.

3.7 Generierung des Hilfetextes

`HelpTextGenerator` generiert einen kurzen Hilfetext, der die Funktionen des Web-Clients erklärt und über die aktuell konfigurierten Beschränkungen informiert. Damit die Verweise zu den einzelnen Funktionen die tatsächliche URL der Web-Anwendung verwenden, werden sie aus der URL der Anfrage generiert. Weil die Web-Anwendung aus dem Internet möglicherweise nicht unter der lokal verwendeten URL erreichbar ist, kann die URL auch fest konfiguriert werden. Dies bietet darüber hinaus den Vorteil, dass sich wegen des konstanten Wertes für die Java Virtual Machine bessere Optimierungsmöglichkeiten bieten.

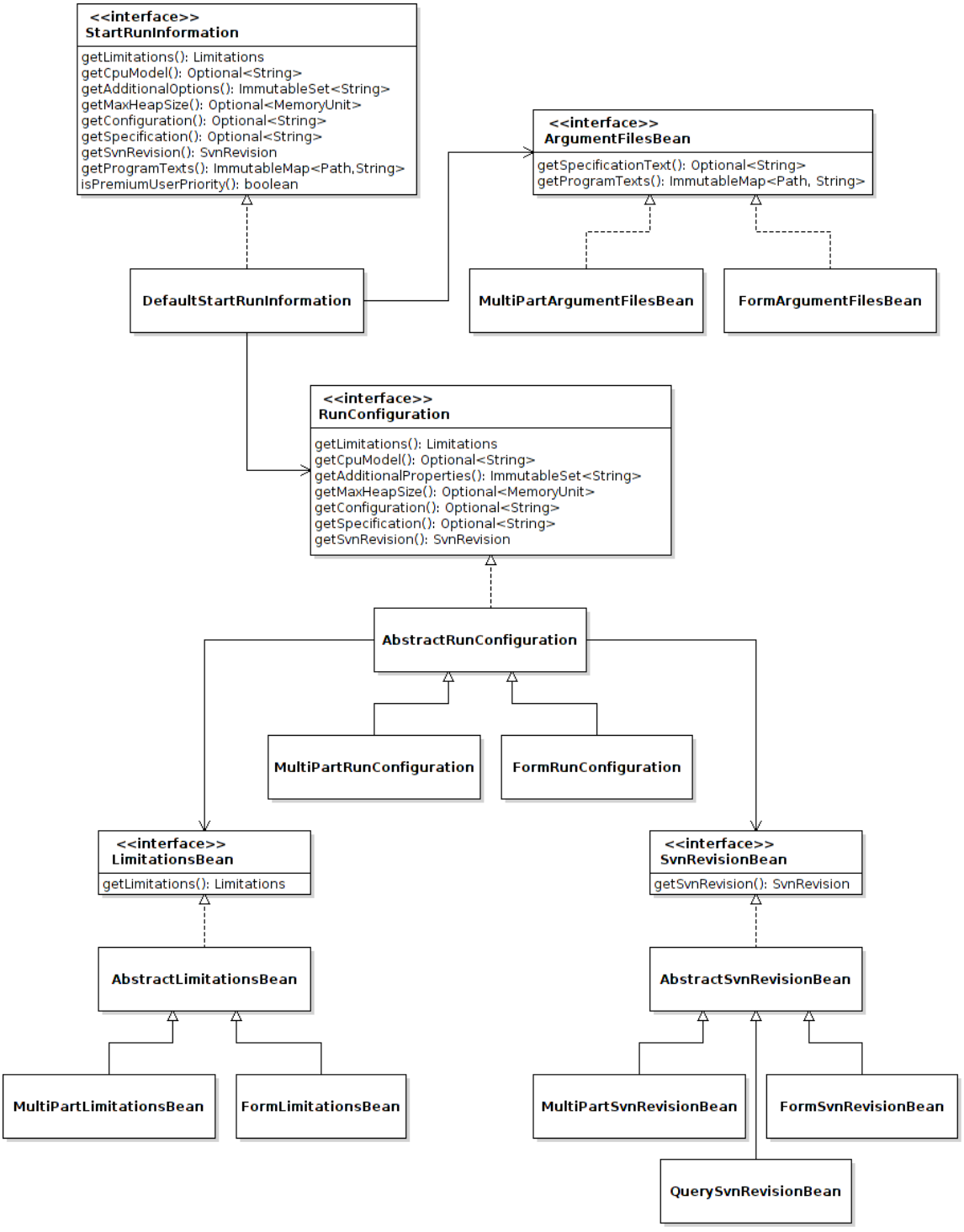

Abbildung 3.3: Klassendiagramm der Bean-Klassen und -Schnittstellen

3.8 Generierung der Master-Informationsseite

Der Web-Client stellt die gleiche HTML-Seite bereit, die der Info-Client generiert. Im Gegensatz zum Info-Client berechnet der Web-Client die Seite nur, wenn sie angefordert wird und behält sie dann für eine einstellbare Zeitspanne in einem Cache. Zur Transformation der durch den Master bereit gestellten Informationen zum Zustand der VerifierCloud in die HTML-Seite wird der bereits vorhandene `HTMLOutputFormatter` genutzt, wodurch für diese Funktion nur wenig neuer Quellcode geschrieben werden musste.
Bisher erstellt der Info-Client alle fünf Sekunden die Seite neu, wozu der Master jedes Mal alle notwendigen Informationen berechnen und dem Info-Client zuschicken muss, und speichert sie im Dateisystem. Die Seite ist durch ein PHP-Skript über das Internet zugänglich.

3.9 Kompilieren des CPAcheckers

Da der CPAchecker im SVN-Repository nur als Quelltext vorliegt, muss er, bevor er ausgeführt werden kann, kompiliert werden. Außerdem werden die meisten benötigten Bibliotheken erst während des Bauprozesses mittels *ivy*[8] nachgeladen. Zum Bauen des CPAchecker wird *ant*[9] verwendet, das auch ein jar-Archiv ausgeben kann.

3.9.1 Kompilieren in der VerifierCloud

Da das Übersetzten des CPAcheckers einige Zeit dauern kann, sollte es nicht durch den Webserver selbst durchgeführt werden. Die VerifierCloud stellt eine große Zahl an Rechnern als Worker bereit, weshalb der Web-Client diese Aufgabe durch einen Worker erledigen lässt.
Der `EagerProgramFilesProvider` stellt die benötigten Dateien für den CPAchecker als Dateihierachie bereit, indem er den CPAchecker vom `GitManager` in der geforderten Revision auschecken lässt, um anschließend einen Run zu erstellen, der den CPAchecker kompiliert. Dieser enthält alle für das Bauen des CPAcheckers benötigten Dateien. Weil während der Ausführung des Runs auf dem Worker keine SVN-Informationen mehr vorhanden sind, wird im Baubefehl die Zeichenkette *$revision* durch die Revisionsnummer ersetzt, damit die sonst beim lokalen Bauen des CPAcheckers verfügbare Versionsnummer auch nutzbar ist. Aus den Dateien des Run-Ergebnisses werden entsprechend der konfigurierten Muster Dateien der Dateihierarchie hinzugefügt. Dateiinhalte aus Run-Resultaten liegen immer am Master vor, wenn der Client das Ergebnis erhält.

[8] `http://ant.apache.org/ivy/`
[9] `http://ant.apache.org/`

Zur Ausführung des CPAcheckers sind zusätzliche Dateien aus dem Repository notwendig, die nicht das Ergebnis des Übersetzungsprozesses sind. Diese werden direkt in die Dateihierarchie der entsprechenden Revision eingefügt ohne sie dem Kompilier-Run hinzuzufügen, weil sie sonst unnötigerweise in der VerifierCloud transportiert würden.
Das CPU-Modell, auf dem der Kompilier-Run ausgeführt wird, wird durch die Konfiguration festgelegt. Dabei können mehrere Werte angegeben werden. Der `EagerProgramFilesProvider` berechnet alle aktuell verfügbaren CPU-Modelle und wählt dann die erste der konfigurierten Zeichenkette aus, die zu mindestens einem mit der VerifierCloud verbunden Worker passt. Dabei werden zunächst nur die Worker als verfügbar betrachtet, die in diesem Moment den Kompilier-Run tatsächlich ausführen könnten, weil sie über ausreichend freie Kapazität verfügen. Falls auf diese Weise keines der konfigurierten CPU-Modelle ausgewählt werden kann, werden alle mit dem Master verbundene Worker in Betracht gezogen. Wenn unter den verbunden Workern kein passendes CPU-Modell gefunden wird, wird das erste der konfigurierten gewählt. Falls kein CPU-Modell für den Kompilier-Run konfiguriert ist, wird das CPU-Modell offen gelassen und der Master weist den Run einem zufälligen Worker zu. Durch dieses Verfahren steht das Verifikationswerkzeug möglichst schnell in gebauter Form zur Verfügung, weil der Kompilier-Run in den meisten Fällen durch den Master sofort einem Worker zugewiesen werden kann.

3.9.2 Vorausberechnung der CPAchecker-Revisionen

Das Übersetzten des CPAcheckers dauert in der Regel auch in der VerifierCloud noch über eine Minute. Weil der Nutzer nicht bei jeder Anfrage so lange warten müssen sollte, werden die Dateihierarchien einmal gebauter Revisionen in einem Cache vorgehalten. Dies benötigt kaum Speicher, da die eigentlichen Dateiinhalte schon zum Master transportiert wurden, der alle erhaltenen Dateiinhalte dauerhaft speichert, und sie somit nicht mehr vom Web-Client gespeichert werden müssen. Außerdem wird dadurch bei gleichzeitigen Anfragen mit derselben Revision der CPAchecker nur einmal übersetzt und das Ergebnis für alle Anfragen verwendet.
Für alle zulässigen `SvnRevisionPattern` berechnet der `EagerProgramFilesProvider` beim Start des Web-Clients die Dateihierarchie einer Revision des CPAchecker vor, von der davon ausgegangen werden kann, dass sie als nächstes genutzt wird. Das ist jeweils die neuste Version eines `SvnRevisionPattern`.

3.10 Ausführen des Verifikationswerkzeuges

Zum Ausführen des Verifikationswerkzeuges muss eine `RunCollection` mit einem `Run` erstellt werden. Dies erledigt jeweils eine `WebRunCollectionBuilder`-Instanz, die durch eine `WebRunCollectionBuilderFactory`-Implementierung aus einer validierten `StartRunInformation`-Instanz, die alle für diese Run spezifischen Informationen enthält, erzeugt wird. Die Konfigurationsoptionen und Instanzen benötigter Klassen werden der `WebRunCollectionBuilderFactory` während der Initialisierung des Web-Client durch das Injection-Framework übergeben.
Der `WebRunCollectionBuilder` holt sich die Dateien, die für die Ausführung des Verifkationswerkzeuges in der gegebenen Revision, notwendig sind, vom `ProgramFilesProvider` in Form eines `TreeFileHierachyBuilders`. Dieser generiert mit den Programmtexten und gegebenenfalls der Spezifikation als zusätzliche Dateien eine `FileHierachy` für den zu erzeugenden Run. Außerdem werden für den `Run` Beschränkungen, Anforderungen an den Rechner, auf dem er ausgeführt wird, und der Befehl entsprechend den gegebenen Informationen gesetzt. Wichtig ist noch die Einstellung, dass die Ergebnisse als Zip-Datei am Master gespeichert werden, da sie später von dort abgerufen werden. Zuletzt wird der Run in eine `RunCollection` gekapselt sowie deren Priorität gesetzt, weil der Master keine einzelnen `Run`s entgegennehmen kann.
Die `DefaultWebClientAPI` sendet diese RunCollection zum Master, der, nachdem er alle benötigten Dateiinhalte erhalten und die RunCollection an den Scheduler übergeben hat, eine Erfolgsmeldung zurück schickt. Wenn die `DefaultWebClientAPI` diese erhält, gibt sie die ID des `Run`s zurück und speichert die Zuordnung von `Run` zu `RunCollection`, da dies für die Bestimmung des Zustandes eines Runs später benötigt wird.

3.11 Bereitstellen der Run-Zustände und der Resultate

Die `DefaultWebClientAPI` berechnet die Run-Zustände aus der `WorkerSummary`, die vom Master angefordert wird. Das Ergebnis der Berechnung wird für fünf Sekunden in einem Cache aufbewahrt, damit der Rechenaufwand nicht zu hoch ist und Anfragen in aller Regel sehr schnell beantwortet werden können. Beides ist wichtig, weil die Clients der Benutzer voraussichtlich so programmiert sind, dass sie die Zustände aller eingereichten Aufgaben in kurzen Intervallen abfragen, bis diese erledigt wurden. Es werden jeweils für alle bekannten, das heißt durch durch die `DefaultWeb-ClientAPI` gestarteten Runs, die Zustände berechnet, wozu die Zuordnung von `Run`-IDs zu `RunCollection`-IDs gespeichert werden muss, weil die `WorkerSummary` nur Informationen zu `RunCollection`s bereit stellt.
Die Resultate fertiger Runs werden am Master als Zip-Datei persistent gespeichert und kön-

nen den Clients als `FileContent` geschickt werden. Der Web-Client reicht also Anfragen bezüglich der Run-Resultate an den Master weiter und gibt das Ergebnis als `StreamingOutput` zurück, weil es potentiell sehr groß sein und deswegen möglicher weise nicht im Speicher gehalten werden kann. Außerdem wird mittels des Header-Parameters *Content-Disposition* der Name des Resultats gesetzt, damit HTTP-Clients einen sinnvollen Dateinamen verwenden können.

3.12 Bereitstellen des gebauten Verifikationswerkzeuges

`ProgramProvider`-Implementierungen stellen das gebaute Verifikationswerkzeug als Zip-Datei bereit. Sie enthält alle Dateien, die bei der Ausführung der Verifikation-Runs auf den Workern verwendet werden. Der Web-Client verwendet einen `CachingProgramProvider`, der einmal erzeugte Zip-Dateien im verwendeten `WebClientFileStorage` ablegt und bei erneuter Anfrage von dort lädt, so dass das Zip-Archiv für jede Revision maximal einmal erzeugt wird.
`CachingProgramProvider` löst die gegebene SVN-Revision mit Hilfe des `GitManager`s zu einer `ImmutableSvnRevision` auf. Der `CachingProgramProvider` holt sich die Hash-Werte und Pfade aller Programdateien dieser Revision vom `ProgramFilesProvider`. Die Dateiinhalte befinden sich entweder im `WebClientFileStorage` oder am Master, weswegen sie erst lokal und, falls sie dort nicht vorhanden sind, am Master angefragt werden. Diese Dateiinhalte werden anschließend an den entsprechenden Pfad in das Zip-Archiv geschrieben. Das Archiv wird in einem temporären Ordner angelegt und nach Hinzufügen aller Dateien dem `WebClientFileStorage` hinzugefügt. Der `CachingProgramProvider` holt sich das Archiv als `FileContent` vom `WebClientFileStorage` und gibt es an den Methodenaufrufer zurück. Jersey wird der Dateiinhalt, wie bei den Verfikationsergebnissen, als `StreamingOutput` zurückgegeben.

3.13 Sicherheitskonzept

Da die Web-Schnittstelle des Web-Clients über das Internet erreichbar ist, kann prinzipiell jeder den CPAchecker mit selbst gewählten Optionen auf Rechnern, auf denen ein Worker der VerifierCloud läuft, ausführen. Es wird angenommen, dass der CPAchecker selbst keine unmittelbaren Schwachstellen besitzt, jedoch können nicht alle Optionen als sicher betrachtet werden, zum Beispiel kann der Präprozessor für die C-Programme konfiguriert werden, wodurch es möglich ist, beliebige vorhandene Programme auszuführen und gegebenenfalls das zu verifizierende C-Programm selbst zu übersetzten und zu starten.
Durch einige Sicherheitsvorkehrungen sollen solche Schwachstellen verhindert werden. Zum

einen können nur in der Konfiguration definierte Revisionen ausgeführt werden und zum anderen können keine benutzerdefinierten CPAchecker-Konfigurationen verwendet werden, sondern nur solche aus dem SVN-Repository. Diese müssen einem konfigurierbaren regulären Ausdruck genügen und einzelne Namen können verboten werden, weil die Namen der Konfigurationen als Parameter übergeben werden. Wenn der Benutzer als Namen eine existierende Option wählt und den darauf folgenden Dateinamen geschickt setzt, können beliebige Optionen gesetzt werden. Für den CPAchecker sind *setprop* und *config* kritisch. Da durch diese beiden Maßnahmen die Benutzer möglicherweise zu stark einschränkt werden, besteht die Möglichkeit einzelne Optionen freizugeben, die durch den Benutzer selbst gesetzt werden können. Für diese gelten jedoch strenge Regeln um Missbrauch vorzubeugen: Es sind ausschließlich Schlüsselwertpaare zulässig, die nicht die Zeichen $<$, $>$, *&*, *|*, *;* oder ␣ enthalten.
Darüber hinaus soll vermieden werden, dass Benutzer des Web-Clients übermäßig viele Ressourcen verwenden. Deshalb gibt es für Zeit, Speicher und CPU-Kerne Begrenzungen der maximal erlaubten Limits.
Diese Überprüfung von Anfragen erfolgt durch `DefaultStartRunInformationValidator`. Bei Verletzungen der definierten Regeln wird eine Ausnahme ausgelöst, die eine Abweisung der Anfrage und eine Benachrichtigung des Benutzers bewirkt.
Es gibt Überlegungen in den CPAchecker einen sicheren Modus einzubauen, der intern dafür sorgt, dass keine sicherheitskritischen Aktionen ausgeführt werden. Die Überprüfung der über die Kommandozeile gesetzten Optionen durch CPAchecker selbst, würde den Konfigurationsaufwand senken, da zur Zeit jede Option einzeln freigegeben werden muss. Außerdem können Dateizugriffe außerhalb der Umgebung des entsprechenden Runs kontrolliert werden. Dies ist durch die VerifierCloud selbst praktisch nicht möglich.

3.14 Konfiguration

Der Web-Client verwendet das vorhandene Konfigurationssystem der VerifierCloud. Im Gegensatz zu den anderen Modi der VerifierCloud, lassen sich die Optionen nur über eine Konfigurationsdatei anpassen und nicht durch Kommandozeilenparameter, da der Web-Client nicht direkt von der Kommandozeile gestartet wird. Für Testzwecke wurde deshalb die Möglichkeit geschaffen über Java-Properties die fest einkodierten Standardwerte aller Optionen zu überschreiben. Neben der allgemeinen Client-Konfiguration, die von allen Clients verwendet wird, enthält die Klasse `WebClientConfiguration` alle Web-Client-spezifischen Konfigurationsoptionen. Diese gliedern sich in drei Gruppen: Einige allgemeine Optionen, Optionen zur Einstellung des Kompilierprozesses des CPAcheckers und Optionen zu Ausführung der Verifikationsaufgaben:

- *localPath*: Pfad unter dem Dateien während der Ausführung gespeichert werden, der Web-Client muss dort Lese- und Schreibrecht haben
- *gitRepository*: Pfad zum lokalen Klon des Git-Repositories
- *requiredFilePatterns*: kommaseparierte Liste von Glob-Mustern der Dateien aus dem CPAchecker-Repository, die für die Ausführung des CPAchecker notwendig sind
- *command*: Befehl zum Starten des CPAcheckers auf den Workern
- *allowedRevisionsAndBranches*: kommaseparierte Liste von SVN-Revisionen des CPAcheckers, die durch den Web-Client in der VerifierCloud ausgeführt werden dürfen:
  - <Branchname>:<Revision>: die angegebene Revision des Branches bzw. des Trunks, Revision kann auch *HEAD* sein
  - <Branchname>:*: alle Revisionen des Branches bzw. des Trunks
  - <Tagname>: Die neuste Revision des Tags
- *allowedOptions*: kommaseparierte Liste von Namen der CPAchecker-Optionen, die durch den Nutzer gesetzt werden dürfen
- *configAndSpecPattern*: regulärer Ausdruck der zulässigen Konfigurationen und Spezifikationen
- *disallowedSpecAndConfig*: Namen, die aus Sicherheitsgründen nicht als Konfiguration oder Spezifikation verwendet werden dürfen, z.B. 'setprop' und 'config'
- *normalPriority*: Priorität mit der normale Runs am Master gescheduled werden
- *premiumPriority*: Priorität mit der über die Premium-URL eingereichte Runs am Master gescheduled werden
- *preDefinedLimitations*: nicht leere, semikolonseparierte Liste von Schlüsselwertpaaren aus Name der vordefinierten Beschränkung und der jeweiligen Beschränkung als kommasepariertes Tripel, bestehend aus Zeit-, Speicher- und Prozessorzahlbeschränkung
- *timeLimitation*: maximal zulässige Zeitbeschränkung
- *memoryRequirementLimitation*: maximal zulässige Speicherbeschränkung
- *coresRequirementLimitation*: maximal zulässige Prozessorzahlbeschränkung
- *buildCommand*: Befehl zum Bauen des CPAcheckers, *${revision}* wird durch die Revisionsnummer ersetzt

- *buildCommandrequiredFilePatterns*: kommaseparierte Liste von Pfaden und Glob-Mustern der Dateien, die zum Bauen des CPAcheckers benötigt werden
- *buildCommandResultFiles*: kommaseparierte Liste von Glob-Mustern der Dateien aus dem Ergebnis des Build-Runs, die für die Ausführung des CPAcheckers notwendig sind
- *buildCommandTimeLimitation*: Zeitbeschränkung des Build-Runs des CPAcheckers
- *buildCommandMemoryRequirementLimitation*: Speicherbeschränkung des Build-Runs des CPAcheckers
- *buildCommandCoresRequirementLimitation*: Beschränkung der Prozessorkernzahl des Build-Runs des CPAcheckers
- *buildCommandCpuModels*: kommaseparierte Liste von CPU-Namensteilen, auf denen die Build-Runs ausgeführt werden sollen. Der Run wird nur auf Prozessoren ausgeführt, in deren Namen die angegeben Zeichenkette enthalten ist. Es wird jeweils der erste CPU-Namensteilen verwendet, der zu einem verbundenen Worker passt, wenn ein neuer Kompilier-Run gestartet wird.
- *buildCommandSchedulingPriority*: Priorität, mit der Build-Runs des CPAcheckers gescheduled werden
- *username*: Benutzername, mit dem der Web-Client sich beim Master anmeldet. Wenn keiner angegeben ist, wird der Name des Benutzers, unter dem der Web-Client betrieben wird, verwendet.

Falls für Speicher- oder Zeitwerte keine Einheiten angegeben sind, werden die Basiseinheiten Byte und Sekunde angenommen.
Wenn die Konfigurationsdatei nicht existiert, wird eine mit Standardwerten und Beschreibung der einzelnen Parametern erstellt.

3.15 Ausnahmebehandlung

Die Ausnahmebehandlung des Web-Clients weist zwei Besonderheiten auf: Zum einen die Reaktion auf unbehandelte Ausnahmen und zum anderen das Erzeugen von Rückmeldungen, die an den Benutzer geschickt werden.
Alle anderen Komponenten der VerifierCloud beenden sich, wenn in einem internen Thread eine nicht behandelte Ausnahme ausgelöst wird. Der Web-Client als Web-Anwendung, der auf einem Java-Anwendungsserver läuft, kann und sollte den Server nicht anhalten, sondern

diese Störung intern bearbeiten. Dafür wird ein `WebAppReloader` bei Jersey während der Initialisierung des Web-Clients als `ContainerLifecycleListener` registriert, dem bei jedem Start und Neuladen des Web-Clients der aktuelle `Container` der Web-Anwendung übergeben wird. Im Falle einer unbehandelten Ausnahme wird der `WebAppReloader` angewiesen den Web-Client neu zu laden. Dazu wird der `Container` mit einer neuen, initialisierten Instanz der Klasse `WebClientResourceConfig` geladen. Das Framework räumt dabei automatisch die alte Instanz des Web-Clients auf.
Außerdem kann, wie von Master und Worker bereits unterstützt, ein Befehl konfiguriert werden, der in einem solchen Fall oder auch bei einer geloggten Ausnahme ausgeführt wird. Dem gestarteten Prozess werden dabei alle vorhandenen Informationen auf der Standardeingabe zur Verfügung gestellt. Auf diesem Weg könnte zum Beispiel der Administrator per E-Mail benachrichtigt werden.
Bei Auftreten eines Fehler während der Bearbeitung einer Anfrage, sollte der Benutzer darüber informiert werden. Dies ist besonders bei Fehlern, die auf eine fehlerhafte Anfrage zurückzuführen sind, wichtig. Jersey bietet die Möglichkeit Ausnahmen automatisch in Antworten zu transformieren, die an den Client des Benutzers geschickt werden. Dafür wurden für alle Ausnahmen, deren Auftreten zu erwarten ist, `ExceptionMapper` implementiert. Diese erstellen aus der Fehlermeldung der Ausnahme den Text der Antwort. Der Typ der Ausnahme bestimmt den HTTP-Statuscode.
Alle unbehandelten Ausnahmen, für die kein `ExceptionMapper` vorhanden ist, resultieren automatisch in einer Antwort mit HTTP-Statuscode *Internal Server Error*.
Jersey weist Anfragen automatisch zurück, bevor sie an den Web-Client übergeben werden, wenn die Parameter nicht in die entsprechenden Java-Typen transformiert werden können, der *Content-Type* für die entsprechende HTTP-Methode und URL nicht unterstützt wird oder die HTTP-Methode für die gegebene URL nicht unterstützt wird.

4 Qualitätssicherung

Die Funktionsfähigkeit des Web-Clients wird, ebenso wie die anderen Komponenten der VerifierCloud, durch JUnit-Tests und einen automatischen Systemtest sichergestellt, die für jede neue Revision im Trunk automatisch vom BuildBot der VerifierCloud[1] ausgeführt werden. Außerdem wird der Quellcode jeweils statisch durch *FindBugs* und *error-prone* überprüft.
Darüber hinaus wurde der Web-Client schon während der Entwicklung von externen Nutzern verwendet, die aufgetretene Fehler gemeldet haben.

4.1 JUnit-Tests

Mit JUnit-Tests wird die Funktionsfähigkeit der einzelnen Komponenten des Web-Clients gesichert. Als Testumgebung für die meisten Klassen sind einige Dummyklassen als Konstruktorparameter ausreichend.
Der `DefaultGitManagerTest` benötigt einen Git-Klon eines SVN-Repositories. Deshalb wird vor jedem Test ein leeres SVN-Repository erstellt und mit *git-svn* geklont. Anschließend werden je nach Test unterschiedliche Dateien in teilweise mehreren Revisionen in das SVN-Repository eingecheckt, um zu überprüfen, ob die `GitManager`-Implementierung diese dann richtig verarbeitet.
Um auch `WebClientRecource` und die Bean-Klassen, die die Anfrageparameter enthalten, testen zu können, wird das Jersey-Test-Framework[2] mit dem In-Memory-Container verwendet. Dabei werden HTTP-Anfragen gestellt ohne eine HTTP-Verbindung aufzubauen, indem sie vom Test-Framework direkt an den Web-Client weitergeleitet werden. Die Anfragen unterscheiden sich aus Sicht des Web-Clients nur darin, dass keine Informationen über die Verbindung zum HTTP-Client verfügbar sind. In diesem Fall setzt `WebClientResource` die Client-Adresse auf *Unkown*. Bei diesem Test wird auch kontrolliert, ob die Bean-Klassen die POST- bzw. Multipart-Parameter korrekt entgegennehmen.

[1] `http://vcloud.sosy-lab.org/buildbot/`
[2] `https://jersey.java.net/documentation/latest/test-framework.html`

4.2 Automatischer Systemtest

Die VerifierCloud verfügt über einen automatischen Systemtest, der die Funktionsfähigkeit des Masters, des Workers und der Clients überprüft. Dabei werden alle Tests in einer komplett leeren Umgebung und außerdem in der Umgebung, die durch den ersten Testlauf entstanden ist, ausgeführt. Der Systemtest wurde erweitert, damit auch die Funktionen des Web-Clients getestet werden.
Der Web-Client setzt einen Java-Anwendungsserver voraus, der den Jersey-Servlet-Container ausführt. Deshalb kann der Web-Client für Testzwecke auch als eigenständige Anwendung gestartet werden. Dafür bindet `WebClientStandalone` den Appache Tomcat in Form von jar-Dateien ein und lädt die war-Datei des Web-Clients. Mit *ant webclient-jar* wird der Web-Client als eigenständige Anwendung gebaut.
Nachdem der Systemtest Master und Worker gestartet und die anderen Clients getestet hat, startet er den Web-Client und wartet bis der Anwendungsserver Anfragen beantwortet. Anschließend wird die Hilfeseite sowie die Informationsseite zum Zustand des Masters angefragt und die Antworten überprüft. Danach versucht der Systemtest CPAchecker in Version 1.3.4 als Zip-Archiv herunterzuladen, dessen Größe überprüft wird. Im Anschluss startet der Systemtest eine Verifikationsaufgabe mit der neuesten Trunk-Version des CPAcheckers, kontrolliert die empfangene ID, mit der er dann den Status fortlaufen abfragt, bis dieser zu *FINISHED* oder *UNKOWN* wechselt, wobei letzteres zum Fehlschlag des Tests führt. Andernfalls wird das Ergebnis heruntergeladen, entpackt und überprüft, ob die Ausgabe auf Standard-Out in Ordnung ist. Zuletzt wird die `WebClientStandalone`-Anwendung gestoppt.

5 Experimentelle Evaluation

Dieses Kapitel zeigt experimentell, dass der Web-Client auch unter realen Bedienungen sinnvoll einsetzbar ist. Dafür wurde ein Lasttest durchgeführt und Vergleiche mit bereits vorhanden Clients der VerifierCloud gezogen.
Für die experimentellen Vergleiche und den Lasttest wurde der Web-Client der normalen VerifierCloud[1] verwendet. Er läuft in einer virtuellen Maschine[2], auf der auch der Master betrieben wird. Die Anfragen werden von einem Apache HTTP-Server, der die Authentifizierung vornimmt, zum Tomcat weitergeleitet. Dabei werden maximal 150 gleichzeitig offene Verbindungen zugelassen. Das *benchmark.py*-Skript, der CPAchecker-, der Benchmark- und der Lasttest-Client liefen jeweils auf einem über das lokale Netzwerk mit dem Server verbunden Rechner[3]. Für die Tests wurde der CPAchecker in Revision 13327 aus dem Trunk verwendet. Damit die JVM-Instanz des Web-Clients Optimierungen durchgeführt hatte, bevor die Tests gestartet wurden, wurden vorher zahlreiche Testanfragen gestellt.

5.1 Lasttest

Für den Lasttest wurde ein kleines Java-Program verwendet, dass die Jersey-Client-API nutzt. Dabei wurde zum einen das Einreichen von Aufgaben und zum anderen das Abfragen von Zustand und Resultat von Runs durchgeführt, wobei jeweils die Antwortzeit auf Seite des Clients gemessen wurde. Die Zahl der parallel durchgeführten Anfragen wurde kontinuierlich gesteigert.
Die Diagramme 5.1 und 5.2 zeigen die Antwortzeit in Abhängigkeit der Anzahl der parallelen Anfragen aufgeschlüsselt nach dem Einreichen von Aufgaben sowie dem Abfragen von Run-Zustand und -Ergebnis. Das erste Bild zeigt zusätzlich die Ausgleichsgerade der mittleren Antwortzeit während der Aufgabeneinreichung und das zweite die der Zustandsabfragen. Die vergangene Zeit bis zum Empfang der Antwort schwankt beim Einreichen von Aufgaben deutlich und weist einige sehr große Ausreißer nach oben auf. Im Bereich um 150 parallel Anfragen steigt die mittlere Antwortzeit auf ca. 7.500 ms. Ansonsten steigt

[1] `http://vcloud.sosy-lab.org/webclient/master/info`
[2] 8 Kerne Intel(R) Xeon(R) CPU E7- 4870 @ 2.40GHz, 8 GB Speicher
[3] Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz, 32 GB Speicher

die mittlere Antwortzeit linear an. Die Abfrage der Ergebnisse weist ebenfalls sehr hohe Ausreißer und kleinere Schwankungen um 150 parallele Anfragen auf. Die Antwortzeit liegt überwiegend unter 50 ms. Beginnend bei 150 parallelen Anfragen steigt die Antwortzeit linear an. Dies ist mit der Beschränkung auf 150 gleichzeitig offene Verbindungen zu erklären. Auf dem zweiten Diagramm ist nur die mittlere Antwortzeit bei der Abfrage der Run-Zustände und die zugehörige Ausgleichsgerade zu erkennen, da dann eine passendere Auflösung der Y-Ache gewählt werden kann. Die Ausgleichsgerade stellt, abgesehen von Ausreißern, die besonderes um 150 parallele Anfragen zu beobachten sind, eine gute Approximation der Kurve dar.
Während der Ausführung des Lasttests ist die Last auf der virtuellen Maschine stark gestiegen, so dass normale Anfragen an den HTTP-Server mehrere Sekunden zur Beantwortung gebraucht haben. Daran ist zu erkennen, dass tatsächlich der Last-Test die Grenze der Leistungsfähigkeit des Web-Clients auf diesem System aufgezeigt hat. Dabei stiegen die Antwortzeiten zwar teils deutlich an, aber die Benutzung des Web-Clients war trotzdem noch möglich. Der vorgeschaltete HTTP-Server sollte eventuell die Zahl der parallelen Verbindungen eines Clients beschränken, da sonst der Master vom Web-Client mehr Aufgaben erhält, als er langfristig abarbeiten kann, wodurch es zu Speicherknappheit kommen kann. Dabei muss auch bedacht werden, dass es noch lokale Benutzer der VerifierCloud gibt, die bei der Bearbeitung ihrer Aufgaben im Allgemeinen bevorzugt behandelt werden.

Diagramm 5.3 zeigt über die Zeit die Anzahl der bearbeiteten HTTP-Anfragen pro Minute, die während der Evaluation des Web-Clients auf dem beschriebenen Produktiv-System beantwortet wurden. Dabei ist zu erkennen, dass in Spitzen während des Lasttestes über 310.000 Anfragen pro Minute vom Web-Client verarbeitet wurden und auch über längere Zeiträume mehrere zehntausend Anfragen pro Minute zu verzeichnen waren. Die Anfragen zwischen halb zehn und viertel nach elf wurden durch den Vergleich mit dem Benchmark-Client verursacht. Das Diagramm ist dem Monitoring-System des Tomcat-Servers entnommen und der Beginn des Tages herausgeschnitten, weil diese Web-Client-Instanz zu dieser Zeit noch nicht lief.

5.2 Vergleich mit anderen Clients

In den beiden folgenden Abschnitten wird der Web-Client mit dem CPAchecker- und dem Benchmark-Client verglichen. CPAchecker lag jeweils in übersetzter Form vor, das heißt der Cache des Web-Clients enthielt die verwendete Revision und lokal war der CPAchecker bereits für den Benchmark-Client bzw. CPAchecker-Client gebaut. Das *benchmark.py*-Skript baut bei Verwendung des Benchmark-Clients den CPAchecker trotzdem jedes mal neu. Dies dauert, wenn alle Bibliotheken und Class-Dateien bereits vorhanden sind, ca. 8 Sekunden.

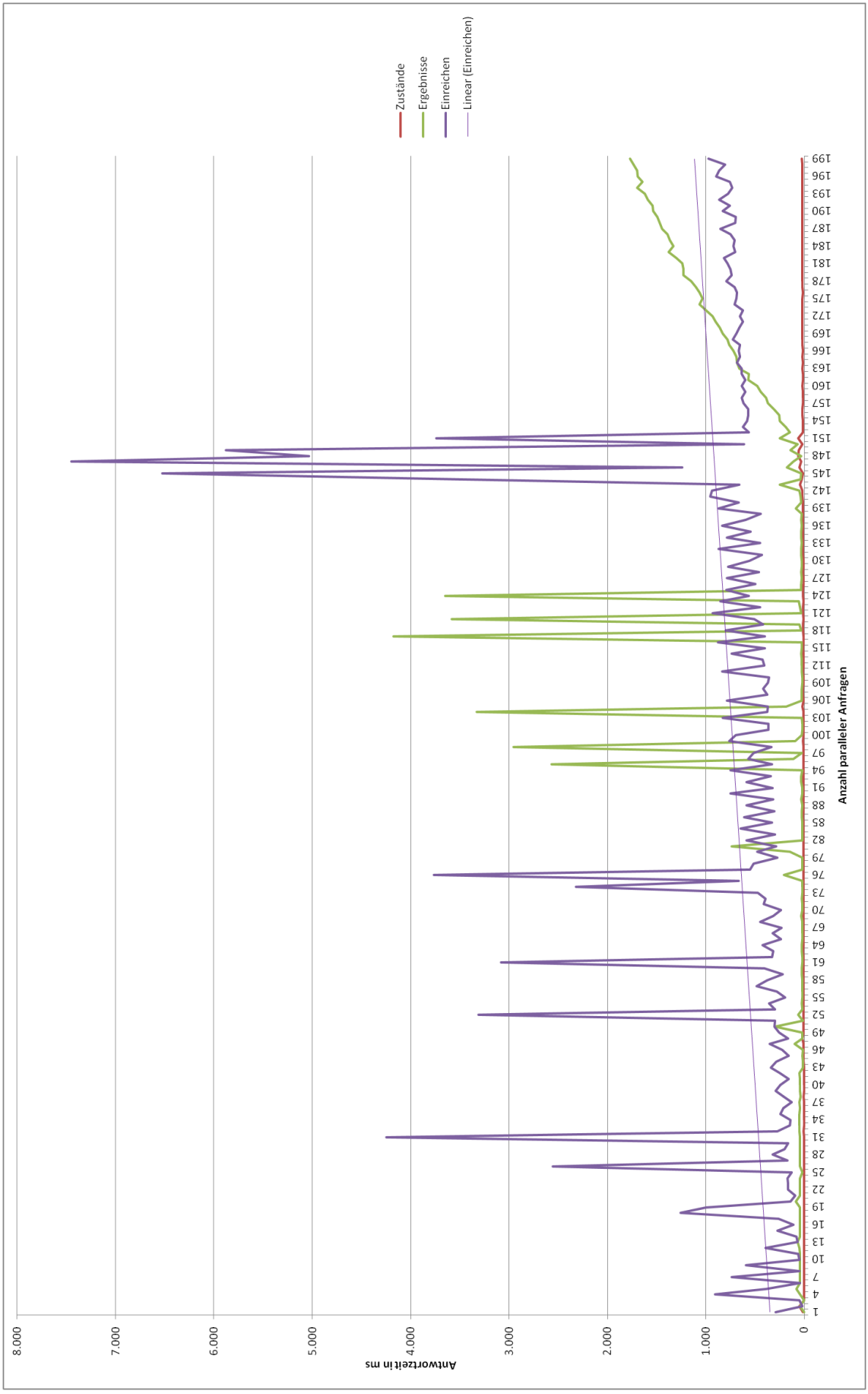

Abbildung 5.1: Antwortzeit über die Anzahl der parallelen Anfragen

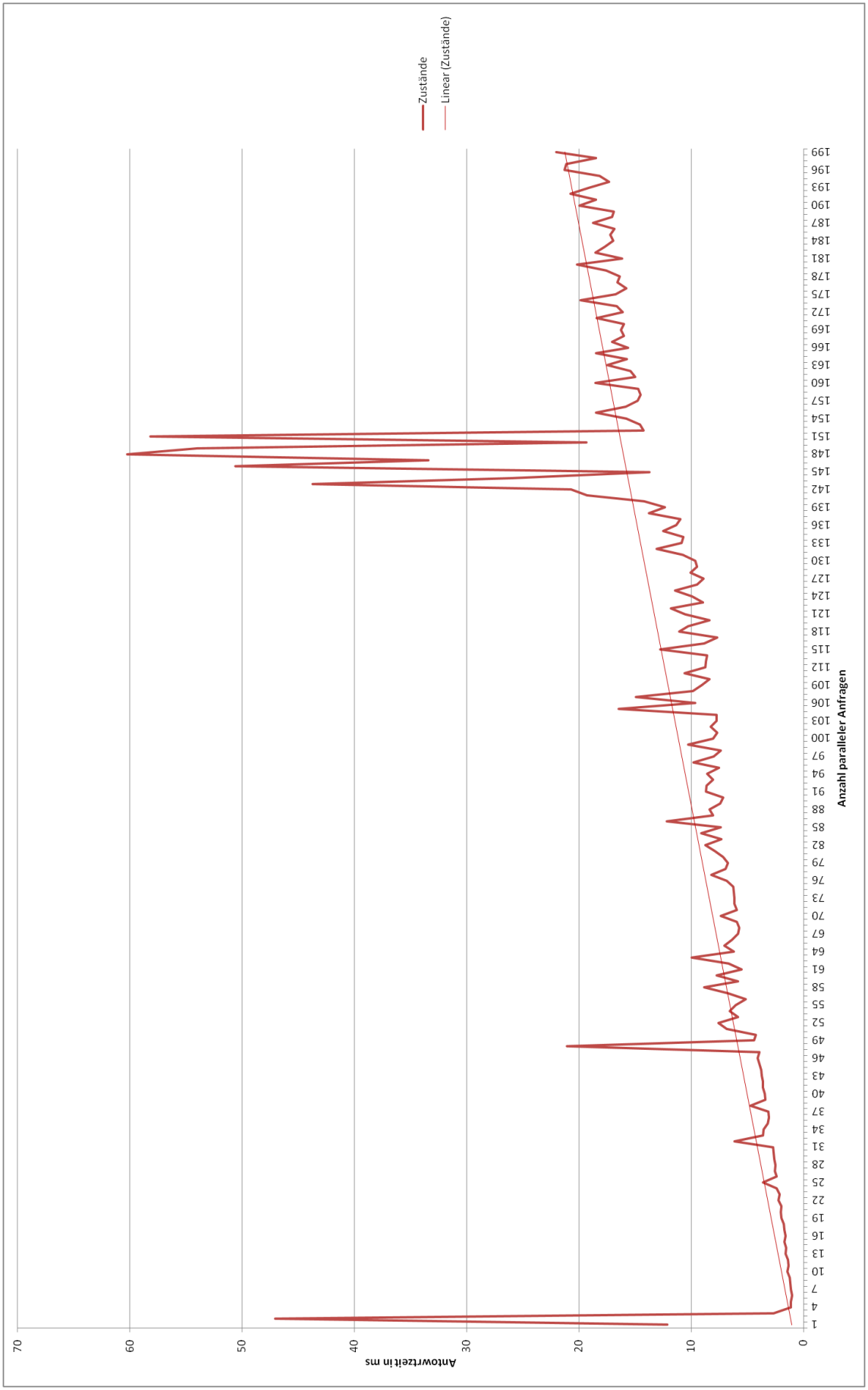

Abbildung 5.2: Antwortzeit von Zustandsanfragen über die Anzahl der parallelen Anfragen

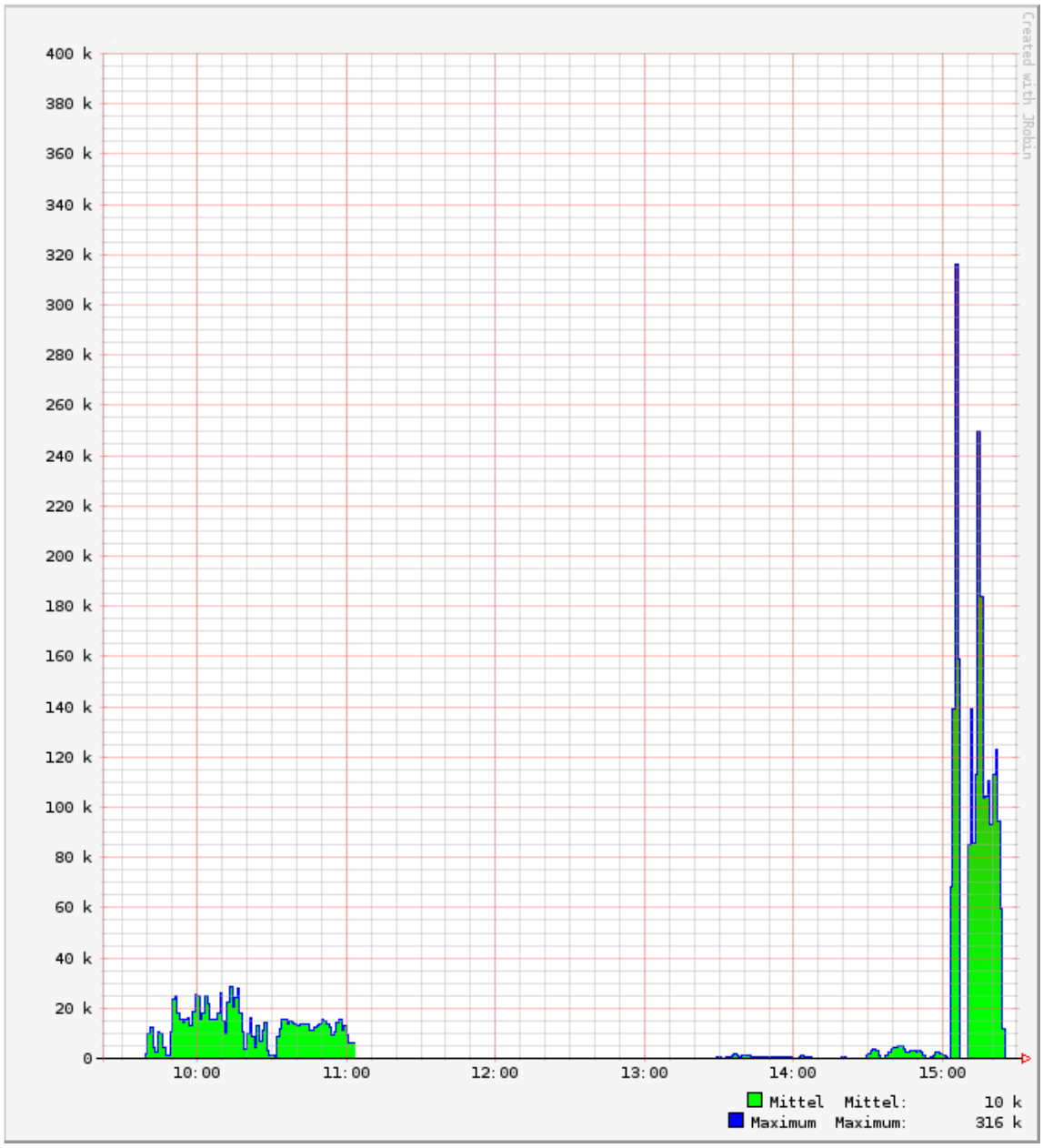

Abbildung 5.3: Anzahl der HTTP-Anfragen pro Minute

5.2.1 CPAchecker-Client

Der CPAchecker-Client erzeugt eine `RunCollection`, mit je einem `Run` für eine Datei oder rekursiv für alle Dateien eines Verzeichnisses. Dabei werden für alle Dateien die selben Optionen und Beschränkungen verwendet. Somit ist der CPAchecker-Client weniger flexibel als der Web-Client und kann nicht in Verbindung mit dem *benchmark.py*-Skript verwendet werden. Außerdem unterstützt er nur eine Programmtext-Datei pro CPAchecker-Lauf.
Da der CPAchecker nicht für komplexe Eingaben konzipiert ist, wird der Vergleich mit einer Datei[4] aus dem CPAchecker-Repository als Eingabe durchgeführt, wobei die ldv-Konfiguration Verwendung findet. Dafür muss beim CPAchecker-Client *-ldv* als Parameter, der Pfad zur Eingabedatei, der Pfad zum Basisverzeichnis des ausgecheckten und gebauten CPAcheckers und der Host-Name des Masters angegebene werden. Der CPAchecker-Client terminiert 30 Sekunden nach dem Start, nachdem die Ausgabe des CPAcheckers in je eine Datei geschrieben worden ist. Die Anfragen an den Web-Client werden manuell mit *curl*[5] gestellt. Für die Anfrage zum Einreichen der Verifikationsaufgabe wird das Formt *multipart/form-data* verwendet. Es muss Benutzername, Passwort, die Eingabe-Datei für den CPAchecker, die Konfiguration als Option, die zu benutzende Revision und die URL des Web-Clients angegeben werden. Die Ausgabe ist die ID der eingereichten Aufgabe, mit der anschließend der Zustand angefragt und das Ergebnis als Zip-Archiv heruntergeladen werden kann. Hierzu ist jeweils ein eigener Aufruf von *curl* notwendig. Es wird deutlich, dass die Benutzung des Webclients ohne ein passendes Skript oder Programm als eher weniger praktisch einzustufen ist.

5.2.2 Benchmark-Client

Das *benchmark.py*-Skript des CPAcheckers wurde erweitert, so dass es Aufgaben auch mittels des Web-Clients an die VerifierCloud übergeben kann. Dabei unterliegt es bereits beschriebenen Sicherheitsbeschränkungen. Es können also insbesondere nur Konfigurationen aus dem Standardordner für Konfigurationen verwendet werden. Alle Kommandozeilenparameter müssen, so weit es möglich ist, in das Eingabeformat des Web-Clients transformiert werden. Es wird `application/x-www-form-urlencoded` als Format für das Einreichen von Verifikationsaufgaben verwendet, weil es von Python direkt unterstützt wird. Falls Python in Version 3 verwendet wird, wird zusätzlich *deflate*-Komprimierung verwendet. Ab Python 3.2 werden die Verifikationsaufgaben parallel eingereicht. Sobald alle Aufgaben eingereicht wurden, werden alle fünf Sekunden die Zustände aller bisher noch nicht beendeten Runs

[4] test/programs/benchmarks/ldv-linux-3.7.3/linux-3.10-rc1-43_1a-bitvector-drivers–atm–he.ko-ldv_main0_true-unreach-call.cil.out.c
[5] `http://curl.haxx.se/`

angefragt. Die Zip-Datei mit allen Ergebnissen wird dann für alle Runs heruntergeladen, die seit der jeweils letzten Zustandsanfrage fertiggestellt wurden. Das *benchmark.py*-Skript entpackt sie, kopiert die Standardausgabe an das Ende der Log-Datei und extrahiert alle Informationen aus *hostInformationen.txt* sowie *runInformationen.txt*.
Der Benchmark-Client wird normaler Weise verwendet, wenn Verifikationsaufgaben durch das *benchmark.py*-Skript in der VerifierCloud gestartet werden. Zuerst wird CPAchecker lokal gebaut und anschließend die benötigten Dateien, auszuführenden Befehle, zu verwendenden Beschränkungen und die Pfade der Ausgabedateien berechnet. Anschließend wird der Benchmark-Client gestartet und jene Informationen auf der Standardeingabe übergeben. Daraus erstellt der Benchmark-Client eine `RunCollection`, die er an den Master sendet. Nach und nach erhält der Benchmark-Client die Ergebnisse vom Master und schreibt sie an die vorgesehenen Pfade im Dateisystem. Nachdem er alle Ergebnisse erhalten hat, beendet er sich, woraufhin das *benchmark.py*-Skript sie analysiert. Im Gegensatz zur Verwendung mit dem Web-Client werden die Resultate erst nach der Terminierung des Benchmark-Clients für den Benutzer sichtbar, indem sie auf der Kommandozeile ausgegeben werden. Da Benchmarks auch mehre Tage zur Fertigstellung benötigen können, ist das durchaus ein nicht unwesentlicher Unterschied.
Für den experimentellen Vergleich wurden zwei vorhandene Benchmarks des CPAcheckers gewählt, zum einen die Integrationstests für die BDD-Analyse und das Benchmark der SV-Competition-14[6]. Bei letzterem wurde die Option zur Deaktivierung der Java-Assertions entfernt, weil diese Option am Web-Client nicht freigeschaltet war. Das *benchmark.py*-Skript wurde mit Python 2 für den Benchmark-Client und Python 3 für den Web-Client verwendet, da es im Modus mit dem Benchmark-Client nicht mit Python 3 funktioniert und im Modus mit dem Web-Client das Hochladen der POST-Daten komprimiert geschehen und parallel für mehre Run durchgeführt werden kann. Dabei wurden 20 Threads verwendet. Die eingereichten Aufgaben wurden durch die VerifierCloud jeweils auf 24 Rechnern[7] parallel bearbeitet.
Diagramm 5.4 zeigt die Anzahl der vom *benchmark.py*-Skript analysierten Ergebnisse über die Zeit bezogen auf den Start des Skriptes. An der Kurve des Web-Clients ist zu erkennen, dass nach 96 Sekunden das Einreichen der Aufgaben abgeschlossen ist und das Herunterladen der Ergebnisse beginnt, wobei bis 04 Minuten 22 Sekunden die VerifierCloud schneller Ergebnisse produziert als sie Heruntergeladen werden. Die Stufe nach ca. 18 Minuten ist auf die Zeitbeschränkung der Verifikationsaufgaben von 15 Minuten zurückzuführen. Nach 30 Minuten und 33 Sekunden beginnt das *benchmark.py*-Skript bei Verwendung des Benchmark-Clients, nachdem dieser alle Ergebnisse empfangen hat, mit der

[6] `http://sv-comp.sosy-lab.org/2014/`
[7] je Rechner: 2 Intel Xeon E5-2650 v2 @ 2.60 GHz mit insgesamt 32 Kernen, 135 GB Speicher

Analyse und schließt sie nach 14 Sekunden ab. Zu diesem Zeitpunkt fehlen bei Verwendung des Web-Clients noch 62 Ergebnisse. Das *benchmark.py*-Skript schickt bei Verwendung des Benchmark-Clients noch ein zusätzliches Skript mit, das ein kürzeres Offset für die Zeitbeschränkung von Runs verwendet als die VerifierCloud, so dass die Runs, die durch die Zeitbeschränkung abgebrochen werden, bei Verwendung des Web-Clients länger laufen. Der Worker wartet eine Minute länger als das Zeitlimit des Runs, bis dieser beendet wird. Es ist geplant die Funktionen des zusätzlichen Skriptes, in das des Workers, das immer verwendet wird, zu integrieren, so dass dieser Unterschied dann nicht mehr besteht.

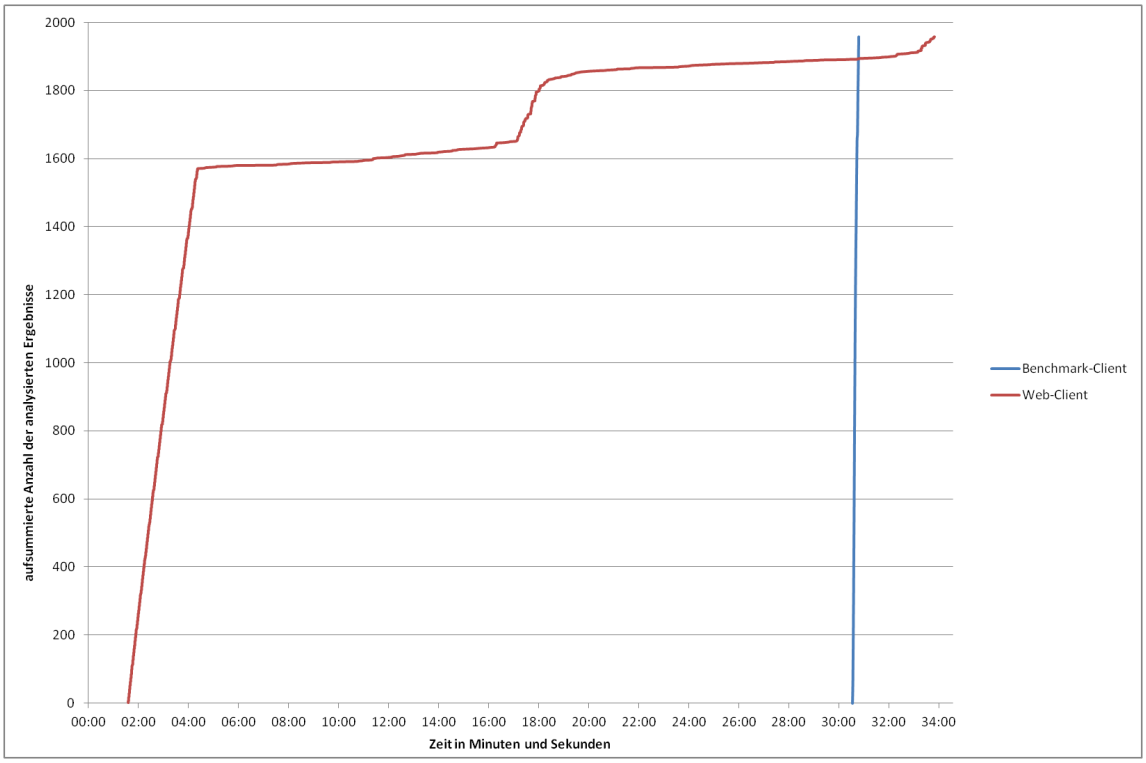

Abbildung 5.4: Aufsummierte Anzahl der analysierter Ergebnisse über die Zeit beim SV-Comp14-Benchmark

Diagramm 5.5 zeigt das Gleiche wie obige Grafik für das BDD-Integrationstest-Benchmark. Die einzelnen Runs haben eine deutlich kürzere Laufzeit, von meist nur wenigen Sekunden, als im vorherigen Beispiel. Beim Einreichen der Aufgaben wurde die Revision explizit angegeben, so dass das Auflösen von *HEAD* zu einer Revisionsnummer am Web-Client nicht notwendig war. An der Kurve des Web-Clients ist eine Treppenform erkennbar, die durch das Cachen der Run-Zustände für 5 Sekunden am Web-Client hervorgerufen wird. Nach 75 Sekunden sind 342 der 344 Ergebnisse vom Web-Client heruntergeladen und analysiert, während bei Verwendung des Benchmark-Clients noch kein Ergebnis durch

das *benchmark.py*-Skript bearbeitet wurde. Die beiden fehlenden Ergebnisse sind erst nach Ablauf der Zeitbeschränkung nach 2 Minuten und 35 Sekunden verfügbar. Wie bereits im vorherigen Absatz beschrieben laufen solche Runs bei Verwendung des Web-Clients länger bis das Beenden erzwungen wird.

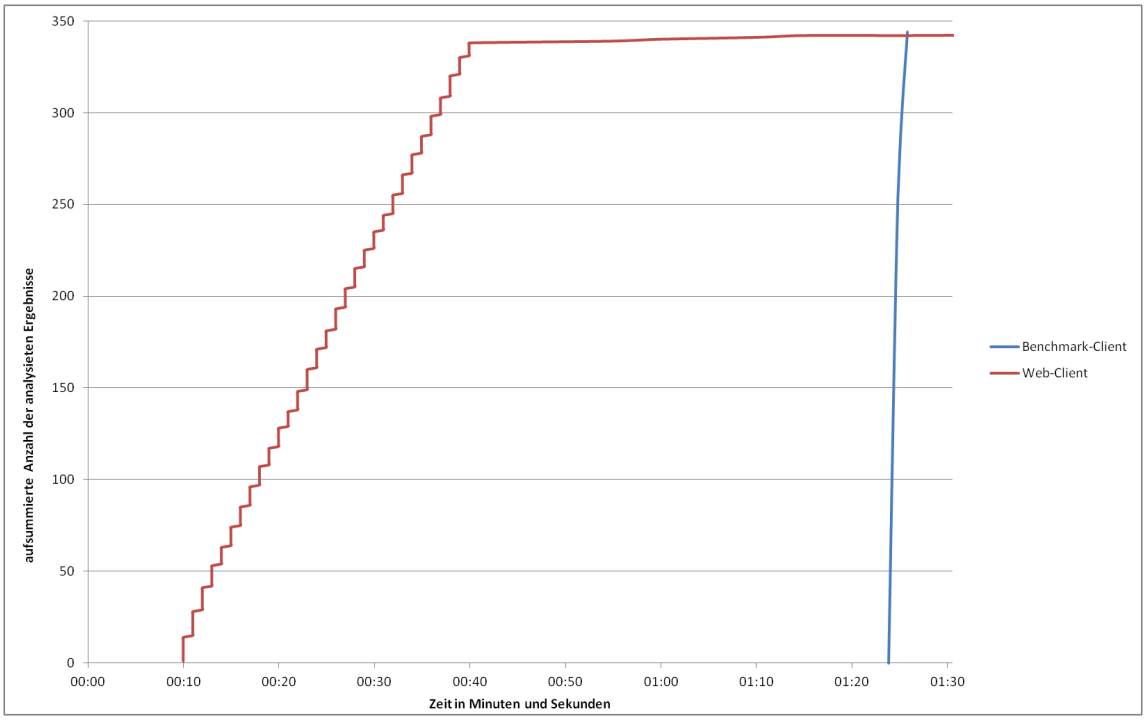

Abbildung 5.5: Aufsummierte Anzahl der analysierter Ergebnisse über die Zeit beim BDD-Integrationstest-Benchmark

Die Auswertung der Log-Dateien ergab, dass der Web-Client in der Lage ist ca. 85 Aufgaben pro Sekunde entgegenzunehmen und zu verarbeiten, falls die Revisionsnummer explizit angegeben ist, sowie ca. zehn Resultate pro Sekunde bereitzustellen. Hierbei ist zu beachten, dass das *benchmark.py*-Skript die Resultate sequentiell herunterlädt und analysiert. Daher gibt es auf Clientseite weitere Möglichkeiten den Gesamtvorgang zu beschleunigen. *HEAD*-Revisionen müssen jeweils zu einer Revisionsnummer aufgelöst werden, so dass dann nur noch ca. 20 Anfragen pro Sekunde beantwortet werden können. Dies könnte durch einen Cache beschleunigt werden, ist aber immer noch mehr als die VerifierCloud zur Zeit abarbeiten kann, wenn keine sehr einfachen Aufgaben eingereicht werden.

6 Beschreibung der Web-Schnittstelle

Dieses Kapitel beschreibt die REST-Schnittstelle des Web-Clients. Der Basispfad ist standardmäßig */vcloud/webclient/*. Alle im Folgenden erläuterten Pfade sind relativ zu diesem. Die Pfade und Parameter orientieren sich an denen des CPAcheckers in der Google App-Engine [3], wobei bereits vorhandene Namen aus der VerifierCloud verwendet werden, weil diese schon ausführlich diskutiert wurden. Wenn eine Option nicht gesetzt ist, wird ein eventuell vorhandener Standardwert verwendet. Tabelle 6.1 bietet eine Übersicht der vorhandenen REST-Ressourcen.

Methode	URI	Ressource
GET	.	Weiterleitung zum Hilfetext
GET	help	Hilfetext
POST	runs	Einreichen einer Verifikationsaufgabe
POST	runs/premium_priority	Einreichen einer Verifikationsaufgabe mit Premium-Priorität
GET	runs/{id}/state	Status des Runs mit {id}
GET	runs/{id}/result	Ergebnis des Runs mit {id}
GET	tool	gebauter CPAchecker in gegebener Revision
GET	master/info	Informationsseite zum Zustand des Masters
GET	update_repository	Update-Trigger des lokalen Git-Klons

Tabelle 6.1: Übersicht der REST-Ressourcen des Web-Clients

6.1 Hilfetext

Mittels einer *GET*-Anfrage auf den Subpfad *help* erhält der Benutzer einen Hilfetext, der die Funktionen des Web-Clients und deren Parameter erklärt. Außerdem informiert jener über die aktuelle Konfiguration der Beschränkungen. Der Basispfad wird auf den Hilfetext weitergeleitet, um Benutzern den Einstieg zu erleichtern.

6.2 Einreichen von Verifikationsaufgaben

Verifikationsaufgaben werden durch eine *POST*-Anfrage auf den Subpfad *runs* eingereicht. Die Parameter können dabei in zwei Formaten übergeben werden: *application/x-www-form-urlencoded* und *multipart/form-data*. Letzteres bietet eine effizientere Kodierung von nicht ASCII-Zeichen und die Möglichkeit die Namen von Dateien anzugeben. Diese werden für die Programmtext-Dateien vom Web-Client übernommen, wodurch in Fehlermeldungen und Log-Dateien die ursprünglichen Namen verwendet werden.
Falls für Speicher- und Zeitwerte keine Einheiten angegeben sind, werden die Basiseinheiten Byte und Sekunde angenommen.
Durch den Web-Client gestartete Runs unterliegen grundsätzlich einer Beschränkung in Zeit, Speicher und Anzahl an Prozessorkernen. Zur einfacheren Nutzung gibt es vorkonfigurierte Beschränkungen, von denen der Nutzer eine auswählen kann. Tabelle 6.2 zeigt die Standardeinstellungen der vorkonfigurierten Beschränkungen.

Name	Zeit	Speicher	Prozessorkerne
LOW	5 min	2 GB	2
MEDIUM	10 min	7 GB	2
HIGH	15 min	15 GB	2

Tabelle 6.2: Standardmäßig Vorkonfigurierte Beschränkungen

Die Parameter aus Tabelle 6.3 werden durch den Web-Client in beiden Formaten unterstützt. Alle Parameter außer *programText*, *specificationText*, *propertyText* und *heap* unterliegen Restriktionen, die auf der Hilfeseite angezeigt werden.
Die Antwort, deren Inhalt die ID des erzeugten Runs ist, wird im Format *text/plain* gesendet.
Tabelle 6.4 zeigt mögliche Fehler und deren Beschreibung.

6.3 Einreichen von Verifikationsaufgaben mit höherer Priorität

Aufgaben, die durch eine *POST*-Anfrage auf den Subpfad *runs/premium_priority* eingereicht werden, erhalten eine andere, entsprechend der Konfiguration des Web-Clients gegebenenfalls höhere Priorität als normal eingereichte Aufgaben. In allen anderen Aspekten ist das Verhalten wie unter 6.2 beschrieben.
Diese Funktionalität ist zur Zeit nicht öffentlich dokumentiert. Der Zugang zu dieser URL kann zum Beispiel durch einen vorgeschalteten Server eingeschränkt werden.

Name	Beschreibung
programText	Inhalt der vom CPAchecker zu überprüfenden Dateien. Es muss mindestens eine Datei angegeben werden. Der Parameter muss je Datei einmal verwendet werden. Die bei Anfragen im Format *multipart/form-data* übertragenen Namen der Programm-Dateien werden übernommen, soweit sie vorhanden sind.
specificationText	Inhalt der vom CPAchecker zu verwendenden Spezifikations-Datei, als Dateiendung wird *spc* verwendet.
propertyText	Inhalt der vom CPAchecker zu verwendenden Property-Datei, als Dateiendung wird *prp* verwendet.
specification	Name einer Spezifikationsdatei aus dem CPAchecker-Repository. Standard ist keine Spezifikation.
configuration	Name einer Konfigurationsdatei aus dem CPAchecker-Repository. Standard ist keine Konfiguration.
cpuModel	Spezifiziert das CPU-Modell, auf denen dieser Run ausgeführt werden soll. Der Run wird nur auf Prozessoren ausgeführt, in deren Namen die angegebene Zeichenkette enthalten ist. Der Standardwert ist keine Beschränkung.
option	Setzt eine Option bzw. überschreibt eine Option aus der gegeben Konfiguration. Es müssen Schlüssel-Wert-Paare angegeben werden. Der Parameter kann mehrfach verwendet werden, je einmal für jede Option.
heap	Setzt die maximale Größe des Java-Heap-Speichers der JVM, auf der der CPAchecker ausgeführt wird. Wenn nichts angegeben ist, setzt das Startskript des CPAcheckers diesen Wert, wobei es dafür aktuell 1200 MB verwendet.
limitations	Setzt eine der vordefinierten Beschränkungen, von denen die erste der Standardwert ist.
timeLimitation	Überschreibt die Zeitbeschränkung, die mit *limitations* gesetzt wurde.
memoryLimitation	Überschreibt die Speicherbeschränkung, die mit *limitations* gesetzt wurde.
coreLimitation	Überschreibt die Prozessorkernbeschränkung, die mit *limitations* gesetzt wurde.
revision	SVN-Revision als nicht negative ganze Zahl oder *HEAD*. Der Standardwert ist *HEAD*.
svnBranch	Name des SVN-Branches oder *trunk*. Der Standardwert ist *trunk*.
svnTag	Name des SVN-Tags. Wenn diese Option gesetzt ist, wird *revision* und *svnBranch* ignoriert.

Tabelle 6.3: Beschreibung der Parameter für das Einreichen von Verifikationsaufgaben

Status-Name	Status-Nummer	Beschreibung
Forbidden	403	ein oder mehrere Parameter der Anfrage sind nicht gestattet, der mitgeschickte Text enthält weitere Hinweise
Internal Server Error	500	CPAchecker steht in der angeforderten Revision nicht zur Verfügung, z.B. wegen eines Fehlers während des Bauens, der Antworttext enthält weitere Hinweise
Service Unavailable	503	Web-Client ist nicht mit dem Master der Verifier-Cloud verbunden und kann deswegen keine Anfragen entgegennehmen

Tabelle 6.4: Beschreibung der Fehlercodes beim Einreichen von Verifikationsaufgaben

6.4 Abfrage des Run-Zustandes

Der Web-Client gibt bei einer *GET*-Anfrage auf den Subpfad *runs/{id}/state* den Zustand des Runs mit der gegeben *{id}* zurück. Tabelle 6.5 erläutert die Run-Zustände und Tabelle 6.6 zeigt mögliche Fehler und deren Beschreibung.

Name	Beschreibung
PENDING	Der Run ist bekannt und wird noch nicht ausgeführt.
PROCESSING	Der Run wird gerade auf einem Worker ausgeführt.
FINISHED	Die Ausführung des Runs ist fertig und das Ergebnis kann abgerufen werden.
UNKOWN	Der Run existiert nicht, wurde nicht von diesem Web-Client gestartet oder wurde vor dem letzten Neustart dieses Web-Clients gestartet. In diesem Fall kann der Run sich in jedem der obigen Zustände befinden.

Tabelle 6.5: Beschreibung der Run-Zustände

Status-Name	Status-Nummer	Beschreibung
Not Found	404	ID des Runs nicht korrekt
Service Unavailable	503	Web-Client ist nicht mit dem Master der Verifier-Cloud verbunden und kann deswegen keine Anfragen entgegennehmen

Tabelle 6.6: Beschreibung der Fehlercodes beim Abfragen eines Run-Zustandes

6.5 Erhalten der Resultate

Nachdem ein Run durch die VerifierCloud ausgeführt wurde, kann das Ergebnis durch eine *GET*-Anfrage auf den Subpfad *runs/{id}/result* als Zip-Archiv heruntergeladen werden. Darin sind alle während der Ausführung neu erzeugten Dateien, die Ausgabe sowie Informationen zur Run-Ausführung und zum Worker, auf dem der Run ausgeführt wurde, enthalten:

- *stdout*: Ausgabe von CPAchecker auf Standard-Out
- *stderr*: Ausgabe von CPAchecker auf Standard-Error
- *runInformation.txt*: Informationen über die Ausführung des Runs:
  - *command*: Ausgeführter Befehl
  - *exitCode*: Exitcode des CPAcheckers
  - *wallTime*: Zeit vom Start des CPAcheckers bis zu dessen Terminierung in Sekunden
  - *cpuTime*: verwendete Prozessorzeit in Sekunden
  - *usedMemory*: verwendeter Speicher in Byte
  - *energy*: verbrauchte Energie in Joule (optional)
  - *timeLimit*: Zeitbeschränkung in Sekunden
  - *memoryLimit*: Speicherbeschränkung in Byte
  - *coreLimit*: Beschränkung der Prozessorkernanzahl
- *hostInformation.txt*: Informationen über den Rechnern, auf dem der Run ausgeführt wurde:
  - *name*: Name des Rechners
  - *os*: Name und Version des Betriebssystems
  - *memory*: Größe des Speichers
  - *cpuModel*: Name des Prozessormodells
  - *frequency*: Taktfrequenz des Prozessors
  - *cores*: Anzahl der Prozessorkerne

Tabelle 6.7 zeigt mögliche Fehler und deren Beschreibung.

Status-Name	Status-Nummer	Beschreibung
Not Found	404	ID des Runs ist nicht korrekt oder das Ergebnis des Runs ist noch nicht verfügbar
Service Unavailable	503	Web-Client ist nicht mit dem Master der VerifierCloud verbunden und kann deswegen keine Anfragen entgegennehmen

Tabelle 6.7: Beschreibung der Fehlercodes beim Abfragen eines Run-Ergebnisses

6.6 Herunterladen des CPAcheckers

Der CPAchecker kann als Zip-Archiv in der Form, wie er auf dem Worker ausgeführt wird, heruntergeladen werden, indem eine *GET*-Anfrage auf den Subpfad *tool* gestellt wird. Dabei können die in Tabelle 6.8 enthaltenen Parameter genutzt werden.

Name	Beschreibung
revision	SVN-Revision als nicht negative ganze Zahl oder *HEAD*, Standardwert ist *HEAD*
svnBranch	Name des SVN-Branches oder *trunk*, der Standardwert ist *trunk*
svnTag	Name des SVN-Tags; falls diese Option gesetzt ist, wird *revision* und *svnBranch* ignoriert

Tabelle 6.8: Beschreibung der Parameter für das Herunterladen des Verifikationswerkzeugs

Die VerfierCloud unterstützt keine Datei-Meta-Eigenschaften. Deshalb werden für die Dateien des Archivs keine solchen gesetzt.
Tabelle 6.9 zeigt mögliche Fehler und deren Beschreibung.

Status-Name	Status-Nummer	Beschreibung
Not Found	404	angegebene Revision existiert nicht
Internal Server Error	500	CPAchecker steht in der angeforderten Revision nicht zur Verfügung steht, z.B. wegen eines Fehlers während des Bauens, Antworttext enthält weitere Hinweise
Service Unavailable	503	Web-Client ist nicht mit dem Master der VerifierCloud verbunden und kann deswegen keine Anfragen entgegennehmen

Tabelle 6.9: Beschreibung der Fehlercodes beim Herunterladen des Verifikationswerkzeugs

6.7 Master-Informationsseite

Der Web-Client stellt unter dem Subpfad *master/info* die gleiche HTML-Seite bereit, die der Info-Client ausgibt. Sie enthält den Status des Masters, eine Übersicht der Runs, die aktuell am Master verarbeitet werden, und eine tabellarische Übersicht über den Zustand der Worker.
Tabelle 6.10 zeigt mögliche Fehlermeldungen und deren Beschreibung.

Status-Name	Status-Nummer	Beschreibung
Service Unavailable	503	Web-Client ist nicht mit dem Master der Verifier-Cloud verbunden und kann deswegen keine Anfragen entgegennehmen

Tabelle 6.10: Beschreibung der Fehlercodes beim Abrufen der Master-Informationsseite

6.8 Update-Trigger für den Git-Klon

Durch *GET*-Anfragen auf den Subpfad *update_repository* wird das Herunterladen aller neuen Revisionen in den lokalen Git-Klon getriggert, wodurch dies nicht mehr während der Bearbeitung von Anfragen geschehen muss. Dadurch wird eine schnellere Antwortzeit bei der Einreichung von Verifikationsaufgaben und dem Herunterladen des CPAcheckers erreicht.
Diese Anfrage sendet immer eine leere Antwort mit HTTP-Status *OK*, da das Herunterladen der neuen Revisionen parallel im Hintergrund geschieht.

7 Benutzer des Web-Clients

Schon während der Testphase wurde der Web-Client von unterschiedlichen Nutzern verwendet. Dazu wurde ein Web-Client, der auf einem Tomcat ausgeführt wird, mit der vorhandenen VerifierCloud-Instanz verbunden.
Das Linux Driver Verification[1] Projekt sucht mit Hilfe von CPAchecker Fehler in Treibern des Linux-Kernels. Durch die Verwendung des Web-Clients stehen erheblich mehr Rechenressourcen zur Verfügung. Um diese nutzen zu können, haben sie ihr Framework erweitert, wodurch die Verifikationsaufgaben an den Web-Client geschickt werden anstatt CPAchecker lokal auszuführen.
Das Fachgebiet Echtzeitsystem der Technischen Universität Darmstadt setzt CPAchecker zur automatischen Generierung von Tests ein[2]. Sie nutzen den Web-Client zu Ausführung dieser Aufgaben, die besonders zeit- und speicherintensiv sind.
Außerdem wird der Web-Client durch den BuildBot des CPAcheckers[3] genutzt, indem er den CPAchecker vom Build-Service des Web-Clients für die Ausführung der Regressionstests herunterlädt. Dadurch werden nicht gleichzeitig zehn Build-Prozesse auf dem BuildBot-Server ausgeführt, sondern nur einer in der VerifierCloud, die in aller Regel über unbeschäftigte Worker-Rechner verfügt.

[1] `http://linuxtesting.org/project/ldv`
[2] `http://www.es.tu-darmstadt.de/forschung/model-based-quality-assurance/`
[3] `http://buildbot.sosy-lab.org/buildbot/`

8 Fazit

Die VerifierCloud wurde erfolgreich um einen Client erweitert, der eine Web-Schnittstelle zur Softwareverifikation bereitstellt. Diese bietet fast die gleichen Möglichkeiten wie die lokale Ausführung des CPAcheckers, wobei die umfangreichen Ressourcen der VerifierCloud verwendet werden können, um eine große Zahl an Verifikationsaufgaben gleichzeitig zu bearbeiteten. Dadurch verkürzt sich die Zeit bis die Ergebnisse aller Verifikationsaufgaben bereitstehen im Vergleich zur lokalen Ausführung auf einem Rechner erheblich. Die Evaluation hat die Leistungsfähigkeit des Web-Client sowie Vorteile gegenüber bisher vorhandenen Clients der VeriferCloud gezeigt. Außerdem können die Ressourcen der VerifierCloud auch zum Bauen des CPAcheckers herangezogen werden, wodurch der Web-Client zusätzlich in der Lage ist, einen Build-Service für den CPAchecker anzubieten. Der Web-Client wurde schon während der Entwicklungsphase und dem Verfassen dieser Arbeit erfolgreich eingesetzt. Die Anbindung an vorhandene Client-Infrastruktur ermöglicht eine leichte und komfortable Nutzung der Web-Schnittstelle.
Es gibt auch Einschränkungen bei der Benutzung des Web-Clients, die überwiegend als Sicherheitsvorkehrungen notwendig sind. Außerdem ist die VerifierCloud auf die Verwendung des Benchmark-Clients optimiert, so dass der Web-Client zum aktuellen Zeitpunkt nicht alle Möglichkeiten vergleichbar umsetzten kann.
Die Möglichkeit mehre Aufgaben auf einmal einzureichen und deren Zustände gesammelt abzufragen könnte im Rahmen einer Weiterentwicklung des Web-Clients realisiert werden. Darüber hinaus wäre die Integration einer Benutzerverwaltung denkbar.

Literatur

[1] Dirk Beyer, Georg Dresler und Philipp Wendler. »Software Verification in the Google App-Engine Cloud«. In: *Proceedings of the 26th International Conference on Computer-Aided Verification (CAV 2014, Vienna, Austria, July 18-22)*. Hrsg. von A. Biere und R. Bloem. LNCS. Springer-Verlag, Heidelberg, 2014. URL: `http://www.sosy-lab.org/~dbeyer/cpa-appengine`.

[2] Bill Burke. *RESTful Java with Jax-RS*. Ö'Reilly Media, Inc.", 2009.

[3] Georg Dresler. »A Google-App-Engine Implementation for CPAchecker«. Bachelorarbeit. Universität Passau, 2014.

[4] Roy T. Fielding und Richard N. Taylor. »Principled Design of the Modern Web Architecture«. In: *ACM Trans. Internet Technol.* 2.2 (Mai 2002), S. 115–150. ISSN: 1533-5399. DOI: `10.1145/514183.514185`. URL: `http://doi.acm.org/10.1145/514183.514185`.

[5] Roy Thomas Fielding. »Architectural styles and the design of network-based software architectures«. Diss. University of California, Irvine, 2000.

[6] Santiago Pericas-Geertsen und Marek Potociar. *JAX-RS: Java^{TM} API for RESTful Web Services*. Techn. Ber. Oracle Corporation, Mai 2013.

[7] Nikolai Tillmann u. a. »Code Hunt: Searching for Secret Code for Fun«. In: *Proceedings of the International Conference on Software Engineering (Workshops)* (Juni 2014). URL: `http://research.microsoft.com/apps/pubs/default.aspx?id=210651`.

Erklärung zur Bachelorarbeit

Hiermit erkläre ich, dass ich die Bachelorarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet habe und dass die Bachelorarbeit in gleicher oder anderer Form noch keiner anderen Prüfungsbehörde vorgelegt wurde.

Passau, 29. September 2014

_____

Sebastian Ott