5 Übungsaufgabe #5: ZooKeeper

In dieser Aufgabe soll ein fehlertoleranter Dienst zur Koordinierung verteilter Anwendungen entwickelt werden. Als Vorbild dient *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);
}
```

`create()` erstellt unter dem Pfad `path` einen neuen Knoten mit den Nutzdaten `data`; `ephemeral` gibt an, ob es sich um einen flüchtigen Knoten (siehe Teilaufgabe 5.4) handelt. Ein Aufruf von `delete()` löscht einen Knoten, sofern dessen aktuelle Versionsnummer `version` entspricht oder `version = -1` übergeben wurde. Mit `setData()` lassen sich einem Knoten neue Nutzdaten zuweisen, falls der Knoten beim Bearbeiten der Anfrage die entsprechende Versionsnummer aufweist. Als Rückgabewert liefert `setData()` ein Objekt der Klasse `MWZooKeeperStat`, das die aktualisierten Metadaten des Knotens (z. B. Versionsnummer, Zeitstempel der letzten Modifikation) enthält. Mit `getData()` lassen sich sowohl die Nutz- und Metadaten eines Knotens auslesen. Die Rückgabe der Metadaten erfolgt über den Ausgabeparameter `stat` (siehe Übungsvideo). Ausnahmesituationen (z. B. ungültige Pfadangaben, veraltete Versionsnummern) werden per `MWZooKeeperException` signalisiert.

Als Ausgangsbasis für die eigene Implementierung sind im Pub-Verzeichnis einige Klassen bereitgestellt. Falls erforderlich, dürfen diese beliebig modifiziert bzw. erweitert werden. Als Orientierungshilfe kann der Überblick über den Nachrichtenfluss von Schreibanfragen auf Folie 2 im Foliensatz "Konsistente Replikation"dienen.

5.1 Verteilung des Diensts (für alle)

Im ersten Schritt soll der entfernte Zugriff auf ZooKeeper ermöglicht werden. Während die als Client fungierende Klasse `MWZooKeeper` bereits existiert (siehe Pub-Verzeichnis), ist der `MWZooKeeperServer` noch zu implementieren. Dieser soll über einen Server-Socket TCP-Verbindungen annehmen und die eigentliche Interaktion mit Clients in separaten Worker-Threads abwickeln. Da Clients mehrere Anfragen über dieselbe Verbindung schicken können, muss ein Worker die Verbindung nach dem Senden einer Antwort offen halten. Um darüber hinaus eine effiziente Kommunikation zu garantieren, sollte durch einmaligen Aufruf von `setTcpNoDelay(true)` am Socket jeder Client-Verbindung der in Java standardmäßig verwendete Nagle-Algorithmus deaktiviert werden.

Aufgabe:

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

5.2 Implementierung der Zustandsverwaltung (für alle)

ZooKeeper unterscheidet bei der Bearbeitung zwischen lesenden (`getData`) und schreibenden (`create`, `delete` und `setData`) Operationen: Leseanfragen werden für eine möglichst effiziente Verarbeitung 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. Die Antwort auf eine Leseanfrage kann folglich auf einem (leicht) veralteten Zustand basieren, falls eine Aktualisierung den antwortenden Server noch nicht erreicht hat. 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, den jedes Replikat vorhält, 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 direkt auf Z_B aus und gibt das Ergebnis bzw. eine Fehlermeldung als Antwortnachricht zurück. Die Methode `processWriteRequest()` führt eine Schreibanfrage auf Z_A aus und erstellt darauf basierend eine (Fehler-)Transaktion `MWZooKeeperTxn` mit eindeutiger ID `zxid`. Mittels `applyTxn()` lässt sich die in der Transaktion `txn` enthaltene Zustandsänderung auf Z_B anwenden.

Aufgabe:

→ Implementierung der Klassen `MWZooKeeperImpl` und `MWZooKeeperTxn`

Hinweise:

- Eine Transaktion soll die durch eine Schreibanfrage verursachten Zustandsänderungen enthalten.
- Um Hauptspeicherplatz zu sparen, darf Z_A nur im Vergleich zu Z_B geänderte Knoten speichern.
- Sobald eine Transaktion auf Z_B angewendet wurde, sind nicht mehr benötigte Knoten in Z_A zu löschen.
- Als Hilfestellung kann mit dem `MWZooKeeperImplTest` ein Teil der Implementierung getestet werden.

5.3 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 zum Einsatz 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 genutzt werden. Bei Verwendung von `MultiZab` sind mindestens 3 Replikate notwendig.
- Szenarien wie die Wiederherstellung ausgefallener bzw. das Hinzufügen neuer Replikate erfordern Mechanismen zum Transfer von Replikatzuständen und sind daher nicht Teil dieser Übungsaufgabe.
- Um Fehlermeldungen von Zab zu erhalten, muss log4j entsprechend den Übungsfolien konfiguriert werden.

5.4 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.
- Das Entfernen flüchtiger Knoten eines ausgefallenen Clients soll atomar erfolgen.

Abgabe: am 10.02.2021 in der Übungssprechstunde