
# Laborprotokoll

# DEZSYS-06: Verteilte Transaktionen

**Systemtechnik**
**5BHITT 2015/16**

**Rene Hollander**

**Version 1.0**

**Note:**

**Betreuer: Micheler**

**Begonnen am 16. Februar 2016**

**Beendet am 17. Februar 2016**

# Inhaltsverzeichnis

# Einführung

Die Übung soll die Grundlagen von verteilte Transaktionen mit Hilfe eines praktischen Beispiels in JAVA vertiefen.

## 1 Ziele

Implementieren Sie in JAVA einen Transaktionsmanager, der Befehle an mehrer Stationen weitergibt und diese koordiniert. Mit Hilfe des 2-Phase-Commit Protokolls sollen die Transaktionen und die Antwort der Stationen verwaltet werden. Der Befehl kann beliebig gewaehlt werden und soll eine Datenquelle (Datenbank oder Datei oder Message Queue) abfragen oder veändern.

Die Kommunikation zwischen Transaktionsmanagers und der Stationen soll mit Hilfe einer Übertragungsmethode (IPC, RPC, Java RMI, JMS, etc) aus dem letzten Schuljahr umgesetzt werden.

## 2 Voraussetzungen

- Grundlagen Transaktionen (allgmein, Datenbanksysteme)
- Anbindung Datenquelle in JAVA (JDBC, File, JMS)
- Kommunikation in JAVA (IPC, RPC, Java RMI, JMS)

## 3 Aufgabenstellung

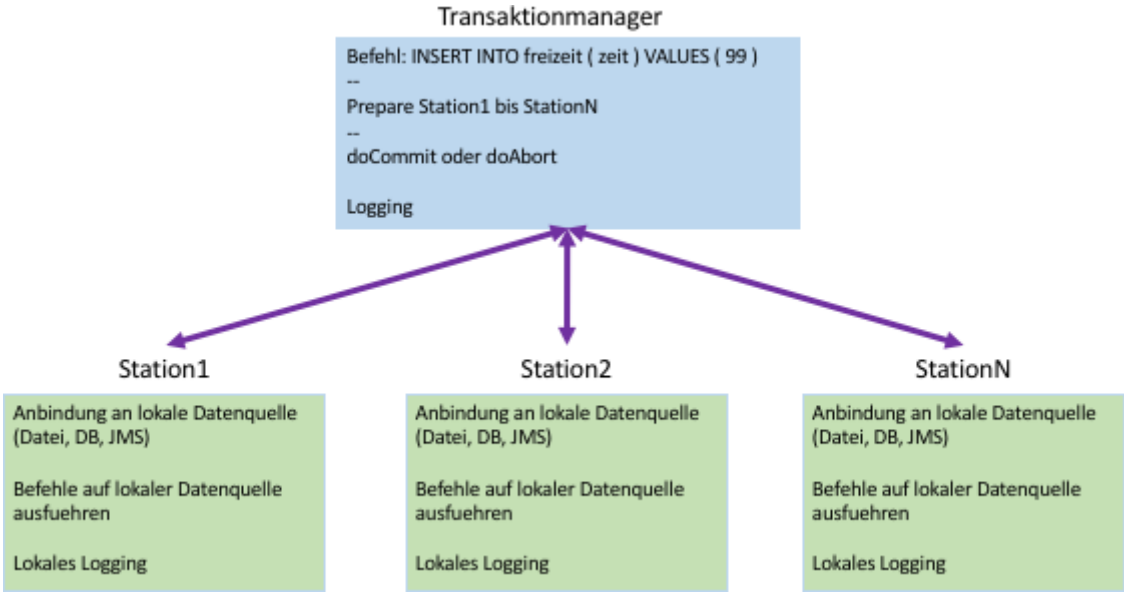
Der Transaktionsmanager läuft auf einer eigenen Instanz (bzw. eigenem Port) und stellt die Schnittstelle zwischen den Stationen und dem Benutzer dar. Der Benutzer gibt über die Konsole oder ein User Interface einen Befehl ein, der danach an alle Stationen verteilt wird. Da das 2-Phase-Commit Protokoll als Transaktionsprotokoll zugrunde liegt, soll der Transaktionsmanager jeweils nach einem Befehl,

- das Resultat nach der PREPARE-Phase (Bsp. 3xYES 0xNO 0xTIMEOUT) ausgeben
- welche Aktion der Transaktionsmanager danach durchfuehrt (doCommit, doAbort)
- das Resultat der COMMIT-Phase (Bsp. 3xACK 0xNCK 0xTIMEOUT)
- danach kann ein neuer Befehl eingegeben werden
-

Logging

Um im Einzelfall die Transaktionen und Resultat nachverfolgen zu koennen, sollen alle Befehle und deren Resultate mit Timestamp geloggt werden. Beim Transaktionsmanager soll dokumentiert werden, welcher Befehl zu welcher Station und zu welchem Zeitpunkt abgesendet wurde, ebenso beim Erhalt der Antwort. Ebenso sollen, bei den Stationen eingehenden Befehle und deren Resultate bei Ausführung an der lokalen Datenquelle mitdokumentiert werden.

Die folgende Grafik soll den Vorgang beim 2-Phase-Commit Protokoll verdeutlichen:




Bewertung: 16 Punkte
- verteilte Transaktion mit 1 Station (8 Punkte)
- verteilte Transaktion mit n Stationen (4 Punkte)
- Logging (in Log-Dateien) Transaktionsmanager und Stationen (2 Punkte)
- Protokoll (2 Punkte)

Alternativ zu dieser Aufgabenstellung kann ein Transaktionssystem in JEE mit Hilfe der JTA entwickelt werden. Details mit Prof. Micheler per Mail abklären.

# Ergebnisse

Zur Kommunikation zwischen dem Manager und den Stations wurde Socket.IO verwendet. Der Vorteil von diesem Framework ist, dass Nachrichten über Events ausgetauscht werden. Bei Bedarf kann bei einem Event auch ein Acknowledgement, direkte Antwort auf ein Event, auch eine Nachricht gesendet werden.

Als DBMS wurde SQLite gewählt. Als API wird JDBC verwendet.

Für das Logging wurde SL4J mit dem Logback Logger verwendet um eine genaue Ausgabe der ausgeführten Befehle zu ermöglichen.

## Code

Der folgende Code wird ausgeführt sobald ein SQL Statement in der Konsole eingegeben wird. Zuerst wird ein execute Event an alle Stationen geschickt. In einem Set werden alle Benachrichtigten Stationen aufbewahrt und sobald eine Antwort kommt entfernt. Wenn eine Antwort eine Fehlermeldung enthält wird diese zur Fehlerliste hinzugefügt. Erst wenn alle Stationen geantwortet haben, wird entschieden ob ein Commit oder ein Rollback gemacht werden muss. Ist die Fehlerliste leer, kann ein Commit ausgeführt werden.

```
server.getBroadcastOperations().sendEvent("execute", Maps.of("timeout", timeout,
"statement", statement), new BroadcastAckCallback<Map>(Map.class, timeout) {
    public void onClientSuccess(SocketIOClient client, Map resultMap) {
        lock.lock();
        LOG.info("Recieved response from " + client.get("name"));
        Map<String, Object> result = resultMap;
        if (result == null) result = new HashMap<>();
        clients.remove(client);
        if (result.containsKey("error")) {
            LOG.error("client " + client.get("name") + " responded with an exception: "
+ result.get("error"));
            exceptions.add("client " + client.get("name") + " responded with an
exception: " + result.get("error"));
        } else {
            LOG.info("client " + client.get("name") + " response after " + statement +
" was ok");
        }
        rollbackOrCommitCheck(clients, exceptions, timeout, callback, triggered);
        lock.unlock();
    }
    public void onClientTimeout(SocketIOClient client) {
        lock.lock();
        clients.remove(client);
        LOG.error("client " + client.get("name") + " timed out");
        exceptions.add("client " + client.get("name") + " timed out");
        rollbackOrCommitCheck(clients, exceptions, timeout, callback, triggered);
        lock.unlock();
    }
});
```

Der Folgende Code Prüft ob schon alle Clients geantwortet haben. Wenn dies der Fall ist wird entschieden ob ein Rollback oder ein Commit ausgeführt werden muss.

```
private synchronized void rollbackOrCommitCheck(Set<SocketIOClient> clients,
List<String> exceptions, int timeout, Callback.NoParamsWithStringError callback,
AtomicBoolean triggered) {

    if (!triggered.get() && clients.isEmpty()) {
        triggered.set(true);
        if (!exceptions.isEmpty()) {
            LOG.info("Executing Rollback on all stations");
            rollback(timeout, callback, exceptions);
        } else if (exceptions.isEmpty()) {
            LOG.info("Executing Commit on all stations");
            commit(timeout, callback, exceptions);
        }
    }
}
```

Ähnlich dem Code für das eigentliche execute wird hier ein rollback Event gesendet. Auch hier wird wieder überprüft ob alle Stationen geantwortet haben und ob der Rollback erfolgreich war. Die Implementierung für das commit ist gleich, verwendet aber das commit Event.

```
server.getBroadcastOperations().sendEvent("rollback", null, new
BroadcastAckCallback<Map>(Map.class, timeout) {

    public void onClientSuccess(SocketIOClient client, Map resultMap) {
        lock.lock();
        Map<String, Object> result = resultMap;
        if (result == null) result = new HashMap<>();
        clients.remove(client);
        if (result.containsKey("error")) {
            LOG.error("client " + client.get("name") + " responded with an exception
while rolling back: " + result.get("error"));
            exceptions.add("client " + client.get("name") + " responded with an
exception while rolling back: " + result.get("error"));
        } else {
            LOG.info("client " + client.get("name") + " response after rollback was
ok");
        }
        callbackCheck(clients, exceptions, callback, triggered, prev);
        lock.unlock();
    }
    public void onClientTimeout(SocketIOClient client) {
        lock.lock();
        clients.remove(client);
        exceptions.add("client " + client.get("name") + " timed out while rolling
back");
        LOG.error("client " + client.get("name") + " timed out while rolling back");
        callbackCheck(clients, exceptions, callback, triggered, prev);
        lock.unlock();
    }
});
```

Folgender Code wird auf einer Station ausgeführt sobald ein Execute vom Manager empfangen wird. Es wird das Statement aus den Daten gelesen und an das DBMS übergeben um ausgeführt zu werden. Sollte ein Fehler auftreten wird dieser über das Acknowledgement zurück gesendet. Die Implementierungen für Rollback und Commit sehen ähnlich aus, führen aber die entsprechenden Operationen am DBMS aus.

```
LOG.info("Recieved statement from Transaction Manager");
Ack ack = (Ack) datas[datas.length - 1];
JSONObject data = (JSONObject) datas[0];
try {
    getDatabaseConnection().execute(data.optInt("timeout", 10),
data.getString("statement"), (err, res) -> {
        if (err != null) {
            ack.call(Maps.of("error", err.getMessage()));
        } else {
            ack.call(Maps.of("res", res));
        }
    });
} catch (JSONException e) {
    LOG.error("An exception occured while parsing request", e);
    ack.call(Maps.of("error", e.getMessage()));
}
```

Ein letzter Interessanter Codeteil ist das öffnen der Verbindung zur Datenbank. Hier muss darauf geachtet werden, dass autocommit auf false ist und eine möglichst starke Isolationsstufe ausgewählt ist.

```
Class.forName("org.sqlite.JDBC");
connection = DriverManager.getConnection("jdbc:sqlite:" + file.getAbsolutePath());
connection.setAutoCommit(false);
connection.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
```

# 1 Verwendung

## Simple Mode

Im Simple Mode wird ein Manager und 3 Stationen automatisch gestartet, sowie der DB Ordner gelöscht und eine Testdatenbank mit einem Eintrag der nur auf Station3 ist erstellt.

```
java -jar transactionmanager-1.0.0.jar
```

## Advanced Mode

Man kann auch Manager und Stationen manuell erstellen. Vorteil dabei ist, dass auch SQL Statements direkt auf einer Station ausgeführt werden können.

Manager starten

```
java -cp transactionmanager-1.0.0.jar
at.renehollander.transactionmanager.manager.ManagerMain <PORT>
```

Station starten

Wichtig ist, dass jede Station einen anderen Namen hat.

```
java -cp transactionmanager-1.0.0.jar
at.renehollander.transactionmanager.station.StationMain <NAME>
<HOSTNAME> <PORT>
```

# Zeitaufwand

Da wir keine Möglichkeit hatten die Übung in der Schule zu machen, mussten 4 Stunden meiner kostbaren Freizeit für diese Aufgabe geopfert werden.