6 Übungsaufgabe #6: ZooKeeper

In dieser Aufgabe soll ein fehlertoleranter Dienst zur Koordinierung verteilter Anwendungen entwickelt werden. Als Vorbild dient dabei *Apache ZooKeeper*, das mit folgender (eingeschränkter) Funktionalität nachgebildet wird:

```
public class MWZooKeeper {
    public String create(String path, byte[] data, boolean ephemeral);
    public void delete(String path, int version);
    public MWZooKeeperStat setData(String path, byte[] data, int version);
    public byte[] getData(String path, MWZooKeeperStat stat);
}
```

Mit `create()` lässt sich unter dem Pfad `path` ein neuer Knoten mit den Nutzdaten `data` anlegen; `ephemeral` legt dabei fest, ob es sich um einen flüchtigen Knoten (siehe Teilaufgabe 6.4) handelt. Ein Aufruf von `delete()` löscht einen Knoten, sofern dessen aktuelle Versionsnummer `version` entspricht. Mit `setData()` lassen sich einem Knoten neue Nutzdaten zuweisen; auch dieser Aufruf ist nur erfolgreich, falls der Knoten bei der Bearbeitung der Anfrage die angegebene Versionsnummer aufweist. Als Rückgabewert liefert `setData()` ein Objekt der Klasse `MWZooKeeperStat`, in dem die aktualisierten Metadaten des Knotens (z. B. Versionsnummer, Zeitstempel der letzten Modifikation etc.) gekapselt sind. Über die Methode `getData()` lassen sich sowohl die Nutz- als auch die Metadaten eines Knotens auslesen. Die Rückgabe der Metadaten erfolgt dabei über den Ausgabeparameter `stat` (siehe Tafelübung). Sämtliche auftretenden Ausnahmesituationen (z. B. ungültige Pfadangaben, veraltete Versionsnummern etc.) werden per `MWZooKeeperException` mit passender Fehlermeldung signalisiert.

Als Ausgangsbasis für die eigene Implementierung sind im Pub-Verzeichnis der Aufgabe einige Klassen bereitgestellt. Falls erforderlich, dürfen diese beliebig modifiziert bzw. erweitert werden.

6.1 Verteilung des Diensts (für alle)

Im ersten Schritt soll der entfernte Zugriff auf den ZooKeeper-Dienst ermöglicht werden. Während die auf Client-Seite als Zugangspunkt fungierende Klasse `MWZooKeeper` bereits existiert (siehe Pub-Verzeichnis), muss der `MWZooKeeperServer` noch implementiert werden. Dieser soll über einen Server-Socket TCP-Verbindungen annehmen und die eigentliche Kommunikation mit Clients in separaten Worker-Threads abwickeln. Da ZooKeeper vorsieht, dass Clients mehrere Anfragen über dieselbe Verbindung schicken können, muss ein Worker-Thread die Verbindung nach dem Senden einer Antwort offen halten. Es darf angenommen werden, dass ein Client erst eine neue Anfrage stellt, wenn er die Antwort (bzw. eine Fehlermeldung) auf seine vorherige Anfrage erhalten hat.

Aufgabe:

→ Implementierung der Klasse `MWZooKeeperServer` in einem Subpackage `mw.zookeeper`

6.2 Implementierung der Zustandsverwaltung (für alle)

ZooKeeper unterscheidet bei der Bearbeitung zwischen lesenden (`getData`) und schreibenden (`create`, `delete` und `setData`) Operationen: Leseanfragen werden unmittelbar von dem Server beantwortet, der sie empfängt. Die Ausführung von modifizierenden Operationen erfolgt dagegen auf einem Anführerreplikat, das für jede Schreibanfrage eine Transaktion erstellt, mit deren Hilfe alle Replikate anschließend ihre Zustände aktualisieren. Als Vorbereitung für die Replikation des Diensts soll hier zunächst die zur Zustandsverwaltung erforderliche Logik in einer Klasse `MWZooKeeperImpl` realisiert werden, die mindestens folgende Methoden umfasst:

```
public class MWZooKeeperImpl {
    public MWZooKeeperResponse processReadRequest(MWZooKeeperRequest request);
    public MWZooKeeperTxn processWriteRequest(MWZooKeeperRequest request, long zxid);
    public MWZooKeeperResponse applyTxn(MWZooKeeperTxn txn, long zxid);
}
```

Wie in der Tafelübung erläutert, muss `MWZooKeeperImpl` zwischen zwei ZooKeeper-Zuständen unterscheiden: Einem bestätigten Zustand Z_B, der auf allen Replikaten existiert, und dem nur vom Anführer verwalteten aktuellen Zustand Z_A, der im Vergleich zu Z_B neue, noch unbestätigte Änderungen umfassen kann. Ein Aufruf von `processReadRequest()` führt die übergebene Leseanfrage auf Z_B aus und gibt das Ergebnis bzw. eine Fehlermeldung in Form einer Antwortnachricht zurück. Die Methode `processWriteRequest()` erstellt auf Basis einer Schreibanfrage und Z_A eine (Fehler-)Transaktion `MWZooKeeperTxn` und weist dieser die eindeutige ID `zxid` zu. Mittels `applyTxn()` lässt sich die in der Transaktion `txn` enthaltene Zustandsänderung auf Z_B anwenden.

Aufgabe:

→ Implementierung der Klassen `MWZooKeeperImpl` und `MWZooKeeperTxn`

Hinweise:

- Um Hauptspeicherplatz zu sparen, ist Z_A in Form von Änderungen gegenüber Z_B zu implementieren.
- Sobald eine Transaktion auf Z_B angewendet wurde, soll die entsprechende Änderung in Z_A gelöscht werden.

6.3 Replikation des Diensts (für alle)

Die aktuelle Implementierung des Diensts bietet keinerlei Schutz vor Rechnerausfällen, da sie sich auf das korrekte Funktionieren eines einzelnen Servers verlässt. Um die Server-Seite des Diensts tolerant gegenüber Ausfällen zu gestalten, soll sie im Rahmen dieser Teilaufgabe repliziert werden. Da sich jeder Client mit einem beliebigen ZooKeeper-Replikat seiner Wahl verbinden kann, muss dabei sichergestellt sein, dass alle Replikate über einen konsistenten Zustand verfügen. In ZooKeeper wird dies dadurch erreicht, dass ein Anführerreplikat alle zustandsmodifizierenden Anfragen bearbeitet und die daraus resultierenden Zustandsaktualisierungen mittels *Zab* an die anderen Replikate verteilt. Zab garantiert hierbei, dass eine solche Zustandstransaktion nur dann ausgeliefert wird, wenn zuvor eine Mehrheit aller Replikate den Erhalt der Transaktion bestätigt hat und weiterhin dem aktuellen Anführerreplikat folgt.

Da in der eigenen ZooKeeper-Implementierung sämtliche Interaktion zwischen Replikaten mittels Zab erfolgen soll, benötigt jeder `MWZooKeeperServer` Zugriff auf einen eigenen Zab-Knoten. Des weiteren muss ein Server die Schnittstelle `ZabCallback` implementieren, um per Zab übermittelte Anfragen und/oder Transaktionen empfangen sowie über den Ausgang von Anführerwahlen informiert werden zu können.

Im letzten Schritt dieser Teilaufgabe ist dafür zu sorgen, dass ein Replikat nach Beendigung einer Anführerwahl die ihm zugewiesene Rolle einnimmt: Für ein Follower-Replikat bedeutet dies, dass es nur Leseanfragen unmittelbar bearbeiten darf, Schreibanfragen dagegen an den Anführer weiterleiten muss. Das Anführerreplikat führt im Unterschied dazu sämtliche (von Clients oder anderen Replikaten) eintreffenden Anfragen aus und schlägt für jede aus einer Schreiboperation resultierenden Transaktion eine neue `zxid` vor.

Aufgaben:

→ Replikation des ZooKeeper-Diensts unter Verwendung von Zab
→ Testen der Implementierung mit drei ZooKeeper-Replikaten auf verschiedenen Rechnern
→ Implementierung von Testfällen, aus denen ersichtlich wird, dass a) die Antwortzeit lesender Anfragen signifikant kleiner ist als die Antwortzeit zustandsmodifizierender Anfragen und b) ZooKeeper keine stark konsistente Sichtweise auf den verwalteten Datenbestand bietet, es also unter Umständen vorkommt, dass Clients beispielsweise veraltete Versionen von Datenknoten lesen.

Hinweise:

- Die zur Verwendung von Zab benötigten Klassen sind in `zab-mwcc.jar` (Pub-Verzeichnis) zusammengefasst.
- Um den geänderten Nachrichtenfluss auf einem Replikat lokal zu testen, kann statt `MultiZab` zunächst ein Objekt der Klasse `SingleZab` als Schnittstelle zu Zab zum Einsatz kommen.
- Der bei TCP-Sockets in Java standardmäßig verwendete Nagle-Algorithmus sollte durch jeweils einmaligen Aufruf von `setTcpNoDelay(true)` an den Sockets der Client-Verbindungen deaktiviert werden.
- Szenarien wie die Wiederherstellung ausgefallener bzw. das nachträgliche Hinzufügen neuer Replikate erfordern Mechanismen zum Transfer von Replikatzuständen und sind daher nicht Teil dieser Übungsaufgabe.

6.4 Flüchtige Knoten (optional für 5,0 ECTS)

Neben den regulären persistenten Knoten, die explizit erzeugt und gelöscht werden müssen, existiert in ZooKeeper mit den „*Ephemeral Nodes*“ eine Kategorie von flüchtigen Knoten, die das System automatisch entfernt, sobald die Verbindung zu dem Client, der sie erzeugt hat, geschlossen wird oder abbricht. Ob es sich bei einem Knoten um einen persistenten oder einen flüchtigen handelt, legt der Client bei der Erzeugung des Knotens fest (siehe `ephemeral`-Parameter der `create()`-Methode).

Die Unterstützung von flüchtigen Knoten macht es auf Server-Seite erforderlich, Client-Verbindungen eindeutig identifizieren zu können. Da das Löschen flüchtiger Knoten eine zustandsmodifizierende Operation darstellt, muss darüber hinaus darauf geachtet werden, dass alle Replikate diese in konsistenter Weise durchführen.

Aufgabe:

→ Erweiterung der bestehenden Implementierung um die Unterstützung flüchtiger Knoten

Hinweise:

- Flüchtige Knoten müssen Blattknoten sein, dürfen selbst also keine eigenen Kindsknoten haben.
- Der Fall, dass eine Client-Verbindung aufgrund eines Replikatausfalls endet, soll nicht betrachtet werden.

Abgabe: am 21.01.2015 in der Rechnerübung