Internet of Cars - In-Vehicle Services mit Open Source IoT Technologien umsetzen

Dipl.-Ing. Günter Obiltschnig
Applied Informatics Software Engineering GmbH
Maria Elend 143
9182 Maria Elend
Austria
guenter.obiltschnig@appinf.com

Abstract

Spätestens mit der verpflichtenden europaweiten Einführung des elektronischen Notrufsystems eCall ab Ende März 2018 wird jedes Neufahrzeug über eine Mobilfunkanbindung verfügen. Von einem eCall System ist der Schritt zu einer universellen Plattform für Telematik-dienste im Fahrzeug (In-Vehicle Services) nicht mehr weit. Der nächste Evolutionsschritt bei solchen Telematiksystemen wird von starren Systemen mit fixem Funktionsumfang zu dynamisch erweiterbaren, flexiblen Plattformen basierend auf IoT Technologien führen, die jederzeit, während des gesamten Lebenszyklus des Fahrzeuges, um neue Dienste, auch von Drittanbietern, erweitert werden können. Dieser Artikel stellt ein mögliches Umsetzungskonzept eines solchen Systems mit Open Source Technologien vor.

Einleitung

Telematikdienste (In-Vehicle Services) basierend auf mobilen Kommunikations-technologien (GSM, UMTS, LTE), wie z. B. die Steuerung der Fahrzeug-klimatisierung oder die Abfrage des Ladezustands der Batterie bei Elektrofahrzeugen per Smart Phone sind im Bereich der Premiumfahrzeuge schon längere Zeit verfügbar. Spätestens mit der verpflichtenden europaweiten Einführung des elektronischen Notrufsystems *eCall* ab Ende März 2018 wird aber jedes Neufahrzeug über eine Mobilfunkanbindung verfügen. Von einem eCall System ist der Schritt zu einer universellen Plattform für Telematikdienste im Fahrzeug nicht mehr weit. Insbesondere gibt es mittlerweile kostengünstige Mobilfunkmodule die auf mindestens einem der Rechenkerne ein Linux System ausführen können, was aus Software-Sicht viele Möglichkeiten für innovative Applikationen öffnet. Der nächste Evolutionsschritt bei solchen Telematiksystemen wird daher von starren Systemen mit fixem Funktionsumfang zu dynamisch erweiterbaren, flexiblen, auf IoT Technologien basierenden Plattformen führen, die jederzeit und während des gesamten Lebenszyklus des Fahrzeuges um neue Dienste, auch von Drittanbietern, erweitert werden können. Beispiele wären Dienste im Bereich Flottenmanagement, Car Sharing oder Rental. Dabei ergeben sich natürlich besondere Herausforderungen hinsichtlich Zugriffsschutz und Datensicherheit. So darf ein nachträglich installierter Dienst eines Drittanbieters keinesfalls die Funktionsweise des Gesamtsystems

negativ beeinflussen. Auch dem Thema Datenschutz ist besondere Aufmerksamkeit zu widmen.

In diesem Artikel wird eine Software-Architektur für eine flexible Telematik-Dienstplattform im Fahrzeug vorgestellt, die auf Linux, bewährten Open Source Software Frameworks und standardisierten und offenen IoT Protokollen basiert. Die Software ermöglicht die dynamische, aber trotzdem sichere Installation von neuen Software Komponenten zur Bereitstellung neuer Services im laufenden Betrieb. Fahrzeughersteller können auf der Telematik-Box im Fahrzeug Programmier-schnittstellen (APIs) bereitstellen, um die Entwicklung von „Telematik-Apps“ und In-Vehicle Services durch Drittanbieter zu ermöglichen. Die Entwicklung der Apps kann dabei in den High-Level Programmiersprachen C++ oder JavaScript erfolgen, wobei eine Sandbox-Umgebung garantiert, dass Software von Drittanbietern die Funktion des Gesamtsystems nicht negativ beeinflussen kann. Die Kommunikation mit Cloud-Diensten erfolgt flexibel und sicher über Protokolle wie MQTT oder HTTPS und REST APIs. Der Entwicklungsaufwand für die Software einer solchen Telematik-Dienstplattform kann damit signifikant reduziert werden.

Eine Open Source IoT Software Plattform für Telematik Dienste

macchina.io [1] ist ein Softwarebaukasten zum schnellen Prototyping, sowie zur Entwicklung industrieller Edge- und Fog Computing Applikationen für das Internet der Dinge (IoT). Diese Applikationen laufen auf vernetzten, typischerweise Linux-basierten Geräten und kommunizieren einerseits mit lokal angebundenen Sensoren, Aktoren und anderen Geräten sowie andererseits auch mit Cloud Applikationen. Zu diesem Zweck bietet macchina.io eine Vielzahl an Softwaremodulen die flexibel kombiniert werden können. So werden z. B. verschiedene Kommunikations-protokolle und -technologien aus dem IoT und Automatisierungsbereich unterstützt, wie z. B. Bluetooth LE, 6LoWPAN, Modbus oder OPC-UA, aber auch Internet Technologien wie HTTP, REST und MQTT. Weiters sind Programmierschnittstellen zur Positionsbestimmung und Anbindung von GNSS/GPS Receivern vorhanden. Die Anbindung an Fahrzeugbussysteme (CAN, FlexRay, Ethernet) ist ebenso möglich. Dies erfolgt aus Sicherheitsgründen üblicherweise über ein herstellerspezifisches Bus Gateway.

macchina.io unterstützt mehrere Programmiersprachen, bzw. Programmiermodelle. Als Programmiersprachen werden derzeit C++ und JavaScript unterstützt. C++ wird vornehmlich für low-level Code (z. B. Geräteanbindungen und Kommunikations-protokolle) verwendet, während JavaScript die Entwicklung der Applikationslogik stark vereinfacht und somit beschleunigt. Zusätzlich steht eine Dataflow Engine mit einem graphischen, web-basiertem Editor zur Verfügung um einfache Applikationen auch ohne Schreiben von Code realisieren zu können.

Ein in macchina.io integrierter Web Server mit diversen Web-Applikationen ermöglicht speziell während der Entwicklungsphase eine vereinfachte Interaktion mit dem Gerät. In der endgültigen Software kann der Web Server dann weggelassen werden.

Komponenten – Bundles und Services

macchina.io basiert auf einer komponentenbasierten, modularen und hierarchischen Architektur (Abbildung 1) deren Kernkonzepte *Bundles* und *Services* sind, implementiert im *Open Service Platform* Framework. Ein Bundle ist ein

Deployment-Container für Software-Komponenten, die aus ausführbarem Code (C++ Shared Libraries oder JavaScript), diversen Ressourcen und Konfigurationsdateien bestehen, realisiert als Zip Datei mit genau definierter Verzeichnisstruktur. Bundles können untereinander Abhängigkeiten definieren und haben einen definierten Lebenszyklus, der dynamisch zur Laufzeit gesteuert werden kann. So können jederzeit neue Bundles im System installiert werden, bestehende Bundles gestartet, gestoppt oder entfernt werden, oder auch durch neuere Versionen ersetzt werden (Abbildung 2). Bundles können kryptographisch signiert werden um den Erzeuger eindeutig identifizieren und Manipulationen ausschließen zu können.

Abbildung 1: macchina.io Architektur

macchina.io selbst besteht aus über 60 Bundles, welche flexibel kombiniert werden können. Auch der Web Server oder das JavaScript Laufzeitsystem sind in verschiedenen Bundles implementiert und können somit weggelassen oder ersetzt werden. Bundles können entweder als Teil des Hauptprozesses laufen, auf mehrere Prozesse aufgeteilt werden, oder in eingeschränkten und abgeschotteten „Sandbox“ Prozessen laufen.

Ein Bundle kann verschiedene Dienste (Services) bereitstellen, die von anderen Bundles über eine *Service Registry* gefunden und verwendet werden können. Die Suche nach Services erfolgt entweder über den eindeutigen Namen (falls ein spezifisches, bekanntes Service gefunden werden soll), oder über Eigenschaften. So kann z. B. nach allen verfügbaren Temperatursensoren gesucht werden. Services werden als C++ Klassen implementiert. Ein in macchina.io integriertes *Remoting* Framework ermöglicht es, diese Services z. B. aus JavaScript Code, aus anderen (Sandbox-)Prozessen (IPC), oder auch remote, z. B. per MQTT/JSON-RPC aufzurufen. Die Vergabe und Kontroller detaillierter Zugriffsrechte ist möglich.

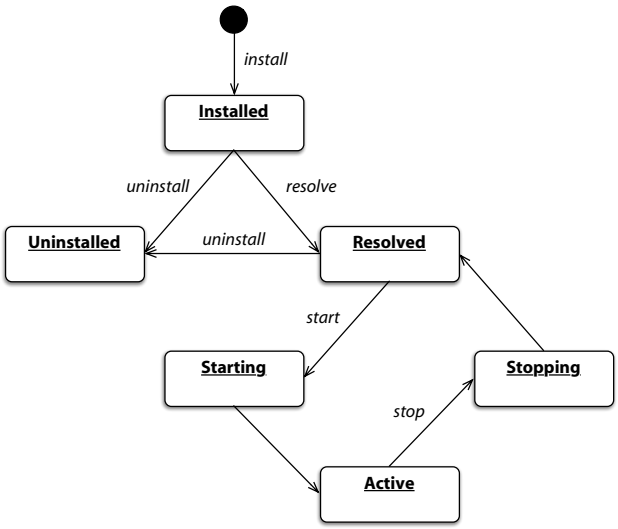

Abbildung 2: Lebenszyklus eines Bundles

Für viele Services gibt es vordefinierte Interfaces zu denen unterschiedliche Implementierungen existieren können. Beispielsweise können Fahrzeugdaten wie z. B. aktuelle Geschwindigkeit, Motordrehzahl, zurückgelegte Entfernung, Verbrauch, Kühlwassertemperatur, usw. über ein vordefiniertes *Sensor* Interface (siehe Listing 1) bereitgestellt werden. Diese Interfaces stellen auch Event-Mechanismen (inklusive Event-Filter) bereit, um eine Applikation bei Änderung eines Wertes notifizieren zu können. Die im Listing ersichtlichen speziellen Kommentare (z. B. „//@ remote“) werden vom Code Generator des Remoting Frameworks verarbeitet und ermöglichen Einflussnahme auf die Code Generierung.

```
//@ remote
class Sensor: public Device
{
public:
    //@ filter=true
    Poco::BasicEvent<const double> valueChanged;

    virtual double value() const = 0;

    virtual bool ready() const = 0;

    void clearValueChangedFilter(
        const std::string& subscriberURI);

    void setValueChangedIsGreaterThanFilter(
        const std::string& subscriberURI, double limit);

    // ...

    void setValueChangedMinimumDeltaFilter(
        const std::string& subscriberURI, double delta);

    void setValueChangedMinimumIntervalFilter(
        const std::string& subscriberURI, long milliseconds);

};
```

Listing 1: Sensor Interface (vereinfacht)

In der Entwicklungsphase können anstatt realer Sensordaten (die über das Fahrzeugbussystem erfasst werden) auch Simulationen verwendet werden, wobei dies aus Sicht der Applikationslogik vollkommen transparent ist.

In JavaScript kann z. B. die aktuelle Geschwindigkeit des Fahrzeuges über folgenden Code ermittelt werden, der zuerst das Service, welches die Geschwindigkeit bereitstellt (*Speedometer*), findet, und dann die Geschwindigkeit abfrägt.

```
// Search for speedometer in the Service Registry
var speedoRefs = serviceRegistry.find(
    'io.macchina.physicalQuantity == "speed"');
if (speedoRefs.length > 0)
{
    // Found at least one speedometer – resolve it.
    var speedo = speedoRefs[0].instance();

    // Get current speed.
    var speed = speedo.value();
    logger.information('Current speed: %f', speed);
}
else
{
    logger.error('No speedometer found.');
}
```

Listing 2: Abfrage der aktuellen Geschwindigkeit mit JavaScript

Telematikdienste

Nach demselben Schema lassen sich auch Telematikdienste realisieren. Üblicherweise besteht ein solcher Dienst aus einem Cloud Backend, welches eine REST API bereitstellt, die z. B. von Smart Phone Apps oder anderen Applikationen genutzt werden kann. Die Kommunikation zwischen Cloud und Telematiksystem im Fahrzeug kann über das bekannte und weit verbreitete MQTT Protokoll [2] erfolgen. MQTT (MQ Telemetry Transport oder Message Queue Telemetry Transport) ist ein offenes und standardisiertes Nachrichtenprotokoll für Machine-to-Machine-Kommunikation (M2M), das die Übertragung von Nachrichten zwischen Geräten, bzw. Systemen ermöglicht und auch bei unzuverlässigen Netzwerkverbindungen, hohen Latenzen und geringen Bandbreiten gut funktioniert. Ein großer Vorteil bei der Verwendung von MQTT ist, dass das Telematiksystem im Fahrzeug von sich aus eine Verbindung zum Server, bzw. MQTT Broker, aufbaut und somit keine extern erreichbaren Ports benötigt werden, die als Eintrittspunkt für Angriffe dienen könnten. Weiters ermöglicht MQTT die sichere, verschlüsselte Kommunikation über TLS, sowie die gegenseitige Authentifizierung über Zertifikate. Sobald eine MQTT Verbindung zwischen Telematiksystem und Cloud-Backend aufgebaut wurde, kann bidirektional kommuniziert werden, wobei Nachrichten an bestimmte „Topics“ gesendet werden, für die sich alle am Broker angemeldeten Systeme (sowohl Cloud Backend, als auch Telematiksysteme) registrieren können („publish-subscribe“). MQTT gibt kein Nutzdatenformat vor. Es lassen sich sowohl binäre Daten, als auch z. B. JSON oder XML-formatierte Daten übertragen. Um nun ein Service auf dem Telematiksystem über MQTT anzusprechen bietet sich z. B. das JSON-RPC Protokoll [3] an.

Als Beispiel soll ein vereinfachtes Service zur Steuerung der Fahrzeugklimatisierung dienen. Das Interface könnte wie in Listing 3 gezeigt aussehen.

```
//@ remote
class ClimateControl
{
public:
      enum Status
      {
            CC_OFF,
            CC_ON,
            CC_HEATING,
            CC_COOLING
      };

    Poco::BasicEvent<const double> targetTemperatureReached;
    Poco::BasicEvent<const Status> statusChanged;

    void setTargetTemperature(double temperature);
    double getTargetTemperature() const;

    double getCurrentTemperature() const;

    void setPower(bool on);
    bool getPower() const;

    Status getStatus() const;
};
```

Listing 3: Vereinfachtes Service zur Fahrzeugklimatisierung

Mit Hilfe des in macchina.io integrierten Remoting Frameworks kann dieses Service über MQTT und das JSON-RPC Protokoll vom Cloud Backend angesprochen werden. Um z. B. die Klimatisierung einzuschalten (aufgrund eines entsprechenden Befehls über eine Smart Phone App) würde das Backend eine JSON-RPC Nachricht an das Topic */vehicle/<VIN>/services/ClimateControl/request* senden (wobei <VIN> für die Fahrzeug-Identifizierungsnummer steht).

```
{
    "jsonrpc": "2.0",
    "method": "setPower",
    "params":
    {
        "on": true
    },
    "id": 1
}
```

Das Remoting Framework im Telematik System erhält diese Nachricht und erkennt anhand des Topics, dass das *ClimateControl* Service angesprochen werden soll. Welche Methode aufgerufen werden soll, sowie die Parameter, sind aus dem Inhalt der JSON-RPC Nachricht ersichtlich.

Nach dem gleichen Prinzip können auch Services bereitgestellt werden, um neue Bundles, die weitere Services anbieten können, im Telematiksystem zu installieren. Entsprechende APIs werden von macchina.io bereitgestellt.

Zusammenfassung

In diesem Artikel wurde gezeigt wie mit Hilfe des Open Source IoT Software Frameworks macchina.io die Software für ein flexibel erweiterbares Telematiksystem mit In-Vehicle Services umgesetzt werden kann. Dabei wurden sowohl die Aspekte Integration in das Fahrzeugbussystem, als auch Kommunikation mit Cloud Systemen angesprochen.

Referenzen

[1] https://macchina.io, https://github.com/macchina-io/macchina.io

[2] http://mqtt.org

[3] http://www.jsonrpc.org/specification

Autor

Günter Obiltschnig ist Gründer der Open Source Projekte macchina.io und POCO C++ Libraries, sowie Geschäftsführer der Applied Informatics GmbH, einem Software-Unternehmen spezialisiert auf Tools und Dienstleistungen rund um das Internet der Dinge. Er verfügt über 20 Jahre Erfahrung in der Entwicklung von Software für verschiedenste Systeme - von verteilten Unternehmensapplikationen bis zu Embedded Systemen und hält regelmäßig Vorträge auf Fachkonferenzen.