Gutes Timing

Java EE Scheduler mit skalierender Verarbeitung

Stefan Jäger, Ivo Kronenberg, Michael Heide Qvortrup

Zeitgesteuerte Auftragsverarbeitung ist ein wichtiges Element in Enterprise-Applikationen. Im folgenden Architekturvorschlag wird aufgezeigt, wie man durch geschickte Kombination diverser Java EE-Techniken einen zu Java EE konformen Scheduler bauen kann, der Aufträge persistent ablegt, in der Verarbeitung skaliert und ausfallsicher ist.

Einführung

Enterprise-Applikationen erfordern eine zeitgesteuerte Auftragsverarbeitung. Dabei sind Aufträge zu einem bestimmten Zeitpunkt, und zwar mit einer gewissen Genauigkeit, zu starten. Diese Aufträge können technischer oder fachlicher Natur sein.

Nehmen wir als Beispiel einen etwas größeren Webshop. Dieser Webshop ist als Java EE-Applikation realisiert, läuft in einem Cluster und hat 200 000 Bestellungen pro Tag. Nach Eingang einer Bestellung sollte nach 30 Minuten geprüft werden, ob in der Logistiksoftware die bestellten Artikel auch gebucht und versendet worden sind. Ist dies nicht der Fall, wird ein Mitarbeiter darüber informiert, welcher die nötigen Schritte einleitet und die Bestellung manuell weiterführt.

In diesem einfachen und fiktiven Beispiel werden bereits einige Anforderungen an die Applikation aufgezeigt:

- Für jede Bestellung muss nach 30 Minuten ein Auftrag ausgeführt werden.
- Die Überprüfung ist möglichst genau nach 30 Minuten durchzuführen.
- Nach einem Deployment müssen die angemeldeten Aufträge noch immer vorhanden sein.
- Die Ausführung muss ausfallsicher sein. Fällt ein Knoten aus, verarbeitet ein anderer Knoten die bestehenden Aufträge.

Warum EJB-Timer nicht ausreichen

Versucht man das Problem ausschließlich über Enterprise-JavaBeans(EJB)-Timer [UTS] zu lösen, wird schnell ersichtlich, dass diese nicht alle Anforderungen erfüllen. Zum einen gilt die Best Practice, dass vorzugsweise wenige EJB-Timer parallel laufen. Somit sollte nicht für jede Bestellung ein EJB-Timer gestartet werden. Weiter überstehen EJB-Timer zwar Abstürze, jedoch nicht Deployments. Alle registrierten Aufträge gehen zum Zeitpunkt des Deployments verloren. Und zu guter Letzt definiert die Java EE-Spezifikation nicht, wie die EJB-Timer in einem Cluster lauffähig sind.

Mit Hilfe eines einzigen EJB-Timers kann jedoch eine generische Lösung realisiert werden, die alle obigen Anforderungen erfüllt, wenn man diesen Timer mit weiteren Java EE-Techniken, wie Messaging und Message-Driven Beans, kombiniert.

Grundprinzip

Abbildung 1 gibt eine Übersicht über das hier vorgestellte Architekturkonzept. Aufträge, so genannte Jobs, können zunächst registriert werden. Dabei wird ein Job definiert durch die drei Elemente:

- Der *Zeitpunkt* definiert, wann der Job verarbeitet werden muss.
- Die *Identifikation* ist eine fachliche, zum Job gehörende Identifikation und dient der späteren Wiedererkennung. Dies kann z. B. die Bestellnummer sein.
- Die *Worker Queue* ist die Verarbeitungsstelle und gibt an, wer den Job verarbeiten soll. In unserem Fall ist das beispielsweise die Komponente, welche die Bestellung bei der Logistiksoftware überprüft. Es ist somit möglich, mehrere unterschiedliche Worker Queues für verschiedene Arten von Jobs einzurichten.

Ein Job wird angemeldet, indem er als JMS-Nachricht an eine Queue gesendet wird [JMS]. Eine Message-Driven Bean (MDB) nimmt diese Jobs entgegen und speichert sie in einer Datenbank ab. Ein EJB-Timer selektiert regelmäßig – z. B. alle 10 Sekunden – aus der Datenbank abgelaufene Jobs. Abgelaufene Jobs werden an die entsprechende Worker Queue weitergeleitet und aus der Datenbank gelöscht.

Asynchrone Verarbeitung

Der Schlüssel dieses Designs liegt in der asynchronen Verarbeitung. Das Anmelden von Jobs ist dank JMS asynchron und blockiert damit die aktuelle Verarbeitung (z. B. die laufende Bestellung) in keiner Weise.

Die einzige Aufgabe des EJB-Timers ist es, abgelaufene Jobs aus der Datenbank zu lesen und an eine Worker Queue zu senden. Dank der asynchronen Weiterleitung der Jobs ist die Jobverwaltung frei von der eigentlichen Verarbeitung der Jobs. Der Timer wird nicht durch eine lang andauernde Jobverarbeitung blockiert. Diese findet in den Message-Driven Beans statt, welche die unterschiedlichen Worker Queues konsumieren. Stellt man fest, dass bei einer Worker Queue mehr Jobs anliegen, als verarbeitet werden können, so kann man einfach die Anzahl der Bean-Instanzen erhöhen. Dadurch skaliert die Auftragsverarbeitung sehr gut.

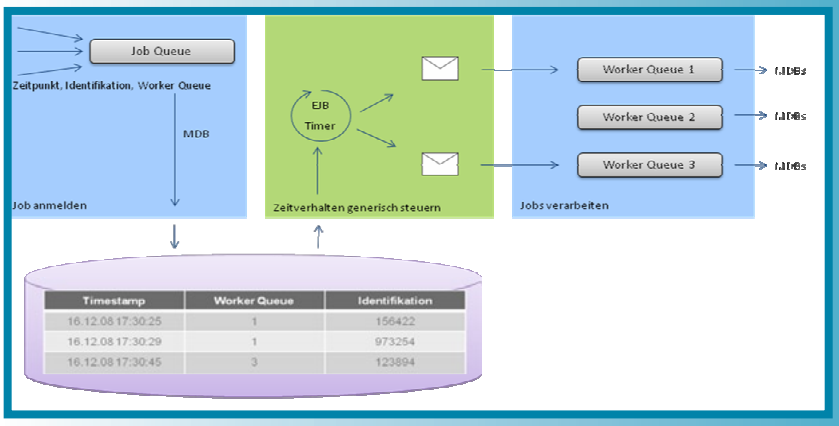

Abb. 1: Architekturkonzept

Transaktionsmanagement

Um zu erreichen, dass keiner der Jobs verloren geht, wird mit verteilten Transaktionen (XA) [CMT] gearbeitet. Der Transaktionsmanager stellt sicher, dass nur Transaktionen abgeschlossen werden, welche von allen beteiligten Ressourcen (Datenbank, JMS-Server) erfolgreich abgearbeitet werden können.

Das Anmelden von Jobs hängt normalerweise mit der zu diesem Zeitpunkt ausgeführten Aktion zusammen. Wenn beispielsweise die Bestellung nicht komplett ausgeführt werden konnte, darf auch der Job nicht angemeldet werden. Dies ist in Java EE einfach mit einer XA-Transaktion zu erreichen.

Etwas komplizierter sieht es bei der Verarbeitung von abgelaufenen Jobs mit dem EJB-Timer aus (s. Listing 1). Alle abgelaufenen Jobs werden innerhalb einer XA-Transaktion an die Worker Queue versendet und aus der Datenbank gelöscht. Diese XA-Transaktion kann nur dann erfolgreich beendet werden, wenn alle beteiligten Ressourcen die Verarbeitung abschließen können. Ist dies nicht der Fall (wenn beispielsweise eine einzige Worker Queue nicht erreichbar ist), dann bedeutet das, dass kein einziger Job mehr weitergeleitet wird. Um in einem solchen Fall die Jobs mit einer lauffähigen Worker Queue trotzdem zu verarbeiten, wird auf eine Einzelverarbeitung der Jobs umgestellt. Dabei wird für jeden Job eine neue Transaktion gestartet. Innerhalb dieser Transaktion wird die JMS-Nachricht versendet und der Datenbankeintrag gelöscht.

```
@Timeout
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public void timeout() {
  try {
    // Auslesen aller Jobs
    List<Job> allExpiredJobs = ...

    // über die eigene Bean-Instanz Massenverarbeitung starten
    // (notwendig, damit im Fehlerfall das Rollback gesetzt wird)
    bean.verarbeiteJobs(allExpiredJobs);
  } catch {
    // Fehler ist aufgetreten, starte Einzelverarbeitung
    for(Job expiredJob : allExpiredJobs) {
      try {
        // über die eigene Bean-Instanz Einzelverarbeitung starten,
        // notwendig, damit das TransactionAttribute eine Wirkung hat
        bean.verarbeiteJobEinzeln(expiredJob);
      } catch(Exception e) {
        // Fehler nur loggen, damit die for-Schleife weiterläuft
      }
    }
  }
}

@TransactionAttribute(TransactionAttributeType.REQUIRED)
public void verarbeiteJobs(List<Job> allExpiredJobs) {
  // alle JMS-Nachrichten versenden,
  // alle Datenbankeinträge mit einer Datenbankinteraktion löschen
}

// Einzelverarbeitung startet eine neue XA-Transaktion
@TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
public void verarbeiteJobEinzeln(Job expiredJob) {
  // JMS-Nachricht an Worker Queue senden,
  // Datenbankeintrag löschen
}
```

Listing 1: Verarbeitung verteilter Transaktionen

Diese Verarbeitung hat den Vorteil, dass sie im Normalfall ziemlich schnell ist. Gibt es einen Teilausfall von Worker Queues, werden die noch funktionierenden dennoch bedient. Die Einzelverarbeitung beeinflusst jedoch durch die erhöhte Anzahl an Datenbankinteraktionen die Performance negativ.

Redundanz und Parallelität

Um eine hohe Verfügbarkeit des hier vorgestellten Mechanismus zu gewährleisten, muss in irgendeiner Form Redundanz erzeugt werden, damit bei einem Ausfall von Hard- oder Software das System weiter funktioniert. Im Enterprise-Applikationsumfeld wird dies typischerweise mit einem Cluster erreicht. Die gesamte Asynchronität – Messaging und Message-Driven Bean – kann in einem Cluster problemlos ausfallsicher betrieben werden. Bei dem EJB-Timer muss dies differenziert gesehen werden. Soll auf eine proprietäre Lösung von Applikationsserver-Produkten verzichtet werden, muss ein Weg für ein Failover-Behandlung mit mehren EJB-Timern gefunden werden.

Wird ein EJB-Timer in einem Cluster auf mehreren Knoten gestartet, laufen diese unabhängig voneinander. Jeder EJB-Timer wird die gleiche Arbeit quasi parallel ausführen, was für unser Design unerwünscht ist.

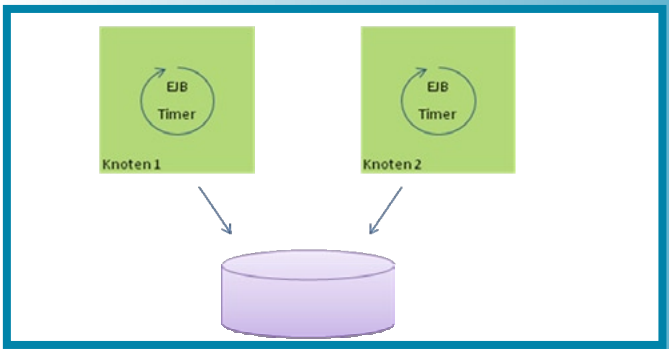

Abb. 2: Timer-Synchronisation über Datenbank

Mit einer Überprüfung, ob bereits ein anderer EJB-Timer die Jobs abarbeitet, kann dieses Problem gelöst werden. Als Synchronisationsmittel eignet sich die Datenbank (s. Abb. 2) am besten, da sie eine gemeinsame Ressource darstellt. In einer Tabelle wird ein Datensatz erstellt, in welchem der zuletzt ausführende Knoten mit einem Zeitpunkt der Abarbeitung gespeichert wird. Dabei ist zu beachten, dass die Systemzeit der Knoten unterschiedlich sein könnte, weshalb hier mit der Zeit der Datenbank gearbeitet werden muss.

Abbildung 3 zeigt die Failover-Behandlung. Nachdem die Intervallzeit des EJB-Timers abgelaufen ist, wird zunächst der Datensatz in der Failover-Tabelle für andere Datenbank-Sessions gesperrt (row lock mit SELECT … FOR UPDATE NO WAIT). Arbeitet bereits eine andere Instanz, kann der Datensatz nicht exklusiv beansprucht werden. In diesem Fall wird keine Abarbeitung der Jobs vorgenommen. Konnte jedoch der Datensatz gesperrt werden, wird im nächsten Schritt überprüft, ob die letzte Ausführung vom gleichen Knoten ausging. Falls dem so ist, wird direkt mit der Abarbeitung der Jobs begonnen. War der zuletzt ausführende Knoten nicht derselbe, wird die Abarbeitung der Jobs nur übernommen, wenn die Zeitspanne zur letzten Ausführung größer als eine Failover-Zeitspanne ist. Somit wird vermieden, dass die Abarbeitung der Jobs sehr kurz aufeinander folgt und ein ständiges Übernehmen der Abarbeitung durch einen anderen Knoten entsteht (Ping-Pong-Effekt).

Fazit

Diese Lösung für die Verarbeitung von zeitgesteuerten Aufträgen bietet viele Vorteile. Einerseits ist die zeitliche Verarbei-

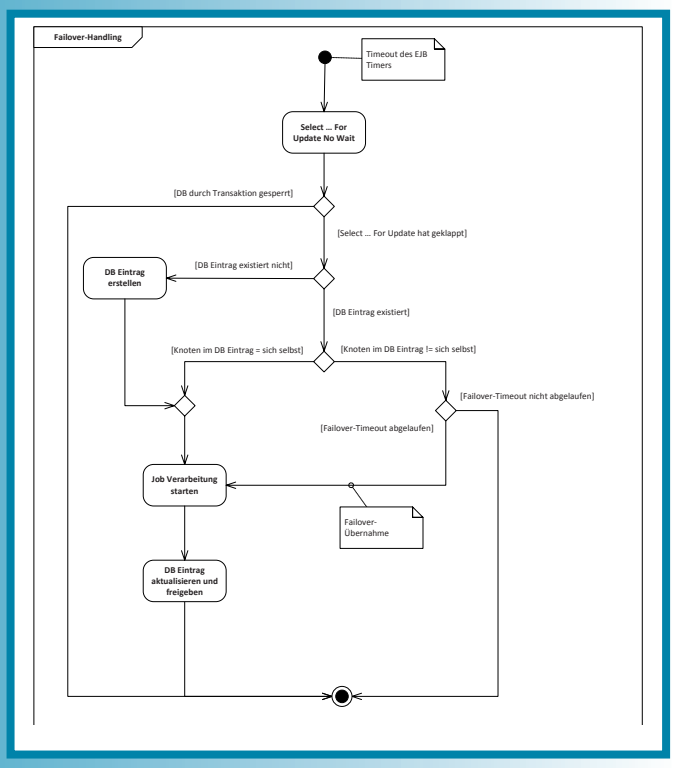

Abb. 3: Failover-Behandlung

tung selbst generisch ausgelagert. Dadurch gibt es keine Vermischung von fachlichen Codeteilen mit dem technischen Teil der Jobverwaltung. Der Scheduler kann für verschiedene Aufgaben verwendet werden. Des Weiteren ist die Jobverarbeitung nicht davon abhängig, wie lang die Ausführungszeit eines Jobs ist. Wächst die Anzahl Jobs, so kann man ganz einfach die Anzahl der Message-Driven Beans an einer Worker Queue erhöhen. Damit ist die Verarbeitung skalierbar. Zudem können keine Jobs verloren gehen, da diese persistent abgelegt werden. Neue Deployments oder Abstürze der Applikation haben nicht zur Folge, dass Jobs verloren gehen.

Literatur und Links

[CMT] Transaktionen unter Java EE,
`http://java.sun.com/javaee/5/docs/tutorial/doc/bncij.html`
[JMS] JMS mit Java EE,
`http://java.sun.com/javaee/5/docs/tutorial/doc/bncdq.html`
[UTS] Timer-Service unter Java EE,
`http://java.sun.com/javaee/5/docs/tutorial/doc/bnboy.html`

Stefan Jäger studierte Informatik an der FH Luzern und ist heute Software-Ingenieur bei der Zühlke Engineering AG in Bern. Im aktuellen Projekt beschäftigt er sich mit der Umsetzung von Konzepten für hochskalierbare Systeme.
E-Mail: stefan.jaeger@zuehlke.com.

Ivo Kronenberg hat an der FH Bern Informatik studiert und ist Software-Ingenieur bei der Zühlke Engineering AG in Bern. Zurzeit ist er im gleichen Projekt wie Stefan Jäger für die Umsetzung von Konzepten für hochskalierbare Systeme verantwortlich. Er war auch an der Definition der Architektur beteiligt.
E-Mail: ivo.kronenberg@zuehlke.com.

Michael Heide Qvortrup studierte Informatik an der ETH Zürich und ist heute Software Engineering Berater bei der Zühlke Engineering AG. Nebenberuflich ist er Lehrbeauftragter der FH Rapperswil für Software-Architektur im NDS (Nachdiplomstudium) Software Engineering.
E-Mail: michael.qvortrup@zuehlke.com.