Performance, Durchsatz und Integrität

Logging konsolidieren und Performance gewinnen

Stefan Bodewig, Phillip Ghadir

Apache Log4j 2.0 ist eine Neuimplementierung von Log4j und zeichnet sich – dank asynchronem Input/Output – vor allem durch einen höheren Durchsatz aus. Die Neuerungen des Frameworks machen Lust darauf, Log4j2 einzusetzen. Aber es drängt sich die Frage auf, wie die Vorzüge von Log4j2 in einem System genutzt werden können, wenn Teile noch Log4j1, Commons-Logging oder Ähnliches verwenden. Neben einer kurzen Vorstellung des Frameworks betrachten wir daher auch diesen Aspekt.

Log4j 2.0 Kurzüberblick

Lange Zeit war es still um das Apache-Log4j-Projekt [Log4j]. Aber seit Juli 2014 gibt es mit Version 2.0, gefolgt von einigen Bugfix-Releases und Version 2.1 im Oktober 2014, eine vollständige Neuentwicklung des Frameworks. Log4j2 behält seit Langem eingeführte Konzepte wie die hierarchischen Logger oder die Aufgabentrennung zwischen Loggern und Appendern bei, sodass sich jeder sofort zurechtfindet, der Log4j 1.x oder [logback] kennt.

Die Entwickler von Log4j2 heben besonders die deutlich bessere Performance hervor. Durch das neue Konzept des Async-Loggers erlaubt Log4j2 einen erheblich höheren Durchsatz von Log-Nachrichten bei gleichzeitig deutlich geringeren Latenzzeiten als bei synchronen Loggern mit asynchronen Appendern.

Aber auch jenseits der Performance hat Log4j2 einiges zu bieten. Anders als die Vorgängerversion oder logback soll Log4j2 auch als Audit-Logging-Framework einsetzbar sein; es ist möglich, Exceptions in Appendern bis in die Applikation propagieren zu lassen oder einen Failover-Appender anzugeben, der Nachrichten aufnimmt, wenn der eigentlich vorgesehene Appender Probleme hat. So geht mit Log4j2 nichts unbemerkt verloren.

Genau wie logback kann Log4j2 die Konfiguration dynamisch aktualisieren. Hierbei gehen keine Log-Meldungen verloren, und sollte es syntaktische Fehler in der neuen Konfiguration geben, so wird weiterhin die alte Konfiguration benutzt.

Die bereits aus Log4j 1.x bekannten Filter können nun nicht erst auf der Ebene des Appenders Log-Nachrichten anhand anderer Kriterien als des Log-Levels ausfiltern, sondern lassen sich auch global oder auf der Ebene eines Loggers konfigurieren.

Die Code-Basis von Log4j2 ist erheblich modularer geworden, und ein neues Plug-in-Konzept erlaubt es, neue Appender, Formatter oder Layouts einfach hinzuzufügen. Eines der neuen Plug-ins ermöglicht es, via JMX Log4j2 zu konfigurieren oder dessen Status-Logger auszulesen.

Performance-Gewinne

Asynchrone Logger setzen noch ein wenig früher an als die schon seit Langem existierenden asynchronen Appender. Neben der eigentlichen Schreiboperation werden bei asynchronen Loggern auch alle Konvertierungs- und Layoutaufgaben in einem eigens für das Logging vorgesehenen Thread bearbeitet. Im Wesentlichen kopiert der AsyncLogger die für die Log-Nachricht notwendigen Daten in eine Queue-Struktur und setzt den eigentlichen Applikations-Thread anschließend fort. Als Nebeneffekt entsteht hierbei auch ein einfacheres Programmiermodell für die Implementierung der Logger – sie sind single-threaded.

Log4j2s AsyncLogger setzt auf den [LMAXDisruptor] anstatt – wie bei den asynchronen Appendern – auf eine Blocking-Queue, um Log-Nachrichten von der Applikation an den Appender als Konsumenten weiterzuleiten. Der Disruptor ist eine Datenstruktur, die darauf optimiert wurde, mit möglichst geringer Latenz Daten zwischen Quellen und Senken zu vermitteln. Erst durch diese Datenstruktur wird es möglich, erheblich höheren Durchsatz zu erreichen.

Wirklich wahrnehmbar wird der Performance-Gewinn durch asynchrone Logger erst dann, wenn mehrere Threads einer Applikation gleichzeitig loggen wollen. Der Unterschied wird in Benchmarks schon ab vier Threads deutlich sichtbar und verstärkt sich mit zunehmender Anzahl von Threads. Abbildung 1 stellt den Performance-Zuwachs verschiedener Strategien gegenüber (siehe auch [Log4jPerformance]). Noch deutlicher fällt das Ergebnis der Latenz-Messung aus, die bei asynchronen Loggern auch bei vielen Threads etwa konstant bleibt, bei synchronen Loggern aber ab etwa sechzehn Threads deutlich zunimmt.

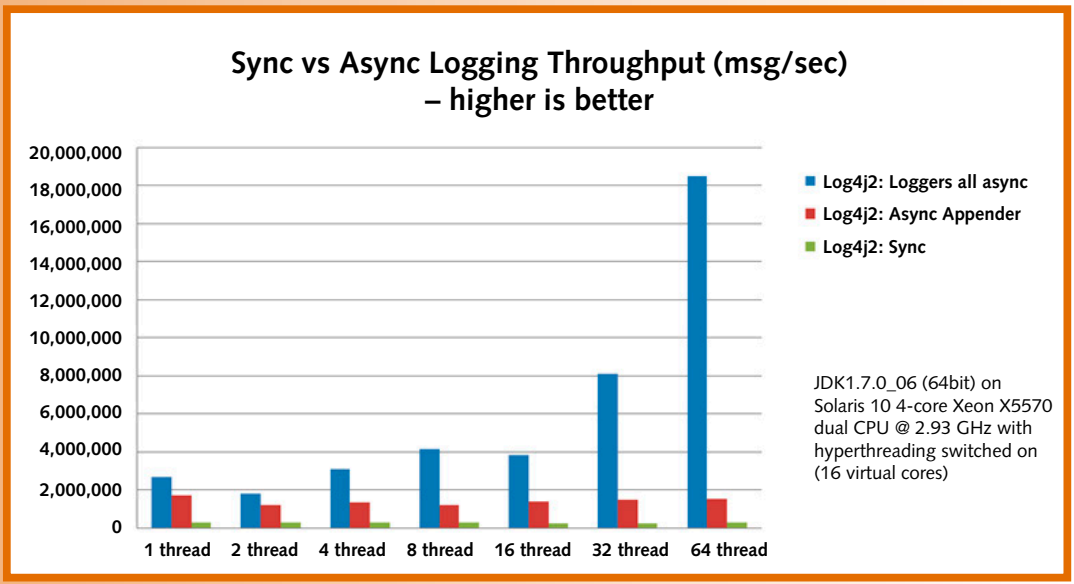

Abb. 1: Log4j2-Durchsatz abhängig von der Anzahl paralleler Threads (von Apache [Log4j])

Preis asynchroner Logger

Asynchrone Logger können eventuell auftretende Probleme mit Appendern nicht als Exceptions an die Applikation zurückmelden. Sie lassen sich allerdings über einen Failover-Appender abfangen und protokollieren. Beim Failover-Appender gehen zwar keine Nachrichten verloren, doch benötigt man einen zusätzlichen Prozess, um die Nachrichten wieder zusammenzuführen. Auch die zeitliche Reihenfolge der Nachrichten geht möglicherweise verloren. Wenn also – wie etwa für Audit-Logs – sichergestellt sein muss, dass eine Log-Nachricht auf jeden Fall geschrieben wird, sind asynchrone Logger daher eher ungeeignet.

Da Log-Nachrichten gegebenenfalls erst zu einem späteren Zeitpunkt abgearbeitet werden, sollten Objekte, die Informationen für die Nachricht bereitstellen, unveränderlich sein. Sonst könnten einige Log-Meldungen aufgrund falscher Daten leicht irreführen.

Einige der Attribute, die in eine formatierte Log-Nachricht eingehen, sind relativ teuer zu ermitteln. Hierzu gehört insbesondere die Location, also die Zeile, Methode und Klasse im Quelltext, die eine bestimmte Nachricht geloggt hat. Log4j verwendet dazu den aktuellen Call-Stack – und diesen zu ermitteln kostet spürbar Zeit. Bei synchronen Loggern kann Log4j das gezielt nur dann leisten, wenn diese Informationen tatsächlich Teil der formatierten Nachricht werden. Für asynchrone Logger müsste der Stack immer ermittelt werden, da er sonst später nicht mehr zur Verfügung steht. Aus diesem Grund haben sich die Entwickler entschieden, Location standardmäßig gar nicht für asynchrone Logger zur Verfügung zu stellen. Sollte man die Informationen doch benötigen, kann man konfigurativ dafür sorgen, dass der Call-Stack immer ermittelt wird, verliert dabei aber einige der Performance-Vorteile.

Es ist möglich, alle Logger der Applikation asynchron auszuführen oder aber synchrone und asynchrone Logger zu mischen. Sofern man gut auf Location und Audit-Log verzichten kann, spricht vieles dafür, alle Logger asynchron zu konfigurieren. Ansonsten lohnt es sich, die Stellen zu suchen, bei denen die Sicherheit eines Audit-Logs notwendig ist, und dort dann gezielt synchrone Logger einzusetzen. Der Rest der Applikation kann dann ruhig asynchrone Logger nutzen.

Log4j2 konfigurieren

Während Log4j 1.x über Property-Dateien konfiguriert wurde, hat man mit Log4j2 die Wahl zwischen den hierarchischen Formaten XML, JSON und YAML. Um JSON und YAML zu verwenden, benötigt die Applikation zusätzlich die Hilfe des Jackson Mappers.

Innerhalb der Konfiguration können statt voll qualifizierter Klassennamen auch Kurznamen verwendet werden, etwa `Console` statt `org.apache.logging.log4j.core.appender.ConsoleAppender`, diese Namen werden über die Plug-in-Konfiguration per Annotation definiert. Einerseits erhält man so übersichtlichere Konfigurationsdateien, andererseits kann so kein Schema mehr definiert werden, da jedes neue Plug-in neue erlaubte Namen hinzufügen kann. Aus diesem Grund gibt es neben dem kompakten ein validierbares Format, das auf Kurznamen verzichten muss; dieses dürfte in der Praxis aber selten verwendet werden.

Innerhalb der Konfigurationsdatei kann durch `${lookup:key}` auf sogenannte Properties zugegriffen werden. Die Werte dieser Properties stammen aus den Lookups genannten Quellen, dazu gehören etwa Environment-Variablen oder System-Properties. Neben diesen eher statischen Lookups können Properties aber auch aus der zu loggenden Nachricht selber stammen.

Besonders sichtbar wird dies im RoutingAppender, der Nachrichten auf der Basis von Eigenschaften der Nachricht an verschiedene andere Appender verteilt. Eine Konfiguration wie

```
<Routing name="file-per-user">
  <Routes pattern="$${ctx:user}">
    <Route>
      <File name="file-per-user-${ctx:user}"
            fileName="logs/${ctx:user}.log">
        <PatternLayout>
          <Pattern>%d %p %c{1.} [%t] (%X{user}) %m%n</Pattern>
        </PatternLayout>
      </File>
    </Route>
  </Routes>
</Routing>
```

legt für jeden Benutzer, der durch die Property `user` in der Log4j-ThreadContext-Map identifiziert werden kann, eine eigene Log-Datei an. Der Umweg durch den RoutingAppender ist notwendig, da Appender unmittelbar nach Lesen der Konfiguration einen eindeutigen Namen benötigen und es keinen Weg gibt, innerhalb des Namens des Appenders Properties zu referenzieren, die erst später einen Wert bekommen.

Properties können Infrastrukturunterschiede in verschiedenen Umgebungen abbilden, ohne dass dazu verschiedene Log-Konfigurationen gepflegt werden müssten. So kann etwa der Hostname eines syslog-Servers aus einer Environment-Variablen oder einer JNDI-Konfiguration bezogen werden.

Log4j2 in Bestehendes integrieren

Viele existierende Bibliotheken verwenden Abstraktionsschichten für das eingesetzte Logging-Framework, damit Benutzer der Bibliotheken nicht festgelegt werden. Besonders verbreitet sind [SLF4J] und [CommonsLogging].

Log4j2 implementiert SLF4J nativ und bietet außerdem Bridges an, die Log4j2 als Ziel hinter der Programmierschnittstelle von Commons-Logging, Log4j 1.x oder gar java.util.logging nutzen. Jede dieser Fassaden wird durch ein eigenes JAR zur Verfügung gestellt, das der Applikation – in der Regel – lediglich im Classpath hinzugefügt werden muss.

Hibernate auf Log4j2

Einen besonders schwierigen Fall stellt Hibernate4 dar.

Hibernate3 verwendet SLF4J für das Logging. Hier reicht es aus, das Log4j-SLF4J-Binding der Applikation hinzuzufügen, um Hibernates Meldungen via Log4j zu loggen. Praktisch alle existierende Dokumentation zu Hibernate und Logging bezieht sich auf Hibernate3.

Mit Hibernate4 haben die Entwickler SLF4J durch JBoss Logging – eine weitere Bibliothek, um vom konkreten Logging-Framework zu abstrahieren – ersetzt. jboss-logging verwendet einen eigenen Algorithmus, um das vorhandene Logging-Framework auszuwählen. Ab der Version 3.2.0.Beta2 wird jboss-logging nativ Log4j2 unterstützen, sodass dann keinerlei Konfiguration mehr erforderlich ist, aber leider gibt es zum Zeitpunkt, da dieser Artikel geschrieben wurde, noch kein Release dieser Version.

Hibernate4 verwendet aktuell jboss-logging in der Version 3.1.3.GA, welches nacheinander prüft, ob eine Reihe von Logging-Frameworks vorhanden sind. Log4j 1.x würde benutzt,

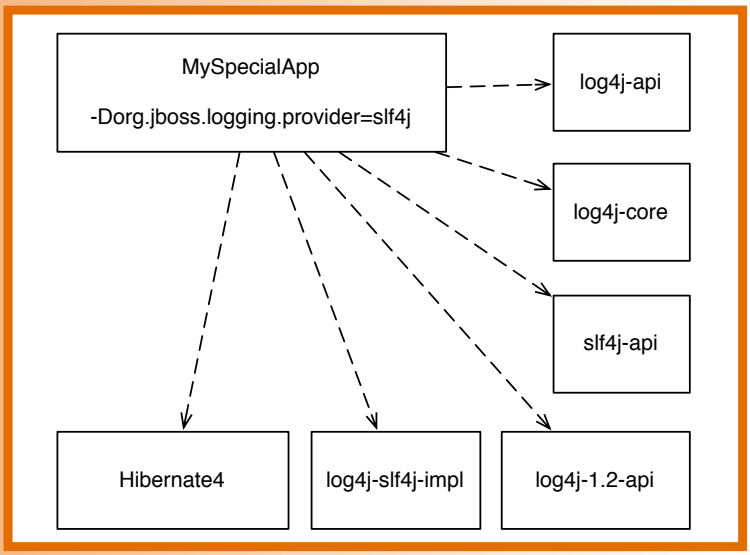

Abb. 2: Setup einer Anwendung, die Hibernate4 über Log4j2 loggen lässt

allerdings prüft jboss-logging, ob eine bestimmte Implementierungsklasse vorhanden ist, weshalb die Log4j1-Bridge von Log4j2 nicht akzeptiert wird. Ähnlich verhält es sich mit dem SLF4J-Binding, da SLF4J nur dann ausgewählt wird, wenn dieses logback verwendet. Glücklicherweise lässt sich jboss-logging mit der System-Property `org.jboss.logging.provider` zwingen, ein bestimmtes Framework zu verwenden – setzt man die Property auf den Wert `slf4j` wird auch das SLF4J-Binding von Log4j2 akzeptiert.

Noch komplizierter wird es, wenn man C3P0 als Datenbank-Connection-Pool verwendet. In der von Hibernate eingesetzten Version 0.9.2.1 verwendet C3P0 Log4j 1.x – hier ist also zusätzlich die Log4j 1.x Bridge von Log4j2 erforderlich.

Damit also Hibernate4 und alle von Hibernate verwendeten Komponenten wirklich über Log4j2 loggen, muss ein Applikation neben log4j-api und log4j-core auch log4j-slf4j-impl, slf4j-api und log4j-1.2-api verwenden und eine System-Property setzen, wie in Abbildung 2 dargestellt.

Der aktuelle Entwicklungszweig von C3P0 wurde ab Version 0.9.5pre2 auf slf4j umgestellt. Sobald dieser stabil genug ist, wird die Log4j 1.x Bridge nicht mehr benötigt.

Logs zusammenführen

Wenn eine Applikation in mehrere (Micro-)Services aufgeteilt wird, wird jeder dieser Services für sich Log-Nachrichten schreiben. Um eine Sicht auf die Gesamt-Applikation zu erhalten, ist es aber notwendig, diese Nachrichten zu konsolidieren.

Idealerweise werden alle Log-Nachrichten in einer speziellen Infrastruktur – etwa [logstash]/Kibana oder [splunk] – gesammelt, die diese indiziert, analysiert und für Auswertungen zur Verfügung stellt.

Folgt man den Grundsätzen der in dieser Kolumne bereits erwähnten Twelve-Factor App [12FactorApp,Ghad14], so schreiben alle Services ihre Log-Nachrichten einfach auf die Standardausgabe und ein externer Prozess kümmert sich darum, diese zusammenzuführen. Ein solcher Prozess kann Teil der Command-and-Control-Infrastruktur sein, die verwendet, wird, um die Services zu starten. Innerhalb der Applikation konfiguriert man lediglich einen ConsoleAppender.

Das relativ einfache Setup erkauft man sich auch damit, dass nun Log-Nachrichten nur formatiert vorliegen. Die Rohdaten wie der ThreadContext oder die aktuelle Zeit sind entweder gar nicht mehr verfügbar (weil sie nicht im Log-Format auftauchen) oder müssen vom Indexer aus dem Text wieder extrahiert werden.

Remote Logs

Die Alternative hierzu besteht darin, Log-Nachrichten „über das Netz" an einen anderen Server zu senden, der diese sammelt. Auf der einfachsten Ebene bietet Log4j hier den SocketAppender, der die Nachrichten beliebig formatiert an einen geeigneten Empfänger sendet. Bestandteil der Log4j2-Distribution ist ein Socket-Server-Gegenstück, das serialisierte LogEvents erwartet und in das eigene Log-System weiterleitet. Hier gelangen Nachrichten der verschiedenen Server in einem Log4j2-Logger.

Für den Transport können UDP oder TCP eingesetzt werden. Letzteres auch via TLS verschlüsselt. Wenn Log-Nachrichten nicht verloren gehen dürfen, muss TCP beziehungsweise TLS (bei Log4j noch als „SSL" bezeichnet) gewählt werden.

Innerhalb des Service muss der SocketAppender lediglich seinen Empfänger kennen. Die folgende Konfiguration

```
<Socket name="socket" host="${env:LOG_HOST}" port="${env:LOG_PORT">
  <SerializedLayout />
</Socket>
```

würde diese etwa aus Environment-Variablen beziehen. `SerializedLayout` sorgt dafür, dass die Log-Nachrichten nicht weiter formatiert, sondern als serialisierte Java-Objekte übertragen werden.

Die Dokumentation für die Server-Implementierung von Log4j2 ist in den Javadocs versteckt. Grundsätzlich startet man den Server mit

```
$ java org.apache.logging.log4j.core.net.server.SOCKETSERVER PORT CONFIG_FILE
```

Hierbei ist `SOCKETSERVER` je nach gewünschtem Protokoll UdpSocketServer, TcpSocketServer oder SecureTcpSocketServer. `PORT` ist der Port, an dem der Server Nachrichten erwartet, und `CONFIG_FILE` ist eine Log4j2-Konfigurationsdatei, die steuert, wie die empfangenen Nachrichten weiter protokolliert werden sollen.

syslog

Häufig werden syslog oder rsyslog eingesetzt, um Log-Nachrichten zu verwalten und zu verteilen. Der SyslogAppender von Log4j2 ist ein spezieller SocketAppender, der das syslog-Protokoll – in verschiedenen Versionen – unterstützt.

Ähnlich zum SocketAppender muss für den SyslogAppender minimal der Empfänger konfiguriert werden

```
<Syslog name="bsd" host="${env:LOG_HOST}" port="514" protocol="TCP"/>
```

Weitere Parameter steuern das konkrete Protokoll, die TLS-Konfiguration und Ähnliches.

Besonderen Charme gewinnt die syslog-Lösung dadurch, dass Server wie logstash oder splunk und Cloud-Lösungen wie [Loggly] als syslog-Server auftreten können. Konfiguriert man den SyslogAppender so, dass ein solcher Service als Ziel angegeben wird, dann landen Log-Nachrichten ohne weiteren Umweg direkt im Indexer der Wahl.

Fazit

Log4j2 erreicht durch die Verwendung des LMAX-Disruptors bei vielen parallelen, asynchronen Log-Produzenten einen sehr hohen Durchsatz an Log-Nachrichten, der weit über das hinausgeht, was mit der Vorgängerversion erreichbar gewesen wäre. Darüber hinaus lässt sich mit Log4j2 sicherstellen, dass keine Log-Nachricht unerkannt verloren geht, wodurch Audit-Logging möglich wird.

Log4j2 bietet Bindings für die gängigen Logging-Fassaden in separaten JARs, die im Idealfall bei Bedarf einfach zum Klassenpfad hinzugefügt werden können. Wie wir am Beispiel von Hibernate4 gezeigt haben, genügt das aber nicht immer. Sind erst einmal alle bestehenden Komponenten und Bibliotheken einer Anwendung auf die asynchronen Logger von Log4j2 umgestellt, kann das die Performance bei Log-intensiven Anwendungen deutlich steigern. Es lohnt sich also, einen Blick auf die neue Version des Klassikers unter den Log-Frameworks zu werfen.

Links

[12FactorApp] `http://12factor.net/logs`
[CommonsLogging] `http://commons.apache.org/proper/commons-logging/`
[Ghad14] Ph. Ghadir, Micro-Services in Java realisieren – Teil 1, in: JavaSPEKTRUM, 4/2014; `http://www.sigs-datacom.de/fileadmin/user_upload/zeitschriften/js/2014/04/ghadir_JS_04_14_uVvU.pdf`
[Hung11] M. Hunger, Disruptorangriff: Hochperformanter Java-Code bringt Prozessoren an ihr Limit?, in: JavaSPEKTRUM, 6/2011
[LMAXDisruptor] `http://lmax-exchange.github.io/disruptor/`
[Log4j] `http://logging.apache.org/log4j/`
[Log4j2Performance] `http://logging.apache.org/log4j/2.x/manual/async.html#Performance`
[logback] `http://logback.qos.ch/`
[Loggly] `https://www.loggly.com/`
[logstash] `http://logstash.net/`
[SLF4J] Simple Logging Facade for Java, `http://slf4j.org/`
[splunk] `http://www.splunk.com/`

Stefan Bodewig arbeitet als Senior Consultant bei der innoQ und entwickelt seit vielen Jahren Software auf der JVM oder der .NET-Plattform. Er ist Mitglied der Apache Software Foundation und Committer bei diversen Open-Source-Projekten.
E-Mail: stefan.bodewig@innoq.com

Phillip Ghadir baut am liebsten tragfähige, langlebige Softwaresysteme. Er ist Mitglied der Geschäftsleitung bei innoQ und hat sich früh auf Architekturen für verteilte, unternehmenskritische Systeme spezialisiert. Darüber hinaus ist er Mitbegründer und aktives Mitglied des iSAQB, des International Software Architecture Qualification Board.
E-Mail: phillip.ghadir@innoq.com