Middleware – Cloud Computing – Übung

ZooKeeper

Wintersemester 2024/25

Harald Böhm, Laura Lawniczak, Tobias Distler

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Systemsoftware)
https://sys.cs.fau.de

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Technische Fakultät

Überblick

ZooKeeper

Apache ZooKeeper

Apache ZooKeeper

- **Koordinierungsdienst** für verteilte Systeme
  - Anfangs entwickelt bei Yahoo! Research, jetzt Apache-Projekt
  - Im Produktiveinsatz unter anderem für:
    - Anführerwahl: z. B. Apache HDFS
    - Konfigurationsdaten: z. B. Kafka
- Verwaltung von Daten
  - **Hierarchischer Namensraum**: Knoten in einer Baumstruktur
  - Knoten sind eindeutig identifizierbar und können Nutzdaten aufnehmen
  - **Keine expliziten Sperren (Locks)**, aber Gewährleistung bestimmter Ordnungen bei konkurrierenden Zugriffen
- **Fehlertoleranz**
  - Replikation des Diensts auf mehrere Rechner (Replikate)
  - Replikatkonsistenz mittels Leader-Follower-Ansatz
  - Leseoptimierung: Jedes Replikat kann Leseanfragen beantworten
- Literatur

Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed
ZooKeeper: Wait-free coordination for Internet-scale systems
Proc. of the 2010 USENIX Annual Technical Conf. (ATC '10), S. 145–158, 2010.

Schnittstelle

- Zentrale Operationen
  - `create` / `delete` — Erstellen / Löschen eines Knotens
  - `exists` — Prüfen auf Existenz eines Knotens
  - `setData` / `getData` — Setzen und Auslesen der Nutzdaten und Metadaten eines Knotens
  - `getChildren` — Rückgabe der Pfade von Kindknoten eines Knotens
  - `sync` — Warten auf die Bearbeitung aller vorherigen Schreiboperationen
- **Persistente Knoten** (*Regular Nodes*)
  - Erzeugung durch den Client
  - Explizites Löschen durch den Client
- **Flüchtige Knoten** (*Ephemeral Nodes*)
  - Erzeugung durch den Client unter Angabe des `EPHEMERAL`-Flag
  - Keine Kindknoten
  - Löschen
    - Automatisches Löschen durch den Dienst, sobald die Verbindung zum Client, der diesen Knoten erstellt hat, beendet wird oder abbricht

    → Anwendungsbeispiel: Erkennen eines Client-Ausfalls
    - Explizites Löschen durch den Client
- **Sequenzielle Knoten** (*Sequential Nodes*) [Siehe Vorlesung]

Verwaltung von Nutzdaten

- Grundprinzipien [→ Unterschiede zu Dateisystemen]
  - Jeder Knoten kann Nutzdaten aufnehmen
    - Speicherung von Nutzdaten ist nicht auf Blattknoten des Baums beschränkt
    - Kleine Datenmengen, üblicherweise < 1 MB pro Knoten
  - Daten werden atomar geschrieben und gelesen
    - {S,Ers}etzen der kompletten Nutzdaten eines Knotens beim Schreiben
    - Kein partielles Lesen der Nutzdaten
- **Versionierung** der Nutzdaten
  - Schreiben neuer Daten → Inkrementierung der Knoten-Versionsnummer
  - Bedingtes Schreiben von Nutzdaten

```
public Stat setData(String path, byte[] data, int version);
```

    - Speicherung der Nutzdaten `data` nur, falls die aktuelle Versionsnummer des Knotens dem Wert `version` entspricht
    - Schreiben ohne Randbedingung: `version = -1` setzen
  - Kein Zugriff auf ältere Versionen möglich

Verwaltung von Metadaten

- Verwaltete Metadaten eines Knotens
  - Zeitstempel der Erstellung
  - Zeitstempel der letzten Modifikation
  - Versionsnummer der Nutzdaten
  - Größe der Nutzdaten
  - Anzahl der Kindknoten
  - Bei flüchtigen Knoten: ID der Verbindung des ZooKeeper-Clients, der den Knoten erstellt hat (*Ephemeral Owner*)
  - ...
- Abruf der Metadaten eines Knotens
  - Kapselung in einem Objekt der Klasse `Stat`
  - Nur in Kombination mit dem Lesen der Nutzdaten möglich
- Implementierungsentscheidung
  - Nutz- und Metadaten werden komplett im Hauptspeicher gehalten
  - Keine Strategie für den Fall, dass der Hauptspeicher voll ist

Benachrichtigung über Ereignisse

- Problemstellung
  - Client wartet darauf, dass ein bestimmtes Ereignis eintritt
  - Aktives Nachfragen durch den Client ist im Allgemeinen nicht effizient
- **Beobachter** (*Watcher*)
  - Registrierung bei Leseoperationen (muss ggf. erneuert werden!)
  - Ereignisarten
    - Erstellen / Löschen oder Ändern der Nutzdaten eines Knotens (`exists`)
    - Ändern der Nutzdaten oder Löschen eines Knotens (`getData`)
    - Hinzukommen oder Wegfallen von Kindknoten (`getChildren`)
  - Aufruf durch ZooKeeper-Dienst bei Eintritt bestimmter Ereignisse
- Schnittstelle für Beobachter-Objekte

```
public interface Watcher {
    public void process(WatchedEvent event);
}
```

ZooKeeper

Aufgabe 6

Aufgabe 6

- Umsetzung eines Koordinierungsdienstes
  - ZooKeeper-Implementierung von Apache als Vorbild
  - Funktionen zum Erstellen, Löschen, Schreiben und Lesen von Knoten

  ⇒ Bedingtes und unbedingtes Schreiben anhand von Versionsnummer

- Vereinfachte Schnittstelle

```
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);
```

- Teilaufgaben
  - Implementierung als Client-Server-Anwendung
  - Zustandsverwaltung inklusive **Leseoptimierung** von ZooKeeper

  ↪ Hilfestellung: Tests für Teilfunktionalitäten in `MWZooKeeperImplTest` bereitgestellt
  - **Konsistente, passive Replikation** unter Zuhilfenahme von Zab
  - Unterstützung flüchtiger Knoten (optional für 5,0 ECTS)

Ausgabeparameter in Java

- Problem
  - Methode (z. B. `getData()`) soll mehr als ein Objekt zurückgeben
  - Nur ein „echter“ Rückgabewert möglich
- Lösungsmöglichkeiten
  - Einführung eines Hilfsobjekts, das mehrere Rückgabewerte kapselt
  - Verwendung von **Ausgabeparametern**
- Beispiel für Ausgabeparameter: ZooKeeper-Methode `getData()`
  - Aufruf: Übergabe eines „leeren“ Parameters

```
MWZooKeeper zooKeeper = new MWZooKeeper([...]);
MWZooKeeperStat stat = new MWZooKeeperStat(); // Leeres Objekt
zooKeeper.getData("/example", stat);
System.out.println("Version:␣" + stat.getVersion());
```

  - Intern: Setzen von Attributen des Ausgabeparameters

```
public byte[] getData(String path, MWZooKeeperStat stat) {
    [...] // Bestimmung der angeforderten Daten
    stat.setVersion(currentVersion);
    [...] // Setzen weiterer Attribute und Daten-Rueckgabe
}
```

Serialisierung & Deserialisierung von Objekten

- Serialisierung & Deserialisierung in Java
  - Objekte müssen das Marker-Interface `Serializable` implementieren
  - {S,Des}erialisierung mittels `Object{Out,In}putStream`-Klassen

- Beispiel: Deserialisierung von Anfragen

```
// Einmaliges Anlegen des Objekt-Stroms
Socket s = [...]; // Socket der Verbindung
ObjectInputStream ois = new ObjectInputStream(s.getInputStream());

while(true) {
    // Empfang und Deserialisierung einer Anfrage
    MWZooKeeperRequest request = (MWZooKeeperRequest) ois.readObject();
    [...] // Bearbeitung der Anfrage
}
```

- **Wichtige Hinweise** zum Einsatz von Object-Streams:
  - Der Konstruktor eines `ObjectInputStream` blockiert, bis auf der anderen Seite ein `ObjectOutputStream` geöffnet wurde $\Rightarrow$ Bei Deadlock Reihenfolge beachten.
  - Object-Streams in Java **puffern Objekte**. Bei Wiederverwendung von Objekten können daher alte Daten übermittelt werden, wenn der Puffer nicht mit `reset()` geelert wird.