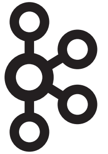
Kafka als IoT-Daten-Plattform



Dr. Ralph Guderlei, eXXcellent solutions GmbH

Viele Firmen setzen auf das Internet of Things, um neue und innovative Produkte zu erstellen. Neben der Entwicklung von Hardware und Sensorik ist das Sammeln und Verarbeiten der anfallenden Sensordaten ein wesentlicher Bestandteil eines IoT-Projekts.

Das Unternehmen des Autors kooperiert seit gut zwei Jahren mit einem lokalen Energieversorger. Dieser will digitale Stromzähler mit Sendern ausstatten, um Verbrauchswerte übertragen zu können. Wenn das Vorhaben vollständig umgesetzt ist, senden rund 150.000 Stromzähler in 15-Minuten-Intervallen Daten. Die gesendeten und aufbereiteten Daten werden dann über Schnittstellen weiteren Systemen zur Verfügung gestellt. Der Energieversorger verspricht sich davon unter anderem eine deutliche Optimierung des Ableseprozesses und eine Verbesserung der Prognosen für den Strom-Einkauf.

Betrachtet man diesen Use Case etwas abstrakter, lassen sich allgemeine Anforderungen an ein System zur Verarbeitung von IoT-Daten ableiten: Eine potenziell große Anzahl von Sensoren versendet mehr oder weniger regelmäßig Daten, die mit geringen Latenzen verarbeitet werden sollen. Oft müssen die gesammelten Daten weiteren Systemen zur Verfügung stehen. Das System sollte in der Lage sein, neue Sensor-Typen einfach integrieren zu können. In diesem Use Case müssen Stromzähler unterschiedlicher Hersteller mit jeweils unterschiedlichen Übertragungswegen unterstützt werden. *Abbildung 1* zeigt die grobe System-Architektur, die sich aus diesen Anforderungen ergibt.

Sensoren übertragen die Daten auf unterschiedlichen Wegen wie LoRaWAN, BLE, WiFi oder GSM. Diese werden zunächst vorverarbeitet, indem man sie etwa in einheitliche Datenstrukturen überführt. Dann werden die eigentlichen Nutzdaten extrahiert und zur weiteren Verwendung bereitgestellt. In unserem Fall kommen die Daten über ein sogenanntes „LoRaWAN-Netzwerk", ein Funk-Netzwerk mit großer Reichweite, das sich besonders für die Übertragung von Sensordaten mit niedriger Datenrate eignet. Diese Art von Funknetz darf von jedermann betrieben werden – vergleichbar mit einem WLAN. Da momentan noch wenig Erfahrung mit dem Betrieb und dem Ausbau dieses Netz-Typs besteht, werden die anfallenden Metadaten gesammelt und zur weiteren Netzplanung herangezogen.

Für die Umsetzung des Systems hat man sich für Apache Kafka entschieden. Neben der weiten Verbreitung und der aktiven Open-Source-Community rund um Kafka war ausschlaggebend, dass mit Kafka Connect und Kafka Streams zwei Teilprojekte existieren, die die Aufgabenfelder „Datenaufnahme und –weitergabe" (Kafka Connect) sowie „Stream Processing" (Kafka Streams) abdecken.

Kafka als Pub/Sub-Queue

Apache Kafka ist zunächst nur eine Pub/Sub-Queue und dient dazu, Daten zwischen den einzelnen Bearbeitungsschritten auszutauschen. Innerhalb von Kafka sind Daten in sogenannten „Topics" organisiert. Diese bilden eine unveränderliche und persistente Sequenz von Nachrichten. Producer fügen neue Nachrichten an das Ende des Topics an, Consumer lesen die Nachrichten sequenziell aus dem Topic. Unterschiedliche Consumer können sich dabei an unterschiedlichen Stellen (Offset) im Topic befinden. Das Format der Nachrichten ist beliebig, typischerweise wird JSON oder Apache Avro als Datenformat verwendet *(siehe Abbildung 2)*.

Kafka ist von Grund auf als verteiltes System ausgelegt. Das bringt zwar bezüglich Skalierbarkeit und Verfügbarkeit deutliche Vorteile, sorgt aber gleichzeitig an der einen oder anderen Stelle für erhöhte Komplexität. Ein wichtiger Aspekt sind sogenannte „Partitionen". Sie dienen dazu, Daten eines Topics über mehrere Kafka-Instanzen zu verteilen.

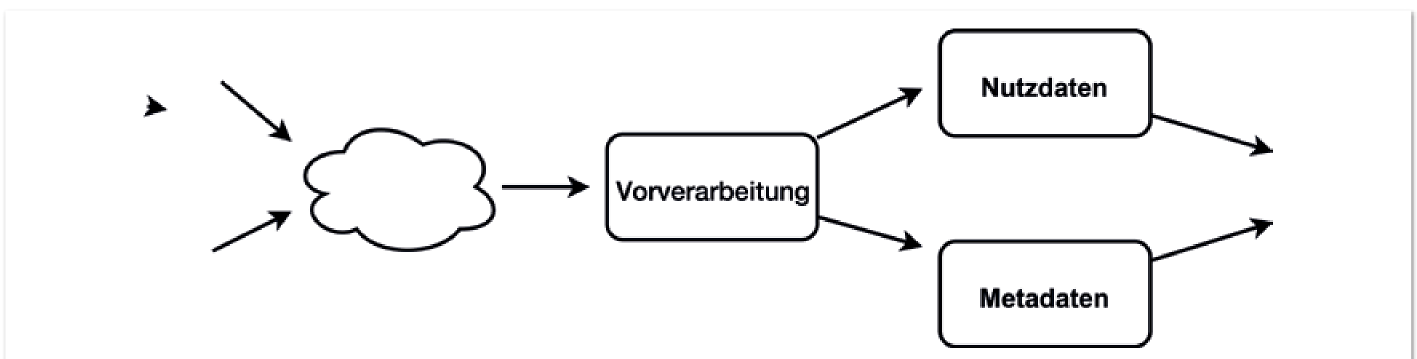


Abbildung 1: System-Architektur

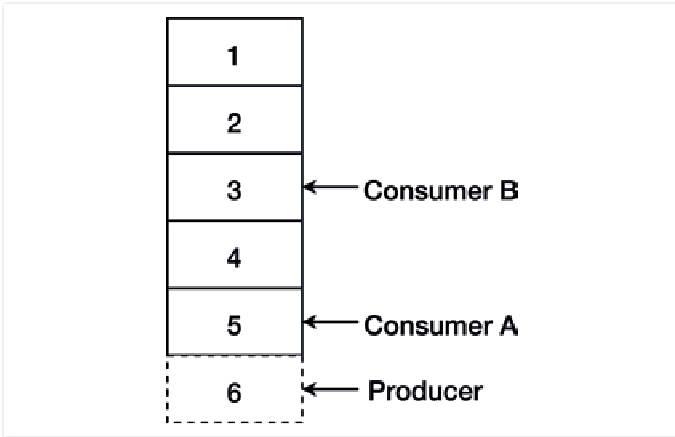


Abbildung 2: Nachrichten-Sequenz

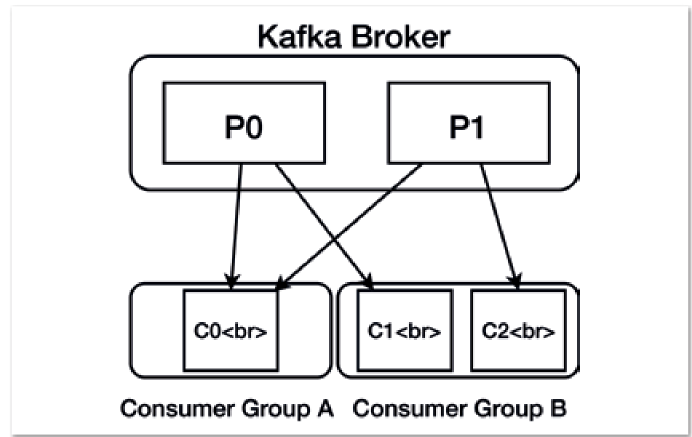


Abbildung 3: Kafka-Cluster

Gleichzeitig bestimmt die Anzahl der Partitionen den Parallelisierungsgrad der Datenverarbeitung in Consumern. Neben dem normalem Publish/Subscribe bietet Kafka die Möglichkeit, mehrere Consumer in einer Consumer Group zusammenzufassen. Die Nachrichten werden dann gleichmäßig über die Consumer einer Consumer Group verteilt und können so parallel bearbeitet werden.

Die maximal mögliche Anzahl der Consumer in einer Consumer Group entspricht der Anzahl der Partitionen eines Topics. Die Anzahl der Partitionen kann zwar im Nachhinein noch geändert werden, das hat aber unter Umständen negativen Einfluss auf die Performance, während der Kafka-Cluster sich neu synchronisiert *(siehe Abbildung 3)*.

Daten rein und raus: Kafka Connect

Eine Möglichkeit, um Daten nach Kafka hinein- oder aus Kafka herauszubekommen, ist Kafka Connect. Dessen Kernstück sind sogenannte „Konnektoren", die es in zwei Ausprägungen gibt: SourceConnectors für die Übernahme von Daten nach Kafka und SinkConnectors für die Übertragung von Daten in andere Systeme. Es existiert bereits eine große Anzahl von Konnektoren, oft als Open-Source-Projekte. Mit diesen lassen sich Datenbanken, andere Message Queues oder andere Systeme rein konfigurativ anbinden. Das Beispiel in *Listing 1* zeigt die Konfiguration für einen Konnektor, der alle Nachrichten des Topics „orders" in eine Tabelle „orders" in einer Sqlite-Datenbank schreibt.

Sollte noch kein Konnektor für ein Drittsystem vorhanden oder man mit der Funktionalität oder der Qualität eines bestehenden Konnektors nicht zufrieden sein, lassen sich eigene Konnektoren mit überschaubarem Aufwand auch selbst erstellen. Ein Konnektor besteht im Wesentlichen aus drei Klassen: dem Rahmen des Konnektors, der Beschreibung der Konfiguration und einer Task-Klasse, die die eigentliche Arbeit übernimmt *(siehe Listing 2)*.

Ein Task besteht aus drei zu implementierenden Methoden, zwei davon werden beim Starten und Beenden des Tasks ausgeführt, die dritte erledigt die eigentliche Verarbeitung der Daten. Im Code-Beispiel ist erkennbar, dass Kafka Connect ein eigenes Format für Daten verwendet. Dieses besteht aus einem Schema zur strukturellen Beschreibung der Daten und den eigentlichen Daten. Durch das Datenschema kann die Logik des Tasks sehr abstrakt und generisch programmiert werden. So wird im Fall des JDBC-Konnektors beispielsweise das notwendige SQL-Statement komplett aus dem Schema abgeleitet.

Kafka Connect stellt dann die Laufzeitumgebung für die Konnektoren zur Verfügung. Unter anderem bietet Kafka Connect auch ein REST-API an. Über dieses können Status-Informationen zu Konnektoren abgefragt und Konnektoren zur Laufzeit hinzugefügt oder entfernt werden. Im sogenannten „Distributed Mode" lässt sich Kafka Connect ebenfalls in einem Cluster betreiben. Kafka Connect verteilt in diesem Modus dann die Konnektoren im Cluster.

Verarbeitung mit Kafka Streams

Nachdem die Daten nun im System sind, geht es im nächsten Schritt darum, den Datenstrom zu verarbeiten. Dazu bietet sich Kafka Streams an, ein API ähnlich dem Java-Streams-API, das speziell

```
{
  "name": "jdbc-sink",
  "config": {
     "connector.class": "io.confluent.connect.jdbc.JdbcSinkConnector",
     "tasks.max": "1",
     "topics": "orders",
     "connection.url": "jdbc:sqlite:test.db",
     "auto.create": "true",
     "name": "jdbc-sink"
  },
  "tasks": [],
  "type": null
}
```

Listing 1

für die Arbeit mit Kafka-Topics entwickelt wurde. Entwickler, die mit dem Streams-API vertraut sind, dürften schnell damit zurechtkommen. Da Kafka Streams nur eine einfache Java-Bibliothek ist, lassen sich Streams-Anwendungen einfach als „executable jar" verpacken und ausführen. Das Beispiel in *Listing 3* zeigt eine einfache Kafka-Streams-Anwendung.

Über ein Fluent-API werden das Quell-Topic und die Art der Deserialisierung der Daten gewählt. Im Beispiel werden die JSON-Nachrichten des Topics automatisch in ein Java-POJO transformiert. Im folgenden Schritt werden dann aus den Rohdaten die eigentlichen Nutzdaten extrahiert. Pro Rohdaten-Objekt entsteht dabei eine Liste von Nutzdaten-Objekten, die mittels „flatMapValues" bearbeitet werden, sodass am Ende eine Sequenz von Nutzdaten-Objekten entsteht. Diese werden am Ende wieder als JSON-Nachrichten im Ziel-Topic publiziert.

Bei spezialisierten Bibliotheken oder Frameworks wie Kafka Streams, die eng mit anderen Systemen verwoben sind, ergeben sich oft Probleme, die mit diesen Werkzeugen erstellten Lösungen mit vertretbarem Aufwand automatisiert zu testen. Für Kafka Streams existiert seit Kafka 1.1.0 eine eigene Test-Bibliothek. Mit deren Hilfe lassen sich einfach Unit-Tests für Streams-Applikationen schreiben. Diese können dann mit den üblichen Mitteln in den normalen Build-Prozess integriert werden *(siehe Listing 4)*.

Im Test werden ein eingebetteter Kafka-Broker gestartet und die zu testende Streaming-Anwendung damit verbunden. Für die Tests können dann Nachrichten in das Quell-Topic der Streaming-Anwendung publiziert werden. Der eigentliche Test besteht aus der Verifikation der Nachrichten im Ziel-Topic der Streaming-Anwendung.

Die reine „1:1"-Verarbeitung von Sensordaten reicht in vielen Use Cases nicht aus. Oft sind in der Verarbeitung Daten aus vorherigen Übertragungen erforderlich. Ein einfaches Beispiel ist die Ermittlung von Paketverlusten. In unserem Fall wird bei jeder Übertragung ein fortlaufender Zähler mitgeführt. Um den Verlust von einem oder mehreren Paketen feststellen zu können, ist der aktuelle Zählerstand mit dem letzten bekannten Zählerstand zu vergleichen.

Normalerweise sind solche Informationen in Datenbanken abgelegt. Kafka Streams bietet dazu ein eigenes Konstrukt: KTables. Dabei

```
public Topology topology(Properties config) {
  final StreamsBuilder builder = new StreamsBuilder();
  final String IN = ...
  final String OUT = ...

  // capture and transform each incoming message
  final KStream<String, DataPoint> values = builder
        .stream(IN, Consumed.with(Serdes.String(),
           new JsonSerde<>(TTNMessage.class)))
        .flatMapValues(ElectricityMessageParser::parse)

  // send data to topic
  values.to(OUT, Produced.with(Serdes.String(),
     new JsonSerde<>(DataPoint.class)));
  return builder.build();
}
```

Listing 3

```
public class MySinkTask extends SinkTask {
  @Override
  public void start(Map<String, String> props) {
        // ...
  }

  @Override
  public void put(Collection<SinkRecord> records) {
     records.forEach(record -> {
        Struct payload = (Struct) record.value();
        Schema schema = record.valueSchema();
        // ... move data out of kafka
     };
  }

  @Override
  public void stop() {
     // ...
  }
}
```

Listing 2

```
Topology topology = ...

Properties config = new Properties();
config.put(StreamsConfig.APPLICATION_ID_CONFIG, "test");
config.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "dummy:1234");
testDriver = new TopologyTestDriver(topology, config);
ConsumerRecordFactory<String, String> recordFactory
  = new ConsumerRecordFactory<>(new StringSerializer(), new StringSerializer());

String payload = ...
testDriver.pipeInput(
  recordFactory.create("ttn", "foo", payload)
);

OutputVerifier.compareKeyValue(
  testDriver.readOutput("ttn-metadata", stringDeserializer,
        new JsonDeserializer<>(TtnConnectionInfo.class)),
     "003E94C6AD72F129",
     new TtnConnectionInfo("exxcellent-uno",
           "2017-04-28T11:30:40.62428417Z",
           "eui-0000024b0b03020e"));
```

Listing 4

```
final KStream<String, TtnDeviceInfo> uplinkMessages = ...
  .selectKey((k, v) -> v != null ? v.id : "no key")
  .filterNot((k, v) -> Objects.equals(k, "no key"));

final KTable<String, TtnDeviceInfo> storedValues = builder.table(
  TABLE,
  Consumed.with(Serdes.String(), new ConnectJsonSerde<>(TtnDeviceInfo.class))
);

final KStream<String, TtnDeviceInfo> updated = uplinkMessage
  .leftJoin(
     storedValues,
     TtnDeviceInfo::updatePacketsDropped,
     Joined.with(
        Serdes.String(),
        new ConnectJsonSerde<>(TtnDeviceInfo.class),
        new ConnectJsonSerde<>(TtnDeviceInfo.class))
  );
updated.to(TABLE,
  Produced.with(Serdes.String(),
     new ConnectJsonSerde<>(TtnDeviceInfo.class)));
```

Listing 5

handelt es sich um Topics mit einem Schlüssel, vergleichbar mit dem Primärschlüssel einer Tabelle einer relationalen Datenbank. Über diesen Schlüssel können dann KTables mit normalen Topics oder mit anderen KTables gejoint werden; fast so, wie man das von relationalen Datenbanken kennt *(siehe Listing 5)*.

Das Beispiel zeigt, wie eine Tabelle mit Daten aus einem Topic aktualisiert wird. Das Joinen findet mit der „leftJoin"-Methode statt. Das zweite Argument dieser Methode ist eine Funktion, die die Logik für das Aktualisieren enthält.

KTables sind damit ein bequemes Mittel, um zustandsbehaftete Daten in Kafka abzulegen. Kafka erledigt aber nicht nur die Speicherung, sondern auch die Replikation der KTables in einem Kafka-Cluster. Damit steht die Tabelle auf allen Knoten des Clusters zur Verfügung und kann von mehreren Instanzen einer Streaming-Anwendung verwendet werden.

Lessons learned

Mit Kafka, Kafka Connect und Kafka Streams lässt sich eine Microservices-Architektur entwickeln. Deshalb sollte man von Anfang an auf ein adäquates Monitoring achten. Sämtliche Bestandteile des Kafka-Ökosystems stellen via JMX Metriken zur Verfügung. Diese lassen sich dann von einem Monitoring-System (etwa die zurzeit beliebte Kombination von Prometheus und Grafana) verwenden.

Eine Metrik ist insbesondere für Streaming-Anwendungen interessant: der sogenannte „Consumer Lag". Sie gibt an, wie weit ein Consumer-Offset vom aktuellen Offset entfernt ist. Damit ist der Consumer Lag der wichtigste Hinweis auf eine zu langsame Datenverarbeitung in einem Stream.

Fazit

Das Unternehmen des Autors beschäftigt sich inzwischen seit gut zwei Jahren mit der Entwicklung und dem Betrieb einer Daten-Plattform auf Kafka-Basis. Kafka selbst hat sich dabei überwiegend als stabile und verlässliche Technologie gezeigt. Einzelne Bugs (etwa im Zusammenspiel von Kafka und Kafka Streams) wurden schon in der nächsten Version behoben.

Bei Kafka Connect ist das Bild gemischt. Einerseits lässt sich Kafka Connect bequem einrichten und einfach durch eigene Konnektoren erweitern, andererseits sind Fehler im Betrieb nur schwer festzustellen. In den neuesten Versionen wurde das Monitoring durch die Bereitstellung von Metriken zwar deutlich verbessert, ist aber immer noch eine Herausforderung.

Kafka Streams ist eine sehr gelungene Lösung, um Stream Processing in Verbindung mit Kafka zu machen. Im bisherigen Projektverlauf konnten alle Anforderungen an die Verarbeitung der Sensordaten mit Kafka Streams abgebildet werden.

Dr. Ralph Guderlei
ralph.guderlei@exxcellent.de

Dr. Ralph Guderlei ist Technology Advisor und Projektleiter bei der eXXcellent solutions GmbH in Ulm. Neben der Arbeit in unterschiedlichen Kundenprojekten berät er Teams in technologischen und methodischen Fragestellungen. Zurzeit beschäftigt er sich intensiv mit Lösungen im IoT- und Smart-City-Umfeld. Seine Erfahrungen gibt er gerne auf Konferenzen und in Fachartikeln weiter.