Einmal ganz anders

Lizenzserver mit verteilten Lizenzservlets

Eric Müller

Oftmals ist ein Lizenzserver ein eigens zu installierendes und zu administrierendes Programm, das für den Anwender keinen spezifischen Nutzen bringt, im Gegenteil die Anwendung sogar durch Absturz blockieren kann. Dieser Artikel zeigt, wie es bei serverbasierten Anwendungen auch anders gehen kann, und erläutert ein paar auch für andere Situationen lehrreiche Implementierungsdetails.

Viele Softwarehersteller lizenzieren ihre Produkte nach maximaler Anzahl der Benutzer, die die Anwendung gleichzeitig verwenden können (Concurrent User). Ein Anwendungsstart erhöht hierzu üblicherweise an einer zentralen Stelle einen Zähler um 1, das Beenden der Anwendung verringert den Zählerstand um 1. Daraus folgen zwei Probleme:

- Fällt die Zählerverwaltung aus (meist ein zentral vorliegendes eigenes Programm – der Lizenzserver), kann niemand mehr arbeiten.
- Stürzt die Anwendung plötzlich ab, hat sie keine Möglichkeit mehr, den Zähler zu verringern (dies muss dann oft der Administrator manuell erledigen).

Verteilter Lizenzserver

Für eine in einem Cluster von Servletcontainern, üblicherweise Tomcats [Tomcat], laufende Anwendung bestand die konkrete Anforderung, einen Lizenzserver zu programmieren, der diese Probleme umgeht. Da die Anwendung ohnehin serverbasiert läuft, bietet es sich an, diese Aufgaben nicht in einen eigenen Prozess auszugliedern, sondern in die ausgelieferte Webapplikation zu integrieren, d. h. bei Clusterbetrieb besteht das Lizenzprogramm aus mehreren gleichberechtigten Bestandteilen – im Folgenden „Lizenzservlets“ genannt – die ihre Informationen selbstständig abgleichen. Damit stehen die Lizenzservlets stets mindestens so lange zur Verfügung wie die eigentliche Anwendung, sodass das Lizenzprogramm nicht durch Absturz die Applikation blockieren kann. Nun bleibt zu klären, wo sich der Zähler der Benutzersitzungen befindet. Naheliegend wäre eine Datenbanktabelle – diese erfährt aber nichts von Rechnerabstürzen. Besser ist es, wenn das Lizenzservlet zu Beginn einer Anwendungssitzung alle anderen Server kontaktiert, um festzustellen, wie viele Sitzungen dort gerade aktiv sind.

Kommunikation und Synchronisation

Diese dezentrale Struktur löst obige Probleme, bedingt aber weitere Anforderungen:

- Unabhängig von der Anzahl der Lizenzservlets soll sich das Lizenzprogramm so verhalten und bedienbar sein, wie wenn es nicht aufgeteilt wäre.
- Werden neue Lizenzinformationen eingespielt, während gerade nicht alle Server in Betrieb sind, sollen deren Lizenzservlets automatisch die neuen Informationen beim ersten Kontakt mit den anderen Lizenzservlets erhalten.

Für die Änderungs- und Abgleichmechanismen benötigt jedes Lizenzservlet neben den eigentlichen Lizenzinformationen also noch Metadaten, um festzustellen, ob die abzugleichenden Lizenzdaten identisch zu denen auf einem anderen Rechner sind bzw. welche aktueller sind.

Die Metadaten sind Bestandteil jeder Anfrage (um nicht stets alle Lizenz- und etwa sonstigen Konfigurationsdaten mitschicken zu müssen). Nur wenn die Metadaten von anfragendem und angefragtem Lizenzservlet übereinstimmen, erfolgt sofort die Antwort. Hat das abgefragte Lizenzservlet ältere Daten, liefert es eine besondere Antwort, welche den Anfrager veranlasst, den kompletten Satz an aktuellen Lizenz- und Konfigurationsdaten mitzuschicken; nach Aktualisieren der lokalen Daten erfolgt die Antwort. Hat das abgefragte Lizenzservlet neuere Daten, schickt es diese zunächst an den Anfrager. Der Anfrager ermittelt dann aus allen Antworten die aktuellsten Lizenzdaten und aktualisiert danach in einem weiteren speziellen Aufruf die Lizenz- und Konfigurationsdaten aller anderen Lizenzservlets (dieser spezielle Aufruf erfolgt übrigens auch beim Einspielen neuer Lizenzdaten). Dann sollte er auch die anfängliche Anfrage auf Basis der aktuellen Lizenzdaten wiederholen.

Wenn die Lizenzservlets ihre Konfiguration und Daten komplett aus den ausgestellten Lizenzdaten beziehen, lässt sich dies gut auf Grundlage der Open-Source-Bibliothek TrueLicense von Christian Schlichtherle [TLC] implementieren. Diese verschlüsselt Lizenzdaten mit einem privaten Zertifikat, das die Anwender auf keinen Fall erhalten dürfen; diese erhalten das entsprechende öffentliche Zertifikat ausgeliefert, womit die Lizenzservlets die Lizenzinformationen auslesen können. Da normalerweise eine zentrale Stelle die Lizenzdaten ausstellt, eignet sich ein dort vergebener Zeitstempel gut für obige Metadaten.

In der konkreten Anwendung war die Situation komplizierter, da die URLs der beteiligten Tomcats mit Rechnername, Port, Webapplikationsname auf Anwenderseite zentral konfigurierbar sein sollten, ohne jeweils neue Lizenzen beantragen zu müssen. Somit hatten die Lizenzservlets nicht nur die reinen Lizenzdaten untereinander abzugleichen, sondern auch noch zusätzliche Konfigurationsinformationen. Ein Zeitstempel in den Lizenzinformationen genügte nun nicht mehr. Um zu entscheiden, welche Information im Konfliktfall aktueller ist, hat sich eine Kombination aus einer eindeutigen Identifikationskennung, z. B. generiert mittels `java.util.UUID`, einem bei jeder Änderung um eins erhöhten Versionszähler sowie dem Zeitstempel der letzten Änderung bewährt: Lizenz- und Konfigurationsdaten zählen als neuer, wenn sie höhere Versionszahl haben oder bei gleicher Versionszahl den späteren Zeitstempel. Der Vergleich von Zeitstempeln ist nur als Notlösung für den unwahrscheinlichen Fall konkurrierender, unabhängiger

Änderungen gedacht. Wenn die Zeiten auf den Rechnern nicht zuverlässig synchronisiert sind, z. B. durch Network Time Protocol [NTP], zeigen lokal erzeugte Zeitstempel möglicherweise nicht die wahren Verhältnisse auf.

Es war im konkreten Fall nicht nötig, die Abfragen der Lizenzservlets besonders zu synchronisieren und zu sichern, wie es z. B. mit den Mechanismen aus der Open-Source-Bibliothek Appia [Appia] möglich ist. Dies verringerte auch den Netzwerkverkehr zwischen den Servern.

Ermittelung der Benutzerzahlen

Wenn das Lizenzservlet beim Anwendungsstart synchron alle anderen Lizenzservlets nach den aktuellen Benutzerzahlen befragt, ist sichergestellt, dass die maximale Gesamtbenutzerzahl stets respektiert wird. Insbesondere bei größeren Netzwerken und wenn einzelne Server nicht antworten, kann dies aber einige Sekunden dauern.

Wenn die Lizenzprüfung nicht parallel zu sonstigen Initialisierungen abläuft, kann es deutlich performanter sein, wenn sich die Lizenzservlets bei jeder Anfrage über ihre Benutzerzahlen gegenseitig informieren und das Lizenzservlet beim Anwendungsstart aufgrund der gespeicherten Benutzerzahlen entscheidet, ob die Anmeldung erfolgreich ist. Dann tauscht es mit allen anderen Lizenzservlets die aktuellen Benutzerzahlen aus, ohne weiter die Anwendung zu blockieren. Zumindest jedoch wenn gemäß der gespeicherten Benutzerzahlen die maximale Anzahl überschritten ist, sollte das Lizenzservlet auf jeden Fall alle anderen Lizenzservlets synchron abfragen. Damit kann es eine eigentlich berechtigte Anmeldung nicht aufgrund veralteter Informationen zu Benutzeranzahlen ablehnen. Bei dieser Vorgehensweise ist es zwar im seltenen Fall, dass sich zwei Benutzer gleichzeitig auf zwei Servern anmelden, möglich, dass zu viele Benutzer gleichzeitig das Programm nutzen, systematische Überschreitungen fallen aber immer noch so häufig auf, dass der Kunde besser Lizenzen nachkauft.

Die folgenden Abschnitte skizzieren die Lösung einiger Teilprobleme von allgemeinerem Interesse.

Derselbe Programmcode für unterschiedliche Java-Versionen

Die Identität von Hardware lässt sich recht einfach mit Hilfe von Daten wie Rechnername, IP-Adresse und den MAC-Adressen [MAC] der Netzwerkkarten testen. Unter Java erhält man letztere mithilfe der erst in Java 6 vorhandenen Methode `NetworkInterface.getHardwareAddress()`. Wie lässt sich diese in einem mit Java 5 und Java 6 gleichermaßen kompilierbaren Code aufrufen?

Dies ist ein guter Anwendungsfall der Reflection-API von Java: Ist obige Methode nicht vorhanden, gibt es bei Aufruf über Reflection-API keinen Kompilierfehler, sondern eine `NoSuchMethodException` zur Laufzeit, bei deren Auftreten andere Wege zu beschreiten sind, etwa der Aufruf nativer Methoden. Ein Codeausschnitt findet sich in Listing 1.

Hierbei ist noch zu beachten: Läuft die Anwendung möglicherweise mehrfach als Webapplikation im Tomcat, ist es nicht ratsam, eine dynamisch geladene Programmbibliothek (DLL bzw. shared library) zu laden – sie kann nämlich nur durch jeweils einen Thread geladen werden (siehe [Forum]). Ist sie bereits geladen (z. B. durch eine andere Webapplikation im Tomcat), kommt der Fehler: „UnsatisfiedLinkError: native DLL already loaded in another classloader“. Eine geladene Bibliothek lässt sich aber nur entladen, indem der zugehörige Classloader durch zweimaligen Aufruf der Garbage Collection entfernt wurde [Forum].

Besser ist es hier, ein externes Programm aufzurufen und auf dessen Standardein/-ausgabe mittels `Process.getInputStream()` und `Process.getOutputStream()` zuzugreifen. Diese externen Programme, üblicherweise in C geschrieben, sind für alle zu unterstützenden Betriebssysteme und Rechnerarchitekturen zu erstellen (ggfs. statisch gelinkt, um von der Betriebssystemversion unabhängiger zu sein). Betriebssystemkennung und Rechnerarchitektur liefert `System.getProperty()`, womit sich zur Laufzeit das richtige Programm auswählen lässt.

Synchronisierte Lese- und Schreibzugriffe

Das Lizenzprogramm ändert recht selten die Daten, liest sie aber oft. Es darf keine zwei gleichzeitigen Schreibzugriffe geben, aber beliebig viele Lesezugriffe. Durch den `synchronized`-Mechanismus ist dies nicht direkt lösbar, wohl aber mittels der Locks aus der in Java 5 eingeführten Concurrent-API. Im einfachsten Fall entspricht ein durch ein Lock-Objekt `lock` geschützter Bereich

```
lock.lock(); // Sperre setzen
try { ... // keine gleichzeitigen Aufrufe
} finally {
  lock.unlock(); // Sperre freigeben
}
```

einem Block (`mutex` ist eine Instanz einer beliebigen Klasse)

```
synchronized(mutex) {
  ... // keine gleichzeitigen Aufrufe
}
```

In beiden Fällen blockiert der Code der ersten Zeile so lange, bis kein anderer Thread mehr den Block ausführt.

Die Locks bieten zusätzliche Funktionalität: Die Methode `lock.trylock()` setzt versuchsweise eine Sperre – ist sie bereits gesetzt, liefert sie, ohne zu blockieren, das Ergebnis `false`. Ersetzt man oben `lock.lock()` durch `lock.lockInterruptibly()`, lässt sich die Blockade durch einen Aufruf von `interrupt()` auf dem Thread beenden – es kommt eine `InterruptedException`. Dies liefert einen Notbehelf, um aufgrund von Programmierfehlern entstehende Deadlocks (zwei Threads blockieren sich gegenseitig) zur Laufzeit aufzulösen. Damit nicht unerwartet von außen kommende `interrupt()`-Aufrufe (sogenannte „spurious wakeups“) den Programmablauf beeinträchtigen, ist der Sperrvorgang

```
import java.util.*;
import java.net.*;
import java.lang.reflect.*;
...
ArrayList<byte[]> macAddresses;
try {
  Method getHardwareAddress =
  NetworkInterface.class.getDeclaredMethod("getHardwareAddress");
  Enumeration<NetworkInterface> intf =
    NetworkInterface.getNetworkInterfaces();
  try {
    while(intf.hasMoreElements()) {
      byte[] mac =
        (byte[]) getHardwareAddress.invoke(intf.nextElment());
      macAddresses.append(mac);
  } catch (InvocationTargetException ex) { ... }
  } catch (NoSuchMethodException ex) {
  ... // Java 5: Aufruf eines externen Programms
}
```

Listing 1: Identität der Hardware mit getHardwareAddress ermitteln

nur zu unterbrechen, wenn dies wirklich erwünscht ist. Mögliche Lösung: Ersetzt man alle Aufrufe `lock.lockInterruptibly()` durch Aufrufe von `LockUtil.lockInterruptibly(lock)` aus Listing 2, so steuert die Variable `LockUtil.interrupt`, ob Unterbrechungen durch `interrupt()`-Aufrufe wirksam sein sollen.

Für das anfangs geschilderte Problem nun besonders praktisch ist die Klasse `java.util.concurrent.locks.ReentrantReadWriteLock`: Die Methode `writeLock()` liefert ein Lock für Schreibzugriffe und `readLock()` eines für Lesezugriffe. Damit können einerseits beliebig viele Threads gleichzeitig Lesezugriff erhalten. Vor einem Schreibzugriff müssen aber erst alle Lesezugriffe beendet sein, und keine zwei Threads erhalten gleichzeitig Schreibzugriff. Somit behindern sich die Lesezugriffe nicht, und es gibt trotzdem keine Inkonsistenz bei Schreibzugriffen.

Exklusive Dateizugriffe

Die Lizenz- und Konfigurationsdaten sind sinnvoll in einer Datei abgelegt, die das Lizenzservlet beim Starten des Tomcats einliest und bei Änderungen überschreibt. Es ist nicht mit Sicherheit auszuschließen, dass mehrere Lizenzservlets in unterschiedlichen Tomcats gleichzeitig die lokal vorliegenden Dateien ändern wollen. Hier hilft also kein regulärer Java-Synchronisationsmechanismus mehr. Das Betriebssystem selbst erlaubt aber, eine Datei exklusiv zu öffnen. Dies lässt sich auch von Java seit Version 1.4 mit der New-I/O-API nutzen:

```
RandomAccessFile raf = new RandomAccessFile("name.txt", "rw");
FileChannel fc = raf.getChannel();
FileLock fl = fc.tryLock();
```

Ist das erhaltene Objekt `fl` ungleich `null`, hat man den alleinigen Zugriff auf die Datei, ist es `null`, hat bereits ein anderer Prozess den Zugriff geholt und man muss warten; ist bereits in einem anderen Thread derselben Java-Umgebung der exklusive Zugriff geholt, kommt eine `OverlappingFileLockException`. Da es nicht möglich ist, einfach blockierend zu warten, bis die Datei frei ist, sollte man mehrmals in unregelmäßigen Abständen versuchen, die Sperre zu erhalten, bis sie verfügbar ist (oder irgendwann aufgeben). Listing 3 zeigt den Algorithmus.

Wenn die Dateiöffnungsversuche unregelmäßig stattfinden, behindern sich Threads, die einmal zeitgleich vergeblich versuchen, die Sperre zu holen, bei weiteren Versuchen mit hoher Wahrscheinlichkeit nicht erneut.

Auf jeden Fall angeraten ist es, die Datei nur so kurz wie möglich exklusiv zu bearbeiten, damit sich die einzelnen Prozesse nur so wenig wie möglich in die Quere kommen können.

```
import java.util.concurrent.locks.*;
public class LockUtil {
  // nur unterbrechen, wenn Wert auf true steht
  static volatile boolean interrupt = false;
  ...
  public static void lockInterruptibly(Lock lock)
  throws InterruptedException {
    while(true) {
      try {
        lock.lockInterruptibly(); // Sperrversuch
        return; // Sperre erfolgreich gesetzt
      } catch (InterruptedException ex) {
        if (interrupt)
          throw ex; // gewünschte Unterbrechung
      } // unerwünschte Unterbrechung – weiterer Sperrversuch
    }
  }
}
```

Listing 2: Unterbrechbare bzw. nicht unterbrechbare Synchronisation

```
import java.util.*;
import java.io.*;
import java.nio.channels.*;
...
Random r = new Random();
FileLock fl = null;
for (int i=0; i<10; i++) { // maximal 10 Versuche
  RandomAccessFile raf = new RandomAccessFile("datei.txt", "rw");
  FileChannel fc = raf.getChannel();
  try {
    fl = fc.tryLock();
  } catch (OverlappingFileLockException ignore) {}
  if (fl != null) {
    ... // Verarbeitung
    break;
  }
  try { Thread.sleep(r.nextInt(1000)); // warte zwischen 0s und 1s
  } catch (InterruptedException ignore) {}
}
```

Listing 3: Exklusiver Dateizugriff

Fazit und Ausblick

Die auf den ersten Blick nicht zu kompliziert erscheinende Aufgabenstellung beinhaltet viele Probleme, deren Lösung viele Bereiche der Java-Plattform involviert.

Bei der konkreten Implementierung löste der verteilte Lizenzserver aufgrund seiner Struktur fast von selbst noch eine zusätzliche Anforderung: Der Kommunikationsmechanismus ist an sich nicht auf lizenzrelevante Daten beschränkt – es lassen sich beliebige synchrone und asynchrone Rundrufe implementieren. Dies führte insbesondere zu einer alle Server vereinigenden Administrationsseite, die die früheren webapplikationsspezifischen Seiten ablöste. Damit führte der Lizenzserver zu einer echten funktionellen Bereicherung der Applikation und nicht nur zur Kontrolle der Anwender.

Literatur und Links

[Appia] Appia Group Communication Library, `http://appia.di.fc.ul.pt`; `http://appia.continuent.org`
[Forum] Entladen einer nativen Programmbibliothek, `http://forums.sun.com/thread.jspa?threadID=5296125`
[MAC] MAC-Adresse, `http://de.wikipedia.org/wiki/MAC-Adresse`
[NTP] Network Time Protocol, `http://www.ntp.org`
[Tomcat] Tomcat und Servletspezifikation, `http://tomcat.apache.org`; `http://java.sun.com/products/servlets`.
[TLC] TrueLicense, `http://truelicense.dev.java.net`

Dr. Eric Müller arbeitet bei der Sage bäurer GmbH in Villingen-Schwenningen an Werkzeugen für die Entwicklung in Java und auf der boa-Plattform. Seine Interessenschwerpunkte sind Automatisierung durch Skripte und Betreuung mathematischer Schülerwettbewerbe.
E-Mail: eric.mueller@sagebaeurer.de.