Miron Kropp
(miron.kropp@akquinet.de)
hat Physik an der TU-Berlin studiert, bevor er bei SIEMENS in Princeton, NY im Bereich Softwareentwicklung arbeitete. Nach Abschluss seiner Doktorarbeit im Bereich Sensoren und Halbleiterentwicklung am IMSAS in Bremen war er zunächst als Projektmanager für Automatisierungstechnik bei BIOTRONIK SE & Co. KG und anschließend als CTO bei zwei Start-ups tätig, bevor er seine jetzige Position als Arbeitskreisleiter für Industrie 4.0 bei akquinet tech@spree GmbH übernahm.

Kai Puth
(kai.puth@akquinet.de)
hat seinen Bachelor in Medieninformatik an der FH Kaiserslautern gemacht mit den Schwerpunkten Usability Engineering und Computergrafik. Nach seinem Master-Abschluss in Medieninformatik an der HTW Berlin mit den Schwerpunkten Computer Vision und Augmented Reality arbeitete er als Softwareentwickler im medizinischen Bereich. 2015 hat er seine jetzige Position bei akquinet tech@spree GmbH als Entwickler mit Expertise in User Experience Design angetreten.

Martin Möller
(martin.moeller@akquinet.de)
hat sein Informatik-Diplom an der TU Berlin gemacht. Schon während des Studiums hat er als Freelancer im Bereich konzeptionelle Gestaltung, Implementierung und Wartung von Webanwendungen für KMUs gearbeitet. Seit März 2012 unterstützt er die akquinet tech@spree GmbH als Entwickler, Architekt und Berater in den Bereichen Webtechnologien, verteilte Systeme und Anwendungsentwicklung. Er ist zertifizierter JBoss-Entwickler.

Maschinensteuerung mit OPC und JBoss Middleware im Browser

Industrie 4.0 wird häufig als die vierte industrielle Revolution bezeichnet. Die Revolution, die sich dahinter verbirgt, bezieht sich auf die Kommunikation von verteilten (industriellen) Systemen. Daher umfasst ein großer Teil von Industrie 4.0 die verteilte Steuerung und Überwachung von Maschinen. In der Regel werden Industrieanlagen heutzutage von einer oder mehreren speicherprogrammierbaren Steuerungen (SPS) kontrolliert. In diesem Artikel zeigen wir, wie man mittels JBOSS EAP Middleware (Java EE) aus einem Webbrowser heraus eine Maschine mit SPS-Controller steuern kann [EAP]. Die SPS wird dabei über das herstellerunabhängige OPC-Protokoll gesteuert [OPC]. Hierzu läuft ein OPC-Server auf einem Rechner, der via Netzwerk mit der Maschine verbunden ist. OPC-Server kommunizieren SPS-spezifisch mit der Maschine und bilden die OPC-Schnittstelle nach außen ab. Um die Brücke in die Java-Welt zu schaffen, wird die Open-Source-Bibliothek Utgard aus dem OpenSCADA-Projekt verwendet, die eine Kommunikation mittels OPC-Schnittstelle ermöglicht [Utgard; openSCADA].

EINFÜHRUNG IN OPC

OPC steht für „OLE for Process Control“ und ist ein Software-Schnittstellen-Standard, der es Microsoft-Windows-Programmen erlaubt, via die „Distributed Component Object Model“-Schnittstelle (DCOM) mit industrieller Hardware zu kommunizieren [OPC; DCOM]. OPC wird dabei immer in einer Server-Struktur verwendet. Dabei übersetzt der Server die Hardware-Kommunikationsprotokolle einer SPS in das OPC-Protokoll, das durch den Klienten implementiert und verstanden wird. Der OPC-Klient nutzt so den OPC-Server, um Daten aus der SPS auszulesen und Kommandos oder Daten zur Hardware zu schicken.

Der OPC- oder auch „OPC Data Access“-Standard (OPC DA), der die Realtime-Datenkommunikation zwischen Hardware und Interface-Geräten (wie zum Beispiel HMIs), SCADA-Systemen (Supervisory Control And Data Acquisition) und ERP-/MES-Systemen definiert, ist dabei nur der erste Teil einer Gruppe von OPC-Spezifikationen. Um auch historische Daten lesen zu können, kann man auf den Standard „OPC Historical Data Access“ (OPC HDA) zurückgreifen. Für Events und Alarme wird der – wie der Name schon sagt – „OPC Alarms and Events“-Standard (OPC AE) spezifiziert.

Heute existiert der weiterentwickelte Standard „OPC Unified Architecture“ (OPC UA). Er bietet einige Vorteile gegenüber der ursprünglichen OPC-Spezifikation. Die Kommunikation über die DCOM-Schnittstelle wird durch Kommunikation über TCP/IP, HTTPS und SOAP ersetzt. Der neue Standard ermöglicht den Einsatz von OPC UA auf jeder Plattform und jedem Gerät, unter anderem Embedded Devices, unabhän-

gig von deren Sprache. OPC UA ermöglicht verbessertes Sicherheitsmanagement und sichere Verbindungen via WAN. Insgesamt vereint OPC UA alle vorherigen Modelle in einem gemeinsamen Datenmodell. Seit 2011 ist der neue Standard auch als IEC-Standard 62541 angenommen.

Trotz all dieser Vorteile von OPC UA gegenüber OPC DA werden heutzutage noch viele Anlagen mit der regulären OPC-Schnittstelle entwickelt, unter anderem weil OPC UA oft zusätzliche Hardware in der Maschine erfordert und Entwickler sich mit der regulären OPC-Struktur besser auskennen. Zusätzlich können ältere Maschinen oft über die OPC-DA-Schnittstelle in Industrie 4.0 eingebunden werden. Im Folgenden erläutern wir, wie eine mögliche Implementierung aussehen kann.

SYSTEMÜBERSICHT

Wie in *Abbildung 1* gezeigt, besteht das System aus einer Webanwendung, die mit einem OPC-Server und einer relationalen Datenbank zusammenarbeitet. Mehrere SPS-gesteuerte Maschinen sind mit dem OPC-Server verbunden, der als Mittelsmann zwischen den einzelnen Maschinen und der zentralen Webanwendung fungiert. Die Datenbank dient als nichtflüchtiger Speicher für ausgewählte Domänenereignisse (Auditlog). Die Benutzeroberfläche ist als Single-Page Application realisiert und läuft asynchron zur Webanwendung im Browser des Benutzers.

Das System kennt prinzipiell zwei Datenflüsse. Zum einen sendet jede Maschine regelmäßig Statusupdates bzw. Messwerte, die über den OPC-Server an die Webanwendung weitergeleitet werden. Der OPC-Server und die Webanwendung befinden sich auf demselben System und kommunizieren über DCOM miteinander. Die Webabwendung wandelt die Messwerte in Domänenereignisse um, die parallel in der Datenbank gesichert und über WebSockets an den Browser des Benutzers geschickt werden. Wenn der Benutzer der Maschine Befehle gibt, werden diese über REST over HTTP an die Webanwendung gesendet. Von dort aus werden die Befehle parallel über den OPC-Server an die Maschine versandt und als Domänenereignisse in die Datenbank gesichert. Die Webanwendung sendet für jeden Befehl einen Erfolgsindikator über REST over HTTP zurück an das Frontend.

Die OPC-Schnittstelle der Anwendung wurde über den Utgard Connector realisiert. Alle der OPC-Schnittstelle nachgelagerten Datentransformationen sind funktional implementiert und werden über Java-8-Streams abgewickelt. Jedes Domänenereignis wird in einer Message-Queue hinterlegt, bis es verarbeitet werden kann. Mit jedem Domänenereignis wird das Modell des Zustands der angeschlossenen Maschine aktualisiert. Für die Kommunikation mit dem Frontend werden ein WebSocket und mehrere REST-Ressourcen angeboten.

Im Folgenden werden einige der genannten Komponenten und eingesetzten Technologien genauer erläutert.

UTGARD

Utgard ist eine Open-Source-Bibliothek aus dem openSCADA-Projekt [Utgard; openSCADA]. Sie stellt OPC-Klienten eine Java-API zur Kommunikation mit OPC-Servern zur Verfügung. Für jede OPC-Adresse kann ein Klient einen Listener registrieren, um auf Änderungen am Datenbestand des OPC-Servers zu reagieren. Der Klient selbst kann über Utgard auch an Adressen auf dem OPC-Server schreiben. Ferner kann der Klient so konfiguriert werden, dass er sich nach Verbindungsabbrüchen selbstständig wieder mit dem Server verbindet. Ob er sich selbstständig verbinden soll und wie groß die Verzögerung zwischen den Verbindungsversuchen sein soll, kann zur Laufzeit konfiguriert werden. Die DCOM-Kommunikation wird von Utgard mittels der Open-Source-Java-Bibliothek j-Interop realisiert [J-Interop].

Utgard liefert die Messwerte als DCOM-Datentypen (Distributed Component Object Model; Klassen aus j-interop). Diese werden zuerst auf einfache Java-Datentypen abgebildet und dann in Domain-Events

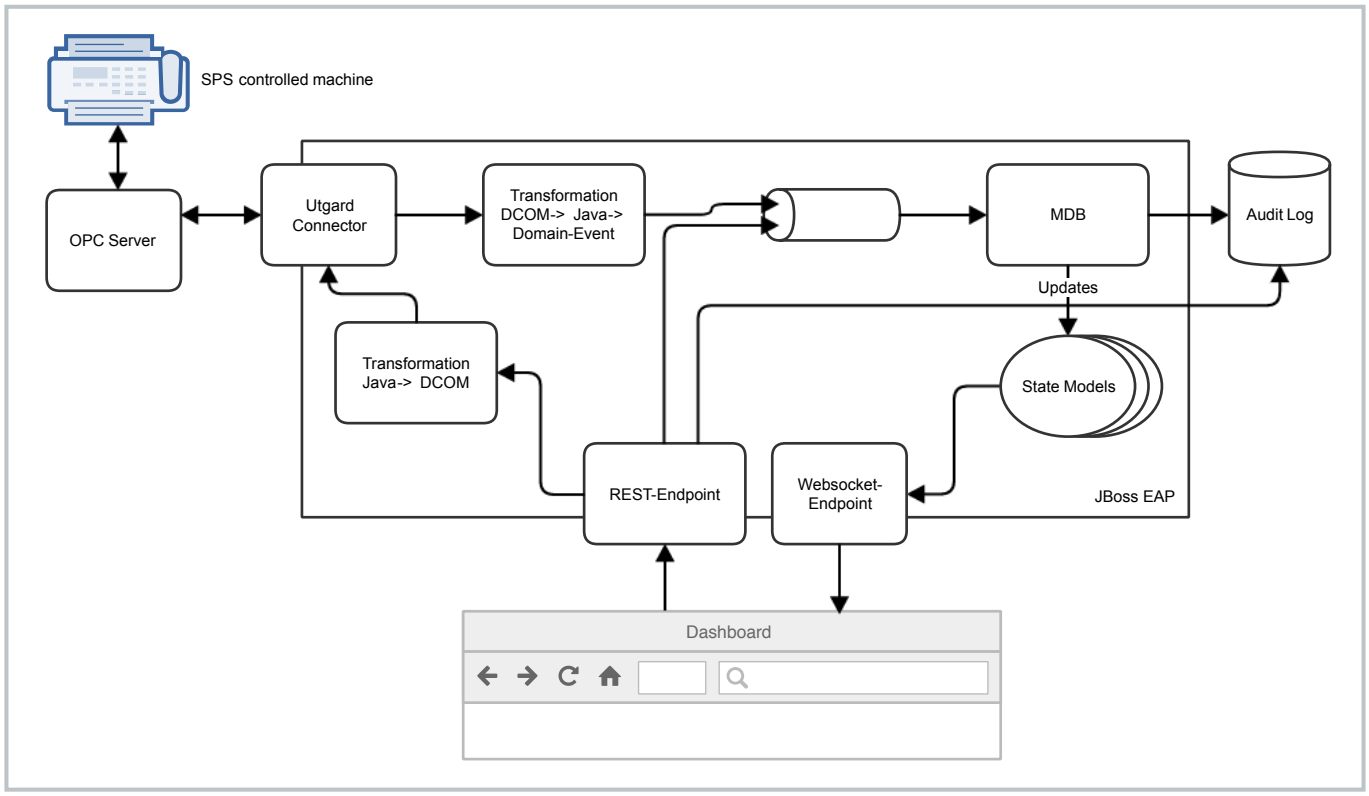

Abb. 1: Überblick über einen möglichen Aufbau eines Systems zum Überwachen und Steuern einer SPS-kontrollierten Maschine

eingebettet. Diese mehrteilige Transformation basiert auf dem mit Java 8 eingeführten *Function*-Interface und *Optional*. Der erste Schritt in der Transformation ist ein sogenannter *Reader*. Die Signatur eines *Readers* ist

```
Function
<ItemState, Optional<T>>
```

Listing 1 zeigt ein Beispiel für eine Implementierung. Die Verwendung von *Optional* erlaubt es, die Verarbeitung zu überspringen, im Fall, dass der ausgelesene Wert leer ist. Auch eine valueChangedFilter()-Funktion gibt *Optional.empty()* zurück, falls sich der Wert nicht geändert hat. Im letzten Schritt wird dann aus dem Inhalt des *Optional* ein Domain-Event, das in die Message-Queue geschrieben werden kann. Die Verwendung von Function<X,Y> erlaubt es, mit wiederverwendbaren und kombinierbaren (*Function.andThen()*)-Einheiten zu arbeiten. Diese lassen sich einfach und isoliert testen.

```
public static final Function<ItemState,
Optional<Boolean>> BOOLEAN_READER = itemS-
tate -> {
  try {
    if (isNullOrEmpty(itemState)) {
      return Optional.empty();
    }

    final boolean value = itemState.getValue().get-
ObjectAsBoolean();
    return Optional.of(value);
  } catch (JIException e) {
    throw new RuntimeException(„failed to read
boolean from", e);
  }
};

public static <T> Function<Optional<T>,
Optional<T>> valueChangedFilter() {
...
}
private static Function<ItemState,
Optional<MachineUpdate>> transformer(int ma-
chineId) {
  Function<Optional<Boolean>,
Optional<MachineUpdate>> toUpdate = optional ->
      optional.map(on -> MachineUpdate.createLig
htStateUpdate(machineId, on));
  return BOOLEAN_READER.
andThen(valueChangedFilter()).andThen(toUpdate);
}
```

Listing 1: *Daten-Transformation (Reader) eines DCOM-Datentyps durch Abbildung auf einfache Java-Datentypen und Einbettung in Domain-Events*

SYSTEMZUSTÄNDE UND IHRE FLÜCHTIGKEIT

Grundsätzlich hängt die Ausgestaltung des Zustandsmodells von der Fachlichkeit der zu steuernden Maschinen ab. Die Systemarchitektur erlaubt gleichermaßen die Abbildung des Systemzustands auf zustandsbehaftete Objekte wie auf unveränderliche Objekte. In der gewählten Umsetzung ist die Anwendung an sich zustandslos. Das heißt, die Systemzustände der Anlage werden von der Maschine selbst verwaltet. Die Anwendung fragt die Zustände über den OPC-Server ab, wobei in der ersten Abfrage alle Informationen verarbeitet werden. In den folgenden Abfragen werden jedoch nur die Änderungen verarbeitet. Die Zustände werden dann In-Memory für die Darstellung vorgehalten. Der Systemzustand wird dabei über zustandsbehaftete Objekte dargestellt, die in der Webanwendung von einer Service-Bean verwaltet und für die gesamte Anwendung nur einmal erzeugt werden. Bei einem Verbindungsausfall werden nach einem Reconnect die Update-Daten erneut geladen und daraus der aktuelle Zustand ermittelt. Bei einem Systemausfall entsteht durch den Verlust des Systemzustands kein Schaden, da der aktuelle Systemzustand von der Maschine verwaltet wird und nach einem Neustart einfach erneut abgefragt werden kann.

Einzig das Auditlog ist in einem nichtflüchtigen Speicher hinterlegt, da hier wichtige Domänenereignisse protokolliert werden, die zum Beispiel der Rückverfolgung auftretender Ereignisse – wie zum Beispiel Alarme – und deren Handhabung dienen. Es enthält keine Systemzustände.

VERARBEITEN DER DOMAIN-EVENTS

Die in der Daten-Transformation erwähnten und erzeugten Domain-Events werden in eine JMS-Queue (Java Messaging Service) geschrieben [JMS]. Verarbeitet werden die Messages aus der Queue von einer Message-Driven Bean (MDB) – eine Bean, die es Java-EE-Applikationen ermöglicht, Nachrichten asynchron zu verarbeiten [MDB]. Die Bean aktualisiert den Systemzustand und protokolliert den Vorgang im Auditlog. Während der Systemzustand nur im Arbeitsspeicher existiert, ist das Auditlog in einer relationalen Datenbank hinterlegt. Der Systemzustand wird auf Enterprise JavaBeans nach dem Singleton-Pattern abgebildet. Die Verwendung dieses Patterns stellt sicher, dass nur eine Abbildung des Systemzustands vorhanden ist und zu keinem Zeitpunkt der Systemzustand mehrfach repräsentiert ist.

KOMMUNIKATION MIT DER BENUTZEROBERFLÄCHE

Das Frontend basiert auf Angular 1 und läuft als Single-Page Application (SPA) im Browser des Benutzers. Frontend und Backend kommunizieren über WebSockets und eine REST-Schnittstelle.

Updates werden vom Server über WebSockets an die Klienten gesendet. Das erfordert eine vorherige Registrierung der Klienten am Server. Um die Verbindung am Leben zu erhalten, sendet jeder Klient in regelmäßigen Zeitabständen ein Lebenssignal an den Server. Wenn dieses für eine gewisse Zeit ausbleibt, schließt der Server den Kanal. Beim ersten Verbindungsaufbau sendet der Server den gesamten Zustand der Maschinen. Im weiteren Verlauf werden nur Zustandsänderungen an den Klienten geschickt.

Schreibkommandos werden vom Frontend nicht über die WebSocket-Verbindung gesendet, sondern über die erwähnte REST-Schnittstelle. Durch die synchrone Natur von HTTP können wir so auf Fehler direkt reagieren. Dadurch wird eine separate Abfrage der Zustände vermieden und über den WebSocket muss nur noch der aktuelle Zustand der Maschine bereitgestellt werden. Das verringert die Komplexität des Kommunikationsmodells, da eine Trennung von Maschinensteuerung und Maschinenüberwachung gegeben ist.

Daneben gibt es ein Auditlog, das den Verlauf aller relevanten Domänenereignisse sammelt. Mittels dieses Logs kann der Benutzer im Browser die Ereignisse einsehen. Das Auditlog wird als separate SPA realisiert, da die zugrunde liegenden Datenmodelle und die verwendeten REST-Ressourcen unabhängig von der zuvor beschriebenen Applikation (Frontend) sind. Dabei sammelt das Auditlog Informationen sowohl aus der Maschinenüberwachung als auch aus der Maschinensteuerung. Es wird in einer Postgres-Datenbank persistent vorgehalten. Die Datenbank wird durch die Java Persistance API (JPA) angesprochen. Zur Sicherung wird das Auditlog parallel in täglich wechselnde CSV-Dateien geschrieben.

TESTEN DER ANWENDUNG

Wie es in Enterprise-Projekten üblich sein sollte, wird die Qualität der Softwareentwicklung durch automatisierte Tests unterstützt. Es werden Unit Tests und Integrationstests der laufenden Anwendung

– unter anderem mit dem Datenbanksystem – durchgeführt. Es werden die REST-Schnittstelle, die WebSocket-Schnittstelle und die Benutzeroberfläche getestet – Letztere mit Selenium Webdriver. Um vor jedem Test die gleiche Ausgangslage zu schaffen, wird die Datenbank durch einen Docker-Container bereitgestellt, der vor jedem Integrationstest in gleichem Zustand neu gestartet wird. Automatisierte Tests mit einer Maschine sind häufig schwer realisierbar, da diese oft nicht verfügbar sind. Daher wurde eine eigene Schnittstelle geschaffen, über die ein einfacher selbst geschriebener Simulator angeschlossen wird. Dieser ermöglicht auch Tests der Gesamtapplikation auf Maschineninteraktion, obwohl keine Anlage zur Verfügung steht. Die letztendliche Testautomatisierung wird über Maven und Jenkins realisiert.

Um während der Entwicklung die Verbindung zur Maschine via OPC-Anbindung testen zu können, wird eine stark abgespeckte Version der Software verwendet. Diese besteht nur aus dem Utgard-Connector und den Transformationen (DCOM → Java → Domain-Event). Die entstehenden Ereignisse werden in ein Logfile geschrieben. Als Fat-JAR verpackt, kann damit die OPC-Anbindung früh in der Entwicklung getestet und verbessert werden.

HERAUSFORDERUNGEN UND ERFAHRUNGEN

Der Entwicklungsprozess ist nicht ohne Herausforderungen. Die Arbeit mit einer SPS via einen OPC-Server ist sehr hardwarenah. Eine SPS verhält sich dabei wie ein Embedded System. Die Konzepte unterscheiden sich stark von den Konzepten, die bei der Java-Enterprise-Entwicklung verwendet werden. Daher müssen die Entwicklungsteams der Maschine und der verteilten Steuerung ein Verständnis der jeweils gegenseitigen Konzepte entwickeln, um Fehler durch „Selbstverständlichkeitsannahmen“ zu verhindern.

Eine weitere Herausforderung kann darin bestehen, dass OPC DA, wie oben bereits beschrieben, sehr Windows-nah ist und Entwickler auf Missverständnisse bei der Entwicklung stoßen. Wenn die Entwicklung der verteilten Steuerung mit der Entwicklung der Maschine zusammenfällt, kann das zunächst ein Vorteil sein. Man kann mit den Maschinenentwicklern gemeinsam auf sinnvoll zu realisierende Applikationen hinarbeiten. Auf der anderen Seite stellt dann das Testen eine Herausforderung dar, im Speziellen wenn die Maschinen nur in geringer Stückzahl oder gar nicht zur Verfügung steht und zum Beispiel wegen ihrer Größe nur beim Kunden vor Ort getestet werden kann. In diesem Fall sind automatisierte Tests nicht möglich oder wie oben beschrieben nur über Simulatoren machbar. Auch Rechts- und Sicherheitsaspekte wie nötige Einweisungen oder Zertifizierungen können das Testen mit den Maschinen vor allem im Produktionsumfeld erschweren.

ZUSAMMENFASSUNG

Industrie 4.0 und vernetzte Systeme sind ein Zukunftsthema. In diesem Artikel haben wir demonstriert, wie man mit Java auch Maschinen mit der relativ hardwarenahen OPC-Schnittstelle verteilt überwachen und steuern kann. Mit Hilfe von Open-Source-Projekten haben wir eine zustandslose Architektur, basierend auf einem JBoss EAP Server, mit REST-Schnittstelle und WebSockets vorgestellt, die mit einer Single-Page Application im Frontend kommuniziert und so die Steuerung und Überwachung einer SPS-basierten Maschine ermöglicht. Weiterhin haben wir diverse Herausforderungen aufgezeigt, von denen einige sich auf Testen mit der Maschine beziehen, da die Verfügbarkeit der Anlagen oft nicht gegeben ist. ■

LINKS

[DCOM] Distributed Component Object Model, https://msdn.microsoft.com/library/cc201989.aspx

[EAP] JBoss Enterprise Application Platform, http://www.jboss.org/products/eap/overview/

[J-Interop] j-Interop a Java Open Source library that implements the DCOM wire protocol, http://www.j-interop.org/

[JMS] Java Message Service, http://www.oracle.com/technetwork/java/jms/index.html

[MDB] Message Driven Bean, http://docs.oracle.com/javaee/6/tutorial/doc/gipko.html

[OPC] OPC Foundation, https://opcfoundation.org/

[openSCADA] openSCADA, an open source Supervisory Control And Data Acquisition System, http://openscada.org/

[Utgard] Utgard, a vendor-independent, 100% pure JAVA OPC Client API, http://openscada.org/projects/utgard/