Ver-Log-end

Log-Daten effektiv verarbeiten mit Apache Kafka

Arne Landwehr, Phillip Ghadir

Mit Apache Kafka stellen wir ein ungewöhnliches Messaging-System vor. Es besticht durch persistente Speicherung der Nachrichten, hohen Transaktionsdurchsatz und gute Skalierbarkeit. Kafkas Architektur ermöglicht dadurch nicht nur den Einsatz in Online-Szenarien mit zeitnaher Verarbeitung, sondern auch in Offline-Szenarien mit stark zeitverzögerter Verarbeitung von Nachrichten. Dieser Beitrag demonstriert, wie Kafka auch mit großem Log-Aufkommen zurecht kommen kann.

Apache Kafka

Ursprünglich bei LinkedIn entwickelt, ist Apache Kafka seit Anfang 2011 ein Apache-Incubator-Projekt. Kafka ist ein in Scala implementiertes, verteiltes Messaging-System, das über ein Java-API auch in Java-Applikationen verwendet werden kann. Im Wesentlichen kann man mit Kafka eine Reihe persistenter Queues mit hohem Durchsatz bereitstellen. Ursprünglich wurde es konzipiert, um bei LinkedIn der Flut von Logs Herr zu werden.

Log-Dateien und ihre Herausforderungen

Je mehr Transaktionen ein System verarbeitet, desto größer werden die Logs, desto höher die Anforderungen an deren Verarbeitung. Rein nachgelagerte Analysen der Log-Dateien reichen für interaktive Anwendungen oft nicht mehr aus. Stattdessen werden Systeme benötigt, die die Verarbeitung der Dateien innerhalb von Sekunden ermöglichen, ohne Hunderte von Gigabyte zu transportieren, zu filtern und zu scannen. Das Volumen der Logs und die Komplexität der Auswertungen erfordern das Skalieren der Log-Datei-Auswertung. Das noch junge Projekt Apache Kafka ist angetreten, um sich mit frischen Designideen dieser Herausforderung anzunehmen.

Log-Dateien gehören zum Alltag in der Softwareentwicklung. Ihre Lektüre bzw. Analyse gestaltet sich dabei oft schwierig, da häufig eine zentrale Log-Datei als Sammelbecken für eine Reihe von heterogenen Informationen dient. Dabei lassen sich die einzelnen Log-Meldungen grob in unterschiedliche Kategorien unterteilen – wie beispielsweise:

- *betriebsrelevante Daten* wie Exceptions, parallel laufende Jobs oder Metriken über die aktuelle Performance des Systems,
- *Aktionen und/oder Status der Anwendung* wie Inhalte von Variablen, genommene Abzweigungen im Code oder besondere Randfälle,
- *Benutzeraktionen* wie Seitenbesuche, neue Kommentare, bearbeitete Seiten oder auch das „Liken" der Inhalte anderer Personen.

Jede dieser Kategorien gruppiert Informationen für unterschiedliche Stakeholder. Auch der Zeitpunkt, zu dem diese Informationen typischerweise ausgewertet werden, ist verschieden:

- *Entwickler* interessieren sich meist Tage nach dem eigentlichen Ereignis für den Status der Anwendung zum Fehlerzeitpunkt.
- *Administratoren* benötigen z. B. aktuelle Daten über das Fehleraufkommen der produktiven Umgebung.
- Das *Management* wiederum verlangt einen wöchentlichen Bericht über Benutzeraktivitäten, wie Seitenaufrufe oder Logins.
- Und schließlich ist eine *interaktive Anwendung* ebenfalls ihr eigener Log-Leser. Komponenten wie Newsfeeds oder Trendgraphen wollen möglichst schnell aktualisiert werden.

Es sind also sowohl zeitnahe (online) Analysen der Log-Daten erforderlich als auch komplexe nachgelagerte (offline) Auswertungen – am Besten auf dedizierten Datencontainern, um die Komplexität des auswertenden Codes zu reduzieren. Klassische Messaging-Systeme ermöglichen zwar per Publish/Subscribe das Online-Verarbeiten der Daten, unterstützen aber nur selten Offline-Verarbeitung und skalieren häufig nicht beliebig. Dabei ist der letzte Punkt entscheidend. Besonders durch ein systematisches Logging der Benutzeraktivitäten (auch bekannt als activity stream data [actis]) steigt das Volumen der Log-Daten rasant an und übersteigt das Datenbankvolumen meist um ein Vielfaches [kaf11]. Fallen in traditionellen Enterprise-Systemen noch mehrere Gigabyte pro Tag an, sind es in bekannten Webanwendungen schon Terabyte [faceb].

Apache Kafka wurde explizit für die beschriebene Problemstellung entworfen und bietet zur Lösung des Problems einige interessante Designideen.

Kafkas Architekturziele

Kafkas hervorstechendstes Qualitätsmerkmal ist der hohe Datendurchsatz. Selbst auf moderater Hardware sollten Hunderttausende von Nachrichten pro Sekunde kein Problem darstellen. Das Versenden von persistenten Nachrichten erfolgt in konstanter Zeit, trotz der notwendigen Einbeziehung des Massenspeichers. Darüber hinaus ermöglicht Kafka das Partitionieren von Nachrichten nach definierbaren Kriterien und erlaubt so eine Skalierung über Partitionen hinweg. Dabei bleibt die relative Reihenfolge der Nachrichten innerhalb einer Partition erhalten.

Hoher Datendurchsatz auf einer JVM?

Kafka wird in der JVM betrieben. Einen konstant hohen Datendurchsatz auf der JVM zu realisieren, erfordert spezielles Augenmerk auf die Strategie des Garbage Collectors (GC). Es gibt JVM-spezifisch verschiedene Strategien. Je nach Volumen und Objekt-Anzahl kann ein vollständiger, blockierender GC-Lauf einige Sekunden Stillstand hervorrufen, was dabei potenziell zu einer Verzögerung mehrerer Millionen Nachrichten führen kann.

Um solche Sperren zu vermeiden, hebelt Kafka den GC direkt nach dem Start aus und nutzt das Speichermanagement des Betriebssystems und insbesondere der Festplatten-Caches, um den Hauptspeicher effektiv zu nutzen (siehe auch [Hick11]).

Kafkas Persistenzstrategie

Kafka schreibt alle zu übermittelnden Nachrichten kontinuierlich auf die Platte(n). Dies ermöglicht auf einer Festplatte eine konstant hohe Schreibgeschwindigkeit. Dieses Verfahren ist analog zur Prepend-Strategie in Clojures List-Datenstrukturen, in denen auch stets dort angehängt wird, wo nur ein Zeiger bewegt werden muss. Über einen Konfigurationsparameter kann man ein Verfallsalter bzw. ein maximales Volumen für Nachrichten konfigurieren.

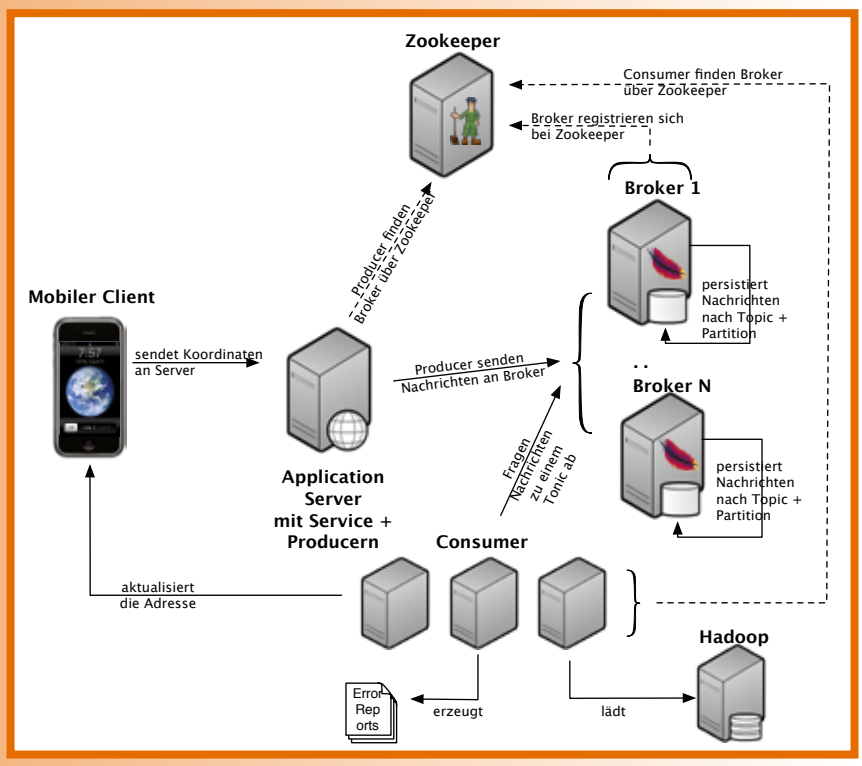

Abb. 1: Überblick über die Apache-Kafka-Umgebung der Beispiel-Anwendung

Kafka ist darauf optimiert, dass Nachrichten mindestens einmal gelesen werden. Daher setzt Kafka auf sogenannte Message Sets (sinnvoll mit Nachrichten-Gruppen übersetzt), dank derer das typische Zugriffsverhalten von Konsumenten zu einem reduzierten Netzwerk-Overhead führt. Dabei lässt Kafka das Betriebssystem die Nachrichten direkt aus dem Disk-Cache auf die Netzwerk-Karte mittels der Methode `FileChannel.transferTo` aus Java-NIO kopieren. Ein unnötiges Kopieren in und aus dem Hauptspeicher entfällt.

Nachrichten werden in einer Struktur repräsentiert, die ohne großen Overhead Byte-Arrays und Dateien abstrahiert.

Kafka-Komponenten

Kafka besteht aus drei Komponenten, die über einen Publisher/Subscriber-Mechanismus miteinander kommunizieren:

- den Producern, die die Daten erfassen und zu einem gewählten Topic publizieren,
- den Brokern, die die Daten entgegennehmen und persistent vorhalten, und
- den Consumern, welche die Daten vom Broker für ein gewähltes Topic abrufen und die eigentliche Verarbeitung durchführen.

Bei Kafka nimmt man stets eine verteilte Umgebung an. Jede der Komponenten kann auf ein oder mehrere physikalische Systeme verteilt sein. Die Registrierung und Verwaltung der einzelnen Komponenten übernimmt Apache Zookeeper [zooke]. Um den Consumern eine maximale Verarbeitungsgeschwindigkeit zu ermöglichen, holt ein Consumer die Nachrichten selbst ab und verarbeitet sie. Kafka stellt die Reihenfolge innerhalb einer Partition sicher. Der Consumer ist dafür verantwortlich, sich zu merken, bis zu welcher Nachricht der Nachrichtenstrom bereits verarbeitet ist. Ab dort - oder auch zu jedem früheren Stand - kann er beliebig aufsetzen und sich die Nachrichten (evtl. erneut) übermitteln lassen.

So können beispielsweise auch zusätzliche Consumer gestartet werden, die zum Beispiel gesonderte Auswertungen fahren. Um hier eine effiziente Verarbeitung zu ermöglichen, können Consumer in sogenannten Consumer-Groups zusammengeschlossen werden. Diese Gruppen werden dann behandelt wie ein Consumer. Eine Nachricht innerhalb einer Partition wird somit von einer Consumer-Group lediglich einmal verarbeitet.

Kafka im Einsatz

Nach dem kurzen Überblick folgt nun ein Beispiel, in dem Apache Kafka als Basis für das Backend einer mobilen Anwendung dient, die die aktuelle Adresse eines Benutzers bestimmen soll. Der auf dem Handy des Benutzers installierte Client sendet bei jeder Positionsänderung die neue Position als zwei-dimensionales Tupel von Geodaten an den Server, der die jeweils passende Adresse ermittelt und die Anzeige auf dem Client aktualisiert.

Angenommen, jeder Client sendet alle 5 Sekunden eine Nachricht, senden 100.000 Nutzer bereits 1,2 Millionen Nachrichten pro Minute. Zusätzlich sollen Exceptions, Performance-Metriken und Debug-Informationen über Kafka protokolliert werden. Einen Überblick über die Anwendung bietet Abbildung 1.

Zookeeper verwaltet die Umgebung

Zum Aufsetzen einer Kafka-Umgebung kann man entweder manuell eine Liste mit Kafka-Broker-Instanzen konfigurieren oder Zookeeper die Verwaltung von Brokern überlassen [zooke]. Dazu registrieren sich die Kafka-Broker lediglich bei einer laufenden Zookeeper-Instanz und sind damit sowohl für Consumer als auch Producer erreichbar.

Die flush-Intervalle sowie die Löschstrategien der Broker werden anwendungsfallspezifisch per Properties-Datei konfiguriert. Hier stehen die Optionen „nach Zeitablauf“ und/oder „nach Anzahl von Messages“ zur Verfügung. Fehlkonfigurationen können zum Datenverlust oder zu Verzögerungen bei der Zustellung führen, da Consumer nur bereits persistierte (geflushte) Nachrichten abrufen können.

Über die Anzahl der Partitionen lässt sich wiederum steuern, auf wie viele Broker ein Topic verteilt werden soll. Jede weitere Partition eines Topics ermöglicht später einen weiteren parallelen Zugriff durch einen Consumer, erhöht allerdings auch die Anzahl von Dateien im System.

Kafka kommt mit einer sinnvollen Zookeeper-Konfiguration, die mit den Aufrufen von

```
bin/zookeeper-server-start.sh config/zookeeper.properties
```

und

```
bin/kafka-server-start.sh config/server.properties
```

direkt verwendet wird, um Kafka zu starten. Damit steht eine Zookeeper-Instanz bereit, bei der sich der Broker registriert hat.

Der GeoService: Nachrichten sauber sortiert

Der GeoService (s. Listing 1) ist der Ausgangspunkt für die Fachlogik auf dem Server. Die vom mobilen Client aus gesendeten Geodaten werden entgegengenommen und über einen Producer als Messages an den Broker weitergegeben. Um eine starke Kopplung zwischen fachlichen und technischen Komponenten zu vermeiden, bietet sich der Einsatz von CDI (Contexts and Dependency Injection, [jsrcd]) an. Der Producer wird über Dependency Injection bereitgestellt und Interceptoren übernehmen die querschnittlichen Aspekte.

Der Service demonstriert, wie das Logging für verschiedene Stakeholder aufgrund der Wahl passender Topics geordnet werden kann:

- Ausnahmen können unter einem eigenen Topic veröffentlicht werden, was z. B. das Fehlermonitoring für den Betrieb vereinfacht.
- Parameter- und Performance-Informationen können durch die asynchrone und skalierbare Infrastruktur querschnittlich gesammelt werden, ohne die Leistung des produktiven Systems zu gefährden.
- Fachliche Events, wie das Eintreffen einer neuen Positionsänderung, können jeweils passenden fachlichen Topics zugeordnet werden.

Die Persistierung der Daten durch den Broker erlaubt es trotzdem, die verschiedenen Kanäle zu einem späteren Zeitpunkt wieder miteinander in Beziehung zu setzen. Zum Beispiel könnte für jeden fachlichen Fehler ein Report erstellt werden, der die passenden Parameter-Traces aller aufgerufenen Methoden enthält. Das dürfte einem Entwickler die spätere Fehlersuche deutlich erleichtern.

```
public class GeoService {
  @Inject
  KafkaProducer producer;
  @Inject
  UserService userService;

  // Logge die Performance der Methode über einen Kafka-Producer
  @PerformanceLogged
  // Logge die Parameter der Methode über einen Kafka-Producer
  @ParameterLogged
  public void submitLocation(double x, double y, String userId) {
    try {
      userService.assertValidUser(userId);
    } catch (InvalidUserException e) {
      producer.sendMessage(System.currentTimeMillis() + " | " +
        e.getMessage(), "ERROR");
        return;
    }
    producer.sendMessage(System.currentTimeMillis() +","+userId+
      ","+x+","+y, "LOCATION");
  }
}
```

Listing 1: GeoService zur Ermittlung einer Adresse

Der Producer als Nachrichtenfabrik

Um die verschiedenen Informationen zum Broker zu schicken, benötigt der GeoService mindestens einen Producer. Dessen Messages sollten dabei möglichst schnell zugestellt werden und am besten auch noch in einem Format vorliegen, das den Consumern eine einfache Verarbeitung ermöglicht.

Entscheidend für die Performance sind sowohl die Anzahl von benötigten Netzwerkverbindungen als auch die Summe der verschickten Bytes. Die Anzahl der Verbindungen kann durch die Verwendung eines asynchronen Producers erfolgen. In diesem Fall schnürt der Producer jeweils *n* Nachrichten zu einem Bündel zusammen und verschickt dieses dann als Bulk-Operation. Die Menge der verschickten Bytes lässt sich wiederum durch eine Komprimierung reduzieren. Kafka bietet hier sowohl GZIP [gzipo] als auch Snappy [snapp] als Algorithmen an.

Messages werden im gleichen Format abgespeichert und gelesen, in dem sie verschickt wurden. Es sollte also möglichst gut zu parsen sein. Klassiker wie XML oder JSON bieten sich an. Weiterhin übernimmt Kafka keine Garantie, dass eine Message nicht zweimal zugestellt wird, sie sollten also durch den Consumer idempotent verarbeitbar sein. Im Zweifel muss damit jede Nachricht eine ID enthalten, die es dem Consumer ermöglich, Duplikate zu erkennen.

Für die eigentliche Implementierung stehen zwei Möglichkeiten bereit. Der Standard dürfte, wie in Listing 2 und Listing 3 dargestellt, die Verwendung des Producer-APIs sein, was dem Entwickler alle Freiheiten lässt. Interessant für bestehende Projekte ist allerdings die zweite Alternative, den womöglich bereits bestehenden Log4J-Logger als Producer zu verwenden. Diese Alternative bietet weniger Freiheitsgrade, bedarf aber lediglich einer entsprechenden Anpassung der Log4J-Einstellungen. Ideal für einen schmerzfreien Umstieg.

```
/**
* Sendet die übergebene Message an einen zufälligen Broker.
* Dieser veröffentlicht sie unter dem gegebenen Topic.
*/
public void sendMessage(Producer producer,
    String message, String topic) {
  ProducerData<String, String> data =
    new ProducerData<String, String>(topic, message);
  producer.send(data);
}
```

Listing 2: sendMessage-Methode in unserem Kafka-Producer

```
@PerformanceLogged
@Interceptor
public class PerformanceInterceptor {
  @Inject
  private KafkaProducer producer;

  @AroundInvoke
  public Object logPerformance(InvocationContext context)
      throws Exception {
    // Einfache Performanceberechnung über die Systemzeit
    final long start = System.currentTimeMillis();

    //Aufruf der eigentlichen Methode
    Object result = context.proceed();
    final long end = System.currentTimeMillis();

    // Dauer des Methodenaufrufes über Topic "PERFORMANCE" senden
    producer.sendMessage(end + " | Method: " + context.getMethod() +
      " | Duration: " + (end - start), "PERFORMANCE");
    return result;
  }
}
```

Listing 3: Die Implementierung des Performance-Interceptors

Der Consumer – Konsumieren nach Bedarf

Die vom Producer erstellten Nachrichten liegen damit persistent beim Broker unter den entsprechenden Topics vor. Die Abfrage erfolgt nun, indem die Consumer die Nachrichten vom Broker abrufen. Ein Consumer erstellt dafür mittels Zookeeper eine Verbindung zum Broker und ruft anschließend parallel von allen vorhandenen Topic-Partitionen die noch nicht gelesenen Nachrichten ab.

Das auf dem Pull-Prinzip basierende Messaging ermöglicht es auch, länger laufende Verarbeitungslogik direkt im Consumer auszuführen. Listing 4 zeigt exemplarisch die Methode `pullMessages`, die parallele Nachrichten über ThreadExecutor abfragt und für jede Nachricht an einen externen Webservice zur Ermittlung der Adresse delegiert.

Kann der Consumer auf dem Broker neu eintreffende Nachrichten nicht schnell genug verarbeiten, fällt er zwar weiter zurück, die Messages sind allerdings nicht verloren. Durch eine längere Laufzeit oder weitere Consumer, die der Consumer-Gruppe hinzugefügt werden, kann der Rückstand jederzeit wieder aufgeholt werden. Das Prinzip verhindert dadurch elegant, dass Consumer zum Engpass der Anwendung werden können, und ermöglicht ebenfalls das erneute Abrufen von fehlerhaft verarbeiteten Messages.

Das Beispiel GeoService demonstriert die Vielseitigkeit der Kafka-Architektur: Die Adressangaben sind auf dem Handy des Benutzers aktuell, es werden Debug-Informationen vorgehalten und gleichzeitig kann man die Laufzeiten des Systems analysieren. Für komplexere Analyselogik, wie die Bestimmung der Lieblingsorte eines Benutzers, könnten die gesammelten Geodaten abschließend noch in Hadoop überführt werden. Für den benötigten ETL-Job (Extract, Transform, Load) steht unter [hadoo] ein geeigneter Consumer bereit.

```
public void pullMessages(ConsumerConnector consumerConnector) {
  // Aufbau der Streams zum Abrufen der Messages von "location"
  Map<String, Integer> topicCountMap =
    Collections.singletonMap("location", partitions);
  Map<String, List<KafkaStream<Message>>> topicMessageStreams =
    consumerConnector.createMessageStreams(topicCountMap);
  final List<KafkaStream<Message>> topicStreams =
    topicMessageStreams.get("location");
  // Für jede Partition von "location" wird ein Thread erstellt,
  // um die Daten parallel abzurufen
  ExecutorService executor = Executors.newFixedThreadPool(partitions);

  for (final KafkaStream<Message> stream : topicStreams) {
    executor.submit(new Runnable() {
      public void run() {
        // Achtung: Blockierender Aufruf von next().
        // Es wird immer auf die nächste Message gewartet
        // Die Schleife wird demnach nicht terminiert
        for (MessageAndMetadata<Message> msgAndMetadata : stream) {
          // Die Daten kommen im gleichen Format an,
          // in dem sie verschickt wurden
          Geodata geodata =
            convertMessageToGeodata(msgAndMetadata.message());
          // die neue Adresse wird über einen Webservice berechnet
          // und an den Client geschickt
          Address address = queryGeoDataWebserviceForAddress(geodata);
          sendAddressToClient(address);
        }
      }
    });
  }
}
```

Listing 4: Kafka-Consumer zur Verarbeitung der Positionsdaten

Das Urteil

Kafka ist eine noch recht junge Persistent-Queue-Implementierung, die es ermöglicht, auf Seiten der Producer, Consumer und auch Broker zu partitionieren. Die API sowie die Implementierung sind auf hohen Datendurchsatz ausgelegt und nutzen die Speicher-Management- und Caching-Strategien des Betriebssystems effizient. Consumer verwenden einen Pull-Mechanismus, um Nachrichten von den Brokern abzurufen. Es liegt in der Verantwortung der Consumer, Buch zu führen, bis zu welcher Nachricht sie den Nachrichtenstrom bereits verarbeitet haben. Nachrichten können daher bequem auch mehrfach abgefragt werden und so parallel sowohl für die Online-Verarbeitung als auch für eine spätere Offline-Verarbeitung verwendet werden.

Apache Kafka verzichtet auf angebliche „must haves“ wie eine Transaktionsunterstützung zu Gunsten von Performance und Skalierbarkeit. Es ersetzt daher kein traditionelles Enterprise-Messaging-System. Ob Apache Kafka - trotz des Incubator-Status - das richtige Werkzeug für einen Einsatzzweck ist, lässt sich im Zweifelsfall herausfinden, indem man sich lokal eine Testumgebung aufsetzt und die Worte von Franz Kafka beherzigt: „Wege entstehen dadurch, dass man sie geht.“

Literatur und Links

[actis] Wikipedia-Seite über Activity Streams, `http://en.wikipedia.org/wiki/Activity_stream`

[faceb] A. Thusoo u. a., Data warehousing and analytics infrastructure at facebook, 2010, in: Proc. of the 2010 ACM SIGMOD Int. Conf. on Management of data

[gzipo] Homepage des GZIP-Projektes, `http://www.gzip.org`

[hadoo] Github-Repository des Kafka Hadoop Consumers, `https://github.com/kafka-dev/kafka/tree/master/contrib/hadoop-consumer`

[Hick11] R. Hickey, Präsentation “Simple made easy” auf InfoQ, `http://www.infoq.com/presentations/Simple-Made-Easy`

[jsrcd] JSR-299: Contexts and Dependency Injection for the Java EE platform, `http://jcp.org/en/jsr/detail?id=299`

[kaf11] J. Kreps, N. Narkhede, J. Rao, Kafka: A distributed messaging system for log processing, 2011, in: Proc. of 6th Int. Workshop on Networking Meets Databases (NetDB), Athens, Greece

[kafka] Homepage des Projekts Apache Kafka, `http://kafka.apache.org`

[snapp] Homepage des snappy-Projekts, `http://code.google.com/p/snappy/`

[zooke] Homepage des Projekts Apache Zookeeper, `http://zookeeper.apache.org`

Arne Landwehr ist bei der innoQ Deutschland GmbH als Berater und Entwickler von Enterprise Anwendungen tätig. Sein besonderes Interesse liegt in der Modularisierung und Restrukturierung von gewachsenen Systemen.
E-Mail: arne.landwehr@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