NEUE KOLUMNE: EclipseHints – Tipps und Tricks für Eclipse › 90

2.16

eclipse MAGAZIN

www.eclipse-magazin.de

Test different.

RCPTT Oberflächentests mit dem Eclipse RCP Testing Tool

JUnit 5 Was kann das neue JUnit Lambda?

SWTBot Funktionales Testen in Eclipse

Geht das? Und wenn ja, wie?

Verteilte Systeme automatisiert testen

DevOps ist heutzutage in aller Munde, Continuous Integration und Continuous Delivery haben inzwischen den Charakter einer Selbstverständlichkeit. Doch wie stellt man sicher, dass die kontinuierlich entwickelte und gelieferte Software auch den Anforderungen genügt, insbesondere, wenn das Gesamtsystem komplex und zusätzlich noch verteilt ist? Testautomatisierung ist eine Möglichkeit, aber wie realisiert man diese konkret?

von Dr. Dehla Sokenou

Bei der Entwicklung verteilter Systeme stellt sich oft die Frage nach effizienten Testverfahren. Angemessene Teststrategien sollten nicht nur das Verhalten der Einzelkomponenten des Gesamtsystems berücksichtigen, sondern auch das Zusammenspiel aller Komponenten im Integrationstest überprüfen. Die Herausforderungen umfassen einerseits das automatische Aufsetzen der Testumgebung, andererseits die Steuerung der Tests gegen die verschiedenen Einzelkomponenten. Will man all dies auch noch möglichst automatisiert unterstützen, ergeben sich weitere Probleme, für die Lösungen notwendig sind.

Automatisierung im Test ist in der agilen Entwicklung unabdingbar. Software muss heutzutage oft kontinuierlich integriert und in der Regel auch in regelmäßigen Abständen an den Kunden ausgeliefert werden. Kurze Releasezyklen werden immer mehr zur Regel. Ohne eine sehr gute Testabdeckung durch automatisierte Tests ist Softwarequalität nur schwer bis unmöglich zu realisieren. Unbezahlbar wertvoll werden automatisierte Tests, wenn Software einem größeren Refactoring unterzogen wird, wie es aufgrund neuer Anforderungen immer einmal auftreten kann. Dies gilt insbesondere bei Software mit Produktcharakter oder bei Frameworks, die einen noch längeren Lebenszyklus haben können als Software, die in einem Projekt für einen Kunden entstanden ist. Hier ist mithilfe automatisierter Tests leicht zu ermessen, ob ein Refactoring zu unerwünschten Nebeneffekten geführt hat.

Neben der effizienten Ausführung der Tests, zum Beispiel im nächtlichen Build durch einen CI-Server, gewährleistet der automatisierte Test eine Wiederholbarkeit der Testergebnisse, die beim manuellen Test nur schwer erreicht werden kann. Es muss nur einmal ein anderer Tester die Tests durchführen, und schon können die Ergebnisse erheblich abweichen – aus eigener Erfahrung kann dies selbst dann der Fall sein, wenn eine detaillierte Testfallspezifikation vorliegt. Wenn die Tests zudem aus der Entwicklungsumgebung heraus automatisiert ausführbar sind, hat es auch ein Entwickler sehr viel leichter, im Fehlerfall die gleiche oder mindestens eine ähnliche Konstellation wie der Tester herzustellen und das Problem zu debuggen.

Im Folgenden sollen anhand eines Praxisbeispiels Lösungsstrategien für den automatisierten Test eines verteilten Systems aufgezeigt werden.

Ein Praxisbeispiel

Eines der Produkte der GEBIT Solutions GmbH ist ein weltweit verteilt eingesetztes System zur Erstellung und Verteilung von Anwendungskonfigurationen (**Abb. 1**). Es besteht im Kern aus einer Serverkompo-

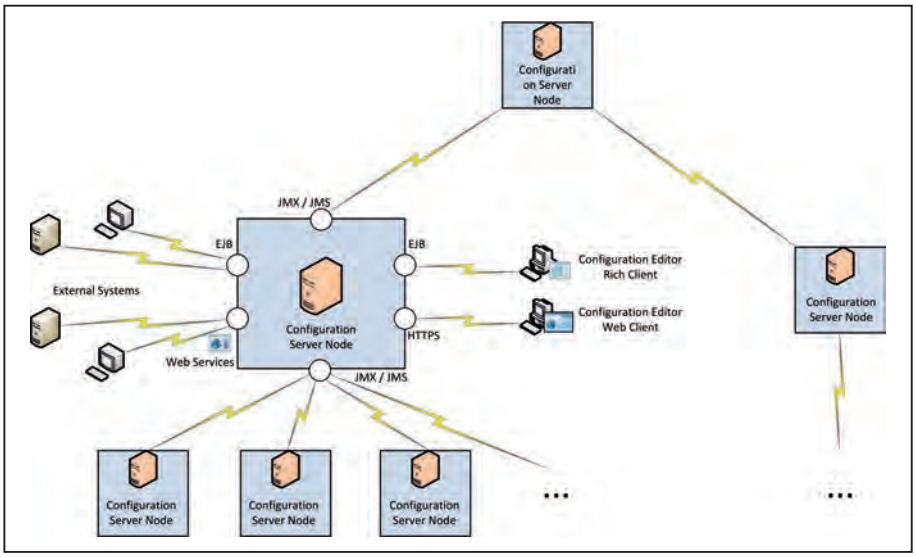

Abb. 1: Komponenten des Konfigurationssystems

nente mit diversen Schnittstellen, einem Webclient und einem Rich-Client. Ein Anwendungsfall ist zum Beispiel die Übertragung wichtiger Betriebsparameter für Kassen- oder Warenwirtschaftssysteme, die in verschiedenen Ländern und Regionen zu unterschiedlichen Zeiten in unterschiedlichen Ausprägungen definiert sein sollen.

Aus Gründen der Betriebssicherheit und Lastverteilung ist üblicherweise die Serverkomponente an verschiedenen Standorten im Einsatz. Die einzelnen Server kommunizieren miteinander, einerseits um Betriebsdaten auszutauschen, andererseits um Konfigurationsdaten weltweit zu replizieren und somit lokal zur Verfügung zu stellen. Die einzelnen Serverinstanzen bilden dabei eine Baumstruktur mit einem Top-Level-Knoten und Kindknoten über mehrere Ebenen; dabei kann ein Netzwerk leicht 10 000 und mehr Knoten umfassen. Für die Kommunikation der Knoten untereinander werden aktuell JMS- bzw. JMX-Schnittstellen zur Verfügung gestellt, zukünftig alternativ auch weitere, beispielsweise EJB-Schnittstellen. Die Kommunikation erfolgt dabei primär zwischen Eltern und den direkten Kindern, für einzelne Funktionen aber auch zwischen Eltern und indirekten Kindern.

Neben den eigenen Clients, die zur Erstellung und Änderung der Konfigurationen sowie zur Überwachung des Servernetzwerks dienen und dazu spezielle eigene Schnittstellen zur Kommunikation nutzen, gibt es auch eine Reihe bekannter sowie auch eine Reihe unbekannter externer Systeme. Diese nutzen zur Kommunikation entweder die öffentlichen EJB- oder Web-Service-Schnittstellen. Dabei umfassen die bekannten externen Systeme sowohl projektbezogene Eigenentwicklungen der GEBIT Solutions GmbH als auch Systeme von Drittherstellern.

Die Herausforderungen beim Test, insbesondere bei der Testautomatisierung eines solch komplexen Systems, umfassen:

- die Kommunikation der einzelnen Komponenten des Systems untereinander – sowohl Server-zu-Server-Kommunikation als auch Client-zu-Server-Kommunikation
- die Simulation von Clients oder alternativ die Einbeziehung bekannter Clients in den Test
- die Berücksichtigung von Laufzeitdifferenzen, da nicht immer davon auszugehen ist, dass Daten zeitnah in den zu testenden Komponenten verfügbar sind
- die effiziente Verteilung und das Setup der Einzelkomponenten

Der Einsatz des Systems bei verschiedenen Kunden unter unterschiedlichen Voraussetzungen erschwert zusätzlich den Test, da durch unterschiedliche kundenspezifische Betriebskonfigurationen des Systems, diverse Datenbanksysteme und kundenspezifische Erweiterungen eine Vielzahl sehr unterschiedlicher Konstellationen entstehen, die in der Regel nicht alle getestet werden können. Hier ist es die Aufgabe des Testmanagements, einen angemessenen Ausgleich zwischen zu wenig und zu viel Testen zu finden und einen ausreichenden Test zu definieren. Gleichzeitig muss der produktspezifische Test vom kundenspezifischen in klarer Weise abgegrenzt werden.

Automatisierung erfordert Testwerkzeuge

Während man manuelle Tests oft auch ohne Testwerkzeuge ausführen kann, erfordert ein automatisierter Test immer den Einsatz eines oder mehrerer Testwerkzeuge. Je nach Einsatzkontext kann das eine oder andere Testwerkzeug geeigneter für den geplanten Zweck sein. Allerdings ist für komplexe und verteilte Systeme mit den unterschiedlichsten Schnittstellen in der Regel ein flexibles Testwerkzeug die beste Wahl (siehe auch Anforderung an das Testwerkzeug).

Aus Entwicklersicht ist das Thema Testen oft eines, das eher stiefmütterlich behandelt wird. Denkt man an Testen und Entwickler, so fällt einem oft das Stichwort Unit Test als einziges ein. Diese werden auch im Kontext des Tests eines verteilten Systems nicht überflüssig. So setzen wir in der Einwicklung des angeführten Produkts aktuell Unit Tests mit JUnit ein, um die Schnittstellen einzeln zu testen, sowie für den Test interner Funktionen und Module.

Allerdings liegt der Fokus beim Test verteilter Systeme auf den anderen Testphasen wie dem Integrations- oder Systemtest. Dazu kommen nicht funktionale Tests wie Last- und Performancetests. Auch hier ist die Rolle des Entwicklers gefragt. Tests für diese Phasen werden entweder ebenfalls von Entwicklern zur Verfügung gestellt oder von speziellen Testern entworfen und anschließend von Entwicklern oder bei Bereitstellung einer entsprechenden

Umgebung von den Testern implementiert. Meist reicht es nicht aus, dem Tester nur das Testwerkzeug an sich zur Verfügung zu stellen. In den meisten Fällen muss zumindest die Infrastruktur für die Tests von Entwicklerseite bereitgestellt werden – sei es nur die Bindung des Testwerkzeugs an die zu testenden Systeme und natürlich die Bereitstellung überhaupt testbarer Software. Die Anforderungen an die einzusetzenden Werkzeuge umfassen also:

- Unterstützung für Mehrsystemtests
- Ansprechen unterschiedlicher Schnittstellen wie Web-Service-APIs, Benutzerinterfaces etc.
- Monitoring der zu testenden Anwendung
- lokale Ausführbarkeit durch Entwickler, zum Beispiel durch gute Integration in eine Entwicklungsumgebung wie Eclipse
- Unterstützung im Build-Server, beispielsweise durch ein entsprechendes Plug-in für Jenkins

In unseren Projekten verwenden wir neben dem bereits erwähnten JUnit üblicherweise das Open-Source-Testframework Integrity [1] für den Integrations- und Systemtest, das speziell für komplexe Testszenarien konzipiert wurde und daher alle vorstehenden Anforderungen erfüllt. So ermöglicht es Integrity im Gegensatz zum weitverbreiteten Integrationstestframework FitNesse [2], verschiedene Teilsysteme transparent innerhalb eines Tests anzusprechen [3] – in unserem Fall sind dies die einzelnen Services, die UI-Schnittstellen und die externen Clients. Zudem ist Integrity in Eclipse integriert, sodass Testfälle auch lokal ausführbar sind.

Nach einem initialem Aufwand, Integrity für das Projekt durch die Implementierung so genannten Fixture-Codes anzupassen, besteht der laufende Aufwand lediglich in der Erstellung neuer und der Wartung existierender Testfälle.

Testbarkeit

Um Software zu testen, muss diese zunächst einmal testbar sein. Auch wenn dies auf den ersten Blick logisch erscheint, wird Software ja nicht primär für dieses Ziel entwickelt, sodass spezielle Anpassungen zur Unterstützung der Testbarkeit die Regel sind. Für die Automatisierung von Testfällen müssen häufig Erweiterungen entweder an der zu testenden Software oder alternativ an den Testwerkzeugen vorgenommen werden. Oft lässt sich jedoch bereits vorhandene Funktionalität in einer für den Test angepassten Version nutzen. Zur Testbarkeit gehört einerseits die Steuerbarkeit der Eingaben, meist durch die entsprechenden Schnittstellen gegeben, andererseits die Beobachtbarkeit der Ausgaben.

In oben skizziertem System werden etwa nur für den Test spezielle Services zusätzlich auf dem Application Server deployt, die einerseits dazu dienen, das System in einen bestimmten Zustand zu setzen, gleichzeitig aber auch bestimmte Ereignisse überwachen und auf Anfrage zurückgeben können, beispielsweise den intern ohnehin verfügbaren Status einer Datenreplikation.

Gibt es bereits eine in das System integrierte Monitoring-Lösung, so kann sich der Test an dieser registrieren, um bestimmte Events abzufangen. Als Beispiel sei hier das Problem des Wartens auf das Ende einer Operation angeführt, das im Test häufig auftritt, wenn Daten zwischen zwei Systemen ausgetauscht werden. So kommunizieren die Knoten im beschriebenen Konfigurationssystem miteinander, zum Beispiel um Daten zu replizieren. Im Test soll die Replikation überprüft werden, etwa daraufhin, ob alle benötigten Daten auf den Knoten verfügbar sind. Die erste Möglichkeit ist, explizit Wartezeiten in den Test einzubauen. Die andere ist eben genau das Abfangen bestimmter Events, beispielsweise ob die Datenreplikation erfolgreich beendet wurde. Achtung: In diesem Fall sollte ein Timeout dafür sorgen, dass der Test mit einem Fehler abbricht, wenn das Event nicht eintritt, anstatt den Test im Negativfall unendlich warten zu lassen.

Das Deployment der Software sowie die Konfiguration des Systems als Ausgangszustand für den Test stellt oft eine zusätzliche Herausforderung dar. So muss für das System aus unserem Beispiel jeder Knoten im Netzwerk aufgesetzt werden. Es muss die eigene Identität des Knotens und die Einordnung in die Baumstruktur (Wer ist mein Elternknoten? Wer sind meine Kindknoten? Welche Daten dürfen auf diesem Knoten bearbeitet werden?) festgelegt werden.

Hier nutzt der Test bereits vorhandene Funktionalität, entweder durch Simulation des manuellen Aufsetzens eines Knotens durch den Test selbst oder durch eine als Service verfügbare Funktion zur automatischen Initialisierung eines Knotens oder eines ganzen Netzwerks. Da die Autoinitialisierung ohnehin bereits als kundenspezifisch austauschbarer Service entwickelt wurde, lag es nahe, für Testzwecke einen angepassten Autoinitialisierungsservice zu implementieren und nur das Service-Mapping im Test zu ändern. Für die Autoinitialisierung im Test werden XML-Dateien verwendet, die entweder alle Knoten mit konkreten Werten oder parametrisiert beschreiben, wobei Letzteres genutzt wird, um Netzwerke zu realisieren, deren Größe und konkreter Aufbau im Vorhinein unbekannt ist. Gerade für die Last- und Performancetests ist die Autoinitialisierung eine deutliche Erleichterung, sodass sich die Bereitstellung einer solchen Funktion auch lohnt, wenn sie im zu testenden System ursprünglich nicht vorgesehen war.

Stufe für Stufe

Bei der Planung, Erstellung und Durchführung der Tests haben wir einen mehrstufigen Aufbau gewählt, der sich bewährt hat. Die unterschiedlichen Teststufen bilden den jeweiligen Reifegrad des Systems ab, vom Test der einzelnen Einheiten bis hin zum Test des Gesamtsystems. Fehler in den einzelnen Einheiten werden bereits im Unit Test erkannt und können so frühzeitig behoben werden, bevor es in die nächste Teststufe geht. Überspringt man die ersten Teststufen und geht gleich in den Integrations- oder Systemtest, so herrscht schnell Frustration. Es ist bei Inte-

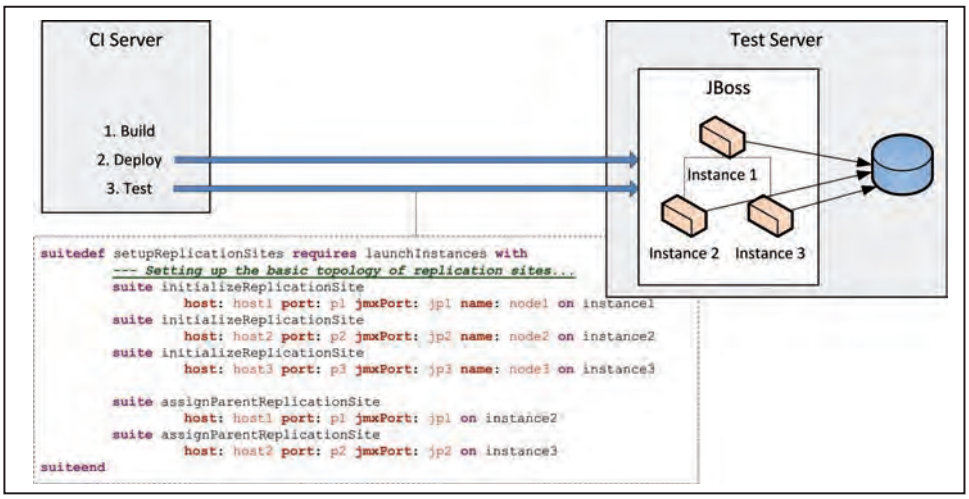

Abb. 2: Deployment und Test auf dem CI-Server

grations- oder Systemtestszenarien nämlich oftmals sehr viel schwieriger, die Ursache eines fehlgeschlagenen Tests zu ermitteln, wenn diese lokal begrenzt in einem Modul liegt. Zusätzlich ist es möglich, dass vorhandene Fehler in den höheren Teststufen gar nicht entdeckt werden, weil sie durch entsprechende Testkonstellationen maskiert werden oder gar nicht erreichbar sind.

Dabei konzentrieren wir uns erst auf den funktionalen Test, bevor nachfolgend die Testszenarien für den Last- und Performancetest skizziert und bewertet werden.

Funktional ...

Neue Funktionalität, etwa neue Schnittstellen, werden zunächst mithilfe von JUnit-Tests auf ihre grundsätzliche Funktionalität überprüft. Ein weiterer Baustein ist der Test der Benutzerschnittstellen, der mit Integrity erfolgt. Für den Unit Test ebenso wie für den Test der Benutzerschnittstellen ist noch kein Server erforderlich: Die benötigten Services laufen in einer Art lokalen Modus, der zwar die vollständige fachliche Funktionalität zur Verfügung stellt, jedoch nicht in einem Container auf einem Applikationsserver läuft. Dieses Verfahren ist ein Feature des von uns genutzten Anwendungsframeworks. Deshalb laufen die Tests für beide Schnittstellen ohne weitere Anpassungen auch in jeder Entwicklungsumgebung und auf dem Build-Server – für die Entwickler werden jeweils passende Run-Konfigurationen zur Verfügung gestellt.

Etwas mehr Aufwand muss in den Test des Gesamtsystems mit unterschiedlichen Knoten und Clients investiert werden, sowohl was die Planung als auch die Automatisierung der Tests betrifft, da hier die Themen Verteilung und Kommunikation im Vordergrund stehen. Die Aufgaben umfassen die Bereitstellung der Testumgebung, den Build und das Deployment von Client und Server und die Ausführung der Testfälle. Dies soll sowohl lokal auf einem Entwicklerrechner möglich sein als auch auf dem Continuous-Integration-Server.

Für diese Teststufe ist das Szenario so aufgebaut, dass nur ein kleines Netzwerk aus bis zu fünf Knoten aufgebaut wird, sowie einige Clients vom Test gestartet werden (**Abb. 2**). Alle Serverinstanzen laufen auf einem vom CI-Server getrennten Server, die einzelnen Instanzen werden auf demselben JBoss-Server mit unterschiedlichen Port-Offsets deployt und verwenden für die Datenhaltung unterschiedliche Schemata auf derselben Datenbank. Die Beschränkung auf ein kleines Netzwerk ist bewusst vorgenommen worden, da hier nur die reine Funktionalität getestet werden soll. Aspekte wie das Verhalten unter Last und Stabilität werden durch einen separaten Test überprüft. Für den rein funktionalen Test sollte zwar die generelle Komplexität des Systems abgebildet werden, aber das Motto „keep it simple“ gelten.

Der Build von Clients und Server-EAR, das Patchen für Port-Offset und Datenbankschema und die Verteilung auf die JBoss-Instanzen übernimmt der Build-Prozess, der anschließend die Tests mithilfe von Integrity ausführt. Dabei kann der Applikationsserver selbst entweder bereits vorinstalliert oder für jeden Build direkt auf dem Testserver aufgesetzt werden, wobei Letzteres den Vorteil hat, dass auch Änderungen am Applikationsserver immer direkt im Test verfügbar sind. In der Testausführung werden sowohl die Services als auch die Clients innerhalb des gleichen Tests angesprochen, da dies mit Integrity problemlos möglich ist.

Ein Entwickler kann das Build-Skript oder alternativ nur die Tests auch lokal aus der Entwicklungsumgebung heraus aufrufen (**Abb. 3**), wobei im letzteren Fall das Bauen und Deployen des EARs separat erfolgen muss.

... und nicht funktional

Da eines der zentralen Themen in unserem System die Replikation von (Massen-)Daten zwischen den einzelnen Instanzen ist, sind neben der reinen Funktionalität auch wichtige nicht funktionale Anforderungen zu testen. Dabei lassen sich bestimmte nicht funktionale Aspekte relativ einfach automatisiert testen, wie beispielsweise Lastverhalten und Performance, andere dagegen eher schlecht bis gar nicht, zum Beispiel die Bedienbarkeit.

Wir konzentrieren uns hier auf die gut automatisierbaren Anforderungen. In unserem Beispielsystem setzen wir für die Kommunikation zwischen den Instanzen unter anderem auf den JMS-Provider HornetQ. Schnell haben wir bemerkt, dass die Standardkonfiguration von HornetQ nicht an unsere Szenarien angepasst ist und hier eigene Anpassungen vorgenommen werden müssen, damit die Replikation via HornetQ stabil und zuverlässig läuft. Um den Erfolg unserer Maßnahmen zu testen, ist es notwendig, größere Netzwerke aufzusetzen und Messungen durchzuführen, wie sich das System mit verschiedenen Konfigurationen verhält.

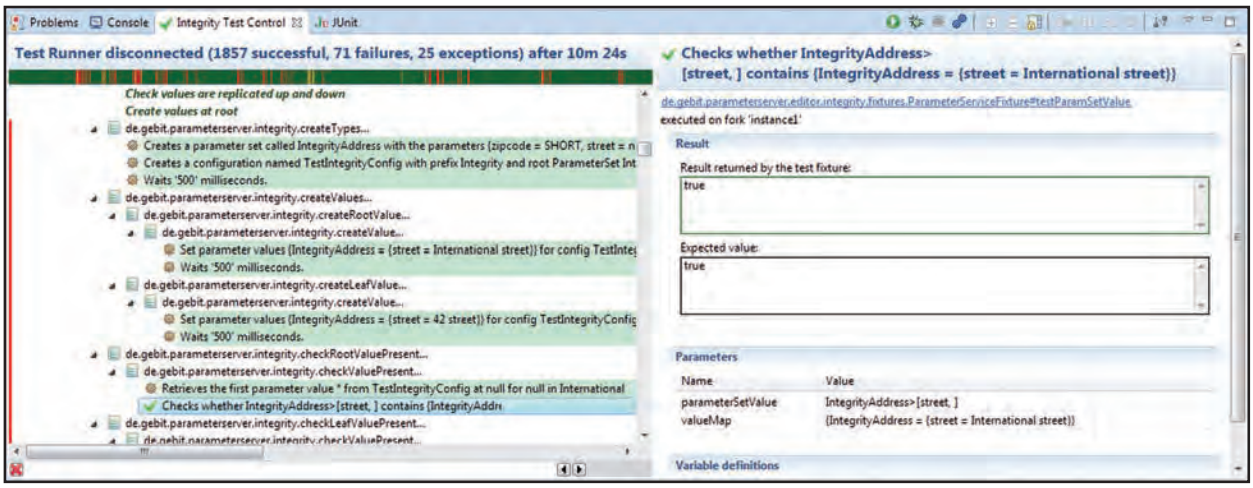

Abb. 3: Ergebnis der Integrity-Tests in Eclipse – im Beispiel wurden Fehler in der Ausführung der Tests durch extrem kurze Intervalle für die Auswertung der Datenreplikation provoziert

Die im Build-Prozess verfolgte Strategie, alle Knoten auf einem Server zu betreiben, ist hier nicht praktikabel. Erstens ist ein Portmapping nicht beliebig oft realisierbar, da man gegebenenfalls in Bereiche kommt, wo Ports bereits anderweitig vergeben sind. Zweitens ist ein Lasttest nur wenig aussagekräftig, wenn nur ein einziger Server alle Anfragen mit seinen eigenen Ressourcen verwalten muss. Als Letztes lassen sich Netzwerkausfälle und -verzögerungen leicht simulieren, wenn die Instanzen auf unterschiedlichen Servern laufen.

Der erste Ansatz ist die Bereitstellung einer Template-VM, auf der ein vollständiger Knoten installiert ist. Diese Template-VM kann nun so oft geklont werden, wie die Ressourcen der darunterliegenden VM-Hosts oder gegebenenfalls gemietete Ressourcen es zulassen. Alle Knoten setzen sich anschließend mithilfe der Autoinitialisierung anhand eines Namensschemas selbst auf. Nun kann der Test, auch automatisiert, gegen diese Instanzen erfolgen. Für uns interessant sind bei diesen Tests insbesondere die Simulation vieler Offlineknoten und die Simulation schlechter Netzwerkqualität, woraus viele Optimierungen an der HornetQ-Konfiguration resultierten.

Inzwischen verfolgen wir den Weg, Docker einzusetzen. Da Docker-Container sehr viel leichtgewichtiger sind als komplette VMs, lassen sich auf der gleichen Hardware deutlich mehr Instanzen betreiben, als wenn komplette VMs genutzt werden – in unserem Fall ohne weitere Optimierungsmaßnahmen etwa um den Faktor 5. Um noch mehr Instanzen parallel zu betreiben, haben wir für die Blattknoten im System eine Stub-Variante entwickelt, die als reine Datensenke fungiert und ankommende Daten leichtgewichtiger verarbeitet, um auf diese Weise noch einmal Ressourcen einzusparen. Der Einsatz von Stubs sollte immer gezielt erfolgen und hinterfragt werden, ob er sinnvoll und zielführend ist. In diesem Fall ist der Einsatz von Stubs problemlos möglich, da diese für den Test der Performance der zuliefernden (vollwertigen) Knoten und nicht der empfangenden (Stub-)Knoten genutzt werden.

Fazit

Verteilte Systeme automatisiert testen – geht das? Die Frage lässt sich mit einem klaren „Ja“ beantworten – und es lohnt sich! Nicht nur, dass ein automatisierter Test Sicherheit für Continuous Delivery gibt und Refactorings unterstützt, zusätzlich lassen sich Kennzahlen wie beispielsweise Codeüberdeckung leicht ermitteln.

Wie geht es? Wichtig ist vor allem ein systematisches Vorgehen, also ein stufenweiser Aufbau des Tests. Zu beachten ist bei der Testautomatisierung immer – und im Besonderen für verteilte Systeme –, dass der Initialaufwand, also die Hürde für den Einsatz, hoch ist. Nichtsdestotrotz lohnt sich der Einsatz. Wird der automatisierte Test von Beginn der Entwicklung konsequent eingeplant und realisiert, so wachsen die Testsuiten während der Entwicklung, und es muss am Ende nicht eine große Zahl manueller Testfälle in automatisierte Testfälle überführt werden.

Durch die Möglichkeiten, die das Klonen von VMs und noch empfehlenswerter von Docker-Containern bieten, lassen sich einfach und schnell kleinere und größere verteilte Systeme aufsetzen und durch parametrisierbare Testsuiten ansprechen und testen. Schlussendlich lassen sich die Arbeitsergebnisse aus den Tests wiederum für andere Zwecke einsetzen – die Nutzung von vorbereiteten Docker-Containern ist ja nicht auf den Test beschränkt, sondern kann auch das Deployment einer Anwendung beim Kunden erleichtern, Stichwort: DevOps.

Dr. Dehla Sokenou promovierte 2005 an der TU Berlin über das Thema UML-basiertes Testen. Seit Anfang 2006 ist sie als Senior-Software-Consultant bei GEBIT Solutions beschäftigt. Ihr Tätigkeitsfeld umfasst neben Projektleitung, Konzeption und Entwicklung großer objektorientierter Softwaresysteme mit modellbasierten Methoden auch weiterhin das modellbasierte Testen.

Links & Literatur

[1] http://fitnesse.org

[2] http://www.integrity-tf.org

[3] Schneider, Rene: „Damit die Kasse stimmt – Praxisbericht über das neue Testautomatisierungsframework „Integrity““, in: Java Magazin 9.2014

Kontakt:

GEBIT Solutions GmbH
Koenigsallee 75 b
14193 Berlin

Telefon: 030 / 896 663 - 00
E-Mail: kontakt@gebit.de

Niederlassung Düsseldorf
Hammer Straße 19
40219 Düsseldorf

Niederlassung Stuttgart
Gropiusplatz 10
70563 Stuttgart

www.gebit.de