Wie ändert sich das Monitoring mit der Microservices-Architektur?

Microservices brauchen ein neues Monitoring

Verteilte Systeme zu betreiben und zu überwachen, kann sehr schwierig sein. Es kommen neue Anforderungen auf den Betrieb und die Entwickler zu. Während das Monitoring bei einer monolithischen Anwendung noch recht einfach und übersichtlich ist, gibt es beim Microservices-Ansatz mehrere Systeme, die überwacht werden müssen. Welche Herausforderungen müssen bei einem Umstieg angegangen werden?

AUTOR: VITALI FICHTNER

Bei Microservices-Ansätzen werden kleine Services feingranular nach fachlichen Domänen geschnitten und können somit von verschiedenen Teams unabhängig entwickelt, veröffentlicht und betrieben werden. Am Ende bildet ein Netz von Microservices ein Gesamtsystem. Dieser Ansatz hat sehr viele Vorteile, bringt aber auch neue Herausforderungen beim Thema Monitoring für die Entwickler, den Betrieb und den Fachbereich mit.

Entwickler sind an Anwendungsmetriken interessiert. Sie möchten nachvollziehen können, wie sich die Anwendung im laufenden Betrieb verhält. Wie ist die Performance unter Last? Wie ist die Entwicklung des Speichers und der CPU seit dem letzten Release? Durch das Monitoringsystem möchten die Entwickler über technische Fehler informiert werden, um schnell reagieren zu können, am besten noch bevor es zum Worst-Case-Szenario kommt. Weiterhin werden nicht funktionale Anforderungen wie die Performance der Anwendung beobachtet.

Für den Betrieb ist das Hauptziel, eine hohe Verfügbarkeit der Anwendung zu gewährleisten. Daher müssen die Betriebsmitarbeiter frühzeitig Bescheid wissen, falls ein Teil der Anwendung nicht funktioniert. Falls eine große Belastung der Anwendung zu langsamen Antwortzeiten führt, kann auch dies eine interessante Metrik für den Betrieb sein. Weiterhin beobachten die Betriebsmitarbeiter die gesamte Infrastruktur der Anwendung wie Hardware, Netzwerk oder Betriebssystem. Werden Probleme erkannt, handeln sie selbst oder sie informieren die Entwickler.

Der Fachbereich definiert in der Anforderungserhebung die nicht funktionalen Anforderungen. So erwartet er, dass die Antwortzeit unter einer gewissen Zeitspanne bleibt oder dass die Anwendung 24/7 verfügbar ist. Aus den nicht funktionalen Anforderungen ergeben sich

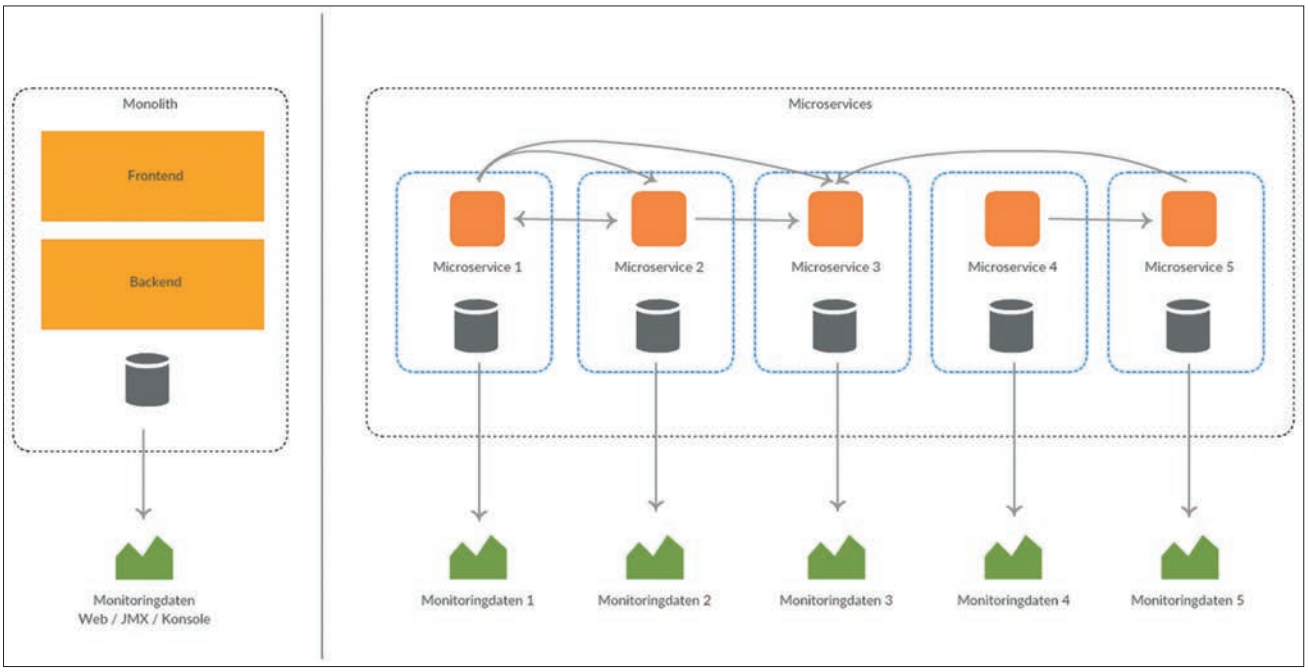

Abb. 1: Eine monolithische Anwendung lässt sich einfacher überwachen als eine Microservices-Architektur

die Schwellwerte für die Metriken, die von Betrieb und Entwicklern beobachtet werden müssen. Ansonsten ist der Fachbereich an keinen technischen Metriken interessiert. Ihn interessieren aber weiterhin fachliche Fragen, die aus der Anwendung generiert werden können, z. B.: „Wie viele Verkäufe wurden vom Produkt X im bestimmten Zeitraum gemacht?“.

Alle Daten werden aus der Anwendung und der Infrastruktur erzeugt. Nicht nur im Zeitalter von Big Data heißt es, so viele Daten wie möglich zu sammeln, auch für die Erhebung von Monitoringdaten gilt diese Prämisse. Da die Daten aber aus unterschiedlichen Quellen kommen und für unterschiedliche Zielgruppen interessant sind, braucht es hier einen standardisierten Prozess.

WIE ÄNDERT SICH DAS MONITORING?

Aus Sicht von Entwicklung und Betrieb ist das Monitoring von monolithischen Anwendungen recht einfach. Es gibt nur ein System, das überwacht werden muss. Das macht die Fehlersuche für die Entwickler übersichtlich: Sie müssen nur an einer Stelle nachsehen. Auch der Betrieb muss die Hardware und die Infrastruktur von nur einer Anwendung überwachen. Manchmal gibt es aus Performancegründen ein Clustering mit mehreren Knoten. Aber auch dann ist das System noch überschaubar.

Bei Microservices hingegen ist die Fachlichkeit auf viele kleine Services aufgeteilt. Jeder Microservice hat seine eigene Mikroarchitektur und lebt in seiner eigenen Infrastruktur. Jeder Microservice erstellt eigene Logs und Metriken und muss zum einen eigenständig und zum anderen im Zusammenspiel mit anderen Microservices beobachtet werden. Dies macht auch die Fehleranalyse schwierig. Denn zunächst muss identifiziert werden, welcher der zahlreichen Microservices die Ursache eines Fehlers ist. **Abbildung 1** zeigt vereinfacht das eigentliche Problem beim Monitoring und Logging von verteilten Systemen auf: Es existieren viele Protokolle mit Logs und Monitoringdaten physisch an unterschiedlichen Stellen. Dazu kommt, dass sich ein fachlicher Prozess über mehrere Services hinwegziehen kann. Dadurch werden Logs und Monitoringdaten zu diesem Prozess an unterschiedlichen Stellen generiert.

Die Lösung ist einfach: Logs und Monitoringdaten müssen zentral abgelegt werden. Die Kommunikation zwischen den Microservices muss protokolliert und konsolidiert werden. All diese Informationen müssen dem Benutzer an einer Stelle zur Verfügung stehen.

RELEVANTE KENNZAHLEN IDENTIFIZIEREN

Bevor man mit der technischen Umsetzung beginnen kann, müssen zunächst die relevanten Kennzahlen ermittelt werden. Grundsätzlich hängen die Anforderungen an die Kennzahlen stark von der Anwendung ab und müssen von den jeweiligen Zielgruppen definiert werden. Tabelle 1 zeigt verschiedene Metriken, die ein Monitoringsystem bei Bedarf bereitstellen sollte. Die Metriken sind nach den Zielgruppen aufgeteilt. Busi-

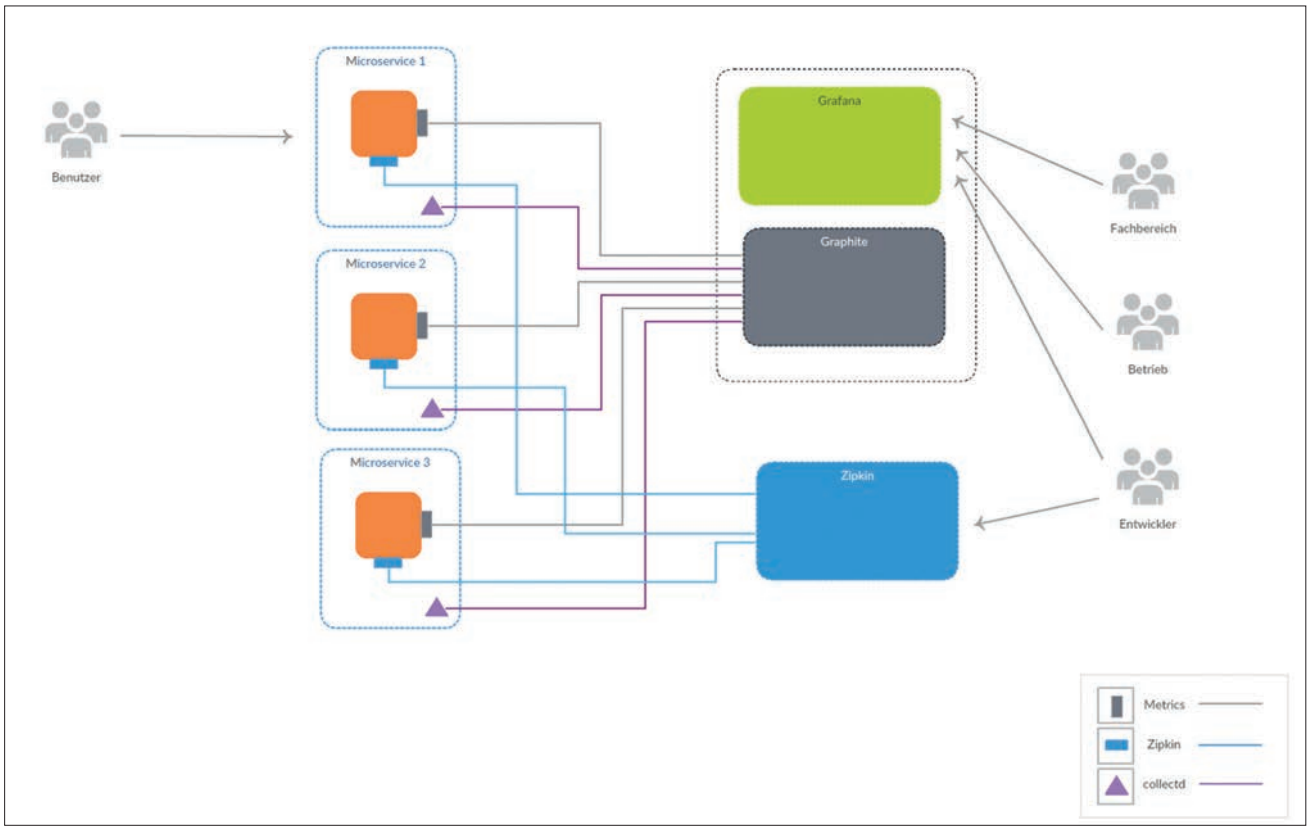

Abb. 2: Beispielhafte Monitoringlandschaft einer Microservices-Architektur

nessmetriken hängen beispielsweise stark von der Fachdomäne der Anwendung ab.

Bei verteilten Systemen muss man die Metriken beim Monitoring aus verschiedenen Perspektiven betrachten können. Zum einen sind Metriken eines Microservice für sich allein interessant, und zum anderen enthält das Gesamtbild konsolidierte Metriken mehrerer Microservices. Fällt ein Microservice aus, kann dies eine Auswirkung auf die direkte Nachbarschaft des Service haben sowie auf übergreifende Prozesse. Dieser Aspekt muss sich beobachten lassen. Dafür sind Metriken wie Health-Checks, Antwortzeiten sowie Tracing von Aufrufen von besonderer Bedeutung. Mit ihnen lässt sich die Kommunikation zwischen den Microservices überwachen. Das ist entscheidend, um eine effektive Fehleranalyse zu betreiben. JVM-Metriken wie Java Heap Space, Garbage Collection, Active Threads und andere sind für die Performance- und Verfügbarkeitsanalyse von Java-Anwendungen ebenfalls äußerst wichtig.

Weiterhin ist es interessant, die erfassten Kennzahlen mit dem Entwicklungsprozess zu verknüpfen und zu beobachten. Falls seit dem letzten Release die Antwortzeiten bei einem bestimmten Microservice gestiegen sind, kann es sich dabei um einen möglichen Flaschenhals handeln, der dringend optimiert werden sollte. Daher ist es wichtig, das Monitoringsystem bereits in der Entwicklung einzusetzen. Auch um beurteilen zu können,

Systemmetriken	Anwendungsmetriken	Businessmetriken
CPU-Auslastung Speicherauslastung Festplattenkapazität JVM-Analyse über JMX Netzwerklatenz	Health-Check Tracing von Aufrufen Antwortzeiten Aktuelle Zugriffe Datenbankzugriffe Cachezugriffe Transaktionsdauer	Anzahl der Log-ins Dauer einer Benutzer-Session Durchlaufzeit eines Auftrags Anzahl verkaufter Güter Gestartete und abgeschlossene Prozesse
Zielgruppen: Betrieb, Entwickler	Zielgruppen: Entwickler, Betrieb	Zielgruppe: Fachbereich

Tabelle 1: Je nach Zielgruppe sollte ein Monitoringsystem unterschiedliche Metriken zur Verfügung stellen

ob die Kennzahlen gut oder schlecht sind, ist es ratsam, die Daten frühzeitig zu sammeln. So können sich Muster von Anfang an abzeichnen.

DIE MONITORINGLANDSCHAFT IST VIELFÄLTIG

Das Erfassen von Monitoringdaten ist eine Querschnittsfunktionalität des gesamten Systems und gehört somit zur Makroarchitektur. Jeder Microservice muss seine Metriken selbst erfassen und an das Monitoringsystem senden. Ein Vorteil der Microservice-Architektur ist, dass sich einzelne Services leicht ersetzen lassen. Daher sollten sie die Aufgabe des Aufbereitens und Präsentierens von Metriken nicht selbst übernehmen, sondern die Daten an einer zentralen Stelle außerhalb der Microservices senden. Bei Ausfall eines Service werden dann zwar keine Daten mehr gesendet, das Monitoringsystem ist aber weiterhin verfügbar.

Für die zentrale Aufbereitung und Sammlung der Daten gibt es verschiedene Tools. In diesem Artikel betrachten wir das Monitoringtool Graphite [1] mit Grafana [2]. Das Tool sammelt mithilfe von collectd [3] und Metrics [4] System-, Anwendungs- und Businessmetriken und stellt sie auf einer Weboberfläche zur Verfügung. Um die Kommunikation zwischen den Microservices zu überwachen, haben wir das Tool Zipkin [5] verwendet. **Abbildung 2** zeigt eine Übersicht der Monitoringlandschaft, bei der auf das Logging bewusst verzichtet wurde.

KONSOLIDIEREN VON KENNZAHLEN MIT GRAPHITE UND GRAFANA

Graphite ist seit Jahren auf die Verarbeitung von Zeitreihen spezialisiert. Es handelt sich um eine Time-Series-Datenbank mit einem effizienten Ansatz zum Speichern von Daten. Es lässt sich z. B. einstellen, ob ein bestimmter Teil der Daten nach einer Zeit automatisch gelöscht werden soll. Das Protokollieren der Metriken kann so eingestellt werden, dass Daten mit einer Genauigkeit von einer Minute geloggt werden. Ein mögliches Szenario ist, dass jede Minute ein Statement geloggt, eine Woche vorgehalten und anschließend gelöscht wird. Zusätzlich können Daten mit einer Genauigkeit von 15 Minuten für einen Monat abgelegt werden und Daten mit einer Genauigkeit von einer Stunde für fünf Jahre. Die individuelle Konfiguration von Genauigkeit und Zeitdauer spart viel Speicherplatz und ermöglicht dennoch, dass Metriken über einen langen Zeitraum protokolliert werden.

Graphite selbst sammelt keine Daten, dafür gibt es mittlerweile viele Plug-ins. Jeder Microservice sendet seine erfassten Daten an Graphite. Graphite legt die Daten unter einem Pfad, getrennt durch Punkte ab. So könnten z. B. die CPU-Metriken für einen bestimmten Microservice abgelegt werden: *graphite.microservice01.cpu*. Im Vorfeld sollte man sich daher genau Gedanken über die Namenshierarchie der Daten machen.

Listing 1: Auszug aus „/etc/collectd/collectd.conf“

```
Interval 10
Timeout 2
ReadThreads 5

// Plug-ins für Systemmetriken
LoadPlugin cpu
LoadPlugin memory
LoadPlugin processes
LoadPlugin disk

// Kommunikation zu Graphite
LoadPlugin write_graphite

<Plugin "write_graphite">
 <Node "endpoint">
  Host "graphite.server.com"
  Port "2003"
  Protocol "tcp"
  LogSendErrors true
  EscapeCharacter "_"
  Prefix "collectd.01."
 </Node>
</Plugin>
```

SYSTEMMETRIKEN MIT COLLECTD ERFASSEN

Für Systemmetriken eignet sich das Tool collectd. Über verschiedene Erweiterungen kann collectd Statistiken über die Maschine und das Betriebssystem einsammeln und sie dem Monitoringsystem bereitstellen. Das Einrichten und Konfigurieren erfolgt relativ einfach, da es sich um einen leichtgewichtigen Daemon auf der Systemebene handelt. Linux-Distributionen wie Debian bieten collectd als Package an. Dann ist die Installation besonders einfach: *apt-get install collectd*.

Durch einen Eintrag in der Konfigurationsdatei wird bestimmt, welche Plug-ins zum Erfassen von Daten geladen werden sollen. Listing 1 enthält einen Auszug aus der Konfigurationsdatei, über den Ausdruck *LoadPlugin <NAME>* lassen sich beliebige Erweiterungen initialisieren. collectd erhebt die Daten lediglich, die Daten müssen noch an Graphite geschickt werden. Hierfür benutzen wir das Plug-in *write_graphite*. Das Plug-in kommuniziert mit dem Daemon Carbon, der dafür zuständig ist, die Daten in Graphite abzulegen. Carbon ist ein Teil der Graphite-Architektur und horcht standardmäßig auf den Port 2003. Den Endpunkt von Carbon konfigurieren wir in der *collectd.conf*-Datei. Des Weiteren wird der Pfad angegeben, in dem Carbon die Daten ablegen soll, und das Zeitintervall, in dem die Daten gesendet werden.

Die meisten Applikationsserver bieten über eine Managementkonsole Zugriff auf JVM-Informationen. In dem Fall ermöglicht collectd über das Plug-in GenericJMX Zugriff auf die MBeans. Dadurch lassen sich Informationen wie Speicherauslastung, Anzahl der geladenen Klassen oder Garbage-Collector-Informationen

an Graphite senden. Nach der Konfiguration wird collectd über den Befehl *collectd -f* gestartet und das Senden der Daten beginnt.

ANWENDUNGS- UND BUSINESSMETRIKEN MIT METRICS GENERIEREN

Anwendungs- und Businessmetriken werden aus der Anwendung heraus generiert. Für Java eignet sich hier das Framework Metrics von Dropwizard, mit dem sich Daten für Gauges, Counter oder Meters erfassen lassen. Metrics enthält verschiedene Reporter, um die erfassten Daten zu veröffentlichen. Neben einem *JMXReporte*r oder *CsvReporter* unterstützt es auch Graphite. Über den *GraphiteReporter* wird die Verbindung zu Graphite konfiguriert (Listing 2). Die Kommunikation erfolgt wie bei collectd über den Carbon Daemon.

Nach der Verbindung zu Graphite erfolgt die eigentliche Implementierung der Datenerhebung. Es kann z. B. aus dem API ein *Counter*-Objekt erzeugt werden, das anschließend mit einem Pfadnamen gefüllt und beim Aufruf von *counter.inc()* inkrementiert wird. Dropwizard Metrics bietet eine intuitive API-Schnittstelle, um verschiedene Metriken zu erheben.

DATEN MIT GRAFANA DARSTELLEN

Graphite bietet zwar von Haus aus eine Weboberfläche zum Rendern der erfassten Daten an, diese ist aber eher rudimentär und bietet keine große Konfigurationsmöglichkeit für das Erstellen von Dashboards. Das Tool Grafana bietet in dieser Hinsicht Abhilfe. Es kann auf Daten von Graphite zugreifen und konzentriert sich hauptsächlich auf eine intuitive Darstellung und Erstellung von individuellen Dashboards. **Abbildung 3** zeigt ein Beispiel eines Grafana Dashboards.

Um Grafana mit Graphite zu verbinden, muss eine neue Datasource in Grafana angelegt werden. Hier wird der Servername mit dem Port eingegeben. Anschließend lassen sich neue Dashboards mit den Daten aus Graphite erstellen. Entwickler, Betrieb sowie der Fachbereich können über Grafana ihre eigenen Dashboards pflegen. Die Daten sind zentral gespeichert, und es gibt nur ein System für alle Beteiligten.

AUFRUFE NACHVERFOLGEN

Bei einem Fehler in einer fachlichen Funktion wie „Bestellung abschicken“ gibt es bei einem Monolithen nur ein Stacktrace, und die Entwickler wissen, wo sie nachsehen müssen. Es wird kein extra Monitoring für die

Listing 2: Beispiel-Snippet für die Registrierung zu Graphite

```
final Graphite graphite = new Graphite(new InetSocketAddress("graphite.server.com", 2003));
final GraphiteReporter reporter = GraphiteReporter.forRegistry(registry)
.prefixedWith("graphite.microservice01.healthcheck")
.convertRatesTo(TimeUnit.SECONDS)
.convertDurationsTo(TimeUnit.MILLISECONDS)
 .filter(MetricFilter.ALL)
  .build(graphite);

reporter.start(1, TimeUnit.MINUTES);
```

Abb. 3: Die mit Grafana erzeugten Dashboards stellen die Daten übersichtlich dar und lassen sich individuell anpassen

Nachverfolgung benötigt: Alles befindet sich in einem System, sodass ein sinnvolles Logging ausreicht. Bei einer Microservices-Architektur kann sich eine solche Funktion über mehrere Services ziehen. Das kann die Fehlersuche anstrengend machen. Bei verteilten Systemen sollte deswegen die Kommunikation zwischen den Services protokolliert und beobachtet werden. Dies betrifft unter anderem das Logging. Bei diesem sollte ein Ansatz gewählt werden, bei dem eine Korrelations-ID durchgereicht wird. So könnte beim ersten Aufruf eines Service ein GUID (Globally Unique Identifier) generiert werden und an weitere Microservices durchgereicht. So funktioniert die Nachverfolgung über verschiedene Logdateien.

Das Sammeln der Kommunikationsdaten ist aber nur die halbe Miete. Man muss auch das große Ganze sehen und verstehen. Durch die richtige Analyse der Kommunikationsdaten wird nicht nur die Abhängigkeit der Services ersichtlich, sondern auch die Laufzeit einzelner Services. Das Tool Zipkin hilft dabei, systemübergreifende Aufrufe nachzuverfolgen. Es sammelt die Kommunikationsdaten und wertet sie auf der Weboberfläche aus. Die Implementierung basiert auf der Google-Dapper-Veröffentlichung [6].

Zipkin bringt eigene Libraries [7] zum Sammeln und Senden von Tracing-Informationen für verschiedene Programmierumgebungen mit. Jeder Microservice muss diese Libraries benutzen, um Tracing-Informationen zu erheben und an Zipkin zu schicken. Um den Start-und-Stop-Aufruf identifizieren zu können, enthalten die Tracing-Informationen folgende Annotationen: *Client Sent*, *Server Received*, *Server Send*, *Client Received*. Die Aufrufe sind oft verschachtelt und bringen eine komplexe Aufrufkette hervor.

Die Bibliothek brave [8] kann für die Frameworks Jersey, RESTeasy, JAX-RS 2, Apache HTTPClient und MySQL die Tracing-Informationen automatisch erheben und an Zipkin senden. Damit muss der Entwickler nur die brave-Bibliothek in jedem Microservice einbinden und konfigurieren. Der Rest erfolgt automatisch. Unter der Haube implementiert brave Client- und Server-Interceptors, die sich an die verschiedenen Schnittstellen der Frameworks hängen, um Logstatements mit Annotationen zu erzeugen. Diese werden anschließend über *SpanCollectoren* an Zipkin geschickt. Als Transportmittel unterstützt Zipkin HTTP, Kafka und Scribe. Zipkin erstellt aus den erfassten Tracing-Daten visuelle Zeitspannen für jeden Aufruf. Die Verschachtelung stellt es in einer Baumstruktur dar (**Abb. 4**).

Durch die visuelle Darstellung der Zeitspannen lässt sich die Laufzeit der Microservices gut beobachten und mögliche Zeitfresser identifizieren. Ein Nebenprodukt von Zipkin ist eine Darstellung der Abhängigkeiten der einzelnen Services. Der Nachteil von Zipkin ist, dass es einen komplett eigenen Technologiestack mit sich bringt. Es enthält eigene Tools zum Erfassen der Daten, eine eigene Datenbank sowie eine eigene Weboberfläche. Zurzeit gibt es keine offizielle Integration mit Graphite oder Grafana.

FAZIT

Während man sich bei der Entwicklung eines Monolithen auf einem Applikationsserver relativ wenig Gedanken um das Monitoring machen musste, spielt es bei Microservices eine umso größere Rolle. Mit der Microservices-Architektur müssen alle Bereiche umdenken: Die alten Tools können nicht mehr verwendet

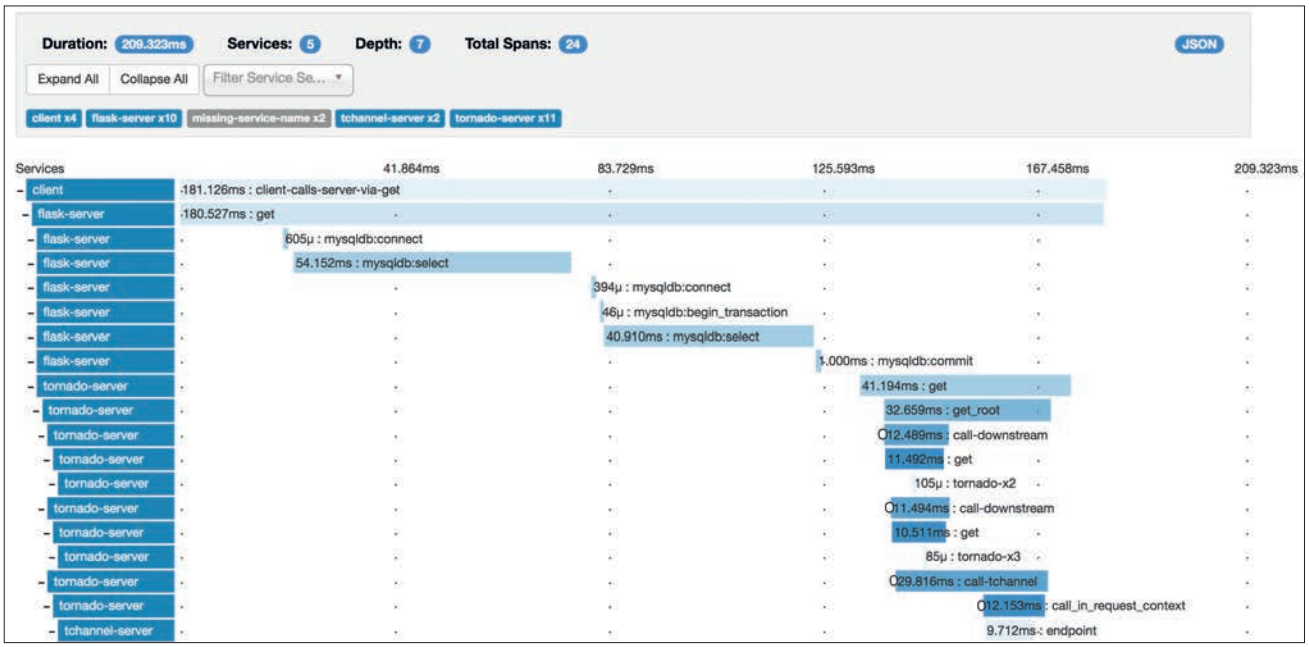

Abb. 4: Das Tool Zipkin hilft dabei, systemübergreifende Aufrufe zu verfolgen

werden. Hier muss Wissen aufgebaut werden, da der Aufwand für den Betrieb der Gesamtplattform steigt und sich die Komplexität somit in Richtung der Infrastruktur verschiebt.

Bei der Umsetzung der Monitoringinfrastruktur muss aber nicht zwingend das Rad neu erfunden und neue Technologien ins Projekt gebracht werden. Idealerweise werden bereits genutzte Technologien im Projekt wiederverwendet. Kommt z. B. für das Logging der ELK Stack (Elastic Stack) zum Einsatz, sollte überlegt werden, ob dieses System vielleicht auch für Performancemonitoring ausgebaut werden kann. Ein Ansatz existiert [9], in dem mithilfe von collectd Daten an den ELK Stack gesendet werden. Ob man das Monitoring und das Logging vermischen sollte und möchte, hängt stark vom jeweiligen Projekt und den Anforderungen an das Monitoring ab.

Fakt ist, dass durch den Umstieg auf eine Microservice-Architektur der Bedarf nach einer einheitlichen Monitoringlösung und dem Tracing von Aufrufen über verschiedene Services hinweg groß ist. Diese Kompetenzen sollten im Team aufgebaut werden, bevor man mit dem Umstieg auf Microservices beginnt. Mit dem vorgestellten Toolset in diesem Artikel lassen sich alle relevanten Metriken und Tracing-Informationen erfassen und intuitiv darstellen.

Links & Literatur

[1] Graphite: http://graphite.wikidot.com/
[2] Grafana: http://grafana.org/
[3] collectd: https://collectd.org/
[4] Metrics: https://github.com/dropwizard/metrics
[5] Zipkin: http://zipkin.io/
[6] Google Dapper: http://research.google.com/pubs/pub36356.html
[7] Libraries von Zipkin: http://zipkin.io/pages/existing_instrumentations.html
[8] brave: https://github.com/openzipkin/brave
[9] Performance Monitoring with the ELK Stack: collectd: https://www.elastic.co/elasticon/2015/sf/performance-monitoring-with-the-elk-stack-collectd

Vitali Fichtner

arbeitet als Senior Consultant bei der OPITZ CONSULTING Deutschland GmbH. Er hat langjährige Erfahrung als Entwickler in der ganzheitlichen Projektabwicklung von Individualsoftware im Java-EE-Bereich.