Übertragung von Sensordaten unter Berücksichtigung von Transaktionskonzepten in der SensorCloud, Zwischenergebnisse [DBAP6]

15.01.2013,

Henning Budde

Thomas Partsch

Gefördert durch:

aufgrund eines Beschlusses
des Deutschen Bundestages

1. Problemstellung

In diesem Arbeitspaket ist eine Familie von Diensten zu entwickeln, die die Übertragung von Sensordaten vom LocationManster (Gateway) in die Cloud Datenbank als auch umgekehrt ausführt. Diese Dienste sollen unter Berücksichtigung des Konzepts von Transaktionen programmiert werden. Auf diese Dienste soll bei der Visualisierung der Sensordaten eines LocationMasters im Web zurückgegriffen werden.

Bei Transaktionen auf ein oder mehrere DBMS müssen die ACID-Eigenschaften unbedingt eingehalten werden. ACID steht für Atomarität, Konsistenz, Integrität und Dauerhaftigkeit. Sie beschreiben erwünschte Eigenschaften von Transaktionen in Datenbankmanagementsystemen (DBMS). Die ACID-Eigenschaften spielen auch bei den Transaktionen im Projekt SensorCloud eine essentielle Rolle. Egal, ob Transaktionen der Sensordaten eines Gateways Richtung eines Rechnerknotens der SensorCloud oder ob Transaktionen von Konfigurationsupdates von einem Rechnerknoten der SensorCloud Richtung Gateway durchgeführt werden, in beiden Fällen müssen die ACID-Eigenschaften eingehalten werden.

Allgemeiner gesagt: Das Einhalten der ACID-Eigenschaften ist ein wichtiges Qualitätsmerkmal für Vertrauen in den Datenaustausch zwischen Rechnerknoten einer Cloud.

2. Fragestellung in Hinsicht auf das Transaktionskonzept

Bei Datenbanktransaktionen unterscheidet man zwischen klassischen Transaktionen, also Transaktionen auf ein DBMS und Datenbanktransaktionen auf mehrere DBMS, den sogenannten geschachtelten bzw. verteilten Transaktionen. Bei verteilten Transaktionen spielen Transaktionsmanager eine zentrale Rolle. Hierbei stellt sich die Frage, welche Open Source Transaktionsmanager existieren, wie funktionieren sie (auf welchen Standards basieren sie) und wie wird eine verteilte Transaktion durchgeführt.

Für den Fall der klassischen Transaktion stellt sich ebenfalls die Frage nach tauglichen Transaktionsmanagern. Können vielleicht dieselben Transaktionsmanager, wie sie bei verteilten Transaktionen eingesetzt werden, auch für eine klassische Transaktion auf einem nicht verteilten System angewandt werden?

Eine weitere Fragestellung ist die Tauglichkeit von Datenbanksystemen in Bezug auf Transaktionssicherheit. Das bedeutet, dass die Speicherengine (oder nur Engine genannt) eines DBMS Transaktionen und Mechanismen wie Beginn, Commit und Rollback, verwalten können muss.

Unterstützen die Datenbanksysteme, die in der Deploymentanalyse untersucht werden, verteilte Transaktionen überhaupt und weisen sie Transaktionssicherheit auf?

3. Ziele

Das Hauptziel des Arbeitspakets der Untersuchung von Transaktionen für Sensordaten ist die Entwicklung von Diensten wie Transaktionen, die zwischen den Gateways und der SensorCloud

Gefördert durch:

aufgrund eines Beschlusses
des Deutschen Bundestages

durchgeführt werden können. Dabei sollen die Transaktionen einerseits von der SensorCloud und andererseits von dem Gateway initialisiert werden können (Pull- und Push-Prinzip), wie die beiden nachfolgenden Abbildungen verdeutlichen sollen.

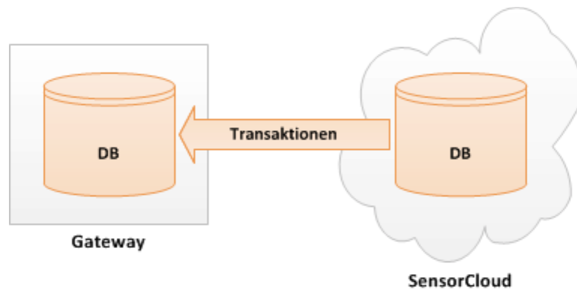

Abbildung 1: Transaktionen initialisiert von der SensorCloud (Pull-Prinzip)

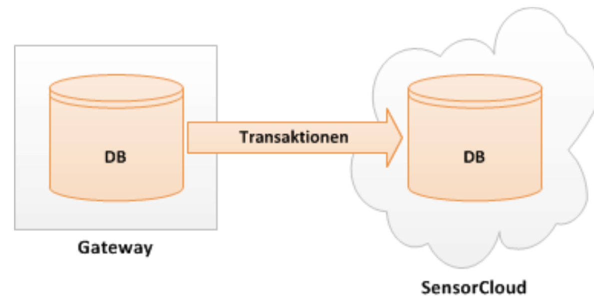

Abbildung 2: Transaktionen initialisiert vom Gateway (Push-Prinzip)

4. Transaktionen initialisiert von der SensorCloud (Pull-Prinzip)

Robuste Transaktionsmechanismen sind heutzutage weit erforscht und eine Selbstverständlichkeit für Schreib- oder Lesezugriffe innerhalb eines Datenbankmanagementsystemd (DBMS). Vorausgesetzt, dass die Engine des DBMS auch Transaktionen unterstützt (Stichwort Transaktionssicherheit). Kommerzielle wie auch OpenSource Transaktionsmanager übernehmen die Verwaltung und Durchführung einer Transaktion auf einem DBMS.
Die bisherigen Untersuchungen in diesem Arbeitspaket, haben sich mit den verteilten Transaktionen beschäftigt. Dabei wurde das Konzept der verteilten Transaktionen von Seiten der SensorCloud in Richtung mehrerer Gateways untersucht. Dabei soll eine globale Transaktion mit mehrere Subtransaktionen von einem Rechnerknoten der SensorCloud initialisiert werden. Auf diese Weise kann die SensorCloud eine globale Transaktion mit n Subtransaktionen auf n Gateways ausführen. Handelt es sich dabei um einen Lesezugriff, so wäre die SensorCloud in der Lage, gleichzeitig die Sensordaten von mehreren Gateways auszulesen, um sie anschließend in ihre eigene Datenbank zu speichern.

Vergleich von geschachtelten- und verteilten Transaktionen

Der Vergleich von geschachtelten und verteilten Transaktionen folgt der Untersuchung von Conrad et al. [1].

- In einem verteilten Datenbanksystem werden die Transaktionen verteilt auf mehreren Rechnern ausgeführt. Jede lokale Transaktion ist Teil einer globalen Transaktion. Es gibt also nur eine Transaktionsverwaltungsebene (Ebene 1).

  Ein verteiltes Datenbanksystem besitzt zudem keine autonomen Knoten. Ein verteiltes Datenbanksystem fungiert fast wie ein zentrales Datenbanksystem, nur mit dem Unterschied, dass die Datenbank auf mehreren Knoten (Rechnern) verteilt ist. Man spricht auch von verteilt implementiertes Datenbanksystem.

Gefördert durch:

aufgrund eines Beschlusses des Deutschen Bundestages

- In einem föderierten Datenbanksystem führt eine globale, also geschachtelte Transaktion mehrere Subtransaktionen auf mehreren Komponentensystemen aus. Auf den Komponentensystemen werden aber noch zusätzlich rein lokale Transaktionen, unabhängig von den geschachtelten Transaktionen (mit ihren Subtransaktionen) ausgeführt.

  Ein föderiertes Datenbanksystem besitzt demnach eine globale Transaktionsverwaltung (Ebene 1) genau wie bei den verteilten Transaktionen, sowie eine lokale Transaktionsverwaltung (Ebene 0) in den Komponentensystemen.

Verteilte und geschachtelte Transaktionen haben einige Gemeinsamkeiten, aber auch Unterschiede. Das Datenbanksystem im Projekt der SensorCloud wird ein föderietes Datenbanksystem sein, da jedes Gateway einen Teil des globalen Datenbankschemas der SensorCloud beinhalten wird. Ein jedes Gateway wird auch selbst, also autonom Transaktionen Richtung SensorCloud durchführen können (Push-Prinzip). Es wird also eine eigene Transaktionsverwaltung besitzen.

Aus der Sicht der SensorCloud (bzw. aus der Sicht eines Rechnerknoten der SensorCloud), ergeben sich aber auch verteilte Transaktionen, falls eine globale Transaktion mit Teiltransaktionen gleich auf mehrere Gasteways angewandt wird.

4.1 X/Open DTP Modell

Verteilte Transaktionen werden durch das X/Open DTP Modell [2] erst möglich. Das X/Open DTP-Modell (DTP = Distributed Transaction Processing) ist ein entwickelter technischer Standard für verteilte Transaktionsverarbeitung. Die X/Open DTP Systemarchitektur definiert mehrere Schnittstellen, von denen hier nur auf die XA, XA+ und TX eingegangen wird.

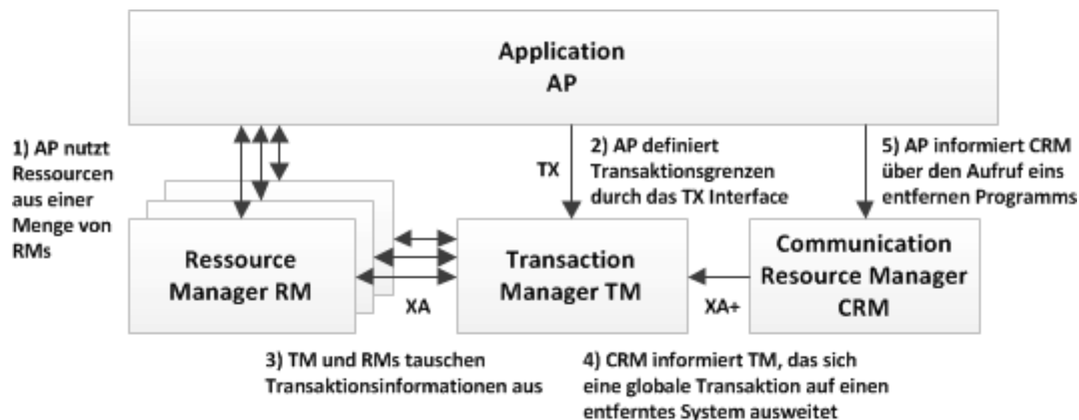

Abbildung 3: X/Open DTP-Modell

Die X/Open DTP XA Spezifikation [3]

Die XA-Schnittstelle ist eine bidirektionale Schnittstelle zwischen einem Transaction Manager und einem Ressource Manager (siehe Abbildung oben). Alle Ressourcen Manager, die an einer globalen Transaktion teilnehmen möchten, müssen die XA-Schnittstelle implementiert haben. Die

Gefördert durch:

Bundesministerium für Wirtschaft und Technologie

aufgrund eines Beschlusses des Deutschen Bundestages

Basis der XA-Schnittstelle stellt das Zwei-Phasen-Commit-Protokoll (verantwortlich für die korrekte Durchführung einer globalen Transaktion und Einhaltung der ACID-Eigenschaften) dar.

Die X/Open DTP TX Spezifikation [4]

Die TX-Schnittstelle beschreibt eine Programmierbare Schnittstelle (API), mit der eine Application mit einen Transaction Manager, für eine globale Transaktion kommunizieren kann. Die TX-Schnittstelle ist ebenfalls für die Abgrenzung globaler Transaktionen und deren Vollständigkeit verantwortlich.

Die X/Open DTP XA+ Spezifikation [5]

Die XA+ Spezifikation beschreibt die gleichnamige XA+-Schnittstelle im DTP-Modell, die durch die Erweiterung des DTP Modells, durch einen Communication Ressource Manager hinzugekommen ist.

4.1.1 Zwei-Phasen-Commit-Protokoll

In föderierten bzw. verteilten Datenbanksystemen müssen wie in lokalen bzw. zentralen Datenbanksystemen die bereits vorgestellten ACID-Eigenschaften erfüllt werden. Die Erfüllung der ACID-Eigenschaften, in verteilten Systemen, garantiert das Zwei-Phasen-Commit-Protokoll (basierend auf der XA-Spezifikation).

1. Die Prepare-Phase

Am Ende einer globalen Transaktion fragt ein Koordinator alle Teilnehmer, ob sie in der Lage sind, die Subtransaktionen erfolgreich durchzuführen. Dazu sendet der Koordinator eine Prepare-Nachricht an alle Teilnehmer und wartet auf die Antworten. Die Teilnehmer haben bis zu diesem Zeitpunkt nur die Ressourcen aggregiert, die sie für die Durchführung der Datenmanipulation ihrer Subtransaktion benötigen.

2. Die Commit-/Rollback-Phase
In dieser Phase gibt es zwei mögliche Verläufe:

- Der Koordinator erhält von allen Teilnehmern ein Commit als Antwort. Daraufhin sendet der Koordinator ein Commit an alle Teilnehmer, welches die Teilnehmer dazu bringt die Datenmanipulation ihrer Subtransaktion durchzuführen. Anschließend antworten alle Teilnehmer mit einem Acknowledgment und der Koordinator beendet die globale Transaktion mit einem Commit.
- Der Koordinator erhält mindestens von einem Teilnehmer ein Abort oder ein Timeout als Antwort. Daraufhin sendet der Koordinator ein Abort, an alle Teilnehmer, welches die Teilnehmer dazu bringt mit einem Rollback ihre Subtransaktion abzuschließen und die aggregierten Ressourcen wieder frei zu geben. Anschließend antworten alle Teilnehmer mit einem Acknowledgment und der Koordinator beendet die globale Transaktion mit einem Rollback.

Gefördert durch:

aufgrund eines Beschlusses
des Deutschen Bundestages

4.2 Java Transaction API

Um eine verteilte Transaktion, wie in den beiden vorgestellten Szenarien durchführen zu können, bedient man sich dem X/Open DTP Modells, genauer dem X/Open XA-Standard, der Open Group, welches in der Java Transaction API (JTA) [6] implementiert ist.

Die JTA beschreibt eine Programmierschnittstelle für verteilte Transaktionen über mehrere XA-Ressourcen (das können z.B. Datenbanken sein). Der Zweck der JTA ist es eine lokale Java Schnittstelle zu definieren, welche der Transaktionsmanager für sein Transaktionsmanagement in verteilten Java Enterprise Umfeld unterstützt.

Die JTA Schnittstelle ist ein Bestandteil der JavaEE. Möchte man anstatt der JavaEE (Java Enterprise Edition) jedoch die JavaSE (Java Standard Edition) verwenden, so muss die Implementierung der JTA in Form eines Transaktionsmanager zugrunde gezogen werden. Bekannte Open Source Transaktionsmanager sind z.B.:

- Atomikos TransactionEssentials [7]
- JBoss Transaction Service [8]
- Bitronix Transaction Manager [9]
- Java Open Transaction Manager [10]

Es muss noch erwähnt werden, dass die einzelnen Transaktionsmanager auf der JTA basieren, jedoch eigene Anpassungen in der Implementierung der JTA beinhalten können

4.3 Datenbanksysteme mit XA-Unterstützung

Datenbanksysteme, die an einer verteilten Transaktion, auf Basis der JTA teilnehmen möchten, müssen die XA-Schnittstelle unterstützen. In der Deploymentanalyse werden Datenbanksysteme wie MySQL, PostgreSQL, Derby, HSQL, Berkeley DB, SQLite und db4o auf Tauglichkeit als Gateway-Datenbank untersucht.

Von den oben genannten sieben Datenbanksystemen unterstützen drei Datenbanksysteme die XA-Schnittstelle. Dies gilt für die MySQL, die PostgreSQL und für die Derby Datenbank. Die Berkeley DB unterstützt XA eingeschränkt nur unter der Verwendung eines JCA (Java Connector Architecture) Ressource Adapters und eines Application Servers. Bei der HSQL ist laut eigenen Angaben die XA Unterstützung in Arbeit. Bei SQLite und db4o existieren in den eigenen Dokumentationen der Datenbanksysteme keine Informationen über eine XA Unterstützung.

Würde man demnach eine XA-Transaktion auf die in der Deploymentanalyse untersuchten Datenbanken durchführen wollen, so wären die Kandidaten MySQL, PostgreSQL und Derby zu wählen.

4.4 Anwendung einer verteilten Transaktion in der FDBS-Teststrecke

Die FDBS-Gruppe der FH-Köln, hat eine eigene Teststrecke für die SensorCloud entwickelt. Die Teststrecke hat den nachfolgend zu sehenden Aufbau.

Gefördert durch:

aufgrund eines Beschlusses des Deutschen Bundestages

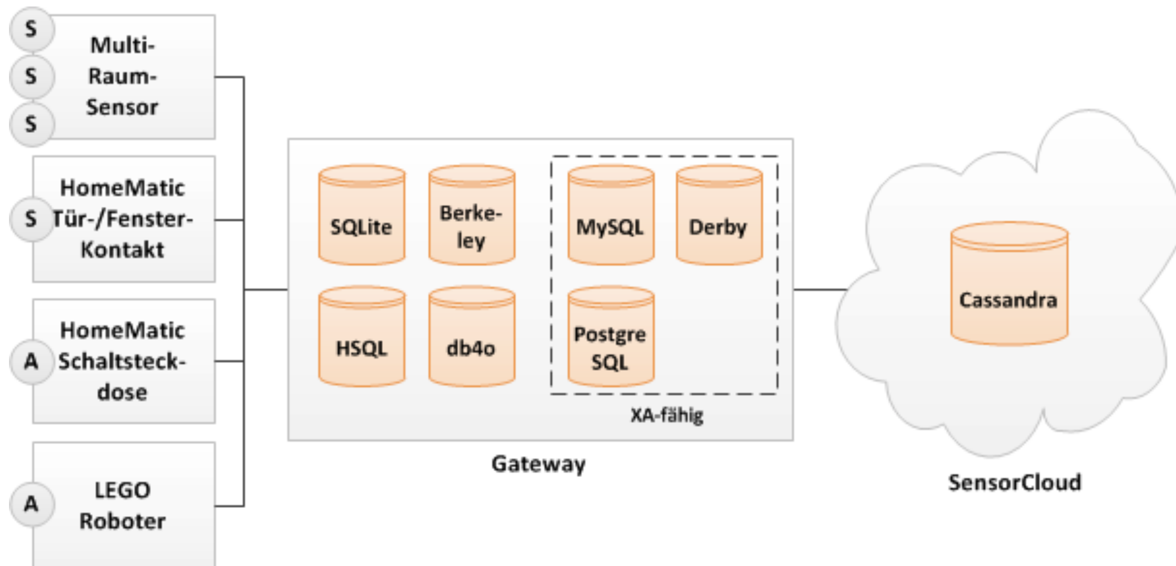

Abbildung 4: FDBS-Teststrecke

In der Teststrecke sind links zwei Sensoren zu sehen, der Multi-Raum-Sensor, der drei Sensormesswerte liefert, nämlich Bewegung, Luftfeuchte- und Lufttemperatur sowie die Luftqualität. Der zweite Sensor, ist der HomeMatic Tür-/Fensterkontakt, der eine Bewegung misst wenn sich die Tür öffnet bzw. wieder schließt. Des Weiteren, sind zwei Aktoren, nämlich die HomeMatic Schaltsteckdose und ein LEGO Roboter vorhanden.

In der Mitte befindet sich das Gateway (ein Einplatinenrechner), welcher die sieben aus der Deploymentanalyse untersuchten DBMS installiert hat. Darunter befinden sich auch die drei XA-fähigen DBMS.

Rechts in der Abbildung befindet sich ein SensorCloud Rechnerknoten, auf dem das Cassandra DBMS installiert ist.

Die Praxistauglichkeit der verteilten Transaktionen wurde anhand eines SensorCloud Rechnerknotens und des Master Gateways in der FDBS-Teststrecke untersucht. Dabei wurde ein Programm namens *XAPull2Cloud* geschrieben, welches eine verteilte Transaktion, mit drei Subtransaktionen, von einem Rechnerknoten der SensorCloud initialisiert (siehe Abbildung unten). Die so gelesenen Sensordaten aus den Gateway-Datenbanken werden anschließend in die SensorCloud Cassandra-Datenbank geschrieben

Gefördert durch:

aufgrund eines Beschlusses
des Deutschen Bundestages

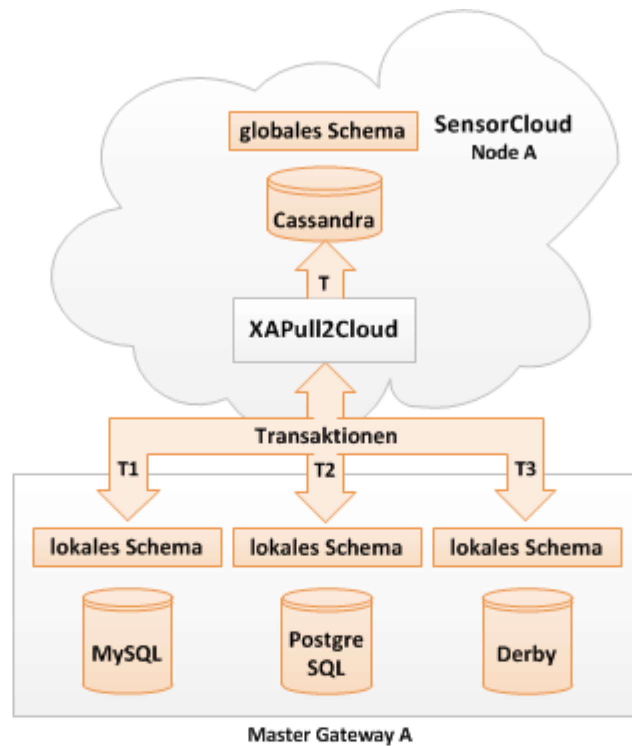

Abbildung 5: XA-Transaktionen in der FDBS-Teststrecke

Als Transaktionsmanager wurden der Atomikos TransactionEssentials und der JBoss Transaction Service verwendet. Beide erfüllten ihre Aufgabe ohne Schwierigkeiten. Der Java Open Transaction Manager erwies sich als zu veraltet (Letztes Softwareupdate 2010) und schied mangels Aktualität und Kompatibilität zum eingesetzten Spring Framework [11] aus.

Die XA-Transaktion wurde, wie auf der oberen Abbildung zu sehen ist, auf einem Gateway getestet und durchgeführt. Im realen Fall befänden sich die drei DBMS auf verschiedenen, geographisch verteilten Gateways.

Klassendiagramm

Für jede der drei DBS in der oberen Abbildung wird eine Interfaceklasse und deren konkrete Implementierung erstellt. Ein Zugriff auf die drei Interfaces stellt der Transaktionsservice (Klasse *TService*) sicher. Die Klasse *InsertInCassandra* ist dafür zuständig, die gelesenen Messwerte der drei DBS in die Cloud-Datenbank Cassandra zu übertragen und die Klasse *XAPull2Cloud* bildet die Main-Klasse zur Ausführung des gesamten Services.

Gefördert durch:

aufgrund eines Beschlusses
des Deutschen Bundestages

<<Java Class>>
XAPull2Cloud
(default package)
log: Log
tServiceContextXml: ApplicationContext
XAPull2Cloud()
executeT():void
main(String[]):void

<<Java Interface>>
TService
de.sensorcloud.transaction.xa.gateway
setDaoT1(DaoT1):void
setDaoT2(DaoT2):void
setDaoT3(DaoT3):void
executeDaoT():void

<<Java Class>>
InsertInCassandra
de.sensorcloud.transaction.nonxa.cloud
log: Log
con: Connection
InsertInCassandra()
connect():void
writeDB(ResultSet,String):void

<<Java Class>>
TServiceImpl
de.sensorcloud.transaction.xa.gateway
TServiceImpl()
setDaoT1(DaoT1):void
setDaoT2(DaoT2):void
setDaoT3(DaoT3):void
executeDaoT():void

-daoT1 0..1
-daoT2 0..1
-daoT3 0..1

<<Java Interface>>
DaoT1
de.sensorcloud.transaction.xa.gateway
readMesswertT1():void

<<Java Interface>>
DaoT2
de.sensorcloud.transaction.xa.gateway
readMesswertT2():void

<<Java Interface>>
DaoT3
de.sensorcloud.transaction.xa.gateway
readMesswertT3():void

<<Java Class>>
DaoT1Impl
de.sensorcloud.transaction.xa.gateway
log: Log
select_all_sql: String
DaoT1Impl()
readMesswertT1():void

<<Java Class>>
DaoT2Impl
de.sensorcloud.transaction.xa.gateway
log: Log
select_all_sql: String
DaoT2Impl()
readMesswertT2():void

<<Java Class>>
DaoT3Impl
de.sensorcloud.transaction.xa.gateway
log: Log
select_all_sql: String
DaoT3Impl()
readMesswertT3():void

Abbildung 6: Klassendiagramm - XAPull2Cloud-Service

Die Interfacearchitektur ermöglicht ein komfortables Hinzufügen und Entfernen von weiteren DBS bei dem Transaktionsservice.

4.5 Anwendung einer klassischen Transaktion in der FDBS-Teststrecke

Als klassischer Ansatz eines Transaktionskonzepts, initialisiert aus der SensorCloud, wurde das Programm *Pull2Cloud* entwickelt. Dieses Programm führt eine klassische Transaktion auf ein DBMS eines Gateways durch. Die so gelesenen Sensordaten werden anschließend in die Cassandra Datenbank der SensorCloud geschrieben.

Bei den DBMS-Systemen auf dem Gateway muss es sich um Client/Server DBS handeln, da ansonsten keine Verbindung aus der SensorCloud auf diese Systeme erstellt werden kann. Aus diesem Grund eignen sich nur die Datenbankprodukte: MySQL, PostgreSQL, Derby, und HSQLDB für dieses Transaktionskonzept nach dem Pull-Prinzip.

Gefördert durch:

aufgrund eines Beschlusses
des Deutschen Bundestages

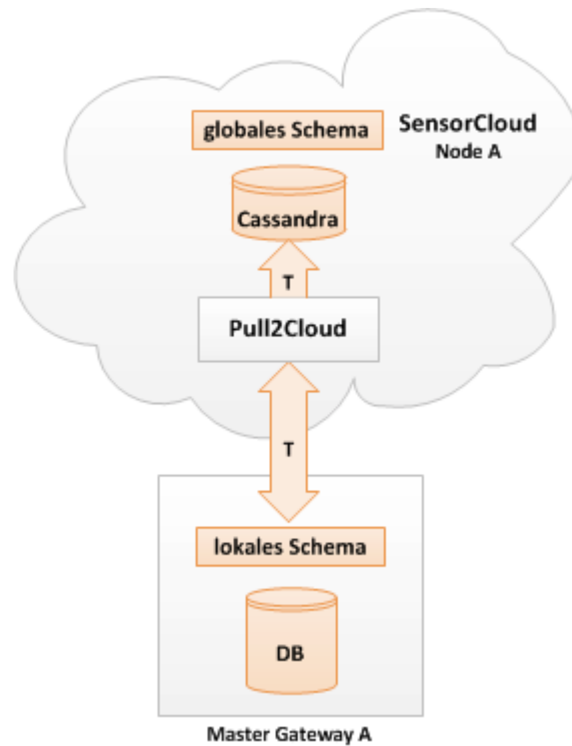

Abbildung 7: Klassische Transaktion, initialisiert von der SensorCloud

Das Gateway bleibt bei den Pull-Transaktionen nicht autonom. Es muss keine Transaktionen selbst nitialisieren, sondern wartet nur darauf, bis die nächste Transaktion von der SensorCloud auf das Gateway initialisiert wird. Auch wenn ein Sensor einen Alarm oder ähnliches meldet, bleibt das Gateway handlungsunfähig. Der Alarmwert kann erst bei dem nächsten lesenden Intervall in die SensorCloud gelangen. Für den Kunden würde es bedeutet, dass er vielleicht zu spät über einen Alarm informiert wird.

Klassendiagramm

Das Klassendiagramm ähnelt stark dem Diagramm des *XAPull2Cloud*-Services bis auf die Tatsache, dass jetzt nur noch ein Interface nötig ist, da die Transaktion nur auf ein DBS des Gateway durchgeführt wird. Aus diesem Grund ist auch keine Transaktionsservice-Klasse mehr vorhanden.

Gefördert durch:

aufgrund eines Beschlusses
des Deutschen Bundestages

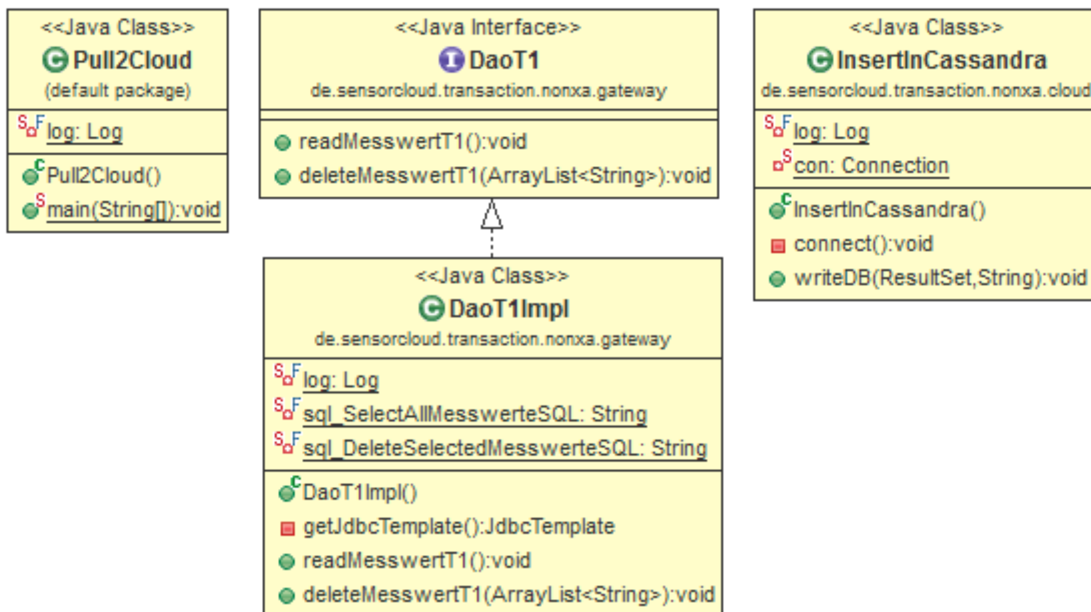

Abbildung 8: Klassendiagramm - Pull2Cloud-Service

Eine Neuerung betrifft die *DaoT1Impl*-Klasse. In ihr befindet sich die *deleteMesswerT1*-Methode, die dafür sorgt, dass die Übertragenen Datensätze vom Gateway in die SensorCloud, auch wieder von der Gateway-Datenbank entfernt werden. Schließlich soll das Gateway nicht ewig die Sensordaten zwischenspeichern. Diese *„Löschstrategie“* ist in dem Meilensteinbericht *„Syncon Commloss - Zwischenspeicherung von Sensordaten bei fehlender Internetverbindung zur SensorCloud, Ergebnisse [DBAP7]“* im Kapitel 1.4 Strategie B1 (A. Stec) ausführlich beschrieben.

5. Transaktionen initialisiert vom Gateway (Push-Prinzip)

Als klassischer Ansatz eines Transaktionskonzepts, initialisiert vom Gateway, wurde das Programm *Push2Cloud* entwickelt. Das Programm läuft auf dem Gateway und kann per Crontab so eingestellt werden, dass es regelmäßig (z.B. alle 10 Minuten), die Sensordaten der Gasteway-Datenbank, selbstständig in die SensorCloud Cassandra-Datenbank schreibt. Hierbei ist es jetzt auch möglich, die Sensordaten eines nicht Client/Server DBMS wie z.B. SQLite, in die Cloud zu transferieren.

Gefördert durch:

aufgrund eines Beschlusses des Deutschen Bundestages

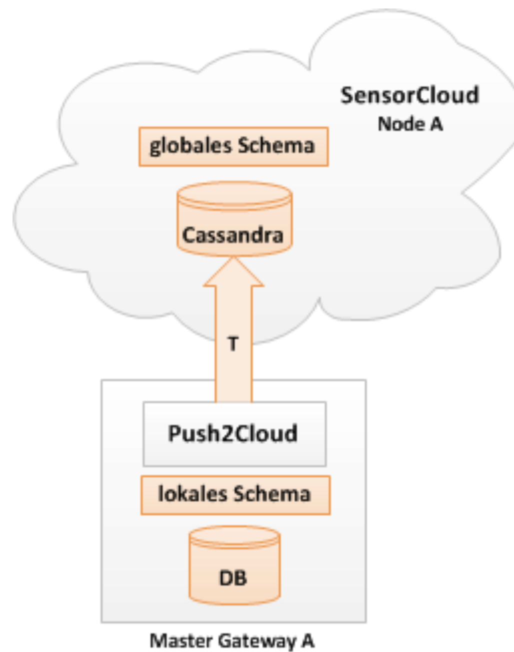

Abbildung 9: Klassische Transaktionen, initialisiert vom Gateway

Das Gateway ist bei der Anwendung des Push-Prinzips autonom. Es wartet nicht nur darauf, dass die Daten wie nach dem Pull-Prinzip von ihnen abgeholt werden, sondern leiten die Transaktionen selbst in die Wege. Das ist vor allem dann ein Vorteil, wenn z.B. ein Sensor einen Alarm meldet, oder ein kritischer Messwert eines Sensors überschritten wird. Anstatt darauf zu warten, bis eine Pull-Transaktion aus der SensoCloud auf das Gateway initialisiert wird, kann das Gateway selbst eine Transaktion initialisieren und den Alarm sofort in die SensorCloud übertragen, damit der Kunde schnellstmöglich informiert werden kann.

Klassendiagramm

Das Klassendiagramm des Push2Cloud-Services unterscheidet sich keineswegs von dem Pull2Cloud-Service. Letztendlich ist es ein Pull2Cloud-Service, welches nur auf dem Gateway und nicht in der SensorCloud angesiedelt ist.

Gefördert durch:

Bundesministerium für Wirtschaft und Technologie

aufgrund eines Beschlusses des Deutschen Bundestages

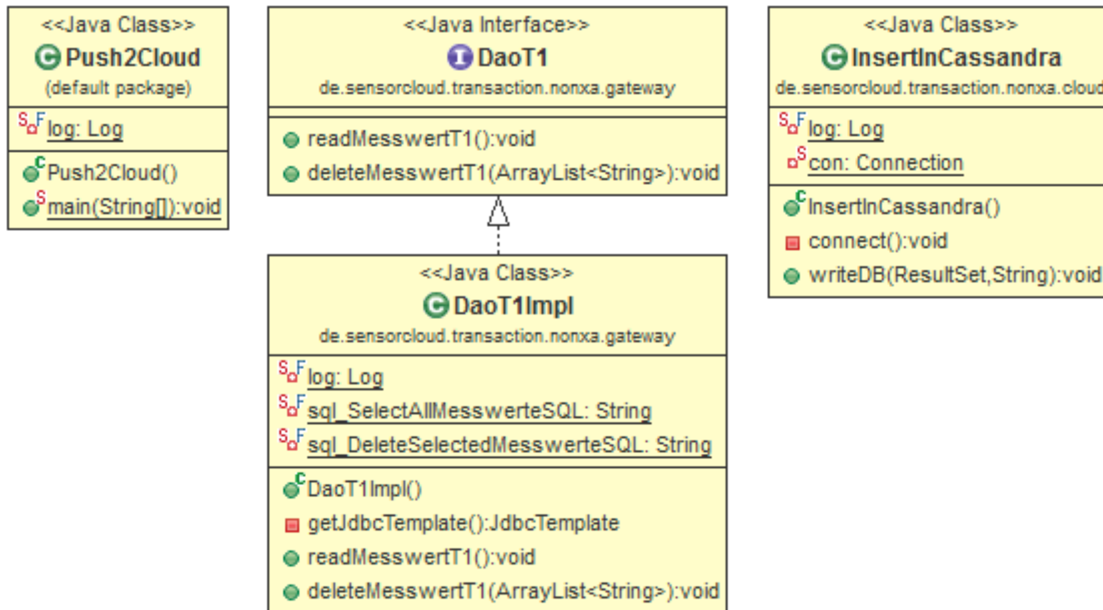

Abbildung 10: Klassendiagramm - Push2Cloud-Service

Wie auch schon beim Pull2Cloud-Service, werden die Sensordaten hier auch nach der erfolgreichen Übertragung in die SensorCloud-Datenbank, wieder von der Gateway-Datenbank entfernt.

6. Crontabs

Die entwickelten ServicesXAPull2Cloud, Pull2Cloud und Push2Cloud sind in der FDBS-Teststecke als Crontabs hinterlegt. Als Crontab wird ein vom cron-Daemon gestarteter Prozess bezeichnet, der zu einer festgelegten Zeit ausgeführt wird. Crontabs können mit dem Befehl crontab -e in eine vorgegebene Liste hinzugefügt werden.

Crontabs auf dem Gateway

Der Push2Cloud-Service ist als Crontab auf dem Gateway angelegt worden. Er ist so konfiguriert, dass der Serice alle 10 Minuten ausgeführt wird und die Sensordaten der der Gateway Datenbank in die Cloud-Datenbank überträgt.

```
# m   h   dom mon dow command
*/10 *   *    *    *    java -jar /home/linaro/Crontab/Push2Cloud/
                        Push2Cloud.jar
```

Crontabs auf dem Cloud-Knoten

Die XAPull2Cloud- und Pull2Cloud-Services sind auf dem Cloud-Knoten angelegt worden. Der XAPull2Cloud-Service ist so konfiguriert, dass er genau 5 Minuten nach Jeder Stunde eine verteilte Transaktion auf die drei XA-fähigen DBS des Gateway durchführt.

Der Pull2Cloud-Service ist so konfiguriert, dass er alle 15Minuten, die Sensordaten von dem Gateway in die Cloud Datenbank befördert.

Gefördert durch:

aufgrund eines Beschlusses des Deutschen Bundestages

```
# m  h  dom mon dow command
# 5 */1 *   *   *   java -jar /home/astec/Crontab/XAPull2Cloud/
                    XAPull2Cloud.jar
*/15 *  *   *   *   java -jar /home/astec/Crontab/Pull2Cloud
                    /Pull2Cloud.jar
```

7. Fazit der Implementierung von Transaktionskonzepten

Die Transaktionskonzepte nach dem Pull- und Push-Prinzip bieten für die SensorCloud eine flexible Art der Transaktionsinitialisierung. Soll die SensorCloud selbst die Initialisierung der Transaktionen auf die Gateways durchführen (Pull-Prinzip), so entlastet dies die Gateways. Die SensorCloud behält die Kontrolle über die zu verwaltenden Transaktionen. Jedoch müssen die DBMS auf dem Gateway einen Zugriff aus der Ferne zulassen (Client/Server Modell).

Für die SensorCloud sind die Gatways nicht autonom. Sie dienen der SensorCloud nur als Datenpuffer, bis die nächste Transaktion aus der SensorCloud auf die Gateways initialisiert wird.

Für die Initialisierung der Transaktionen vom Gateway aus (Push-Prinzip), spricht einerseits, dass auch DBMS-Systeme, die keinen Zugriff aus der Ferne zu lassen, auf dem Gateway verwendet werden können (z.B. SQLite). Hierbei muss das Gateway selbst die Transkationen Richtung SensorCloud initialisieren und kontrollieren.

Die Gateways sind bei der Anwendung des Push-Prinzips autonom. Sie warten nicht nur darauf, dass die Daten wie nach dem Pull-Prinzip von ihnen geholt werden, sondern leiten die Transaktionen selbst in die Wege. Das ist vor allem dann ein Vorteil, wenn z.B. ein Sensor einen Alarm meldet, oder ein kritischer Messwert eines Sensors überschritten wird. Ein Analyse Programm könnte diesen Alarm erkennen und die Transaktion in die Cloud initialisieren, so dass der Kunde direkt über diesen Messwert informiert wird.

8. Übertragung und Visualisierung von Sensordaten am Beispiel eines Stromerfassungssensors in der Sensor Cloud

Für die Visualisierung, der in der Datenbank der SensorCloud persistierten Sensordaten, wurde eine Webanwendung implementiert, die die übertragenen Messwerte grafisch als Diagramm darstellt. Hierbei werden die Diagramme auf Client-Seite generiert, um die alleinige Entschlüsselung von verschlüsselten Messwerten (siehe Kapitel 9) durch den Anwender zu gewährleisten.

Die Anwendung benutzt die Entitäten Sensor, SensorProdukt, Messwert und MessLinie. Über die Entitäten Sensor und Messwert werden die Sensoren ermittelt, die Messwerte in die SensorCloud übertragen haben. SensorProdukt enthält die benötigte Semantik eines Sensors (Physikalischer

Gefördert durch:

aufgrund eines Beschlusses
des Deutschen Bundestages

Name des Messwerts, Aggregatfunktionen und Umrechnungsfunktion). Die Entität MessLinie liefert die Tage, an denen Messwerte zur Verfügung stehen. In der Anwendung können so nur Tage mit Messwerten ausgewählt werden. Für die Generierung des Diagramms werden über die Entität MessLinie zudem die für das Diagramm relevanten Sensordaten in der Entität Messwert identifiziert.

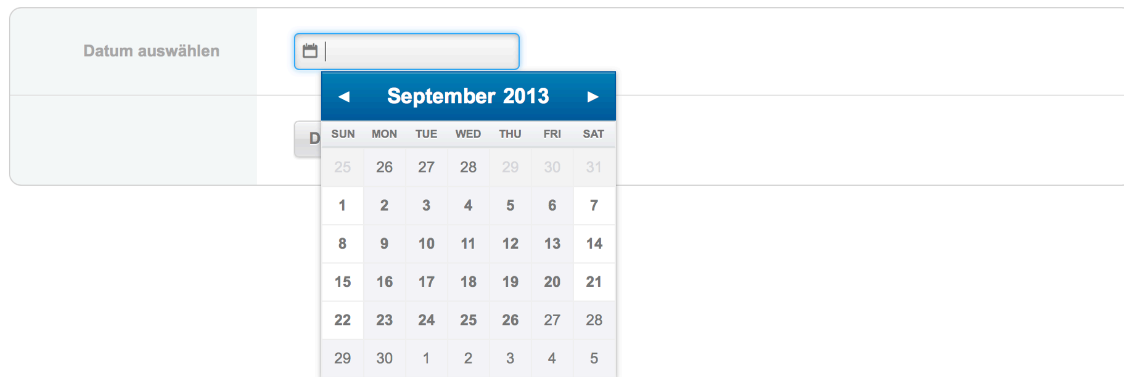

Abbildung 11: Auswahl des zu visualisierenden Zeitraums

Die Abfolge der Anwendung ist die Auswahl eines Sensors, die Auswahl des Messwerttyps (Physikalischer Name eines Messwerts), die Auswahl des darzustellenden Zeitraumes und die abschließende Erstellung des Diagramms.

Die Webanwendung[1] ist mit PHP als AJAX-Anwendung programmiert. Ihr modularer Aufbau ist in Abbildung 12 dargestellt.

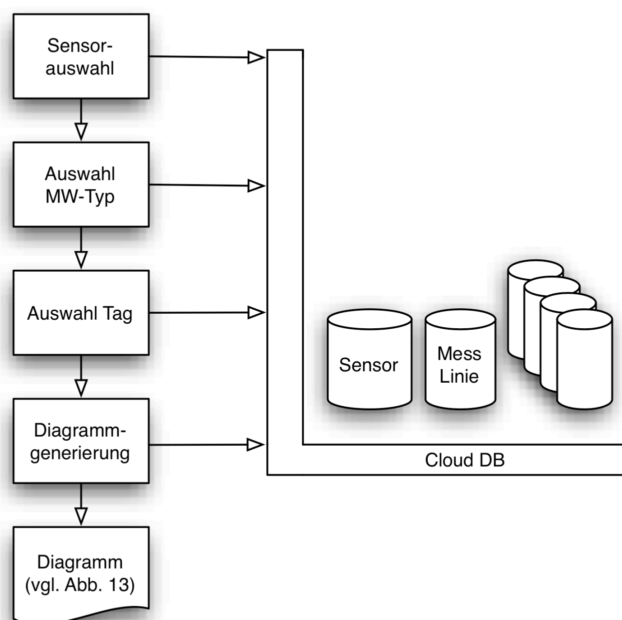

Abbildung 12: Modularer Aufbau der Webanwendung

[1] http://babeauf.nt.fh-koeln.de/scstat

Gefördert durch:

aufgrund eines Beschlusses des Deutschen Bundestages

Abbildung 13 zeigt das Diagramm über die Messwerte eines Stromerfassungssensors, das die gewonnene Leistung in Watt bzw. in Wattstunden einer Photovoltaik-Anlage darstellt.

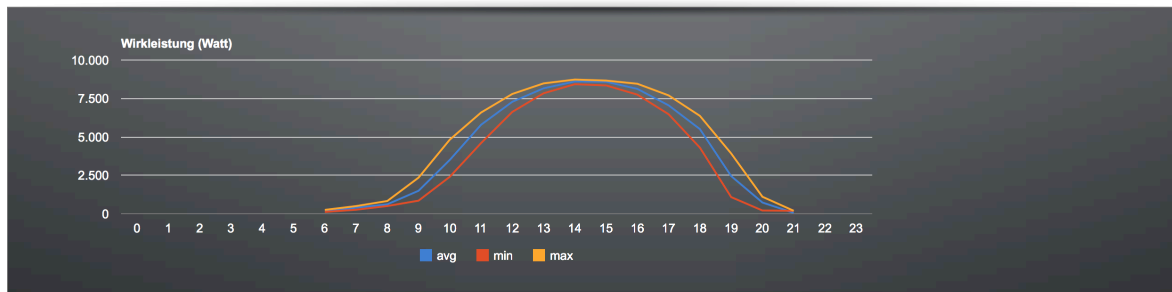

Abbildung 13: Photovoltaik-Visualisierung

Es ist angedacht für den Zugriff auf die genannten Entitäten der Cloud DB Webservices zu entwickeln und diese zu nutzen.

9. Verschlüsselte Übertragung von Sensordaten

Zum Schutz der Vertraulichkeit und unter dem Aspekt der Trusted Cloud werden die zu Sensordaten verschlüsselt übertragen. Bei einer Verschlüsselung der Sensordaten ist es vorher ratsam zu wissen, welche Konsequenzen verschlüsselte Sensordaten in einer Gateway-Datenbank haben. Eine Konsequenz ist, dass auf verschlüsselten Sensordaten keine direkten Aggregationen oder Analysen durchgeführt werden können, außer man möchte die Sensordaten jedes Mal wieder entschlüsseln. Aus diesem Grund wurden die *Pre- und Post- Verschlüsselungsstrategien* entwickelt.

9.1 Pre-Verschlüsselungsstrategie

Bei der Pre-Verschlüsselungsstrategie werden die Sensordaten noch vor dem abspeichern in die Gateway-Datenbank verschlüsselt.

Gefördert durch:

aufgrund eines Beschlusses
des Deutschen Bundestages

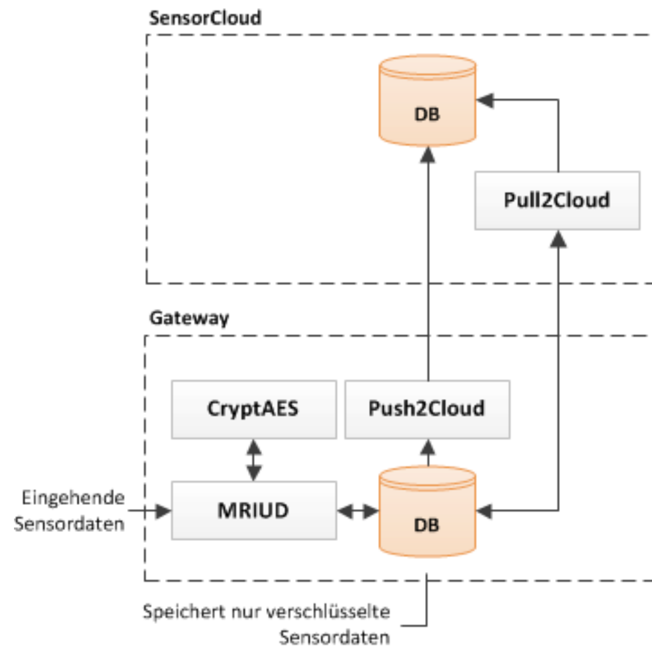

Abbildung 14: Pre-Verschlüsselungsstrategie

Das MRUID-Modul nimmt die eingehenden Sensordaten am Gateway entgegen und ruft zur Verschlüsselung jedes Datensatzes, das *CryptAES*-Modul auf. Den symmetrischen Schlüssel erhält das MRUID-Modul von der Gateway-DB. Anschließend werden die Sensordaten verschlüsselt in der Gateway-DB abgespeichert. Die Push2Cloud bzw. Pull2Cloud Dienste können die Sensordaten wie gewohnt vom Gateway in die SensorCloud-DB übertragen.

9.2 Post-Verschlüsselungsstrategie

Bei der Post-Verschlüsselungsstrategie werden die Sensordaten unverschlüsselt in der Gateway-DB abgespeichert. Erst vor der direkten Übertragung in die SensorCloud wird eine Verschlüsselung der Sensordaten durchgeführt. Bei dieser Strategie ist es jetzt noch möglich Aggregationen oder Analysen auf den Sensordaten, die sich in der Gateway-DB befinden durchzuführen.

Gefördert durch:

Bundesministerium für Wirtschaft und Technologie

aufgrund eines Beschlusses des Deutschen Bundestages

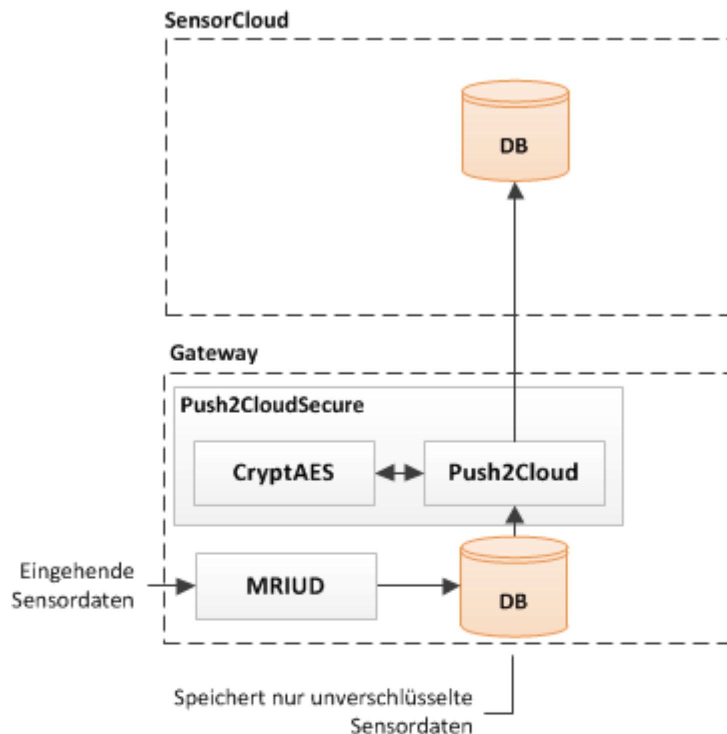

Abbildung 15: Post-Verschlüsselungsstrategie

Die Verschlüsselung der Sensordaten übernimmt wieder das CryptAES-Modul, welches von dem Push2Cloud-Dienst verwendet wird. Zusammen bilden die beiden Programme den Push2CloudSecure-Dienst.

9.3 CryptAES-Modul

Das CryptAES-Modul führt eine 128 Bit AES (Advanced Encryption Standard) Verschlüsselung im CBC (Cipher Block Chaining) Modus durch. Für jeden zu verschlüsselnden Messwert wird ein 16 Byte langer Random Array generiert, aus dem dann der Initialisierungsvektor IV berechnet wird. Der IV sorgt dafür, dass gleiche Klartexte unterschiedliche Chiffretexte liefern. Das Random Array ist auch ein Bestandteil des späteren Chiffretextes, da er für die Entschlüsselung wieder verwendet werden muss.

Soll beispielsweise der Messwerttupel `0;15;25;57` verschlüsselt werden, so wird zu aller erst ein Random Array generiert und mit Hilfe des daraus erzeugtem IV der Messwerttupel verschlüsselt.

Zur Speicherung des Random Arrays und des Chiffretextes empfiehlt es sich eine Base64 Kodierung vorzunehmen, damit z.B. keine Hochkommata, die in einem Chiffretext vorkommen könnten, nicht in eine Datenbankzelle gespeichert werden. Dies könnte als Abschluss einer Zeichenkette vom DBMS interpretiert werden. Den Präfix des letztendlich gespeicherten Chiffretextes bildet das Base64 kodierte Random Array, gefolgt vom ebenfalls Base64 kodiertem Messwerttupel.

Random Array	Messwerttupel
+GCxJSyns641x23cxdX3RA==	**1AS8mWleOY08+yTtEqWmlg==**

Gefördert durch:

Bundesministerium für Wirtschaft und Technologie

aufgrund eines Beschlusses des Deutschen Bundestages

9.4 Zeitaufwand der Verschlüsselung

Da eine zusätzliche Verschlüsselung der Sensordaten, auch einen zusätzlichen Zeitaufwand bedeutet, wurde eine Zeitmessung durchgeführt. Bei der Zeitmessung wurde der Transaktions-Dienst Push2Cloud mit dem Transaktions-Dienst Push2CloudSecure verglichen.

Datensätze	von GW-DB lesen [s]	in Cloud-DB schreiben [s]	Gesamtlaufzeit [s]
5	15,2 / 17,0	0,1 / 0,7	15,3 / 17,7
50	14,6 / 16,3	0,3 / 1,0	14,9 / 17,3
500	14,7 / 16,6	2,1 / 4,1	16,9 / 20,6
5000	15,4 / 17,1	19,7 / 32,7	35,1 / 49,8
50000	19,4 / 21,1	186,3 / 317,8	205,6 / 338,9

Push2Cloud / Push2CloudSecure

Abbildung 16: Zeitmessung Push2Cloud vs. Push2CloudSecure

In der oberen Abbildung ist deutlich zu sehen, dass eine zusätzliche Verschlüsselung der Sensordaten, vor der Transaktion in die SensorCloud, auch einen zusätzlichen Zeitaufwand mit sich bringt. Bei der Zeitmessung ist zu berücksichtigen, dass für jede Messreihe, jeder Transaktionsdienst als JAR-Datei neu ausgeführt wurde. Das hat natürlich zur Folge, dass sich bei nur wenig zu übertragenden Datensätzen, die Initialisierungsphase des Programms, im Verhältnis zur eigentlichen Verschlüsselung und Transaktion in die Cloud, negativ auswirkt.

Abschließend kann man sagen, dass diese Zeitmessung nur deutlich stellen sollte, dass eine Verschlüsselung der Sensordaten auch eine zusätzliche Rechenzeit benötigt, die in den gesamten Transaktionsprozess in die Cloud mitberücksichtigt werden sollte. Wie viele Sensordaten ein Gateway, bis zur Transaktion in die Cloud, zwischenspeichert hängt natürlich auch von dem Einsatzzweck der betriebenen Sensoren und den Anwendungsfall ab.

Gefördert durch:

aufgrund eines Beschlusses
des Deutschen Bundestages

Quellen

[1] S. Conrad, W. Hasselbring, A. Koschel, R. Tritsch: Enterprise Application Integration; Grundlagen – Konzepte, Entwurfsmuster - Praxisbeispiele, Spektrum Akademischer Verlag, 2006

[2] Distributed Transaction Processing: Reference Model, Version 3, X/Open Company Ltd., U.K., UK ISBN 1-85912-170-5, 1996

[3] Distributed Transaction Processing: The XA Spezification, X/Open Company Ltd., U.K., UK ISBN 1-87263-024-3, 1991

[4] Distributed Transaction Processing: The TX Spezification, X/Open Company Ltd., U.K., UK ISBN 1-85912-094-6, 1995

[5] https://www2.opengroup.org/ogsys/jsp/publications/PublicationDetails.jsp?publicationid=11221Distributed Transaction Processing: The XA+ Specification, Version 2, The Open Group,
ISBN 1-85912-046-6, 1994

[6] https://www2.opengroup.org/ogsys/jsp/publications/PublicationDetails.jsp?publicationid=11221Java Transaction API Specification, Sun Microsystems Inc. 2002

[7] https://www2.opengroup.org/ogsys/jsp/publications/PublicationDetails.jsp?publicationid=11221Homepage zum Atomikos TransactionEssentials: www.atomikos.com

[8] https://www2.opengroup.org/ogsys/jsp/publications/PublicationDetails.jsp?publicationid=11221Homepage zum JBoss Transaction Service: www.jboss.org/jbosstm/

[9] https://www2.opengroup.org/ogsys/jsp/publications/PublicationDetails.jsp?publicationid=11221Homepage zum Bitronix Transaction Manager: www.bitronix.be

[10] https://www2.opengroup.org/ogsys/jsp/publications/PublicationDetails.jsp?publicationid=11221Homepage zum Java Open Transaction Manager: jotm.ow2.org

[11] Homepage zum Spring Framework: www.springsource.org

Gefördert durch:

aufgrund eines Beschlusses
des Deutschen Bundestages