Der Tomcat 4 Coyote HTTP-Connector

Auf den Hund gekommen

Die „Tore“ – sprich Connectoren - des Tomcat-Servers stellen einen der wesentlichen Architekturaspekte innerhalb des Jakarta Servlet-Containers dar, denn erst durch sie erlangt er die notwenige Flexibilität bei der Anbindung an die Außenwelt. Dank eines kompletten Neudesigns während der Entwicklung des Tomcat 4.1-Releases sind diese „Tore“ deutlich breiter, schneller und vor allem flexibler geworden. In dieser Ausgabe des Java Magazins wird mit der Coyote Connector-Familie der heutige State of the Art der Tomcat-Connectoren vorgestellt und es wird gezeigt, wie deren Konfiguration für verschiedene Szenarien mit Hilfe von Lasttests überprüft werde kann.

Um die volle Bedeutung der Tomcat-Connectoren zu verstehen, sollte man sich zunächst noch einmal die Gesamtarchitektur des Tomcat 4.x-Servers vor Augen halten. Wie Abbildung 1 verdeutlicht, stellen die Connectoren die „Einfalltore“ in den Tomcat 4.x und somit die Schnittstelle zwischen dem Server und der Außenwelt – sprich dem Client – dar.

Mit Hilfe der Connectoren werden beispielsweise die Sockets, Threads oder die Protokoll-Parser bereitgestellt, welche die Clients nutzen, um mit dem Server über einen bestimmten Port zu kommunizieren. Spricht man im Zusammenhang mit dem Tomcat-Server von Connectoren, so muss man zwischen zwei verschiedenen Typen unterscheiden. Zum einen gibt es Connectoren, welche die klassischen HTTP-Protokolle unterstützen und somit den Tomcat zu einem stand-alone Web Server werden lassen. Zum anderen existieren Connectoren, welche als Bindeglied zwischen dem Tomcat-Server und einem Web Server, wie zum Beispiel dem Apache oder dem IIS, fungieren und somit den Browsern keinen direkten Zugriff auf den Tomcat erlauben (AJP und Warp). Die HTTP-Connectoren sind direkter Bestandteil des Tomcat-Servers und stehen als JAR-Datei zur Verfügung. Die Web Server-Connectoren dagegen setzen sich – wie sollte es anders sein – aus zwei Teilen zusammen, von denen der Tomcat-Teil in Java und der Web Server-Teil in C realisiert ist (Abb. 2).

Damit eine „genormte“ Kommunikation zwischen Web Server und Tomcat

überhaupt erst möglich wird, existieren entsprechende Protokolle auf Seiten der Server:

- WARP – Web Access Remote access/control Protocol
- AJP – Apache JServ Protocol (aktuell in der Version 1.3)

Tabelle 1 zeigt eine Übersicht der verschiedenen HTTP-Connectoren des Tomcat-Servers, Tabelle 2 die beiden Seiten der Web Server-Connectoren.

Mit den Connectoren der Coyote-Familie wurden im Rahmen der Tomcat 4.1-Neuentwicklung für beide oben vorgestellten Welten – Tomcat als stand-alone und Tomcat via Web Server – Lösungen geschaffen, welche in Geschwindigkeit und Flexibilität deutlich über die bis dato bekannten Connectoren hinausgingen. Neben dem HTTP-Protokoll Version 1.1 im Standard nach RFC 2616 [1] wird durch die Coyote Connectoren ebenfalls das Apache JServ-Protokoll Version 1.3 (AJP 1.3) unterstützt. Und auch die Anwendung einer verschlüsselte Variante (HTTPS) auf Basis der Java Secure Socket Extension (JSSE) oder aber der frei erhältlichen Implementierung der Protokolle SSL Version 3 und TLS Version 1 [2] namens PureTLS [3] ist möglich.

Im folgenden Verlauf möchten wir im *Thema des Monats* einen tieferen Einblick in die Familie der Coyote HTTP-Connectoren vermitteln. Ob die neuen Connectoren wirklich halten was sie versprechen und ob sich deren Einsatz in Ihrer Serverumgebung tatsächlich durch eine verbesserte Performance auszeichnet, sollten Sie selber ausgiebig testen. Das nötige Rüstzeug dazu erhalten sie durch die im *Profi Know-how* vorgestellte Anwendung des Werkzeuges JMeter.

Thema des Monats
Coyote HTTP-Connectoren

Die wesentlichen Neuerungen der Coyote Connectoren-Familie liegen in der Modularisierung und dem konsequenten Willen, die Geschwindigkeit der Protokollumsetzung zu erhöhen und zu flexibilisieren. Das Design dringt darauf, die gemeinsamen Bestandteile zwischen HTTP und AJP zu nutzen und entsprechende

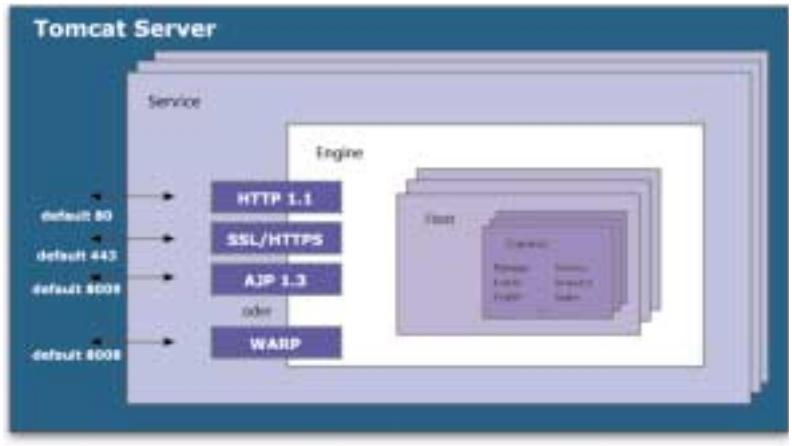

Abb. 1: Tomcat-Architektur

Unterschiedlichkeit durch Adaptoren, Filter oder Handler anpassen zu können. Im *jakarta-connector*-Projekt, deren Bestandteil der Coyote Connector ist, werden parallel dazu die Implementierungen aller aktiven Tomcat-Releases gepflegt [4]. Dies erleichtert zwar die Portierungsprobleme und Bugfixes der Gemeinsamkeiten, aber für uns „normalsterblichen" Nutzer ist es eine schwere Verständnisaufgabe. Der Erwerb der Kenntnisse zur Nutzung des Java-Netzwerk APIs ist nicht immer offensichtlich. Gerade die Portierung für verschiedene Betriebssysteme der Webserver-Komponenten in C macht die Übersetzung der *jakarta-connector* Module zu einer nicht unbedingt geliebten Aufgabe. Leider schlagen hier die Unterschiedlichkeiten der C-Plattformen für die Integration in bestehende Webserver dem Anwender ein echtes Schnippchen. Die einzige Hilfe sind die Mailinglist-Archive oder ein anderes C-Projekt auf derselben Plattform [5]. Hier wünschen wir uns, dass mehr Personen Ihre erfolgreichen Konfigurationen dokumentiert bereitstellen.

Ein wesentliches Augenmerk bei der Entwicklung der Coyote Connectoren ist auf die Erhöhung der Geschwindigkeit gelegt worden. Hier sind vor allem die Verbesserung im Filtern und Parsen der HTTP-Anfrage und die Rückführung auf *Byte*-Felder für Vergleiche, statt *Strings* zu nennen. Das HTTP-Protokoll ist eben 7 Bit ASCII-basiert und somit können alle Zuordnungen der Parameter auf den Datentyp *Byte* zurückgeführt werden. Eine wirklich unspektakuläre Einsicht mit enormen Folgen, denn jede Anfrage enthält im Schnitt zwischen acht und fünfzehn Header-Parameter. In den Filtern, Pools und

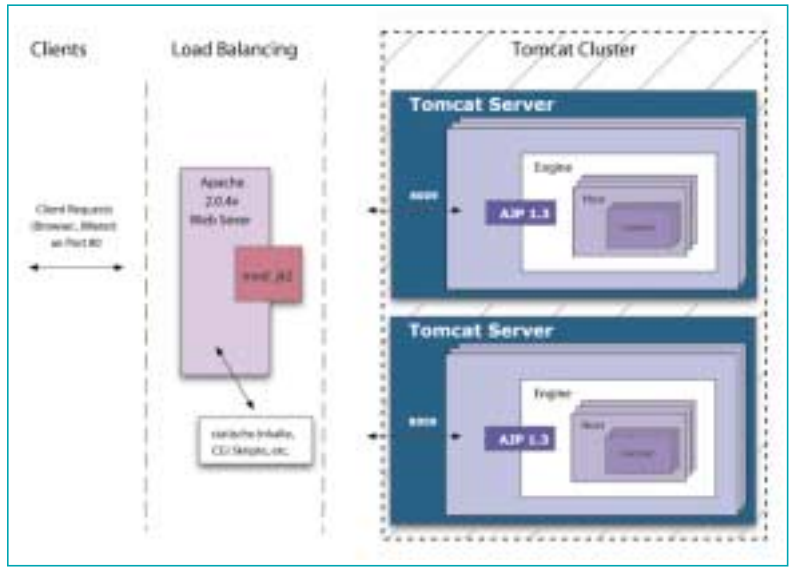

Abb. 2: Indirekte Kommunikation mittels Web Server-Connector

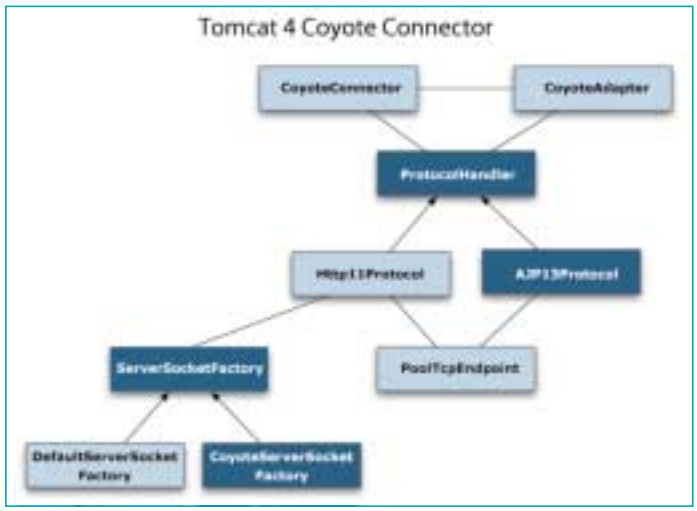

Abb. 3: Design der Coyote Connector-Familie

Transferklassen ist darauf geachtet worden, möglichst wenig Speicher anzufordern und damit den Garbage Collector der JVM zu entlasten. Man versucht weiterhin einmal aufgebaute Zwischenspeicher (Threads) wiederzuverwenden und konsequent zu nutzen. Falls es doch mal zu Fehlern kommt, berichteten dies die jeweilige Klasse der Coyote Connectoren via Commons Log API oder dem Engine Logger. Einer verfeinerten Debug-Aussteuerung via JDK 1.4 Logging oder Log4J steht also nichts im Wege [6].

Das zentrale Konfigurationselement ist die Klasse *CoyoteConnector*, die jeweils in einer spezifischen Ausprägung pro Tomcat-Version existiert (Abb. 3 und Listing 1). Diese Klasse nimmt die Parameter aus der *server.xml* Connector-Konfiguration entgegen und instanziert den Adapter und *ProtocolHandler*. Die Klasse *CoyoteAdapter* hat die Aufgabe, das Request/Response Paar für die Tomcat-Engine vorzubereiten und die Bearbeitung zu veranlassen [7], [8]. Der Adapter bildet die Schnittstelle zwischen der Engine des Catalina-Containers und den technischen Connectoren.

Der *ProtocolHandler* existiert je nach gewähltem Kommunikationsprotokoll für HTTP oder AJP. Jede externe Anfrage, die über einen *ServerSocket* des Connectors hereinkommt, muss im Handler in die internen Datenstrukturen des Catalina-Containers umgewandelt und interpretiert werden. Der Handler nutzt zur Instanzierung entweder die *DefaultServerSocketFactory* oder für den SSL-Support die *CoyoteServerSocketFactory*. Weiterhin steuert der *ProtocollHandler* einen Pool (*PoolTcpEndpoint*) von Threads, um die Nebenläufigkeit der Protokollinterpretation im Container effizient bereitzustellen. Die Handler sind durch geeignete Filter in der Lage, noch Spezialaufgaben wie die *gzip*-Komprimierung der Ausgabe oder eine Pufferung der Eingabe und Ausgabe einzuschalten. Eine detailliertere Betrachtung des Designs kann momentan leider nur durch das Selbststudium des Quellcode erfolgen. Eine exakte Beschreibung aller internen Möglichkeiten des Connector APIs existiert momentan nicht. Dies haben wir zum Anlass genommen, den Einsatz einer Kompression via *gzip* einmal nähr zu beleuchten.

Unterstütztes Protokoll	Java-Klasse	Bemerkung
HTTP 1.0	*org.apache.catalina.connector.http10.HttpConnector*	Veraltet
HTTP 1.1	*org.apache.catalina.connector.http.HttpConnector*	Veraltet
HTTP 1.1	*org.apache.coyote.connector.tomcat4.CoyoteConnector*	Empfohlene Version

Tab. 1: Tomcat HTTP-Connectoren

Unterstütztes Protokoll	Web Server-Part	Tomcat-Part	Bemerkung
AJP 1.3	mod_jk	CoyoteConnector mit *JkCoyoteHandler*	Veraltet, es wird empfohlen mod_jk2 zu verwenden
AJP 1.3	Mod_jk2	CoyoteConnector mit *JkCoyoteHandler*	Empfohlene Version und nutzt die Fähigkeiten des Apache 2 besser.
WARP 1.0	Mod_webapp	WarpConnector	Veraltet

Tab. 2: Web Server-Connectoren

Konfiguration des HTTP-Connectors

Die Beschreibung aller Parameter des Connectors befindet sich in der Tomcat-Dokumentation [9] und auf unserer Tomcat-Referenzkarte [10]. Die gebräuchlichsten Parameter sind sicherlich die Wahl des Socketports (*port*), die minimale und maximale Anzahl der gleichzeitigen Threads dieses Connectors (*minProcessor*, *maxProcessor*) und die Länge der Wartequeue von noch unbearbeiteten Anfragen vor dem Socket (*acceptCount*). Für die HTTPS-Variante wird das Protokollschema (*scheme*) auf den Wert *https* und *secure=true* gefordert. Für HTTPS muss durch das Subelement *Factory* noch eine andere *ServerSocketFactory* mit entsprechenden Sicherheitskonfigurationen angegeben werden. Der normale HTTP-Connector kann mit der Angabe des Parameters *redirectPort* automatisch einen HTTP-Redirect für den Fall erzeugen, dass eine Web-Anwendung in ihrer *WEB-INF/web.xml* einen *SecurityConstraint* mit „CONFIDENTIAL“ Transportgarantie angegeben hat [11]. Dies funktioniert auch, wenn der Tomcat hinter einem Webserver mittels AJP-Adapter betrieben wird. Natürlich lassen sich die diversen Timeout-Situationen oder die Anzahl der maximalen Keep-Alive-Requests einer Verbindung beeinflussen.

Eine recht schöne Ergänzung bietet die Möglichkeit, die direkte Komprimierung der Ausgabe einzustellen. Hier kann bei langsamer Modem-Anbindung des Clients Leitungskapazität zurückgewonnen werden [12]. Fast jeder heutige Browser kann standardmäßig seine Antworten komprimiert mittels *gzip*-Verfahren empfangen. Der Coyote Connector kann die Kompri-

mierung für textuelle Inhalte mit den Wert *on* oder einer Zahlenangabe in Kilobyte einschalten. Der Wert der Zahl legt dann die Mindestkomprimierungsgröße der Antwort fest, wobei 0 keine Kompression bedeutet und sonst ein Wert größer als 128 Bytes angegeben werden soll. Mit dem Wert *force* wird jede Anfrage komprimiert. Dieser Modus ist natürlich für Bilder wenig sinnvoll und sorgt im Gegenteil für wesentlich mehr Datenvolumen. In der Tomcat 5-Version ist der Handler endlich via API direkt erreichbar und damit ist die Möglichkeit gegeben, die User Agent (Clients bzw. verschiedene Browser-Versionen) und die Mimetypes für die Komprimierung flexibler einzustellen. Mit der Version 5.0.13 sind die weiteren Kompressionsattribute im Connector nun direkt konfigurierbar. Eine weitergehende, Tomcat-unabhängige Steuerung bietet sich nur durch den Einsatz entsprechender Servlet Filter in einer Web-Anwendung. Ein entsprechend einfaches Beispiel liefert der Tomcat im Verzeichnis *webapps/examples/WEBINF/classes/compressionFilter* gleich mit [8]. Mit Hilfe dieses Filters ist es möglich die Komprimierung auf einzelne Mappings der Web-Anwendung zu beschränken. Leider fehlen in dieser Version des Kompressionsfilters die Einschränkungen auf User Agents oder Mimetypes des Connectors. Bei der Konfiguration sollte darauf geachtet werden, dass nicht gleichzeitig der Connector und eine Anwendung die Kompressionsfilterung eingeschaltet haben. Auf der Heft-CD befinden sich zur Demonstration die vollständige Anwendung *myapps-filter* mit dem Tomcat-Beispiel *CompressionFilter*.

Listing 1

Konfigurationsparameter eines HTTP1/1 Coyote Connectors in der Datei *conf/server.xml*

```
<Connector
  className="org.apache.coyote.tomcat4.
                                    CoyoteConnector"
  protocolHandlerClassName="org.apache.coyote.
                                    http11.Http11Protocol"
  scheme="http"
  port="7380"
  address="192.168.0.12"
  secure="false"
  redirectPort="7443"
  minProcessors="5"
  maxProcessors="20"
  enableLookups="false"
  acceptCount="10"
  debug="0"
  compression="off"
  disableUploadTimeout="false"
  maxKeepAliveRequests="100"
  serverSocketTimeout="0"
  connectionUploadTimeout="300000"
  connectionTimeout="20000"
  connectionLinger="-1"
  useURIValidationHack="false"
  tcpNoDelay="true"
/>
```

Die Definition des Filters findet in der Web-Anwendung im Descriptor *WEB-INF/web.xml* statt (Listing 2).

Mit dem Parameter *compressionThreshold* wird die Minimalgröße der Kompression eingestellt. Wir haben für unseren Test

Listing 2

Servlet Filter für die Kompression der Inhalte des Mappings /filter/*

```
<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//
                                DTD Web Application 2.3//EN"
  "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
...
  <filter>
    <filter-name>CompressionFilter</filter-name>
    <filter-class>compressionFilters.
                          CompressionFilter</filter-class>
    <init-param>
     <param-name>compressionThreshold</
                                              param-name>
     <param-value>120</param-value>
    </init-param>
    <init-param>
     <param-name>debug</param-name>
     <param-value>3</param-value>
    </init-param>
  </filter>
  <filter-mapping>
   <filter-name>CompressionFilter</filter-name>
   <url-pattern>/filter/*</url-pattern>
  </filter-mapping>
...
</web-app>
```

absichtlich nur 120 Bytes gewählt, um alle Debug-Ausgaben des Filters mit dem Wert drei aufzeigen zu können. Die Ausgabe wird in die *System.out*-Ausgabe geschrieben. Nach dem Start des Beispiels *webdev-server* von der Heft-CD kann mit der Anfrage *http://localhost:7380/myapps-filter/gziptest* geprüft werden, ob der Browser die *gzip*-Kompression überhaupt unterstützt. Mit der Anfrage *http://localhost:7380/myapps-filter/filter/index.html* wird die Inhaltskompression aller Ressource im Verzeichnis *filter* veranlasst. Zur Prüfung haben wir den seit dem Tomcat-Release 4.1.27 verfügbaren *ExtendedAccessLogValve* eingesetzt. Mit diesem Valve sind sehr flexible *AccessLog*-Ausgaben möglich. Im Filterbeispiel haben wir den Browser-Anfrage-Header *Accept-Encoding* und den Antwort-Header *Content-Encoding* aufgezeichnet. Schön sehen wir im Protokoll, dass nur die erste Anfrage jedes Client-Browsers zu einer Kompression führen. Danach stellt der Tomcat durch den Vergleich der *Last Modified*-Angabe in der nächsten Browser Anfrage keine Änderung fest und der Browser präsentiert die Inhalte aus seinem Client Cache (Listing 3).

Der Einsatz von Kompression lohnt sicherlich nicht bei sehr schnellen Anbindung oder stark ausgelasteten Servern. Hier ist die zusätzliche Aufgabe der Kompression sicherlich aus Gründen des Speichermehrbedarfs und der CPU-Belastung zu vermeiden, aber für Kunden mit Modemanschluss ist diese Option oftmals eine echte Hilfe. Leider ist ohne eigenen Aufwand in einen besseren Filter der sinnvolle Einsatz mit dem Standardbeispiel zu prüfen.

Einige eher selten benutzte Einstellungen sind die Parameter *bufferSize*, *connectionLinger*, *disableUploadTimeout* und *useURIValidationHack*. Über den *bufferSize*-Parameter kann man die Größe des Buffered-Input-Streams kontrollieren, jedoch sei hier erwähnt, dass diese Einstellung von System zu System sehr unterschiedliche Auswirkungen haben kann. Man sollte diese Einstellung daher auf jeden Fall vorher ausprobieren. Um die Wartezeit bis zum Schließen einer Verbindung festzulegen, kann im Parameter *connectionLinger* ein Wert in Millisekunden angegeben werden. Standardmäßig steht dieser auf -1. Bei Anwendungen mit vielen Uploads besteht die Möglichkeit der Timeout-Unterdrückung. Dies ist vor allem bei sehr großen Upload-Anfragen notwendig. Die Einstellung *useURIValidationHack* ist, wie der Name schon andeutet, ein Workaround für die Tomcat 4.0.x-Versionen. Im Tomcat Version 4.1 sollte diese Einstellung immer *false* sein.

Profi Know-how
Viele Konfigurationsoptionen und welche ist die Richtige?

Welcher Connector mit welchen Konfigurationseinstellungen jetzt genau der Richtige ist, lässt sich leider nicht pauschal beantworten. Sicherlich kann man die Aussage treffen, dass der Coyote Connector weiterentwickelt wurde als der ursprüngliche HTTP1.1-Connector. Das bedeutet aber nicht, dass vielleicht nicht doch auf genau Ihrem System der alte HTTP1.0-Connector genau der Richtige ist. Um dem Connector Ihrer Wahl auf die Spur zu kommen, sind sicherlich in erster Linie die konfigurierbaren Elemente des Connectors maßgebend. Was aber danach? Verlässliche Aussagen liefern hier nur Performance-Vergleichsmessungen. Hiermit haben Sie die Möglichkeit, auf Ihrem konkreten Zielsystem die Auswirkungen von verschiedenen Connector-Konfigurationen zu vergleichen. Leider ist das Messen der Performance leichter gesagt als getan: Oft ist das Zielsystem nicht für einen Lasttest verfügbar und kleine Unterschiede in der Konfiguration oder der verwendeten Anwendung können große Wirkung im Ergebnis hinterlassen. Man kommt also nicht umher, dass Zielsystem zu nutzen, oder so exakt wie nur möglich nachzustellen. Um die Ergebnisse nicht zu verfälschen sollte man für diesen Test eine Anwendung wählen, welche die typischen Verhaltensweisen der Anwendung aufweist. Außerdem sollte man auch den Einsatz verschiedener Testwerkzeuge in Betracht ziehen, da auch in den Werkzeugen selbst Unterschiede in der Herangehensweise oder der Protokoll-Implementierung liegen können.

Erst die Ergebnisse verschiedener Tests bieten die Möglichkeit, einigermaßen objektiv über seine beste Konfiguration urteilen zu können. Es kostet meist einige Zeit, die verschiedenen Werkzeuge und System auszuprobieren. Wir empfehlen schon zu Beginn eines Web-Projekts mit dem Aufbau des Instrumentariums zu starten und ebenfalls mit dem gesamten Projekt inkrementell zu wachsen. Als kleinen Anreiz haben wir eine einfache Testumgebung mit dem Jakarta-Projekt JMeter [13] in der Version 1.9.1 und mit der Filter-Anwendung MyApps realisiert. Die Apache-Gruppe stellt mit JMeter ein in Java implementiertes Werkzeug zur Verfügung, mit dem Client/Server-Anwendungen unter Last gesetzt werden können. Die Simulation von sehr großer Last auf den Systemen, Netzwerken oder auch Objekten sollte man nutzen, um auf Probleme oder Grenzen der Anwendung und der Systeme aufmerksam zu werden. JMeter bietet mehrere Möglichkeiten die Anfragen abzusetzen:

- FTP,
- HTTP,
- JDBC,
- Java-Objekt,
- LDAP,
- SOAP/XML-RPC,
- Web Service (SOAP) (Alpha Implementierung).

Mit Hilfe von JMeter und Ant haben wir ein parametrisierbares Szenario aufgestellt, um die Konfiguration des Tomcat-Connectors zu testen (Abb. 4). Die Anfragen an MyApps bestehen dabei lediglich

Listing 3

```
#Fields: c-ip date cs-uri cs(Accept-Encoding) sc-status sc(Content-Encoding) sc(Content-Type) sc(Content-Length)
#Version: 1.0
#Software: Apache Tomcat/4.1.27
127.0.0.1 2003-10-05 /myapps-filter/filter/index.html "gzip, deflate" 200 "gzip" "text/html" "195"
127.0.0.1 2003-10-05 /myapps-filter/filter/tomcat.gif "gzip, deflate" 200 "gzip" "image/gif" "1464"
127.0.0.1 2003-10-05 /myapps-filter/filter/index.html "gzip, deflate" 304 - - -
127.0.0.1 2003-10-05 /myapps-filter/filter/tomcat.gif "gzip, deflate" 304 - - -
```

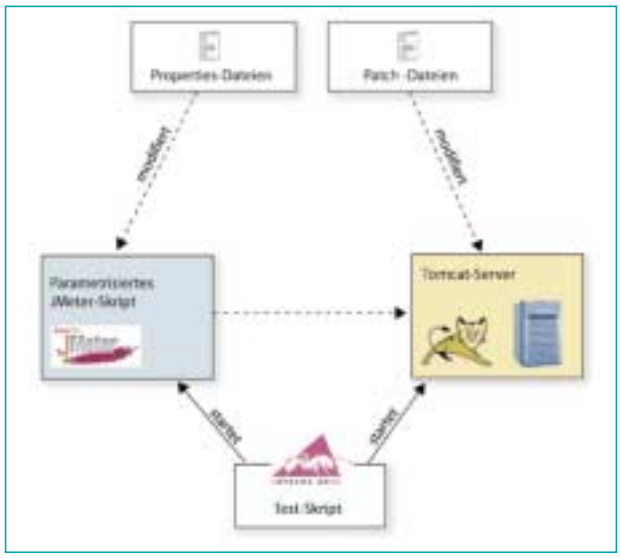

Abb. 4:
Lasttest-Szenario

aus HTTP-Anfragen auf *eine* HTML-Datei, *eine* JavaServer Page und *ein* Servlet. Das JMeter-Skript startet parallel zehn Threads, welche zehnmal diese drei Seiten sequenziell aufrufen. Für dieses Beispiel haben wir das Skript so parametrisiert, dass wir den Test mit der Anzahl Threads, Wiederholungen und dem Protokoll starten können. Der Test wird mit dem Aufruf *ant execute* gestartet und mit *ant result* wird ein Browser mit dem Ergebnissen aufgerufen. Standardmäßig wird dann die Datei *build.properties* genutzt, welche die Anfragen über das HTTP-Protokoll mit zehn Threads und zehn Wiederholungen abgesetzt. Indem man *ant execute* den Parameter *-Ddeploytarget=More* übergibt, kann man den Test mit einer zweiten Konfiguration starten, welche in unserem Beispiel den Test auf dem HTTPS-Protokoll durchführt. Mit diesem kleinen Test könnte man nun verschiedene Connector-Szenarien ausprobieren und die Ergebnisse vergleichen. Die Ergebnisse, welche JMeter hier liefert, sind natürlich nicht allein ausschlaggebend. Damit man sich ein noch genaueres Bild seiner Konfiguration machen kann, sollte der Test nicht nur lokal auf einem Rechner ausgeführt werden, sondern auch über ein Netzwerk, wenn möglich auch mit entsprechenden Knoten dazwischen.

Ein interessanter Punkt ist sicherlich das zusätzliche Monitoring auf dem System selber, um das Verhalten der CPU, des Speichers und des Netzes analysieren zu können, außerdem können Profiler dazu dienen, den Status der VM zu überwachen. Letztlich fällt es meist nicht leicht die richtige Konfiguration für die Connectoren zu finden. Unsere Erfahrung mit dem Speicherbedarf des Tomcat bei der Verwendung von HTTPS und die damit verbundene Auslastung des Systems sind unerfreulich. Vielleicht haben Sie schon ähnliche Erfahrungen gemacht? Mit der hier vorgestellten Lasttestumgebung könnten Sie auf Ihrem System die Ursache eingrenzen. Lassen Sie uns die Ergebnisse und Szenarien im Forum diskutieren [14]. Eine Alternative zur Lasterzeugung, mit der wir gute Erfahrungen gesammelt haben, ist das Grinder Sourceforge-Projekt [12].

Und beim nächsten Mal ...

In der nächsten Ausgabe des Java Magazins werden wir uns eingehend mit der Thematik der JMX-Fähigkeit des Tomcat 4.x beschäftigen. Darüber hinaus findet eine Betrachtung einiger Eigenschaften und Neuerungen der nächsten Generation des Tomcat-Servers statt, die natürlich auch – passend zum aktuellen Thema der Kolumne – einen Blick auf die erweiterten JMX-Fähigkeiten des Tomcat 5 beinhaltet.

Wir freuen und schon auf Kommentare und Anregungen – besuchen sie also unsere Tom C@ Site und das Tom C@ Forum oder die Tomcat-Mailinglisten [10], [14], [5].

Links & Literatur

[1] Spezifikation des HTTP 1/1-Protokolls: www.w3.org/Protocols/rfc2616/rfc2616.html

[2] Spezifikation des TLS-Protokolls Version 1: www.ietf.org/rfc/rfc2246.txt

[3] www.rtfm.com/puretls/

[4] jakarta.apache.org/builds/jakarta-tomcat-connectors/

[5] www.mail-archive.com/tomcat-user@jakarta.apache.org/

www.mail-archive.com/tomcat-dev@jakarta.apache.org/

[6] jakarta.apache.org/log4j/

[7] P. Roßbach, L. Röwekamp, M. Kloss: „Kater a la Carte", *Java Magazin* 9.2003

[8] P. Roßbach (Hrsg.); A. Holubek, T. Pöschmann, L. Röwekamp, P. Tabatt: „Tomcat 4X: Die neue Architektur und moderne Konzepte für Webanwendungen im Detail", Software und Support Verlag, 2002

[9] jakarta.apache.org/tomcat/tomcat-4.1-doc/connector-howto.html

[10] tomcat.objektpark.org/

[11] java.sun.com/products/servlet/download.html

[12] Peter Zadrozny Philip Aston, Ted Osborne; „J2EE Performance Testing With BEA Weblogic Server" Expert Press Birmingham UK 2002

[13] jakarta.apache.org/jmeter/

[14] www.javamagazin.de/tomcat/