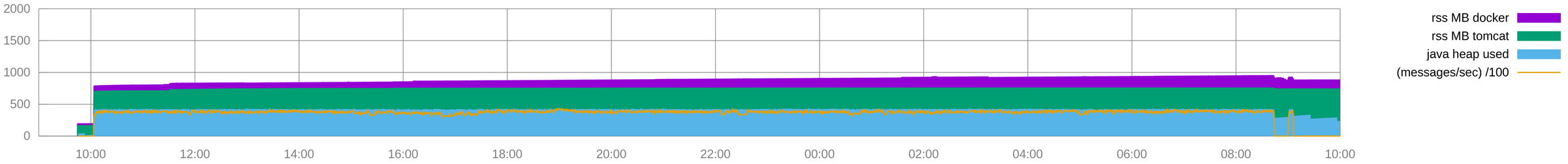
Off-Heap Memory Leaks in Servlet-Containern

Modul Qualitätssicherung, Oliver Radfelder & Karin Vosseberg

Hochschule Bremerhaven

Abbildung 1: Java-Webanwendung im *tomcat 10* im Docker-Container mit Java SE 17, -Xmx512m, 5000 aktiven Websockets und deaktiviertem *per-message-deflate*: der verbrauchte Speicher des Java-Prozesses bleibt unter 1 GB



Problemstellung

In der Infrastruktur der Informatik an der Hochschule Bremerhaven werden den Studierenden Docker-Container zur Verfügung gestellt, in denen als Servlet-Engines *tomcat*, *wildfly* und *jetty* installiert sind. Die Java-Versionen 8, 11 und 17 stehen ebenfalls bereit. Die Container selbst werden mit je vier virtuellen CPUs und zwei GB virtuellem Speicher gestartet.

In aktuellen Webanwendungen werden Websockets als Technologie eingesetzt. Seit einigen Jahren ist bereits zu beobachten, dass eine Webanwendung mit Websockets in einer so RAM-beschränkten Umgebung im *tomcat* in den Versionen 8, 9 und nun auch 10 unter Last schnell so viele Ressourcen verbraucht, dass der OOM-*Killer* (*OutOfMemory*) den Java-Prozess beendet. Das betrifft alle drei verfügbaren Java-Versionen.

Testkonstellation

Um festzustellen, ob das Problem durch einen Programmierfehler im *tomcat* oder der Java-Runtime verursacht wird, oder aber die beschränkten Ressourcen als solche der Grund sind, wird eine kleine Webanwendung deployt. Sie besteht lediglich aus einem Websocket-Endpunkt und einem kleinen ContextListener, der sekündlich den aktuell genutzten Heap und die Anzahl der empfangenen Nachrichten ins Log schreibt.

Zusätzlich werden zwei *node*-Prozesse auf einem weiteren Host gestartet, die jeweils 2500 Websockets öffnen und sequenziell Nachrichten an den Endpunkt senden und die Antwort abwarten (Echo-Service). Nach 100 Nachrichten wird der jeweilige Websocket geschlossen und ein neuer geöffnet, so dass insgesamt durchgängig über Stunden etwa 5000 aktive Websockets verbunden sind.

Erste Beobachtung

In der Standardkonfiguration des *tomcat* 10 unter Java SE 17 stellt sich heraus, dass bei 2 GB virtuellem Speicher der OOM-Killer den Java-Prozess innerhalb von 10 Minuten beendet. Bei zugewiesenen 3 GB virtuellem Speicher läuft der Prozess auch nach Stunden noch. Allerdings steigt der Verbrauch auf über 2.5 GB (siehe Abb. 2 mit Legende aus Abb. 1).

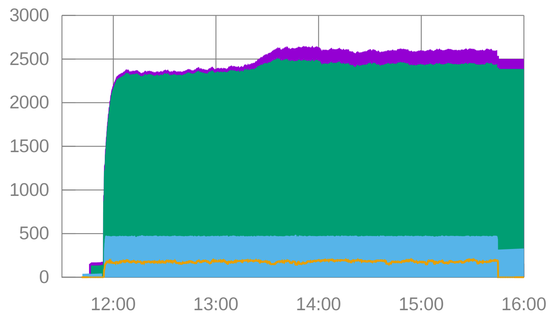


Abbildung 2: *tomcat* 10 im Docker-Container: der Off-Heap Speicherverbrauch steigt über 2 GB bei maximalem Heap von 512 MB

In der Horizontalen sind die Uhrzeiten aufgezeichnet. Die Experimente liefen jeweils etwa vier Stunden und wurden zu unterschiedlichen Tageszeiten durchgeführt.

Zusätzlich ist zu beobachten:

- die Anzahl der Nachrichten pro Sekunde liegt bei etwa 20.000
- der Java-Heap liegt tatsächlich nahezu durchgehend bei 500 MB

Diese Kombination lässt darauf schließen, dass sowohl auf den Garbage-Collector als auch auf das Verarbeiten der Websocket-Nachrichten nicht unerhebliche Anteile an dem CPU-Verbrauch fallen.

Die gleichen Ergebnisse sind mit *tomcat* in den Versionen 8 und 9 sowie den Java-Versionen 9 und 11 zu beobachten. Als Werkzeuge haben sich nach mehreren Experimenten u.a. mit *pmap* und *ps* als besonders geeignet letztlich *docker stats*, *pidstat* und *jcmd* herausgestellt, mit denen die Daten für die Aufbereitungen erzeugt wurden.

Die erste Arbeitshypothese: Irgendwo in der Implementierung des *tomcat* existiert ein Problem, das zu einem übermäßigen Off-Heap Verbrauch führt.

Prüfung der Hypothese mit *jetty*

Wenn es denn an der Implementierung des *tomcat* liegt, sollte eine andere Servlet-Engine ein anderes Verhalten zeigen. Daher wird die gleiche Anwendung mit den gleichen Parametern (512 MB max Heap, etc.) in *jetty* deployt und mit den gleichen Clients getestet.

In Abbildung 3 ist zu erkennen, dass beim *jetty* das gleiche - oder zumindest ein vergleichbares - Problem auftritt: nach sehr kurzer Zeit steigt der von *docker stats* angezeigte Speicherverbrauch auf deutlich über 2 GB und erreicht nach etwas über vier Stunden 2.5 GB. Bekommt der Docker-Container lediglich 2 GB virtuellen Speicher zugewiesen, wird der Java-Prozess nach weniger als 10 Minuten von dem OOM-Killer beendet.

Hier ist weiterhin zu beobachten:

- auch hier beträgt die Anzahl der abgearbeiteten Nachrichten pro Sekunde etwa 20.000
- der tatsächlich gebrauchte Heap beträgt im Mittel nur etwa die Hälfte von dem, den der *tomcat* benötigt

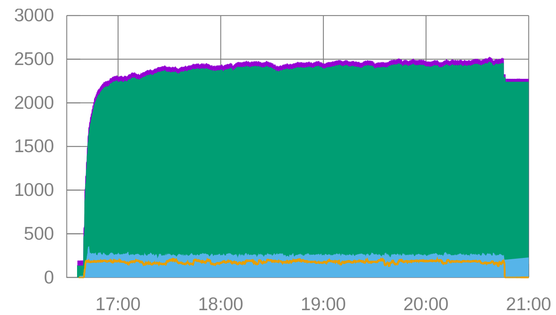


Abbildung 3: *jetty* 11 im Docker-Container

Damit ist zwar die Hypthose des Implementierungsproblems im *tomcat* nicht widerlegt, aber da beide Servlet-Engines das gleiche Phänomen zeigen, könnte hier auch ein fundamentaleres Problem zu Tage treten: es könnte ein generelles Problem in der JVM-Implementierung bzw. in den Standardbibliotheken sein.

Prüfung der Hypothese mit *wildfly*

Als weitere Variante wird die gleiche Webanwendung im *wildfly* deployt. Interessanterweise zeigt sich hier das Verhalten der beiden anderen Servlet-Engines nicht. Wie in Abbildung 4 zu sehen, steigt der Gesamtverbrauch auf knapp unter 800 MB (und bleibt auch nach zwölf Stunden dort).

Zusätzlich ist zu beobachten:

- hier beträgt die Anzahl der abgearbeiteten Nachrichten pro Sekunde etwas unter 20.000
- der tatsächlich gebrauchte Heap beträgt im Mittel noch einmal deutlich weniger als bei *jetty*

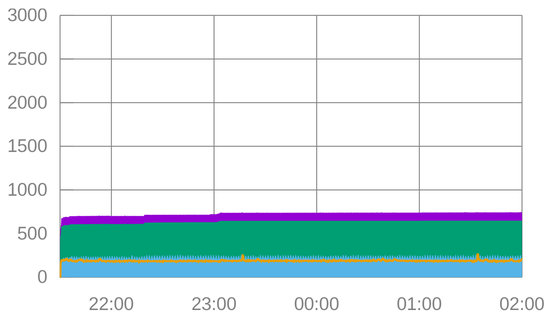


Abbildung 4: *wildfly preview* 26 im Docker-Container

Analyse und zweite Hypothese

Die beiden Servlet-Engines *jetty* und *tomcat* nutzen in der jeweiligen Standardkonfiguration *per-message-deflate* bei Websockets. Bei *wildfly* ist das nicht der Fall.

Mindestens seit Java SE 8 gibt es offensichtlich häufiger Schwierigkeiten mit Off-Heap Memory Leaks bei Anwendungen, die die Kompressionsalgorithmen aus dem Paket *java.util* nutzen. Augenscheinlich ist die Implementierung recht schwierig, korrekt anzuwenden, so dass die Vermutung naheliegt, dass die Servlet-Engines zur Implementierung des *per-message-deflate*-Features diese Klassen nutzen und es daher zum übermäßigen Speicherverbrauch kommt.

Um die Hypothese zu prüfen, wird das Feature im *tomcat* deaktiviert. Das Ergebnis ist in Abbildung 1 zu sehen: Über zwölf Stunden hinweg bleibt die Testanwendung im vertretbaren Rahmen unter 1 GB. Besonders interessant ist, dass die Anzahl der abgearbeiteten Requests pro Sekunde sich mit knapp unter 40.000 fast verdoppelt.

Um die Hypothese abschließend zu prüfen, könnte noch versucht werden, bei *jetty* das Feature zu deaktivieren und es beim *wildfly* zu aktivieren. Beides ist schwierig und mit erheblichem Aufwand verbunden.

Technisches Zwischenfazit

Für eine produktive Anwendung, in der mit Websockets gearbeitet wird und die über Tage - vielleicht sogar Wochen stabil laufen soll, sind *wildfly* in der Standardkonfiguration und *tomcat* mit deaktiviertem *per-message-deflate* vermutlich gleichermaßen geeignet. *tomcat* stellt sich als leistungsfähiger in Sachen *Messages per Second* heraus, *wildfly* scheint in Bezug auf Heap-Auslastung deutlich effizienter. Hier wären weitere Experimente durchzuführen, um herauszufinden, wie sich die beiden Engines in realeren Anwendungen mit Sessions und mehr fachlichen Objekten verhalten: es mag sein, dass der *tomcat* auf Dauer mehr und mehr Resourcen auf den Garbage-Collector verwendet und der Heap deutlich vergrößert werden müsste.

In allen Varianten steigt der von *docker stats* angezeigte Speicher sehr langsam aber kontinuierlich an. Keiner der Prozesse im Docker-Container rechtfertigt diese Beobachtung, die beim *tomcat* am deutlichsten auftritt. Zudem - auch wenn es nicht mehr so rapide ist - steigt der Speicher des Java-Prozesses selbst noch immer stetig an. Ein gelegentliches *jcmd $(pidof java) System.trim_native_heap* scheint da etwas Abhilfe zu schaffen.

In jedem Fall gehört diese Art des Systemmonitorings auch in produktiven Systemen zum soliden Handwerk: die üblichen Werkzeuge (*JConsole*, *visualvm* etc.) bieten gewiss hübschere Oberflächen, bieten aber im Zweifelsfall genau die Informationen nicht, die man braucht. Ein einfaches Beispiel, um den gebrauchten Speicher eines Prozesses sekündlich zu loggen:

```
$ pidstat -p $(pidof java) -r -h 1 \
  | ts '%F %T' | tee pidstat.log
```

Listing 1: Prozessmonitoring der Servlet Engines

Einordnung in Qualitätssicherung

Im Rahmen des Moduls Grundlagen des Qualitätsmanagements haben wir den Schwerpunkt auf das Zusammenspiel von Maßnahmen konstruktiver und analytischer Qualitätssicherung gelegt.

In den Diskussionen und der Literatur zu Maßnahmen und Verfahren der analytischen Qualitätssicherung wird sehr stark auf die funktionalen Eigenschaften von Software fokusiert. Hier sind etablierte Verfahren bekannt, die einen systematischen Zugang zur Qualitätssicherung ermöglichen (siehe z.B. [2]). Solche Verfahren wurden in der Veranstaltung vorgestellt und diskutiert, wie diese Erkenntnisse in der Programmierung der umzusetzenden Funktionalität genutzt werden können, um typische Fehler zu vermeiden.

Die Qualitätssicherung nicht-funktionaler Eigenschaften von Softwaresystemen wird oft als *Sonderaufgabe* abgeschoben und geht nicht wirklich in Alltagshandeln der Projekte über, insbesondere in agilen Projekten scheinen Tests von nicht-funktionalen Eigenschaften, wie beispielsweise Last- und Performance oder effizienter Umgang mit Ressourcen kaum eine Rolle zu spielen. Dies spiegeln auch die Ergebnisse der Umfrage *Softwaretest in Praxis und Forschung* aus den Jahren 2011, 2016 und 2020 wider [3]. Im Vergleich über die Jahre ist erkennbar, dass die Bedeutung von nicht-funktionalen Tests seit 2011 sogar abnimmt (siehe Abb. 5).

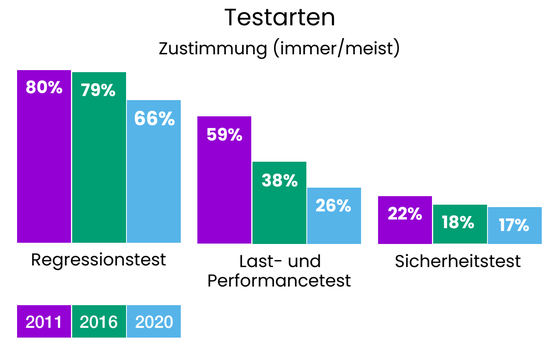


Abbildung 5: Bedeutung von nicht-funktionalen Tests in agilen Projekten

In Diskussionen mit Projektmitarbeiter:innen aus Unternehmen wird die Lösung von Last- und Performanceproblemen in der horizontalen Skalierung oft mit Parallelisierung und Hinzufügen von weiteren Ressourcen gesehen, ohne den Blick auf die entstehende zusätzliche Komplexität zu nehmen.

Für eine solche Untersuchung ist eine analytische Betrachtung des Gesamtsystems - Applikation und Laufzeitumgebung - notwendig. Dies scheint jedoch in den Projekten, die den Blick haben auf der Entwicklung neuer Features ihrer (Teil-)Applikation, kaum Raum zu finden. Leider liegen solche Fragestellungen oft auch nicht im Bereich von zentralen Qualitätssicherungsteams, deren Aufgabe ist, eine Laufzeitumgebung zur Verfügung zu stellen und weiterzuentwickeln.

Die Umsetzung nicht-funktionaler Eigenschaften eines Softwaresystems ist eine wesentliche Aufgabe in der Softwareentwicklung und gehört zu den Aspekten der konstruktiven Maßnahmen der Qualitätssicherung. Werden nicht-funktionale Qualitätseigenschaften eines Systems nicht bereits in der Entwicklung und Umsetzung von Softwarearchitekturen berücksichtigt, wird es schwierig diese Eigenschaften zu erreichen.

Dabei spielen gerade die Erkenntnisse aus analytischen Maßnahmen der Qualitätssicherung nicht-funktionaler Eigenschaften eine wesentliche Rolle und geben wertvolle Hinweise für die Konstruktion der Softwaresysteme und sollten in das Alltagshandeln von Entwickler:innen verankert werden. Aus diesem Grund haben wir in der Veranstaltung neben den Verfahren der analytischen Qualitätssicherung funktionaler Qualitätseigenschaften auch die Analyse und das Monitoring von Gesamtsystemen thematisiert und Umsetzungsbeispiele gemeinsam erarbeitet.

Das beschriebene Problem des *Off-Heap Memory Leaks in Servlet-Containern* ist ein konkretes Beispiel, um ein Gesamtsystem zu analysieren und die Ursachen der Probleme zu lokalisieren, die oft nicht unbedingt in der Umsetzung einer Applikation liegen, sondern im Zusammenspiel mit der Laufzeitumgebung. Um dieses Problem zu lösen wird häufig die Strategie verwendet eine containerisierte Laufzeitumgebung mit zusätzlichen Ressourcen zu versehen, in diesem Fall zusätzlicher Speicher, und die Umgebung regelmäßig neu zu starten. Unser Ansinnen ist, das dahinterstehende Problem zu verstehen, zu lokalisieren und mit weiteren Lösungsmöglichkeiten zu vergleichen. Ziel ist den Studierenden Handlungsoptionen sichtbar zu machen.

Gerade im Kontext der Betrachtung einer ökologischen Nachhaltigkeit in der Entwicklung von Softwaresystemen [1] wird die Analyse des Gesamtsystems ein wichtiger Bestandteil in der Beurteilung dieses Qualitätskriteriums sein, um die Hinweise für eine Einordnung zu nutzen aber auch um in der Weiterentwicklung des Gesamtsystems die Ergebnisse einfließen zu lassen. Ein Aufsetzen eines entsprechenden Monitorings unterstützt, dass Veränderungen, die durch Weiterentwicklungen oder Migrationen in eine neue Laufzeitumgebung entstehen, sofort erkannt werden können. Damit begleiten solche Maßnahmen intensiv den Softwareentwicklungsprozess und unterstützen die kontinuierliche Qualitätsverbesserung. Insbesondere im Kontext von *DevOps* ensteht so eine enge Verbindung zwischen der (Weiter-)Entwicklung von Software und deren Betrieb.

Literatur

[1] J. Gerstlacher, I. Groher und R. Plösch. “Green und Sustainable Software im Kontext von Software Qualitätsmodellen”. In: *HMD Praxis der Wirtschaft* 59.4 (2022), S. 1149–1164.

[2] A. Spillner und T. Linz. *Basiswissen Softwaretest, 6. Auflage*. Heidelberg: dpunkt-Verlag, 2020. ISBN: 3864905834.

[3] M. Winter, K. Vosseberg und F. Simon. *Technischer Report: Umfrage 2020 - Softwaretest in Praxis und Forschung*. de. Dez. 2021. URL: `https://www.softwaretest-umfrage.de/pdf/Technischer-Report-Umfrage-2020-V11Linked.pdf` (besucht am 03. 06. 2022).