MQTT als Brücke

Ein Technologie-Stack für IoT-Anwendungen

Miron Kropp, Heinz Wilming

Alle IoT-Anwendungen haben ähnliche Herausforderungen auf dem Weg vom eigentlichen Sensor über Gateways bis hin zur Aggregation, Nutzung und gegebenenfalls Weiterleitung von Sensordaten zu bewältigen. Dieser Artikel stellt eine Architektur und insbesondere einen konkreten Technologie-Stack anhand einer Beispielanwendung vor. Als ein Standardprotokoll für das Internet of Things (IoT) hat sich MQTT etabliert. Daher wird hier das IoT mit standardisierter Middleware über MQTT integriert. Es wird ein kombinierter Temperatur- und Luftfeuchtigkeitssensor mit dem Bosch XDK und einer MICA-Box der Firma Harting eingesetzt. Ein maschinennaher MQTT-Broker und die JBoss-Middleware bringen diese zwei Welten zusammen.

Das Internet der Dinge

„Internet of Things" (IoT oder Internet der Dinge) und Industrie 4.0 sind derzeit in aller Munde. Eine Standardisierung für Anwendungen oder ausgereifte Plattformen in diesen Bereichen finden sich bisher nicht. Speziell im Bereich IoT sind solche Plattformen bisher nicht etabliert. Im Allgemeinen ist ein entscheidender Vorteil von Technologien aus dem Bereich IoT die schnelle Realisierung von vorzeigbaren Prototypen beziehungsweise „nearly production ready" Systemen, die Hardware und Software integrieren und sich im Weiteren dann ausbauen und skalieren lassen. Diese schnell vorzeigbaren Prototypen erleichtern es zum Beispiel, die in der heutigen Zeit schwierig zu erlangende Zustimmung für neue Projekte vom Management oder Kunden zu bekommen.

Im Folgenden stellen wir eine Architektur und einen Technologie-Stack anhand einer Beispielanwendung vor, die den Herausforderungen von IoT-Anwendungen gerecht wird. Sie ermöglichen sowohl die Erstellung schnell vorzeigbarer Prototypen als auch die Implementierung kritischer IoT-Anwendungen, die auch den Skalierungsanforderungen größerer Umgebungen gerecht werden.

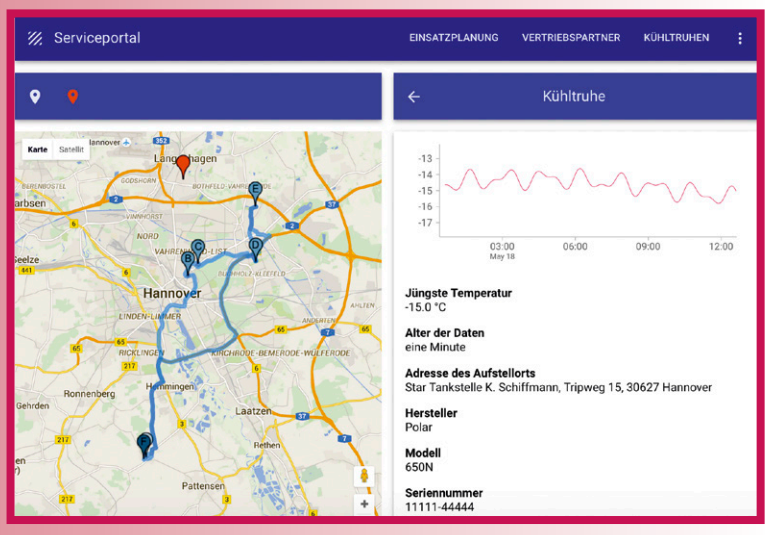

Abb. 1: Serviceanwendung für die Überwachung von Kühltruhen

Die Anwendung

Die Anwendung dient zur Überwachung der Kühlung von verderblichen Lebensmitteln über eine MICA-Box von Harting [MICA] und Bosch-XDK-Sensoren [XDK]. Neben der Fernüberwachung der optimalen Kühlung und der Fernwartung durch Parametrisierung werden die Serviceprozesse unterstützt und optimiert.

Bei einer Unterbrechung der Kühlkette und einer Überschreitung eines konfigurierbaren Schwellenwertes wird ein Alarm ausgelöst. Ein Serviceprozess wird initiiert und ein Wartungsauftrag erzeugt. Der Wartungsauftrag wird je nach Dringlichkeit in der Tages- und Tourenplanung eines verfügbaren Servicemitarbeiters berücksichtigt und eingeplant (s. Abb. 1).

Klassische Anwendungsarchitekturen und wie sich das IoT in die Architektur schleicht

In einer typischen Anwendungsarchitektur für klassische datenzentrierte Geschäftsanwendungen ist in der Regel die Geschäftslogik über einen zentralen Applikationsserver gekapselt und die Hauptlast entsteht eher durch die Verarbeitung der Daten sowie Abfragen und Änderungsanfragen. Die Maschinen- und Sensorintegration bietet jedoch ein paar Besonderheiten. Zum Beispiel senden Maschinen konstant Daten. Maschinen verfügen häufig über geringe Rechen- und Speicherressourcen (häufig im MByte-Bereich oder weniger) und die Kommunikation läuft mit geringerer Priorität als die Maschinensteuerung. Bei einem Verbindungsabbruch, Lastspitzen oder einem Anwendungsausfall darf die Maschine nicht beeinträchtigt werden und es dürfen keine Informationen verloren gehen.

Eine enge Integration und direkte Kopplung von Maschinen beziehungsweise Sensoren mit klassischen Anwendungsarchitekturen ist daher eine nicht so gute Idee und diese Anwendungsarchitekturen sind nicht geeignet für IoT-Anwendungen. Es bietet sich daher der Einsatz klassischer Pipeline/Farm-Muster an, um den Architekturanforderungen hinsichtlich Skalierbarkeit, Durchsatz und Ausfallsicherheit gerecht zu werden. Die einzelnen Anwendungskomponenten können dabei unabhängig voneinander betrieben und durch den Einsatz von Nachrichtendiensten entkoppelt werden. Ein solcher Verteilungsschnitt sollte von Beginn an berücksichtigt werden, auch wenn die Anwendung noch nicht verteilt betrieben wird und die Last zunächst gering ist.

Die in der Abbildung 2 dargestellte Architektur integriert das IoT, in unseren Beispiel Sensoren des XDKs von Bosch mit Temperatur- und Feuchtesensoren mit der MICA-Box von Harting, über MQTT [MQTT] als Nachrichtenprotokoll. Die Sensorpakete veröffentlichen dabei ihre Sensordaten via MQTT über WLAN und senden diese an die MICA-Box, auf der ein lokaler MQTT-Broker bereitgestellt ist.

Die entgegengenommenen Sensordaten werden über den lokalen MQTT-Broker auf der MICA-Box gepuffert und über eine Bridge an einen zweiten serverseitigen MQTT-Broker weitergeleitet.

Die beschriebene und in Abbildung 2 und 3 dargestellte Architektur ermöglicht, dass die Daten über MQTT-Nachrichten effizient von der Maschine zur ser-

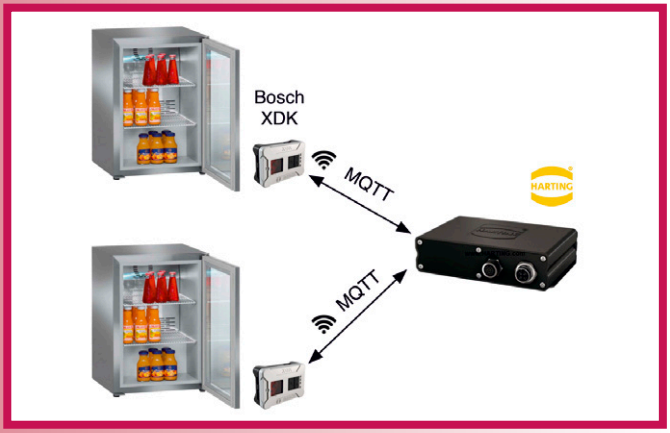

Abb. 2: Sensorintegration über die MICA-Box und MQTT

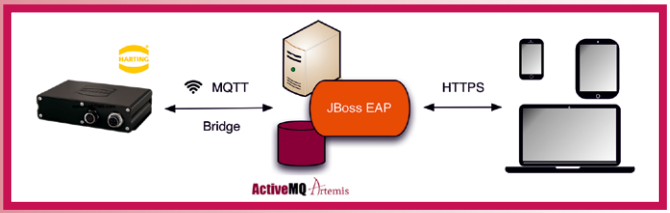

Abb. 3: Integration der MICA-Box

verbasierten Anwendung gelangen. Bei Lastspitzen oder Serverausfällen werden die Daten maschinennah zwischengespeichert. Die Integration der Sensoren über die MICA-Box ermöglicht eine hohe Skalierung, da die Daten bereits lokal aggregiert werden und somit die Serveranwendung entlastet wird. Der Einsatz von MQTT bietet auch in Umgebungen mit instabiler Verbindung eine robuste Kommunikation. Die MQTT Bridge zwischen dem lokalen MQTT-Broker auf der MICA-Box und dem serverseitigen Broker entkoppelt die Komponenten. Auf der Serverseite kommen die JBoss Enterprise Application Platform und Java EE als standardisiertes Programmiermodell zum Einsatz.

MQTT-Protokoll als Bindeglied

Message Queue Telemetry Transport (MQTT) ist ein offenes Nachrichtenprotokoll für Machine-to-Machine-Kommunikation (M2M). Das Protokoll hat sich als Standard für das IoT etabliert. Der Standard wird seit 2013 über die Organization for the Advancement of Structured Information Standards (OASIS) spezifiziert.

Das MQTT-Protokoll ist ein sogenanntes PubSub-Protokoll. Das heißt, Daten werden von Datenquellen an einen Vermittler, den Broker, veröffentlicht (published) und von interessierten Zuhörern vom Broker abonniert (subscribed). Der Broker sammelt via MQTT Daten von verschiedenen Datenquellen. Die Datenquellen, welche meist Sensoren sind, die drahtlos oder zum Beispiel per USB angeschlossen werden, veröffentlichen (publish) also ihre Daten an den Broker. Dieser läuft lokal wie hier auf einer MICA-Box von Harting, welche eine Fog-Cloud bildet, also eine lokale Cloud.

MQTT hat kleine Header und ermöglicht, dass ein MQTT-Client auch in 4 KByte Speicher von günstigen Sensor-Boards genutzt werden kann, wie zum Beispiel auf ESP2688 basierten Systemen. Offene MQTT-Client-Implementierungen sind in verschiedenen Sprachen wie Java oder C verfügbar und auch offene gerätespezifische Implementierungen werden angeboten.

Allgemein sollte man im IoT immer davon ausgehen, dass die Datenverbindungen nicht stabil sind und dass Daten nicht zwangsweise nach dem Absenden auch empfangen werden. Da MQTT ein TCP basiertes Format ist, ist es auch ein half-open Format. Es gibt keine Bestätigung für den Erhalt einer Nachricht. Daher gibt es im MQTT-Protokoll eine „Quality of Service"-Vereinbarung (QoS) zwischen dem veröffentlichenden Client und dem Broker oder dem Broker und dem abonnierenden Client. Dabei gibt es drei QoS-Niveaus:

- 0. "At most once": Die Nachricht wird einmal versendet und es wird nicht kontrolliert, ob sie ankommt. Daher muss die Nachricht nicht immer ankommen. Dieser Modus wird auch gerne „fire and forget" genannt: absenden und vergessen.
- 1. "At least once": Die Nachricht wird so oft versendet, bis der Empfang einmal bestätigt wurde. Dabei kann es jedoch auch vorkommen, dass die Nachricht mehrmals ankommt, da der Empfang erst nach dem Versand der nächsten Nachricht bestätigt wurde.
- 2. "Exactly once": Die Nachricht wird genau einmal versendet. Das erfordert Mehrfachkommunikation von Client und Broker (vier Nachrichten, zwei in jede Richtung), damit beide Seiten wissen, wann die Nachricht gelöscht werden kann (Empfänger zum Vergleich mit vorherigen Nachrichten; Sender, um sicher zu stellen, dass die Nachricht auch ankommt). Dies ist die aufwendigste und langsamste Übertragungsmethode.

XDK von Bosch

Das Cross Domain Development Kit [XDK] ist eine von Bosch verkaufte Prototyping-Plattform für das IoT in einem kleinen Package. Das XDK besteht unter anderem aus einem ARM Cortex M3 Board mit Bluetooth LE, WLAN, USB, und wird betrieben mit einer Li-Ionen-Batterie. Als entscheidende Merkmale hat das System acht Sensoren, wie Temperatur, Feuchtigkeit, Licht und Beschleunigung, die mittels eines SDK über die verschiedenen Schnittstellen ausgelesen werden können. Durch die integrierte Batterie, die per USB geladen werden kann, kann das System auch ohne externe Stromversorgung betrieben werden. Die Laufzeit ist abhängig von der eingesetzten Schnittstelle. Dabei ist das Sensorpaket mit einer IP30-Zertifizierung sehr robust ausgelegt und kann von -20°C bis 60°C bei einer relativen Luftfeuchtigkeit von 10 bis 90 Prozent eingesetzt werden.

Aufsetzend auf dem XDK wurde ein MQTT-Client implementiert, der alle Sensordaten einmal pro Sekunde an den Broker schickt. Listing 1 zeigt den MQTT-Nachrichten-Payload im JSON-Format.

```
{
  "sn": "20:C3:8F:F5:F7:79",
  "data": {
  "acc": {
    "x": 11,
    "y": 41,
    "z": 1030,
    "unit": "mG"
    },
    "gyro": {
      "x": -2197,
      "y": 3540,
      "z": -1831,
      "unit": "mdeg/s"
    },
    "mag": {
      "x": 58,
      "y": 13,
      "z": 23,
      "unit": "uT"
    },
    "light": {
```

```
      "value": 97920,
      "unit": "mLux"
    },
    "temp": {
      "value": 29190,
      "unit": "mCelsius"
    },
    "pressure": {
      "value": 100578,
      "unit": "Pascal"
    },
    "humidity": {
      "value": 21,
      "unit": "%rh"
    }
  }
}
```

Listing 1: Beispiel einer via MQTT übertragenen JSON-Nachricht mit den Messdaten aller Sensoren

Die MICA-Box

Die Harting IIC MICA (Modular Industry Computing Architecture) ist eine modulare Plattform aus offener Hard- und Software, die es möglich macht, Daten direkt aus dem Umfeld von Maschinen und Anlagen zwischenzuspeichern, auszuwerten und zu verarbeiten. Das MICA-System ist für ein raues Umfeld in der Industrie und im Bahnbereich ausgelegt, hat keinen Lüfter, ist beständig gegen Staub, Feuchtigkeit und Temperaturschwankungen (zertifiziert nach IP67) und wartungsfrei. Somit muss man bei der Box auch in schwierigen IoT-Anwendungsfällen, wie auf Baustellen, keine Ausfälle fürchten.

Das Hardwarekonzept ist entgegen Einplatinen-Computern dreigeteilt, wobei eine Platine individuell bestückt werden kann. Die MICA bietet eine Virtual-Industry-Computing-Technologie basierend auf LXC-Linux-Containern, durch die Applikationen in virtuellen Containern parallel nebeneinander laufen können (s. Abb. 4.). Es gibt bereits Container für MQTT-Broker, OPC UA, Hadoop und Java.

Aufsetzend auf dieser Softwaretechnologie wird ein Mosquitto-MQTT-Broker in einem Container gestartet, der als lokaler Datenaggregator dient und die Daten der Sensoren, im speziellen die Daten des Bosch XDK, sammelt. Mosquitto ist ein Open-Source-Messaging-Broker und implementiert MQTT in den Versionen 3.1 und 3.1.1.

Der Broker, ein Mosquitto-Container, der auf der MICA-Box läuft, sendet automatisch alle Daten, die dieser erhält, an alle Geräte, die diese Daten abonnieren (subscribe). Eine zweiter in JBoss laufender Broker abonniert die Daten des MICA-Brokers, sodass alle Sensordaten, die lokal auf der MICA eingehen, via MQTT an das Serversystem weitergeleitet werden.

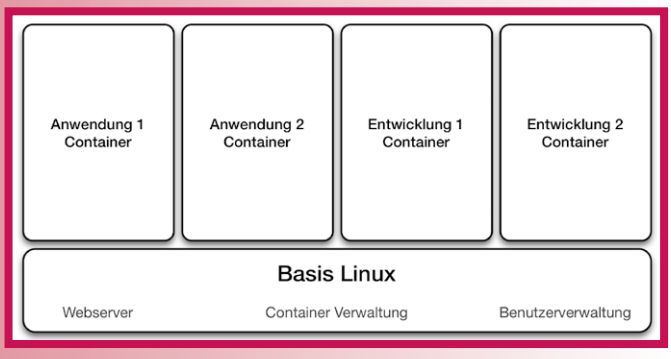

Abb. 4 Virtualisierung mittels LXC-Containern

Dieses Pattern von zwei MQTT-Brokern, ein lokaler und einer in der Cloud, hat einen entscheiden Vorteil: Bricht aus irgendeinem Grund die Verbindung zwischen lokalem und Cloud-Broker ab, sammelt der lokale Broker so lange die Daten weiter, bis die Verbindung wieder besteht (queuing), und sendet dann alle aufgelaufenen Daten an die Cloud. So gehen keine Daten verloren, obwohl keine dauerhafte Verbindung bestehen muss. Dies ist bei einigen IoT-Anwendungsszenarien, zum Beispiel in schlechter versorgten ländlichen Gebieten, von Vorteil.

Das Queuing zwischen zwei Brokern ist nur möglich, wenn das QoS-Niveau 1 oder 2 verwendet wird, da das Niveau 0 kein Queuing zulässt. Sollte Geschwindigkeit in der Datenübertragung dabei wichtig sein, ist das Verwenden von QoS 1 sinnvoll. Die Applikation sollte dann so ausgelegt sein, dass diese mit mehrfachen identischen Nachrichten umgehen kann. Ist es wichtig, dass die Nachrichten genau einmal und in der richtigen Reihenfolge ankommen, so verwendet man QoS 2, muss dann aber mit deutlich langsamerer Datenübertragung zurecht kommen.

MQTT-Client – Eclipse Paho

Die verwendete MICA-Box bietet unter anderem USB-Anschlüsse. Die Box ermöglicht daher das Anschließen von USB basierten Sensoren. In unsere Anwendung nutzen wir einen USB-Sensor zur Messung der Temperatur und Luftfeuchtigkeit. Die Daten des Sensors werden mit einer einfachen Java-Implementierung mit der Bibliothek RXTX JAVA Communication API [RXTXJCAPI] über den USB-Port ausgelesen:

```
NRSerialPort serial = new NRSerialPort( "/dev/ttyUSB0", 4800 );
serial.connect();
```

Die Werte werden zu einem JSON-String konvertiert und über einen MQTT-Client an den lokalen Broker gesendet. Als MQTT-Client wird die Java-Implementierung Eclipse Paho [Paho] verwendet. Das Eclipse-Paho-Projekt bietet neben einer Implementierung in Java weitere in C, Python, JavaScript und .Net an. Die MQTT-Nachricht ist nicht typisiert und der JSON-String mit den Sensorwerten wird als Byte-Array übergeben (s. Listing 2).

```
public class MqttPublisher implements AutoCloseable {
  private static final String topic =
    System.getProperty( "mqtt.topic", "freezer/sensordata" );
  private static final String broker =
    System.getProperty( "mqtt.broker.url", "tcp://192.168.0.1:1883" );
  private static final String clientId = System.getProperty(
    "mqtt.clientId", "freezer-" + UUID.randomUUID().toString() );
  private static final int qos = 2;
  private MqttClient sampleClient;

  public MqttPublisher() throws MqttException {
    MemoryPersistence persistence = new MemoryPersistence();
    sampleClient = new MqttClient( broker, clientId, persistence );
    MqttConnectOptions connOpts = new MqttConnectOptions();
    connOpts.setCleanSession( true );
    System.out.println( "Connecting to broker: " + broker );
    sampleClient.connect( connOpts );
    System.out.println( "Connected to broker: " + broker );
  }
  public void publishMessage( final String msg )
      throws MqttException {
    MqttMessage message = new MqttMessage( msg.getBytes() );
    message.setQos( qos );
    sampleClient.publish( topic, message );
  }
  @Override
  public void close() throws MqttException {
    sampleClient.disconnect();
    System.out.println( "Disconnected from Mqtt broker: " + broker );
```

```
    }
    @Override
    public void finalize() {
      try {
        close();
      } catch ( final MqttException e ) {
      ...
      }
    }
  }
```

Listing 2: Implementierung eines MQTT-Clients mittels Eclipse Paho zum Versenden von generierten JSON-Nachrichten

JBoss EAP 7 und ActiveMQ Artemis

Die zentrale Serverapplikation wird auf der JBoss Enterprise Application Platform [EAP] in der Version 7 betrieben. JBoss EAP ist die supportete und qualitätsgesicherte Variante des bekannten Wildfly-Applikationsservers der JBoss Community.

Der Applikationsserver enthält ActiveMQ Artemis [Artemis] als Nachrichtendienst. Neben anderen Protokollen unterstützt ActiveMQ Artemis ebenfalls MQTT. Daher ist die Kommunikation zwischen den beiden Nachrichtendiensten Mosquitto und ActiveMQ Artemis möglich.

Der Applikationsserver ist vollständig zu Java EE 7 konform. Die MQTT-Nachrichten lassen sich daher nahtlos in das Java EE-Programmiermodell integrieren und können mittels Message-Driven Beans weiter verarbeitet werden.

Zusammenfassung

Das Internet der Dinge und Industrie 4.0 schleichen sich in die Anwendungsarchitektur. Die Integration von Maschinen und Sensoren ist immer häufiger eine Anforderung. Dabei sind einige Besonderheiten hinsichtlich der Architektur zu berücksichtigen. Darüber hinaus empfehlen wir den Einsatz standardbasierter Middleware statt Eigenimplementierungen und proprietärer Lösungen. Geeignete Open-Source-Technologien sind vorhanden, insbesondere standardisierte Protokolle, Nachrichtendienste und Integrationsmiddleware.

Links

[Artemis] ActiveMQ Artemis, `https://activemq.apache.org/artemis/`
[EAP] JBoss Enterprise Application Platform, `http://www.jboss.org/products/eap/overview/`
[MICA] Modular Industry Computing Architecture, `http://www.harting-mica.com/startseite/`
[MQTT] Message Queue Telemetry Transport, `http://mqtt.org/`
[Paho] `http://www.eclipse.org/paho/`
[RXTXJCAPI] RXTX JAVA Communication API, `http://www.jcontrol.org/download/rxtx_de.html`
[XDK] Cross Domain Development Kit, `http://xdk.bosch-connectivity.com`

Dr. Miron Kropp hat Physik an der TU Berlin studiert, bevor er bei Siemens in Princeton, NY im Bereich Softwareentwicklung arbeitete. Darauf hat er seine Doktorarbeit im Bereich Sensoren und Halbleiterentwicklung am IMSAS in Bremen abgeschlossen, um anschließend als Projektmanager für Automatisierungstechnik bei BIOTRONIK SE & Co. KG zu arbeiten. Im Anschluss half er als CTO zweier Start-ups, die ersten Wachstumsschmerzen zu überwinden, bevor er seine jetzige Position als Arbeitskreisleiter für Industrie 4.0 bei der akquinet tech@spree GmbH angenommen hat.
E-Mail: miron.kropp@akquinet.de

Heinz Wilming ist Leiter des JBoss Competence Center der akquinet AG. Er beschäftigt sich dort mit Technologien und Architekturen für verteilte Anwendungen. Sein Fokus liegt dabei auf der Java EE-Plattform, JBoss Middleware und Open-Source-Technologien.
E-Mail: heinz.wilming@akquinet.de