5 Übungsaufgabe #5: ZooKeeper

In dieser Aufgabe soll ein fehlertoleranter Dienst zur Koordinierung verteilter Anwendungen entwickelt werden. Als Vorbild dient dabei die Implementierung von *Apache ZooKeeper* (siehe `http://svn.apache.org/viewvc/zookeeper/trunk/src/java/main/`), die mit eingeschränkter Funktionalität nachgebildet werden soll.

5.1 Verwaltung von Daten (für alle)

ZooKeeper verwaltet Informationen in Form von Datenknoten in einer Baumstruktur, die vollständig im Hauptspeicher liegt. Ein Knoten wird dabei eindeutig über einen Pfad (z. B. „`/parent/child`“) adressiert; der Wurzelknoten ist über den Pfad „`/`“ erreichbar. Jeder Knoten bietet die Möglichkeit, eine geringe Menge an Daten (üblicherweise weniger als 1 KB) aufzunehmen. Die Daten eines Knotens werden ausschließlich atomar gelesen bzw. geschrieben, jedem Schreibvorgang wird dabei eine knotenspezifische Versionsnummer zugeordnet.

Zunächst soll eine Hilfsklasse `MWDataTree` realisiert werden, die auf Server-Seite die Verwaltung der Datenknoten übernimmt und dafür über folgende Methoden verfügt, die jeweils eine `MWZooKeeperException` werfen können:

```
public class MWDataTree {
    public String create(String path, byte[] data, long time);
    public void delete(String path, int version);
    public MWStat setData(String path, byte[] data, int version, long time);
    public byte[] getData(String path, MWStat stat);
}
```

Mit der Methode `create()` lässt sich unter dem Pfad `path` ein neuer Knoten mit den Nutzdaten `data` anlegen; `time` enthält dabei den Zeitstempel der Operation. 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 dann erfolgreich, wenn der Zustand des Knotens bei der Bearbeitung der Anfrage die angegebene Versionsnummer aufweist. Als Rückgabewert liefert `setData()` ein Objekt der Klasse `MWStat`, 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 dem Aufrufer über eine `MWZooKeeperException` mit passender Fehlermeldung signalisiert.

Aufgabe:

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

5.2 Verteilung des Diensts (für alle)

Im nächsten Schritt soll der entfernte Zugriff auf ein `MWDataTree`-Objekt ermöglicht werden. Hierzu ist zunächst ein ZooKeeper-Server (`MWZooKeeperServer`) zu implementieren, der über einen Server-Socket TCP-Verbindungen annimmt. Die eigentliche Bearbeitung der Anfragen soll in separaten Worker-Threads erfolgen. 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.

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

Auf Client-Seite fungiert die Klasse `MWZooKeeper` als Zugangspunkt für den ZooKeeper-Dienst. Hierzu bietet `MWZooKeeper` die Methoden `create()`, `delete()`, `setData()` und `getData()` (siehe Teilaufgabe 5.1) an und übersetzt jeden Aufruf dieser Methoden in einen Nachrichtenaustausch mit dem ZooKeeper-Server.

Aufgabe:

→ Implementierung der Klassen `MWZooKeeperServer` (Server-Seite) und `MWZooKeeper` (Client-Seite)

Hinweise:

- Anzahl und Art der Nachrichten, die Client und Server zur Kommunikation nutzen, sind freigestellt.
- Es darf angenommen werden, dass ein Client erst dann eine neue Anfrage stellt, wenn er die Antwort (bzw. eine Fehlermeldung) auf seine vorherige Anfrage erhalten hat.
- Aufgrund der parallelen Bearbeitung von Anfragen in Worker-Threads ist unbedingt auf korrekte Synchronisation der gemeinsam genutzten Datenstrukturen zu achten.

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 Fehlertoleranz des Diensts zu erhöhen, soll dieser nun dreifach aktiv repliziert werden. Da sich jeder Client mit einem beliebigen ZooKeeper-Replikat seiner Wahl verbinden kann, muss auf Server-Seite sichergestellt werden, dass alle Replikate über einen konsistenten Zustand verfügen. Wie in aktiv replizierten Systemen üblich, wird dies dadurch erreicht, dass alle Replikate die zustandsmodifizierenden Anfragen in der selben Reihenfolge bearbeiten; lesende Anfragen sind davon nicht betroffen (siehe Tafelübung). Die ZooKeeper-Implementierung von Apache greift zur Erstellung einer einheitlichen Anfragenreihenfolge auf allen Replikaten auf *Zab* zurück, das auch in dieser Aufgabe zum Einsatz kommen soll.

Für die Replikation des ZooKeeper-Diensts ist die Klasse `MWZooKeeperServer` so zu erweitern, dass alle eintreffenden zustandsmodifizierenden Anfragen erst dann bearbeitet werden, wenn sie von Zab geordnet wurden. Um die per Zab verteilten Anfragen zu empfangen, muss `MWZooKeeperServer` die Schnittstelle `org.apache.zookeeper.zab.ZabCallback` implementieren. Zudem ist sicherzustellen, dass sich die Reihenfolge der über die Methode `deliver()` zugestellten Anfragen nicht im Nachhinein ändert.

Da lesende Anfragen hinsichtlich der Konsistenz der Replikatzustände keine Rolle spielen, sieht ZooKeeper aus Performanzgründen vor, diese unmittelbar nach ihrem Empfang zu bearbeiten, unabhängig davon, ob sich andere Anfragen noch in Bearbeitung befinden. Diese Optimierung soll auch im `MWZooKeeperServer` ausgenutzt werden.

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) Clients u. U. unterschiedliche Versionsstände von Datenknoten sehen, je nachdem mit welchem Replikat sie verbunden sind.

Hinweise:

- Die zur Verwendung von Zab benötigten Klassen sind in `zab-dev.jar` (Pub-Verzeichnis) zusammengefasst.
- Um den geänderten Nachrichtenfluss auf einem Replikat zu testen, kann statt `MultiZab` zunächst ein Objekt der Klasse `SingleZab` als Schnittstelle zu Zab zum Einsatz kommen.
- Zab erzeugt zur Sicherung seines internen Zustands die Ordner `zabdata` und `zabsnap`. Um zu verhindern, dass vorherige Programmausführungen den aktuellen Zustand beeinflussen, sollten beide Ordner jeweils vor dem Start eines ZooKeeper-Replikats gelöscht werden.
- Aus Konsistenzgründen ist darauf zu achten, dass die Zeitstempel aller Replikate eines Knotens identisch sind. Zusätzlich soll sichergestellt werden, dass sämtliche Zeitstempel (nicht notwendigerweise streng) monoton steigen, um Sprünge in die Vergangenheit zu verhindern.

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 um einen persistenten oder einen flüchtigen Knoten handelt, wird bei der Erzeugung des Knotens festgelegt. Hierzu ist die `MWZooKeeper`-Methode `create()` um den Parameter `ephemeralNode` zu erweitern:

```
public String create(String path, byte[] data, boolean ephemeralNode);
```

Auf Server-Seite lässt sich die Unterstützung flüchtiger Knoten durch Erweiterung bzw. Hinzufügen der folgenden beiden Methoden der Klasse `MWDataTree` realisieren:

```
public String create(String path, byte[] data, long ephemeralOwner);
public void killSession(long ephemeralOwner);
```

Der Parameter `ephemeralOwner` identifiziert hierbei jeweils eindeutig die Verbindung zu einem bestimmten Client. Wird diese unterbrochen, sollen durch einen Aufruf der Methode `killSession()` die zugehörigen flüchtigen Knoten gelöscht werden. Es ist dabei darauf zu achten, dass bei der Einbeziehung von flüchtigen Knoten das Ende einer Client-Verbindung somit eine zustandsmodifizierende Operation (`killSession()`) nach sich zieht, die auf allen ZooKeeper-Replikaten konsistent ausgeführt werden muss.

Aufgabe:

→ Erweiterung der bestehenden Implementierung um die Unterstützung flüchtiger Knoten

Abgabe: am 3.2.2012