Zielführende Fragmentierung mit Microservices und Docker vs. Herausforderungen des Monitorings – die Bedeutung des Journald von Linux

Roland Grieder, GRADUAL.CONSULTING

Während immer mehr Konsens darüber entsteht, dass Software-Entwicklung mit (Micro-) Services effektiv ist, ergeben sich seitens des System-Managements zusätzliche Herausforderungen: Wie überwacht man am besten die Vielzahl von Komponenten, deren Zahl sich auch dynamisch ändern kann, insbesondere mit Docker. Der Artikel zeigt, wie man mit Bordmitteln von Linux, insbesondere Journald, hochwertig (nicht nur) Logs zentralisieren und auswerten kann. Darüber hinaus leitet der Autor eine schlanke, moderne Architektur für das Monitoring ab, die auf Journald aufbaut – und in der Praxis funktioniert.

Über Microservice-Architekturen und flankierende Maßnahmen wie DevOps oder Transformation zu agilen Organisationsformen gibt es mittlerweile detaillierte Erfahrungen. Die Beteiligten unterstützen den allgemeinen Konsensus, dass mit Maß verteilte Services in einem passenden Kontext attraktiv sind.

Je mehr Systeme verteilt sind, desto aufwendiger ist es, diese zu überwachen. Das Monitoring genauer betrachtet ergibt, dass sich heutzutage eine Vielzahl von Systemen etabliert hat, wobei die Funktionalität überlappt, sodass sich der Aufwand pro Tool und Komponente unbefriedigend multipliziert: Alarmierung, Logs zentralisieren und korrelieren, Systeme für Audit-Trails; in Bezug auf Security gehören im weiteren Sinne Konsolen-Logger dazu, hinzu kommen Überwachungstools für Prozesse auf dem Betriebssystem (nebst Systemd) etc. Einige dieser Komponenten können nicht ohne konzeptionelle Überlegungen zu entsprechender Spezialkonfiguration mit dynamisch gestarteten beziehungsweise gestoppten Komponenten umgehen.

Verschiedene Arten des Monitorings

Monitoring lässt sich mit einer Reihe von Attributen qualifizieren, wobei sich die gängig verwendeten Begriffe überschneiden. Nebst einer vollständigen Sicht lässt sich der Zielumfang für die Monitoring-Architektur abstecken, die dieser Artikel herleitet:

- **Aktiv vs. passiv**
  Eine Applikations-Komponente wird von einer aktiven Komponente nach deren Status gefragt oder ein Benutzer löst betrieblich relevante Log-Records aus, die ein Monitoring-System (passiv) einsammelt
- **Whitebox vs. Blackbox**
  Innenleben oder Außenwirkung
- **Negativ, positiv**
  Fehlermeldungen vs. Status-Information; horizontal entlang eines Ablaufs vs. vertikal, pro Server
- **Layer**
  Information kann aus Ablauf- beziehungsweise Benutzer-Perspektive abgegriffen werden (Probe), pro Applikations-Komponente, auf Ebene von Middleware, System-Prozessen oder Kernel inklusive Netzwerk
- **Log-Level beziehungsweise Zweck**
  Von Alarmierung, Warnung, Information, Hilfe beim Debuggen bis zu detaillierter Ablauf-Analyse (Trace)
- **Technisches Format**
  Gängig sind Files, Meldungen (SNMP, Syslog etc.), Messaging und seit einigen Jahren Journald
- **Veränderbarkeit**
  Monitoring-Information lässt sich gegen nachträgliche Änderungen schützen
- **Struktur**
  Text, strukturierter Kopf mit „Tail" oder gänzlich strukturiert, wobei die Struktur im Record mitgeliefert wird oder implizit gegeben ist
- **Situation einer Komponente**
  Information zur Installation, Starten/Stoppen oder Laufzeit
- **Art der überwachten Kenngröße**
  Neben der Information über das Ablaufverhalten von Programmen werden auch Kapazitäten überwacht (auf verschiedenen Layern) mit Schwellwerten sowie spezifische Systemgrößen (CPU nice, Angaben zu Paketverlusten auf dem Netzwerk etc.)
- **Aufbereitung**
  Direkt abgegriffene Information, aufbereitet und ergänzt, gerechnete Indikatoren oder korrelierte Information
- **Mengengerüst**
  Monitoring-Information kann diskret erfolgen, also Ereignis-basiert, periodisch, als Zeitreihe oder laufend (stream)
- **Technische Domäne**
  Applikation, Betriebssystem, System-Ressourcen, Netzwerk, Security etc.
- **Zeitversatz**
  Monitoring in Echtzeit oder unter Umständen verzögert

Ein Einschub: Besonders hinweisen möchte der Autor im Kontext von Microservices auf passive Probes, ein Instrument, mit dem SLAs unmittelbar ausgewertet werden können inklusive Ursachen-Analyse, statistisch oder pro Einzelfall. Benutzer lösen Logs aus, die IDs zur Session, zum Geschäftsablauf, Request, aufrufender beziehungsweise aufgerufener Komponenten etc. enthalten. Passive Probes geben also Auskunft über die geschäftliche Nutzung oder Auswirkung einer Störung bis hinunter auf die Ebene der Komponenten-Interaktion; zum Beispiel lässt sich das Antwortzeitverhalten einfach als etappierte Sicht von Round-Trips auswerten. Dynamisch ist also deutlich effizienter und zuverlässiger, als Bottom-up-Impacts zu korrelieren, zudem nützlich für die Security.

Praktische Erfahrung mit Log-Zentralisierung – die gute Seite …

Angefangen hat alles in einer Bank, die den Großrechner durch ein modernes Kernbankensystem abgelöst hat, mit einer verteilten Architektur für Front-Komponenten und Zusatzdienste. Es kamen Linux-Systeme zum Einsatz, das Deployment wurde vollständig mit RPM und insbesondere Ansible automatisiert.

Der Aufwand für die Fehler-Abklärungen wäre mit „remote Login", „find" für Logs, „tar" und trivialen Werkzeugen wie „grep" zu groß geworden, deshalb hat das Engineering-Team die Logs zentralisiert. Dabei wurde eines der gängigen Tools eingesetzt mit Agent, Server (Logs parsen), No-SQL-Datenbank und Portal zur Auswertung. Es wurde festgestellt, dass die Tools in der Grund-Architektur austauschbar sind; Ansatz, Nutzen wie auch die Probleme sind allerdings überall gleich.

Doch zunächst zum Nutzen; alle Logs wurden sorgfältig getaggt: Umgebung, Applikation, Unterkomponente, Version, Start/Run/Stop, Server-Name etc. Damit war man tatsächlich in der Lage, technische Incidents inklusive Problem-Analyse innerhalb einer Viertelstunde zu lösen. Man stand kurz davor, dass die Entwickler per Self-Service ihre Tickets effizient bearbeiten können, statt Konstellationen oft mühsam mit zusätzlichen Daten in ihrer Umgebung zu reproduzieren beziehungsweise iterativ Logs vom Operating-Team zu bestellen.

…und die schlechte – das war zu lösen

Das Engineering-Team meldete, der Log-Server sei bereit. Dann geschah es: „Bis wann läuft das eBanking (auf der Abnahmeumgebung) wieder?" Die Antwort lautete „Hm … wir schauen umgehend und melden uns." Ein sehr unangenehmes „Hm", es gab keinerlei

Anzeichen einer Störung. Noch schlimmer eine Woche später: Der Job-Scheduler hatte sich verabschiedet, wortreich wie eine Diva (2 GB Logs), worauf der Log-Server sich verschluckt hatte, er wurde instabil. Das Engineering-Team hat wieder nichts davon erfahren. Auch das Problem-Management klappte nicht, da diese interessanten Daten teilweise unterwegs verloren gingen – inakzeptabel.

Die erste Regung bestand darin, den Code der Open-Source-Lösung anzuschauen, um Bug-Fixes beizusteuern. Dabei wurde festgestellt, dass diese Art von Log-Tools als architektonisches Prinzip die Applikationen möglichst nicht stören soll, was sie mit Zuverlässigkeit und damit Wertschöpfung für den Betreiber bezahlen:

- Als Protokoll kommt standardmäßig das verlustbehaftete UDP zum Einsatz, wobei die Log-Tools auf dem applikatorischen Layer den Datenverkehr nicht zusätzlich absichern. Bei viel Netzverkehr profitiert also TCP ein Stück weit davon, dafür werden Logs zunehmend unzuverlässiger übertragen. Also TCP.
- Die Agents arbeiten mit eher kleinen, flüchtigen Puffern, die interne Verarbeitung im Server (ohne Not) auch. Überlauf oder Ausfälle führen unmittelbar zu Datenverlust.
- Durch das Parsen ist der Server recht langsam. Die vielen Retries der wartenden Agenten haben ihn soweit nachvollziehbar instabil gemacht. Der Log-Server muss generell hoch dimensioniert sein, sonst baut sich bei Last der Eingangs-Puffer immer mehr auf. Wenn das Telefon klingelt, bevor ein Alarm eintrifft, kann man zu Recht über den Sinn der Lösung diskutieren. Wenn Logs mit etwas zufälligem Zeitverzug eintreffen, stellt sich auch die Frage, wie zuverlässig die sofortige Log-Korrelation noch funktioniert.
- Um täglich ein Log-File zu hinterlassen, wird das laufend gefüllte Log-File umbenannt. Damit nun der Log-Agent das neue File nicht neu auswertet, arbeitet er nicht mit Dateinamen, sondern mit den dahinterliegenden IDs, den „Inodes". Diese bleiben nämlich bei diesem Vorgang stabil. Konzeptionell unsauber wird es, wenn man alte Log-Files löscht: Das File ist weg, der Inode entsprechend frei zur erneuten Vergabe, aber die Buchführung des Agent erfährt davon nichts. Das Engineering-Team fühlte sich trotz geringer Wahrscheinlichkeit auch mit dieser Entdeckung schlecht, da die künftige Betriebsfirma hohe Ausfallentschädigungen riskiert; niemand würde doch einen Webshop ohne Transaktionen für Bestellungen und Zahlungen implementieren, wo es um deutlich geringere Beträge geht.

Noch eine andere Eigenschaft ließ am Ansatz dieser gängigen Lösungen zweifeln: In den Java-Applikationen kommt Log4J V.2 mit einem sauber gepflegten Thread-Context zum Einsatz. Diese direkt maschinell auswertbare Information wird im ersten Schritt zu Fließtext umgewandelt, also massiv entwertet, und das teilweise irreversibel: Je nach Situation zum Zeitpunkt eines Absturzes sehen Thread-Context wie Log-Message anders aus. Es gibt also keine klare Spezifikation, sondern einen Medienbruch: Man soll-

te auf dem Log-Server einen Parser schreiben, der – generell gedacht pro Version der Applikations-Komponente – irgendwie mit der Vielfalt der möglichen Log-Meldungen klarkommen sollte. Die Realität hat gezeigt, dass es möglich ist, einige Felder zu extrahieren, aber irgendwann der Rest eines Log-Record im Textformat belassen werden musste.

Es schien, als seien die Logs einst erfunden worden, das Thema jedoch dann lieblos gehandhabt wurde. Um nur zwei, aber wesentliche Faktoren zu nennen: Betreiber stellen traditionell keine Anforderungen an Logs, geschäftsferne Logik wird zurückhaltend finanziert. Denn eigentlich wäre eine saubere Lösung doch gar nicht so schwierig …

Journald – ein strukturierter, störungsresistenter und sich selbst verwaltender Puffer

Das Ziel ist, mit möglichst nur einer hochverfügbaren Lösung diverse Monitoring-Informationen – strukturiert und direkt maschinell verwertbar – zuverlässig und in Echtzeit zentral auszuwerten. Starten wir mit dem Problem des Puffers: Weder soll die Geschäftsfunktionalität der Applikations-Komponente durch mögliche Staueffekte für das Monitoring beeinträchtigt werden, noch soll das Memory des Agenten überlaufen. Also brauchen wir einen persistenten, störungsresistenten Puffer für strukturierte Daten. Dieser möge sich selbst verwalten, ohne Skript-Wunderwerk, um alte Logs zu managen, und idealerweise zentral sein.

Hier kommt nun der Clou, ein großer Schritt für das Monitoring, noch erstaunlich wenig bekannt, aber in unserem Zusammenhang bestens geeignet: Alle Linux-Derivate haben sich auf Journald geeinigt als zentrales Subsystem für Logging mit mehr Benefits als nur den erwähnten Charakteristika. Ein Journal ist wie eine Datei, die für diverse Szenarien von Störungen gesichert ist. Neben den genannten Kriterien verhindert das Subsystem nachträgliche Änderungen, ideal für Audit-Trails. Mit Rechten lassen sich Bedürfnisse unterschiedlicher Leser konfigurieren.

Journald erfährt umfassend, was auf dem System geschieht. Man kann etwa nachvollziehen, wer sich eingeloggt und wann welche Commands beziehungsweise Komponenten aufgerufen hat; dank der technischen Güte schwierig kompromittierbar. Journald arbeitet mit Key-Value Pairs unterschiedlicher Datentypen. Nützliche Felder wie Hostname, Zeitstempel etc. fügt es selbst hinzu. Außerdem ist Journald die sauberste Lösung im Umgang mit Docker (Varianten gibt es mit automatischen Managern wie Kubernetes), indem die Log-Information lokal geschrieben, aber zentral auf das unterliegende Betriebssystem abgelegt wird; insbesondere dafür wurde Journald ursprünglich erfunden.

Die Logs sind nun also strukturiert persistiert, man muss sich um nichts mehr kümmern, da Journald alte Records automatisch gemäß konfigurierbaren Regeln löscht. Wichtig für die weitere Verarbeitung: Das Niveau der technischen Güte halten. Nur so kann man Auditing, Konsolen-Logger, Alarming etc. zusammenlegen. Also keine Files mehr einbauen. Auch hier hilft Journald: Records können in ein beliebiges Journal (eben nicht File) ausgelagert oder von einem Server auf andere Journalds konzentriert werden. Eine umfassende Log-Zentralisierung nur mit Board- Mitteln von Linux.

Was das mit Java zu tun hat

Log4J Version 2 und höher ist auf einem guten Stand, wir müssen nur noch den wohlstrukturierten Thread-Kontext ins Journald übertragen, wofür Log4J einen speziellen „Appender" vorsieht. Dazu nimmt man den Logger des Applikations-Servers und konfiguriert einen Appender dazu für Journald. Dann legt man einen Log4J-Logger zu Journald für alle Applikationen zusammen an, als geteilte Ressource. Als Basis für den erwähnten Appender kann eine Open-Source-Minimalversion aus dem Internet geladen werden. Um diese zu integrieren, schaut man im Internet für seinen Application Server unter dem Stichwort „custom appender" nach. Achtung, Stolperstein: Ein Java-Appender integriert die Adapter-Library für Journald mit JNA. Der Adapter wird in einem anzugebenden Verzeichnis zwischengespeichert, das ausführbar sein muss; für einen speziellen User finden wir dazu eine gute Lösung. *Listing 1* zeigt die Konfiguration des Pfads.

Der Autor hat in seiner Implementation zusätzlich Funktionalität eingebaut, um Keys vorzugeben, zu denen er die Werte (Values) erhalten möchte, oder um alle verfügbaren Felder abzurufen. Außerdem fügte er Metadaten-Felder hinzu: Umgebung, Applikationskomponente, Version etc. Einige Daten lassen sich direkt beschaffen (Manifest-File für den Applikationsnamen sowie die Version), andere mit Regeln ableiten (das Namenskonzept der Server gab Hinweise zum Verwendungszweck einer Umgebung) und einige Felder setzt man statisch (Run-Log). Fazit: Ein Appender und vernetzte Journalds – und ein Entwickler hat alle Informationen beisammen, die er zum Debuggen braucht.

Journald für Entwickler

Nachdem die Information im Journald liegt, wäre es gut zu wissen, wie man sie wieder ausliest. Im Internet gibt es eine Vielzahl von Tutorials – für Administratoren. Für die Bedürfnisse der Entwickler folgen die wesentlichen Kommandozeilen, sodass man Journald gleich ausprobieren kann.

Wichtig: Zunächst erhält ein Benutzer mit dem zentralen Command „journalctl" meist gar nichts zurück, weil ihm die Berechtigungen fehlen. Der erste Schritt besteht also darin, zum Administrator zu

```
JAVA_OPTS="$JAVA_OPTS -Djna.tmpdir=/path/to/loaddir"
JAVA_OPTS="$JAVA_OPTS - Djava.io.tmpdir=/path/to/loaddir
```

Listing 1

```
Sudo nano /etc/system/journald.conf #Im Konfigurations-File:
…
[Journal]
Storage=persistent    #Damit wird alle Information persistiert.
```

Listing 2

```
ExecStartPost=/bin/sh -c 'tail -f "/var/logs/legacyfile.log" | systemd-cat -t "beispiel-app"' #-t für Attribute, “tags”.
```

Listing 3

```
Journalctl -r #Umgekehrte Reihenfolge, neueste Einträge zuerst (mein Favorit…).
journaldctl -n 20 #Analog tail, die letzten n Einträge.
journalctl --since “2018-03-28” --until “20187-01-11 17:21”
journalctl --since yesterday --until “1 hour ago”
journalctl _UID=55 _PID=8088 #Eigne, reservierte Felder Journald beginnen mit dem Zeichen “_”.
journalctl /user/bin/bash #Commands gefiltert nach Verzeichnis.
journalctl -p err #Priority level, wie Syslog: 0 emerg, 1 alert, 2 crit, 3 err, 4 warning, 5 notice, 6 info, 7 debug.
Journalctl -p 0..4 #Die höchsten fünf Severity-Stufen.
Journalctl -no-pager #Output am Stück für Weiterverarbeitung.
jouirnalctl -u nginx -u elastic.service --since today -o json
#Liefert JSON-Records, -o json-pretty (tatsächlich) schön formatiertes JSON im Zeilenformat.
Journalctl -f #Wie tail -f, laufendes Auslesen neuer Einträge.
```

Listing 4

gehen. Dabei ist zu überprüfen, dass Journald persistent aufgesetzt ist *(siehe Listing 2)*. *Listing 3* zeigt, wie man Legacy-Logs aus Services mit „systemd-cat“ ins Journald umleitet (mäßig strukturiert), und *Listing 4* die Abfragen.

Die Zukunft der traditionellen Log-Tools

Wenn man zu einem traditionellen Log-Tool einen Agenten für Journald sucht, findet man den nicht. Logisch: Traditionelle Log-Server können Logs parsen, was ja mit Log4J zu Journald wegfällt, aber nicht zuverlässig und unverfälscht übertragen; das macht viel vom Wertangebot des Journald aus. Falls es also doch einmal einen solchen Agenten geben sollte, ist Vorsicht geboten. Mit dieser Feststellung wird unmittelbar die Zukunft der aktuellen Architektur dieser Tools fraglich.

Im Verbund mit Journald ergibt eine No-SQL-Datenbank mit Portal weiterhin Sinn: Man kann feiner granular als nur ganze Feld-Inhalte suchen, auch mit Wildcards oder statistisch. Außerdem erhöht eine grafische Benutzerschnittstelle die Effizienz sowie die von Dritten deutlich. Zudem sind flankierende Maßnahmen zu treffen, um die technische Güte auch in dieser Komponente so hochzuhalten, dass eine moderne Log-Architektur ihren breiten Einsatzzweck ohne Bedenken erfüllen kann *(siehe Abbildung 1)*.

Zur Implementation: Mit „At-least-once Guarantee“ (mehr geht mit einer No-SQL-Datenbank aufgrund deren Konsistenz-Mechanismen nicht, sonst kann man beispielsweise JSON in PostgreSQL verwenden) verbindet ein schlanker Agent in Go Journald durch eine ebenfalls sehr schlanke Server-Komponente mit dem Bulk-Interface der No-SQL-Datenbank ElasticSearch. Der Agent kann Legacy-Logs parsen, Werte anpassen (etwa die Severity von Testsystemen deckeln), ein Datenpaket aus Journald zum Transfer anbieten und bei Störungen den Server wechseln.

Der Server vermittelt über viele Agenten hinweg den Datentransfer in das Bulk- Interface, wobei erst das Commit der Datenbank an die Agenten zurückgesendet wird. Zudem sortiert der Server bereits Alarme aus. Beim Datentransfer wurde darauf geachtet, dass Paketgrößen zu denen auf Netzwerk-Ebene passen, um die Effizienz weiter zu erhöhen. Einige positive Charakteristika dieser Vorgehensweise sind:

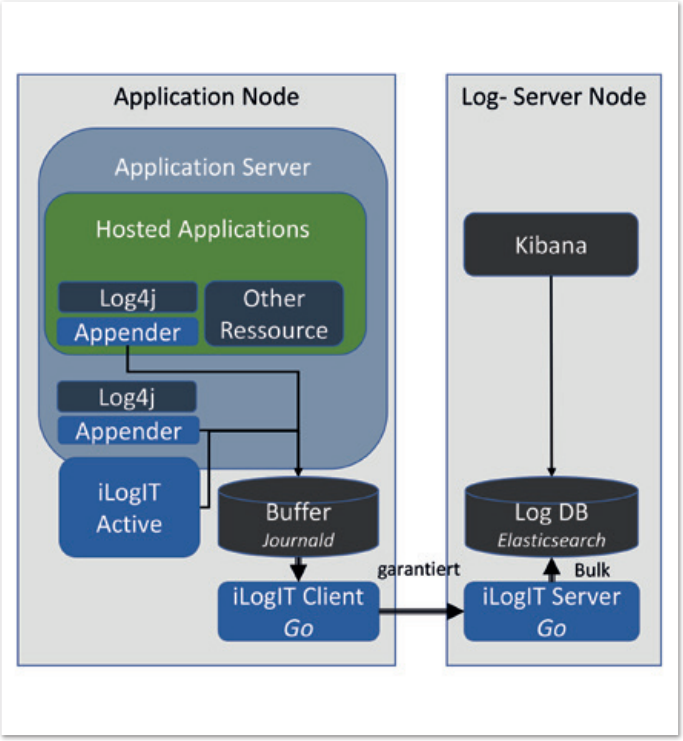

Abbildung 1: Die Komponenten einer modernen, konsolidierten Monitoring-Architektur

- Der Durchsatz dieses Log-Servers ist enorm (gegenüber einem parsenden Server im Leerlauf bereits Faktoren von mehr als 10.000), daher kaum Retries. Der Server selbst ist mit seinen wenigen Code-Zeilen robust.
- Dank der getakteten Übertragung in Echtzeit und des Verzichts auf einen weiteren Buffer auf dem Server sind die Probleme mit Zeitverzug wie Synchronität, um unmittelbar eintreffende Logs zu korrelieren, gelöst.
- Auch der Server ist in Go implementiert, sodass er direkt auf den Knoten von ElasticSearch installiert werden kann, ohne relevante technische Abhängigkeiten. Der Server und Inserts in Elastic sind abwechselnd tätig; man spart eigene VMs für den Server.
- Mit dem Bulk-Interface indexiert ElasticSearch wesentlich effizienter, die Bandbreite nimmt massiv zu, der Ressourcen-Bedarf beziehungsweise die Kosten ab.

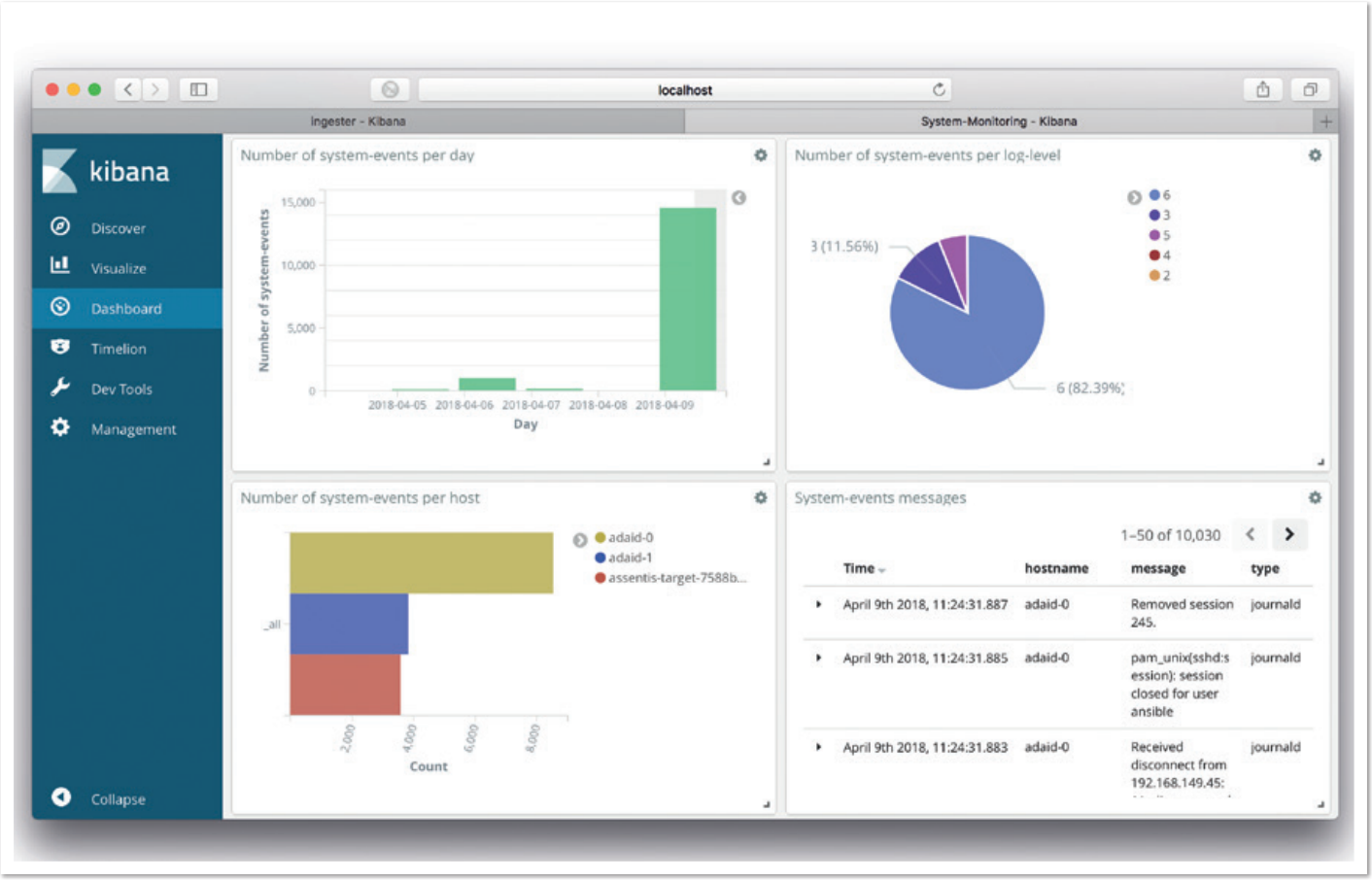

Abbildung 2: Mit wenigen Mausklicks lassen sich solche Dashboards herstellen

- Ohne Logs und Audit-Trails brauchen die Online-Services das Dateisystem gar nicht mehr, sobald diese installiert sind. Nicht nur applikatorisch sind die Services damit state-less, sondern auch betrieblich. Im Sinne der Komplexitätsreduktion wurde gleich auch auf Server-Backups verzichtet; geht auf einer VM etwas schief, wird diese neu angelegt und automatisiert installiert.

Der moderne Log-Server wird zum Systems-Management-Tool

Da nun umfassende Information pro Server zuverlässig, unveränderlich und in Echtzeit zentralisiert vorliegt, kann man auf diverse Spezialtools verzichten. Mit dem Essen kommt der Appetit: Was kann man sonst noch weglassen? Beim Blick auf die Liste mit Eigenschaften des Monitorings sind alle Aspekte abgedeckt, außer aktives Monitoring, da ist noch eine abfragende Komponente erforderlich: JMX, Messaging- Kapazitäten, Betriebssystem-Prozesse und der Status des Kernels inklusive Netzwerk. Eine aktive Komponente ist attraktiver, als eine traditionelle Alarmierungslösung für diesen Zweck aufzubauen, mit großer funktionaler Überlappung, wodurch mehrfach konfiguriert werden muss.

Diese letzte Komponente der Lösung fragt und schreibt beziehungsweise rechnet also periodisch Statuswerte ins Journald. Sie überwacht direkte oder gerechnete Messgrößen gegenüber Schwellwerten. Was fehlt noch? Die Funktion, die entscheidet, welche Prozesse laufen dürfen, sollen oder verhindert werden. Dann bleiben die File-Systeme, die aber meist in anderer Zuständigkeit und für eine Log-freie Installation auch nicht mehr wichtig sind. Deren Zustandsüberwachung noch zu integrieren, wäre sehr einfach.

Damit ist bereits der Stand einer ausgereiften Systems-Management-Suite in Bezug auf Monitoring erreicht – sogar übertroffen *(siehe Abbildung 2)*. In der Praxis hat sich diese moderne Monitoring-Architektur bislang durchweg bewährt. Sie schont das Nervenkostüm, Aufwände und Betriebskosten; daher heißt das Tool freundlich „iLogIT".

Roland Grieder
roland.grieder@gradual.consulting

Roland Grieder hat an der ETH Zürich Elektrotechnik studiert, bei IBM unter anderem die Lehrgänge zu Consulting und Architektur absolviert sowie an der HSG einen Abschluss für General Management, CSA, absolviert. Eine Spezialisierung ist die (Neu-) Planung von Rechenzentren zusammen mit der Architektur für Betriebsautomatisierung, so als Head of Projects einst bei Swisscom Wholesale zur Überwachung des Schweizerischen Festnetzes. Diverse Projekte folgten bei Banken und Versicherungen, die die IT komplett umgestellt haben.