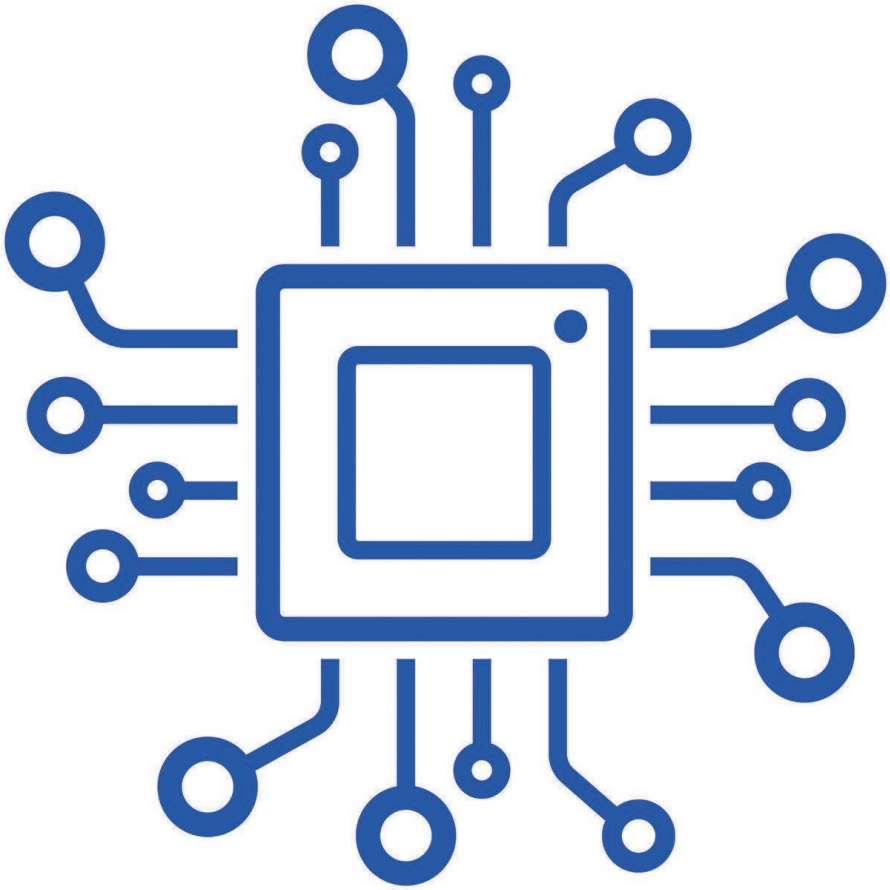

Hallo Microservices, gibt's euch auch transaktional?

Matthias Koch und Markus Grabert, sidion

Es ist 2019. Microservices sind kein Novum mehr und längst in viele Bereiche durchgedrungen. Allerdings hat der Ansatz, viele verteilte kleine Services zu verwenden, auch seine architektonischen Herausforderungen. In den letzten Jahren haben wir gelernt, mit Consumer Driven Contracts, Service Discovery und API-Gateways umzugehen. Für all diese Probleme gibt es mittlerweile fertige Lösungen aus der Werkzeugkiste. Kommt hingegen Datenkonsistenz (Eventual Consistency) über Servicegrenzen hinweg zur Sprache, dann sieht es anders aus. Zuallererst wird überlegt, ob Transaktionen in diesem Kontext überhaupt notwendig sind und nicht ein Indiz für zu harte Kopplung darstellen. Aber: Transaktionen wird es weiterhin geben und das rein fachlich bedingt.

In diesem Artikel wollen wir auf einen alternativen Weg hinweisen, der es ermöglicht, logische Transaktionen in Microservice-Architekturen abzubilden: das Saga-Pattern, das mithilfe von kompensatorischen Operationen eine Art Rollback nachbildet.

Abschließend soll ein Ausblick auf das Proposal der Spezifikation „Long Running Actions for MicroProfile" gegeben werden. Deren API verspricht die Koordination von Services zu vereinheitlichen und damit die Implementierung zu vereinfachen. Hierbei wird ein global konsistenter Zustand sichergestellt, ohne dass Locks auf Daten benötigt werden.

Was sind Transaktionen?

Transaktionen können als Folge von Aktionen beschrieben werden, die als logische Einheit betrachtet werden müssen. Das heißt, entweder alle Aktionen werden in ihrer Gesamtheit umgesetzt oder keine der Aktionen. Auf diese Weise soll in jedem Fall ein konsistenter Zustand sichergestellt werden. Üblicherweise werden Transaktionen in Prozessen eingesetzt, bei denen Daten- oder

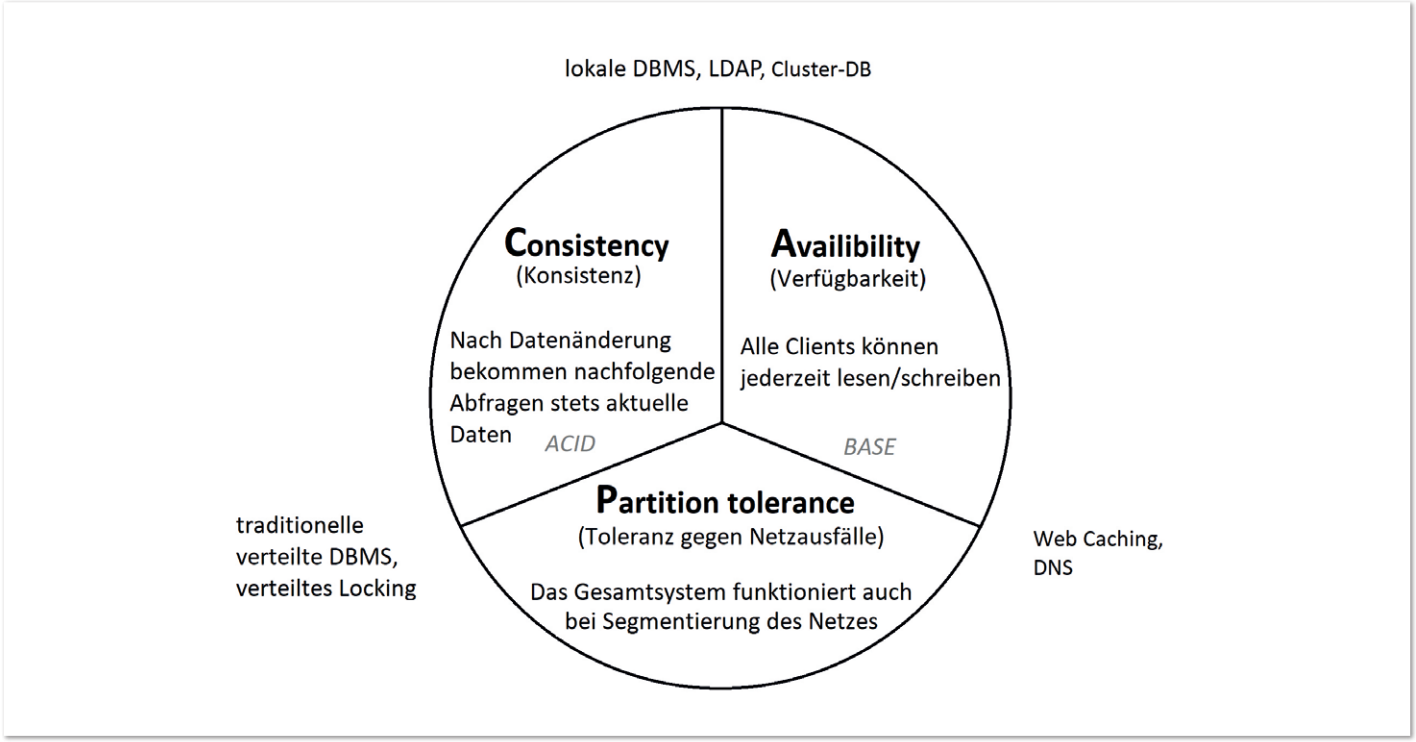

Abbildung 1: Das CAP-Theorem (Quelle: Matthias Koch)

Zustandsinkonsistenzen – verursacht durch eine partielle Umsetzung der Aktionen – große Risiken bergen würden. Das Spektrum der möglichen Risiken reicht von irreversiblen Datenverlusten, Blockierung von Folgeprozessen und Services bis hin zu eingehenden finanziellen Verlusten und so weiter.

Gibt es keine Probleme, wenn nur ein paar und nicht alle Aktionen umgesetzt werden, dann liegt auch keine Transaktion im Sinne der Definition vor. Am Beispiel Bürgerbüro wird dies deutlich: Wer dort ein Ticket zieht, aber nicht zum Schalter geht, wenn die eigene Nummer aufgerufen wird, hat in der Regel nicht mit nennenswerten Auswirkungen zu rechnen. Eine Transaktion ist für diesen Prozess also nicht nötig. Anders ist es bei Banküberweisungen: Wenn das Geld dem Empfängerkonto gutgeschrieben wird, dann sollte immer gleichzeitig das Senderkonto belastet werden. Auch bei Online-Flugbuchungen sollte sichergestellt sein, einen Sitz reserviert zu bekommen, wenn dafür bezahlt wurde. Beides sind Prozesse, bei denen Transaktionen sinnvoll zum Einsatz kommen.

Wie werden Transaktionen nun in IT-Systemen umgesetzt?

Aktionen einer Transaktion gehen immer mit einer Daten- oder Zustandsänderung einher. Diese Änderungen müssen wiederum sofort atomar und konsistent umgesetzt werden, damit die Folge-Aktionen darauf aufbauen können. Bei Monolithen ist es relativ einfach, einen konsistenten Zustand zu garantieren. Dies unterstützen zum Beispiel die traditionellen relationalen DBMS auch schon seit langer Zeit; das Stichwort heißt „ACID“: Atomicity, Consistency, Isolation und Durability. Bei verteilten Systemen, insbesondere Microservices, gibt es jedoch zusätzliche Herausforderungen: Wie kann garantiert werden, dass innerhalb einer Transaktion alle Aktionen auf den beteiligten Systemen wirklich ausgeführt werden? Schließlich sind die Systeme nicht immer zuverlässig erreichbar (Netzwerkprobleme, Überlastung).

Das CAP-Theorem

Mit der Problematik „Datenverarbeitung in verteilten Systemen“ befasste sich 2000 auch Dr. Eric Brewer und stellte sein CAP-Theorem *(siehe Abbildung 1)* in einer Keynote vor *[1]*. Später wurde das CAP-Theorem nochmals formalisiert und bewiesen *[2]*.

Das Theorem in kurz: Bei Systemen, die Daten mit anderen Prozessen teilen, gibt es drei wesentliche, beeinflussende Faktoren: Konsistenz (C für Consistency), Verfügbarkeit (A für Availibility) und Toleranz gegen Netzwerk-Segmentierung (P für Partition Tolerance). Zwischen Konsistenz und Verfügbarkeit muss jedes Mal erneut entschieden werden. Dies wird im Folgenden ausgeführt.

Die Qual der Wahl

Leider gibt es bei verteilten Systemen in Netzwerken immer die Möglichkeit von Netzwerkstörungen. So kann es passieren, dass Teile des Netzwerks abgetrennt werden und ein verteiltes System plötzlich Kommunikationsprobleme hat.

Dann gibt es nur noch zwei Optionen: Die erste ist, auf Konsistenz zu setzen und nur noch mit einem ausgewählten Teil des Systems und der Daten weiterzuarbeiten. Der Rest des Systems wird in diesem Fall deaktiviert, bis die Netzwerkprobleme wieder behoben sind. Abschließend wird alles synchronisiert.

Alternativ ist das verteilte System auf Verfügbarkeit ausgelegt und die unterschiedlichen, abgetrennten Bereiche arbeiten erst einmal unabhängig voneinander weiter. Dies führt längerfristig unweigerlich zu Dateninkonsistenzen innerhalb des Gesamtsystems, die zu einem späteren Zeitpunkt behoben werden müssen. Dies ist oft nicht automatisiert, sondern nur manuell möglich. Die bisherigen Ansätze, verteilte Datenspeicherung zu implementieren, waren daher meist basierend darauf, wie die ACID-Konsistenz-Garantie auf mehrere verteilte DBMS ausgebreitet werden kann.

XA-Transaktionen

X/Open XA (eXtended Architecture) ist wohl das meistverbreitete Protokoll, um verteilte Transaktionen zu implementieren. Es wurde schon 1991 von X/Open, jetzt OpenGroup, veröffentlicht *[3]*, also lange vor SOA und Microservices. Bei Ausführung von verteilten Datenbank-Transaktionen wird immer noch das bereits erwähnte ACID-Prinzip gewährleistet. Wie funktioniert das? XA verwendet dazu das 2-Phase Commit Protocol (2PC). Vereinfacht gesagt werden in der ersten Phase die Datenänderungen bei allen beteiligten DBMS vorbereitet. Wenn alle ein „Okay" melden, folgt die Phase zwei, in der die Datenänderungen wirklich durchgeführt werden.

Geeignet für Microservices?

Leider ist der Prozess ab dem „Okay" blockierend, was es bei Microservices unbedingt zu vermeiden gilt, da dies mit Einschränkungen in der parallelen Verarbeitung verbunden ist. Außerdem führt die 2PC-Kommunikation zu höheren Latenzen. Dazu kommt, dass Ressourcen in verteilten Systemen nicht immer verfügbar sein müssen, was mit einer Fehlerbearbeitung einhergeht, die sehr komplex ist. Dies macht die Fehleranalyse wiederum teuer, und das Ganze wird schwer testbar. Schließlich werden die Systeme eng gekoppelt, was dem Prinzip der Isolation und losen Kopplung von Microservices entgegensteht.

Geeignet für Transaktionen?

Dem 2PC geschuldet, gibt es bei XA-Transaktionen eine komplizierte Fehlerbehandlung, da es verschiedene Fehlerquellen und -ursachen geben kann. In einigen Fehlerfällen muss sogar manuell eingegriffen werden, um die Daten zu korrigieren. Eine atomare Konsistenz ist also nicht immer garantiert. Die unterschiedliche Verfügbarkeit der Services und das heterogene Laufzeitverhalten der Systeme sind weitere Faktoren, die einen möglichen Einsatz einschränken können, vor allem bei Transaktionen mit Anforderungen an starke Konsistenz oder Echtzeit, wie zum Beispiel beim Aktienhandel an der Börse.

Zusammengefasst sind XA-Transaktionen für Microservices nicht besonders geeignet. Letztendlich wurden sie auch rund 20 Jahre vor der Einführung von Microservices konzipiert und eher auf monolithische Systeme zugeschnitten.

Idealerweise bräuchte es also ein Framework, das besser zum Microservice-Ansatz und zur Microservice-Architektur passt. Der nachfolgende Abschnitt gibt einen Überblick darüber, was dafür konkret benötigt wird.

Anforderungen eines Transaktions-Frameworks für Microservices

- Zunächst sollen keine Datensätze gelockt werden. Das würde nicht nur die parallele Verarbeitung behindern, sondern auch potenziell das Risiko mit sich bringen, dass Daten gegebenenfalls für immer gelockt bleiben und damit gar nicht mehr verfügbar sind.
- Auch die lose Kopplung der Services soll nicht aufgegeben werden. Das System soll weiterhin einfach änderbar und erweiterbar sein.
- Da in verteilten Systemen die Zustellung einer Nachricht nie garantiert werden kann, braucht es auch dafür Mechanismen, mit denen sich das leicht handhaben lässt.
- Den Autoren schwebt ein generischer Ansatz vor, der die Möglichkeit bietet, die Datenkonsistenz zu erhalten, und der auch bei der nächsten gebauten Anwendung wieder gleichartig funktioniert. Ziel ist es, das sprichwörtliche Rad nicht bei jeder Anwendung wieder neu erfinden zu müssen.

Zeit sich umzuorientieren?

Bevor diese Frage beantwortet werden kann und geschildert wird, ob und wie das geht, erst einmal zurück zum CAP-Theorem. Der bisher gewählte Ansatz, Transaktionen strikt nach dem ACID-Prinzip durchzuführen und die atomare Datenkonsistenz zu priorisieren, geht mit einigen Nachteilen einher, die im Microservices-Umfeld nicht gewünscht sind.

Wer sich einmal etwas vom ACID-Prinzip entfernt, stößt früher oder später auf den Begriff „BASE". Er stellt quasi den Gegenpol zu ACID dar. Die Abkürzung setzt sich wie folgt zusammen:

- **B**asically **A**vailable
- **S**oft-State
- **E**ventual Consistency

„Eventual Consistency" klingt auf den ersten Blick nach einem Zustand, der in IT-Systemen nicht erstrebenswert ist. Schließlich sollen eine Vorhersagbarkeit und Garantien erreicht werden. Bei tieferer Recherche wird jedoch deutlich, dass sich das Prinzip der „Eventual Consistency" in der IT-Welt schon verbreitet hat: Viele NoSQL-Datenbanken, Elasticsearch Index und nicht zuletzt das DNS-System basieren und vertrauen darauf. Es stellt sich die Frage, ob es denn überhaupt immer notwendig ist, dass verteilte Transaktionen sofort – und atomar als Einheit – umgesetzt werden. Genügt es nicht bisweilen sicherzustellen, dass (innerhalb eines Zeitraumes) verteilte Transaktionen wirklich als Gesamtheit durchgeführt wurden?

Sagas

Es gibt tatsächlich eine Beschreibung, wie eine verteilte Transaktion auf so eine Art abgebildet werden kann: das Saga-Pattern! Zuerst soll ein Blick auf die Ursprünge des Saga-Patterns geworfen werden. Der Begriff „Saga" wurde in einer wissenschaftlichen Abhandlung von Héctor García-Molina und Kenneth Salem bereits 1987 als Alternative zu Long Lived Transactions (LLT) in der Datenbank-Programmierung vorgeschlagen *[4]*. Das Problem dabei war, dass LLT-Datenbank-Tabellen blockierten und dadurch andere (parallele) Prozesse behinderten. Die Idee: Die LLT sollte zur „Saga" werden, indem sie in (möglichst) separate Teil-Transaktionen getrennt wird, die sequenziell oder auch parallel verarbeitet werden können. Diese Datenveränderungen dürfen aber nur dauerhaft in der DB bestehen, wenn sie in ihrer Gesamtheit umgesetzt wurden.

Der Vorteil, der sich durch Aufteilung in einzelne Teil-Transaktionen ergibt, ist die Möglichkeit für andere Anwendungen, auf Datenbank-Tabellen zuzugreifen, die ansonsten durch eine LLT längere Zeit blockiert wären.

Der Nachteil besteht in der Komplexität. Falls Teil-Transaktionen einer Saga fehlschlagen, müssen kompensatorische Transaktionen für alle anderen bisher umgesetzten Teile dieser Saga stattfinden, um den Originalzustand wiederherzustellen.

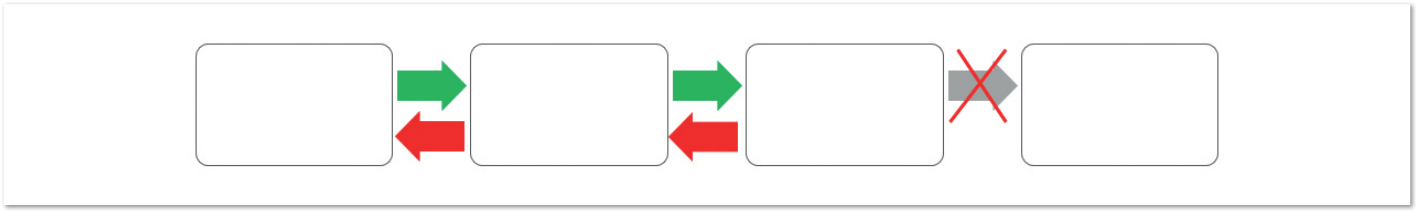

Abbildung 2: Saga Interaction Pattern (Quelle: Matthias Koch)

Saga-Pattern

In seinem Buch „SOA Patterns" von 2012 hatte Arnon Rotem-Gal-Oz schließlich dieses Prinzip auf verteilte Transaktionen angewendet und als „Saga-Pattern" *[5]* vorgestellt. Er beschreibt, dass das Saga-Pattern eigentlich keine neue Erfindung sei, sondern nur eine Anwendung verschiedener schon bestehender „Interaction Patterns". Ab 2013 wurden über das Saga-Pattern und dessen Implementation im SOA- und Microservices-Umfeld vermehrt Diskussionen geführt *[6]*. Es wurden unterschiedliche Vorgehensweisen (Orchestrierung, Choreografie) vorgestellt, wie Sagas implementiert werden können. Darauf wird im späteren Verlauf des Artikels noch eingegangen. 2015 gab es außerdem die Abhandlung „Verteilte Sagas" von McCaffrey, Kingsbury und Dr. Narula *[7]*, bei der eine orchestrierte Saga auf ein verteiltes IT-System angewendet wird.

Welche Ideen stecken nun hinter dem Saga-Pattern?

Beim Saga-Pattern werden naturgemäß kompensatorische Transaktionen durchgeführt, die zu jeder Aktion beziehungsweise jedem Command registriert werden. Im Falle eines Fehlers werden dann alle Transaktionen in inverser Reihenfolge ausgeführt *(siehe Abbildung 2)*. Es soll sichergestellt sein, dass kein möglicher Zustand vergessen wird, betrachtet zu werden. Das würde unweigerlich irgendwann zu einem Fehlverhalten der Anwendung führen.

Angenommen es liegt eine einfache Transaktion vor, die aus vier Einzelaktionen besteht, die sequenziell abgearbeitet werden. Jede Einzelaktion wird immer noch nach dem ACID-Prinzip ausgeführt oder könnte wiederum eine Transaktion sein.
Wie wird eine Transaktion dieser Art mit Sagas implementiert? Und wie orchestriert/choreografiert man so etwas mit Sagas?

Orchestrierung und Choreografie

Es gibt zwei unterschiedliche Vorgehensweisen, um Sagas zu implementieren: Orchestrierung *(siehe Abbildung 3)* und Choreografie *(siehe Abbildung 4)*. Orchestrierung einer Saga bedeutet, dass es eine zentrale Stelle gibt, welche die Fortschritte der einzelnen Aktionen verfolgt und vor allem steuert. Alle Prozesse, die diese Aktionen durchführen, müssen also mit dieser zentralen Stelle in Kontakt stehen und wissen unter Umständen nichts von den anderen beteiligten Aktionen. Bei der Choreografie hingegen müssen die Aktionen selbstständig dafür Sorge tragen, dass sie in der richtigen Reihenfolge ausgeführt werden. Kommunikation zwischen den Prozessen, die die Aktionen durchführen, ist also unbedingt notwendig.

Implementierungen

Derzeit gibt es leider noch nicht viel Auswahl an Frameworks und Werkzeugen, die einen bei der Implementierung des Saga-Patterns unterstützen. Hier sei nur auf das **Eventuate Tram Saga** API *[8]* und das **Axon**-Framework *[9]* verwiesen. Auf beides wollen wir wegen des anspruchsvollen Programmiermodells und der schlechten Kapselung nicht eingehen.

Schließlich gibt es noch den Eclipse Microprofile Standard. Deren Arbeit an Sagas befindet sich noch in der Spezifikationsphase und sieht eine Implementierung mittels sogenannter **Long-Running Actions** (LRA) vor *[10]*. Für Eclipse Microprofile spricht, dass es von

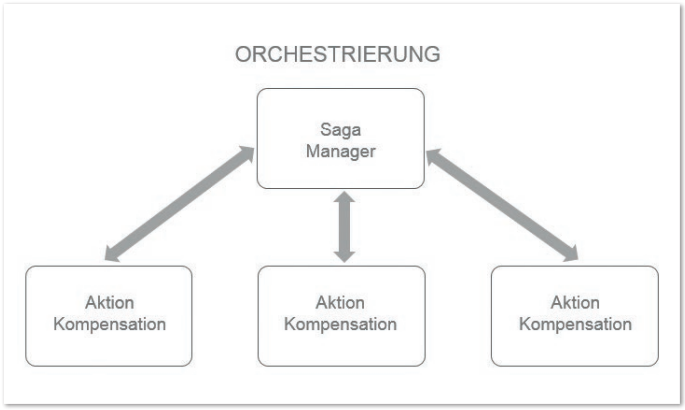

Abbildung 3: Orchestrierung (Quelle: Matthias Koch)

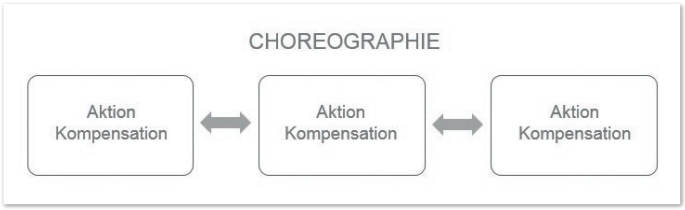

Abbildung 4: Choreografie (Quelle: Matthias Koch)

Abbildung 5: Beispiel Flugbuchung (Quelle: Matthias Koch)

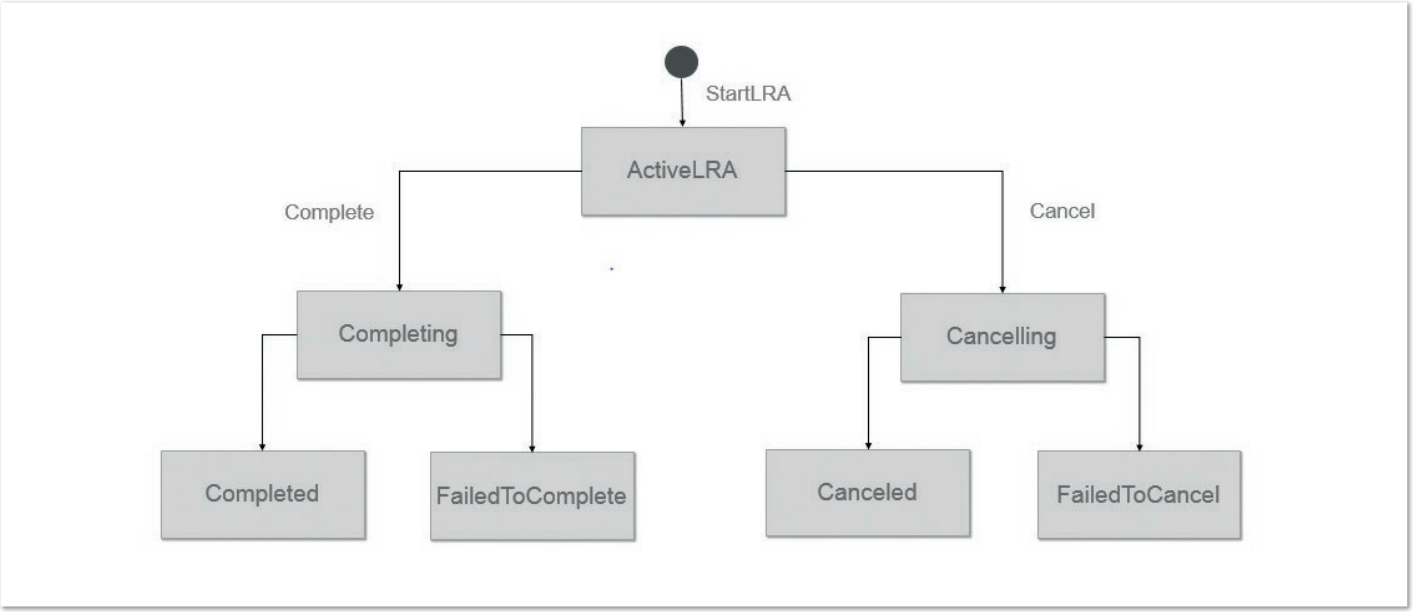

Abbildung 6: LRA-Zustandsmodell (Quelle: Matthias Koch)

vielen namhaften Firmen unterstützt wird und es viele Implementierungen für Microservices (Thorntail, Open Liberty, Payara Micro) dafür gibt. Wenn also die LRA-Spezifikation in den Eclipse Microprofile Standard aufgenommen wird, wird es eine große Auswahl an Implementierungen des Saga-Patterns geben.

Abgrenzung der Zuständigkeiten LRA-Framework vs. Applikation

Da das LRA-Framework die Fachlichkeit der Anwendung nicht kennen kann, muss die Applikation sich weiterhin um einige Dinge selbst kümmern. Daher ist diese Abgrenzung von zentraler Bedeutung. Das LRA-Framework definiert die Zustände und Trigger, die benötigt werden, um eine Transaktion durchzuführen. Die Abfolge der Trigger ist genau definiert. Es wird garantiert, dass die Trigger nicht mehrfach ausgeführt werden, denn das hätte in einem transaktionalen System gravierende Folgen. Das Framework macht keine zeitlichen Zusicherungen, wann die Trigger ausgelöst werden.

Die Applikation hingegen muss für jeden Command (Action) innerhalb einer LRA sicherstellen, dass dieses invertiert werden kann. Bei der Definition des API muss die Kompensierbarkeit bereits mitberücksichtigt und natürlich muss dies auch implementiert werden. In allen Zuständen hat die Anwendung das Verhalten zu definieren. Außerdem hat die Applikation für die Persistenz derjenigen Daten zu sorgen, die für die Kompensation notwendig sind. Dabei muss sichergestellt werden, dass diese Daten auch über einen Neustart des Service hinaus (gegebenenfalls auch durch einen JVM-Crash) verfügbar sind.

Beispiel Flugbuchung

Anhand eines einfachen Beispiels *(siehe Abbildung 5)* soll nun die Funktionsweise einer LRA erklärt werden. Die Beispielanwendung bucht Flüge, aktualisiert die Passagierlisten und stellt schlussendlich die Bezahlung sicher. Jeder Command interagiert mit einem eigenständigen Drittsystem. Daher wurde entschieden, jeden in einen eigenen „bounded context" zu verschieben. Das Payment soll am Ende der Kette stehen, um die Anzahl der stornierten Bezahlungen zu minimieren.

LRA-Zustandsmodell

Das Zustandsmodell einer LRA *(siehe Abbildung 6)* berücksichtigt alle möglichen Fälle und sieht auf den ersten Blick bezüglich der Anzahl der Zustände recht komplex aus. Deswegen werden wir versuchen, dies für die Praxis etwas zu vereinfachen.

Nachdem die LRA begonnen wurde, wird irgendwann im Verlauf der Verarbeitung das Ende eingeleitet. Das kann entweder „complete" oder „cancel" sein. Prinzipiell sollte die Anwendung so geschrieben werden, dass beim „complete" keine Commands mehr ausgeführt werden – somit sollte der Zustand „FailedToComplete" gar nicht eintreten. Keineswegs sollte an dieser Stelle versucht werden, eine XA-Transaktion nachzubilden. Wenn etwas beim „cancel" schiefgehen sollte, müsste in jedem Fall korrigierend eingegriffen und nachbearbeitet werden. Eine jederzeit verfügbare Möglichkeit ist es, die Daten hierzu in eine BackOutQueue zu schreiben. Um den Fokus auf Transaktionen zu halten, wird darauf jedoch nicht weiter eingegangen.

Ähnlich sieht es für einen Participant aus, also einen Service, der an einer LRA teilnimmt *(siehe Abbildung 7)*. Hier liegt die Konzentration ebenfalls auf „complete" und „compensate".

LRA-Annotationen

Das LRA-Framework verwendet für die Implementierung der Sagas Annotationen. Hier nur die wichtigsten:

- **@LRA:** Annotiert die Arbeitsmethode einer LRA
- **@Complete:** Callback für den Fall des Erfolgs einer LRA
- **@Compensate:** Callback im Fehlerfall

Identifiziert wird eine LRA über `LRA_HTTP_CONTEXT_HEADER`. Nur die mit LRA annotierten Services nehmen daran auch teil. Jeder Downstream-Service (vom Request her betrachtet) hat aber potenziell die Möglichkeit, einer LRA beizutreten. Irgendwann terminiert die LRA; mit einem Statuscode 200 wird die LRA erfolgreich beendet. Ein 5xx-Statuscode leitet dann die Kompensation ein. Exceptions werden ebenfalls in einen 5xx verpackt.

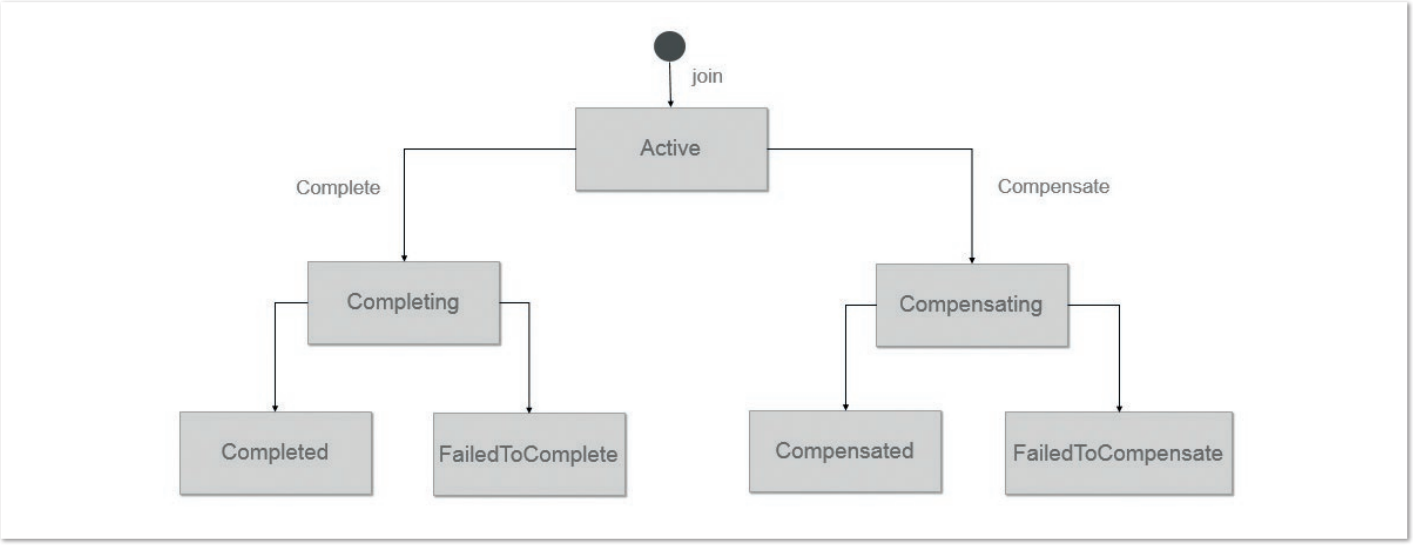

Abbildung 7: Zustandsmodell eines LRA-Participant (Quelle: Matthias Koch)

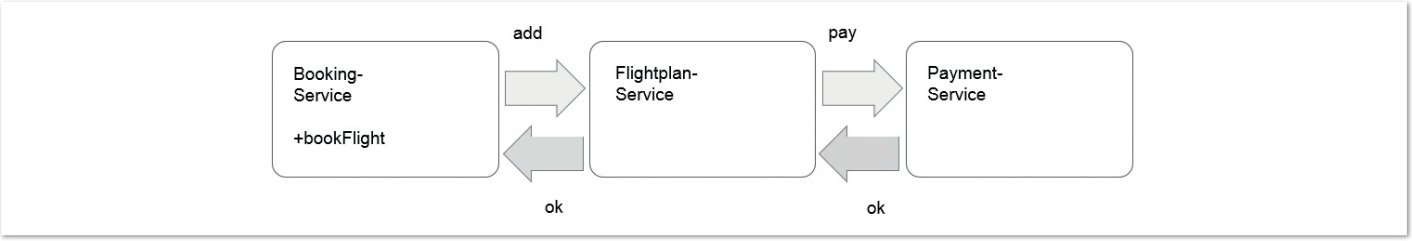

Abbildung 8: Flugbuchung im Erfolgsfall (Quelle: Matthias Koch)

```
@Path("/")
@ApplicationScoped
public class BookingService {

  @LRA(value = LRA.Type.REQUIRES_NEW, timeLimit = 30, timeUnit = ChronoUnit.SECONDS)
  @Path("/book")
  @PUT
  public Response bookFlight(@HeaderParam(LRA_HTTP_CONTEXT_HEADER) String lraId, BookInfo bookInfo) {

    if (book(bookInfo.getFlightInfo())) {

      // Call downstream service.
      addFlightlist(bookInfo);
    }
    return Response.accepted().build();
  }
  ...
```

Listing 1: Start einer LRA (Quelle: Matthias Koch)

```
@Path("/")
@ApplicationScoped
public class FlightplanService {

  @LRA(value = LRA.Type.REQUIRED, timeLimit = 30, timeUnit = ChronoUnit.SECONDS)
  @Path("/add")
  @PUT
  public Response addFlightlist(@HeaderParam(LRA_HTTP_CONTEXT_HEADER) String lraId, BookInfo bookInfo) {

    if (addPassenger(bookInfo.getPassengerInfo())) {

      // Call downstream service.
      payFlight(bookInfo.getPaymentInfo());

      // OK(200) would terminate the LRA. So use ACCEPTED(202) here.
      return Response.accepted().build();
    } else {

      // Finish LRA early.
      return Response.serverError().build();
    }

  }
  ...
```

Listing 2: Downstream-Service innerhalb einer LRA (Quelle: Matthias Koch)

```java
@Path("/")
@ApplicationScoped
public class PaymentService {


  @LRA(value = LRA.Type.REQUIRED)
  @Path("/pay")
  @PUT
  public Response payFlight(@HeaderParam(LRA_HTTP_CONTEXT_HEADER) String lraId, PaymentInfo paymentInfo) {

    if (pay(paymentInfo)) {

      //Initiate completion of the LRA:
      return Response.ok().build();
    } else {

      //Initiate compensation of the LRA.
      return Response.serverError().build();
    }
  }
  ...
```

Listing 3: Das Ende der LRA wird eingeleitet (Quelle: Matthias Koch)

```java
@Complete
@Path("/complete")
@PUT
public Response completeFlightlist(@HeaderParam(LRA_HTTP_CONTEXT_HEADER) String lraId) {

  // Resources can be released here.
  return Response.accepted().build();
}
```

Listing 4: Complete eines Participant (Quelle: Matthias Koch)

```java
@Complete
@Path("/complete")
@PUT
public Response completeBooking(@HeaderParam(LRA_HTTP_CONTEXT_HEADER) String lraId) {

  // Resources can be released here.
  return Response.accepted().build();
}
```

Listing 5: Positiver Abschluss der LRA (Quelle: Matthias Koch)

Abbildung 8 zeigt den Erfolgsfall unseres Beispiels. Der Flug wird gebucht, ein Passagier der Passagierliste hinzugefügt und das Ganze bezahlt. Alles funktioniert fehlerlos.

Jetzt wird es Zeit, den Code anzusehen *(siehe Listing 1)*. Die Werte der LRA-Annotationen sind schon von den EJBs her bekannt – es sind jedoch keine EJBs. Der Timeout wird später noch betrachtet. Nun wird in der Arbeitsmethode der LRA-Header „injected", der die LRA identifiziert. Die Buchung selbst wird durchgeführt und im Erfolgsfall der Downstream-Service, also der Flightplan-Service, gerufen.

Ist auch die Aktualisierung der Passagierliste erfolgreich, wird schließlich die Bezahlung ausgeführt *(siehe Listing 2)*. Wichtig ist, dass hier ein „accepted", also ein 202, zurückgegeben wird. Bei einem 200 würde ja sonst schon das Ende der LRA ausgelöst. Bei einem Fehler wird sofort kompensiert.

Hiermit wird in jedem Fall das Ende der LRA eingeleitet *(siehe Listing 3)*. Je nach Erfolg des Payment endet die LRA als „complete" oder als „cancel". Die LRA soll so einfach gehalten werden, dass im Erfolgsfall nichts mehr zu tun ist *(siehe Listing 4)*. Mit dem „complete" der letzten Action wird die LRA als „completed" beendet *(siehe Listing 5)*.

Sollte nun ein „compensate" ausgelöst werden *(siehe Abbildung 9)*, so müssen immer alle aktiven Actions einer LRA kompensiert werden *(siehe Listings 6, 7* und *8)*. Das kann auch schon früher enden *(siehe Abbildung 10)*, ohne das Payment zu berücksichtigen.

Abbildung 9: Flugbuchung mit Kompensation (Quelle: Matthias Koch)

```java
@Compensate
@Path("/compensate")
@PUT
public Response compensatePayment(@HeaderParam(LRA_HTTP_CONTEXT_HEADER) String lraId) {

  //Service is responsible for availability of data for compensation.
  PaymentInfo paymentInfoCompensate = getPaymentInfo(lraId);
  // Check here if payment was accomplished.
  if (!undoPay(paymentInfoCompensate)) {

    // Can not be resolved immediate.
    backOut(lraId, paymentInfoCompensate);
  }
  return Response.accepted().build();
}
```

Listing 6: Kompensation eines Participant (Quelle: Matthias Koch)

```java
@Compensate
@Path("/compensate")
@PUT
public Response removeFlightlist(@HeaderParam(LRA_HTTP_CONTEXT_HEADER) String lraId) {

  // Service is responsible for availability of data for compensation.
  PassengerInfo passengerInfoCompensate = getPasssengerInfo(lraId);
  // Check if passenger was added first.
  if (!removePassenger(passengerInfoCompensate)) {

    backOut(lraId, passengerInfoCompensate);
  }
  return Response.accepted().build();
}
```

Listing 7: Kompensation eines Participant (Quelle: Matthias Koch)

```java
@Compensate
@Path("/compensate")
@PUT
public Response compensateBooking(@HeaderParam(LRA_HTTP_CONTEXT_HEADER) String lraId) {

  // Service is responsible for availability of data for compensation.
  BookInfo bookInfoCompensate = getBookInfo(lraId);
  if (!unbookFlight(bookInfoCompensate)) {

    backOut(lraId, bookInfoCompensate);
  }
  return Response.serverError().build();
}
```

Listing 8: Abschluss der LRA mit Kompensation (Quelle: Matthias Koch)

Abbildung 10: Flugbuchung mit vorzeitiger Kompensation (Quelle: Matthias Koch)

Abbildung 11: Flugbuchung mit Timeout (Quelle: Matthias Koch)

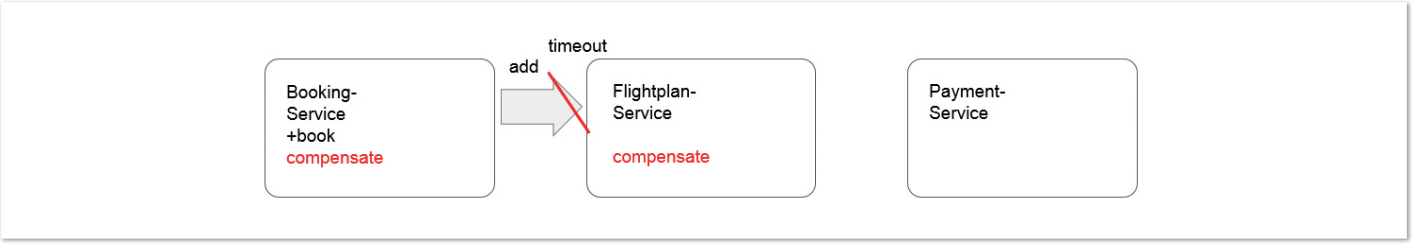

Abbildung 12: Flugbuchung mit Timeout (Quelle: Matthias Koch)

Außerdem kann ein Timeout in einem verteilten System immer auftreten *(siehe Abbildung 11)*. Wichtig ist hierbei, dass auch das Payment gegebenenfalls kompensiert wird, und das bereits als Erstes. Tritt der Timeout schon früher ein *(siehe Abbildung 12)*, ist das Payment gar nicht mit involviert.

Fazit

Abschließend werden noch einmal die Ziele betrachtet. Es wurde kein Fall vergessen, was durch das State-Modell der LRAs gewährleistet wird. Und letztendlich wird auch immer wieder ein global konsistenter Zustand erreicht. Den Code-Beispielen ist zu entnehmen, dass an keiner Stelle Daten gelockt werden, und das Framework kommt nahezu ohne Boilerplate-Code aus.

Zusammenfassend lässt sich feststellen, dass es leider immer noch wenige Lösungsansätze gibt, um Transaktionen in verteilten Microservices zu implementieren. Klassische Lösungen, die stark auf dem ACID-Prinzip basieren, wie zum Beispiel X/Open XA, sind nicht sinnvoll anwendbar.

Das Saga-Pattern und insbesondere die betrachtete Eclipse-Microprofile-LRA sind dagegen eine mögliche Variante, Transaktionen in Microservices zu implementieren. Allerdings gilt auch hier die Einschränkung, dass Transaktionen mit Echtzeit-Charakter oder starken Konsistenzansprüchen nicht sinnvoll implementierbar sind.

Quellen

[1] Dr. Eric Brewer (2000): *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing.* ACM New York, New York, USA.

[2] Seth Gilbert & Nancy Lynch (2002): *Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services.* ACM SIGACT News Volume 33 Issue 2, ACM New York, New York, USA

[3] XA-Spezifikation: *http://pubs.opengroup.org/onlinepubs/009680699/toc.pdf*

[4] Héctor García-Molina, Kenneth Salem: *Sagas.* *ftp://ftp.cs.princeton.edu/reports/1987/070.pdf*

[5] Arnon Rotem-Gal-Oz (2012): *SOA Patterns.* Manning, Shelter Island, New York, USA

[6] Saga Pattern: *https://cararuseugeniu.blogspot.com/p/saga-pattern.html*

[7] Narula, McCaffrey, Kingsbury: *Distributed Sagas*: *https://github.com/aphyr/dist-sagas/blob/master/sagas.pdf*

[8] Eventuate Tram Saga: *https://github.com/eventuate-tram/eventuate-tram-sagas*

[9] Axon Framework: *https://axoniq.io/*

[10] Long Running Actions for MicroProfile: *https://github.com/eclipse/microprofile-lra*

Matthias Koch
sidion
matthias.koch@sidion.de

Matthias Koch ist Diplom-Informatiker und arbeitet als Senior Software Developer bei der Firma sidion. Er entwickelt seit mehr als 20 Jahren Geschäftsanwendungen für unterschiedlichste Kundenprojekte – vor allem mit Java. Bei seinen Code- und Architekturreviews setzt er auf das Prinzip „Clean Code". Ferner interessiert er sich für funktionale Programmierung, Secure Coding und Continuous Delivery. Außerdem ist er Dozent an der Hochschule für Technik in Stuttgart.

Markus Grabert
sidion
markus.grabert@sidion.de

Markus Grabert ist Diplom-Informatiker und arbeitet als Senior Projektmanager bei der Firma sidion. Seit über 20 Jahren ist er in verschiedenen Rollen, als Netzwerk- und Unix-Admin, meist aber als Softwareentwickler tätig. Derzeit arbeitet er in Kundenprojekten als Softwarearchitekt mit Java. Seine Schwerpunkte liegen in und rund um die Themen Agile Project Management, Java, DevOps und Linux.