Middleware - Cloud Computing – Übung

Tobias Distler, Klaus Stengel,
Timo Hönig, Christopher Eibel

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)

www4.cs.fau.de

Wintersemester 2015/16

Überblick

ZooKeeper
- Einführung
- Konsistenzwahrung in ZooKeeper
- Aufgabe 6

Apache ZooKeeper

- **Fehlertoleranter Koordinierungsdienst** für verteilte Systeme
  - Anfangs entwickelt bei Yahoo! Research, jetzt Apache-Projekt
  - Im Produktiveinsatz (z. B. bei Yahoo und Facebook (Cassandra))
- Verwaltung von Daten
  - **Hierarchischer Namensraum**: Knoten in einer Baumstruktur
  - Knoten sind eindeutig identifizierbar und können Nutzdaten aufnehmen
  - **Keine expliziten Sperren oder Transaktionen**, aber Gewährleistung bestimmter Ordnungen bei konkurrierenden Zugriffen
- Fehlertoleranz
  - Replikation des Diensts auf mehrere Rechner
  - Replikatkonsistenz mittels Leader-Follower-Ansatz
- Literatur

Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed
ZooKeeper: Wait-free coordination for Internet-scale systems
Proceedings of the 2010 USENIX Annual Technical Conference, 2010.

Schnittstelle

- Zentrale Operationen
  - `create` Erstellen eines Knotens
  - `exists` Überprüfung, ob ein Knoten existiert
  - `delete` Löschen eines Knotens
  - `setData` Setzen der Nutzdaten eines Knotens
  - `getData` Auslesen der Nutz- und Metadaten eines Knotens
  - `getChildren` Rückgabe der Pfade von Kindknoten eines Knotens
  - `sync` Warten auf die Bearbeitung aller vorherigen zustandsmodifizierenden Operationen [Siehe später.]
- Aufrufvarianten
  - Synchron
  - Asynchron
- ZooKeeper-API

  `http://zookeeper.apache.org/doc/r3.4.6/api/`

Kategorien von Datenknoten

- **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
  - Löschen
    - Explizites Löschen durch den Client
    - Automatisches Löschen durch den Dienst, sobald die Verbindung zum Client, der diesen Knoten erstellt hat, beendet wird oder abbricht
  - Anwendungsbeispiel: Benachrichtigung über Knotenausfall
- **Sequenzielle Knoten** (*Sequential Nodes*)
  - Erzeugung durch den Client unter Angabe des `SEQUENTIAL`-Flag
  - Automatische Erweiterung des Knotennamens um eine vom System vergebene Sequenznummer
  - Anwendungsbeispiel: Herstellung einer Ordnung auf Clients

[Hinweis: Das `EPHEMERAL`- und das `SEQUENTIAL`-Flag sind miteinander kombinierbar]

Verwaltung von Nutzdaten

- Grundprinzipien [→ Unterschiede zu Dateisystemen]
  - Jeder Knoten kann Nutzdaten aufnehmen
    - Kleine Datenmengen, üblicherweise $< 1\,\text{KB}$ pro Knoten
    - „Verzeichnisknoten“ (also Knoten mit Kindknoten) können ebenfalls Nutzdaten direkt aufnehmen
  - 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);
```

    - Nutzdaten `data` werden nur geschrieben, falls die aktuelle Versionsnummer des Knotens `version` entspricht („test and set“)
    - 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

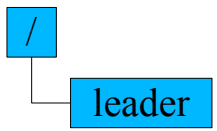
Benachrichtigung über Ereignisse

- Problemstellung
  - Client wartet darauf, dass ein bestimmtes Ereignis eintritt
  - Aktives Nachfragen durch den Client ist im Allgemeinen nicht effizient
- **Wächter** (*Watches*)
  - Umsetzung von Rückrufen (*Callbacks*) in ZooKeeper
  - Aufruf durch ZooKeeper-Dienst bei Eintritt bestimmter Ereignisse
  - Registrierung bei Leseoperationen (muss ggf. erneuert werden!)
  - Ereignisarten
    - Erstellen oder Löschen eines Knotens (`exists`)
    - Änderung der Nutzdaten eines Knotens (`getData`)
    - Hinzukommen oder Wegfall von Kindsknoten (`getChildren`)
- Schnittstelle für Wächter-Objekte

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

Anwendungsbeispiel: Wahl eines Anführers

- Problemstellung
  - In einer Gruppe von ZooKeeper-Clients soll ein Anführer gewählt werden
  - Bei Ausfall des Anführers muss ein neuer Anführer bestimmt werden
- Umsetzung
  - Erstellen eines „Verzeichnisknotens“ `/leader` für die Gruppe



  - Vorgehensweise beim Hinzukommen eines neuen Clients
    - Erstellen eines flüchtigen Kindknotens `/leader/node-<Sequenznummer>`
    - Suche nach Kindknoten mit kleineren Sequenznummern
    - Existiert kein Kindknoten mit kleinerer Sequenznummer → Client ist *Leader*
    - Sonst: Client ist *Follower* → Setzen eines Watch auf den Kindknoten mit der nächstkleineren Sequenznummer
  - Bei Knotenausfall
    - Automatische Löschung des zugehörigen flüchtigen Knotens
    - Genau ein Client wird per Watch über den Ausfall benachrichtigt

Anwendungsbeispiel: Wahl eines Anführers

Beispielablauf

- Client 1 kommt neu zur Gruppe hinzu
  - Erstellen eines flüchtigen Kindknotens `/leader/node-1`
  - Client 1 wird zum Leader, da sein Kindknoten die kleinste Sequenznummer aufweist [bzw. in diesem Fall keine weiteren Kindknoten vorhanden sind]

Anwendungsbeispiel: Wahl eines Anführers

Beispielablauf

- Client 2 kommt neu zur Gruppe hinzu
  - Erstellen eines flüchtigen Kindknotens `/leader/node-2`
  - Client 2 wird zum Follower
  - Client 2 setzt Watch auf Kindknoten mit nächstkleinerer Sequenznummer ($\rightarrow$ `/leader/node-1`)

Anwendungsbeispiel: Wahl eines Anführers

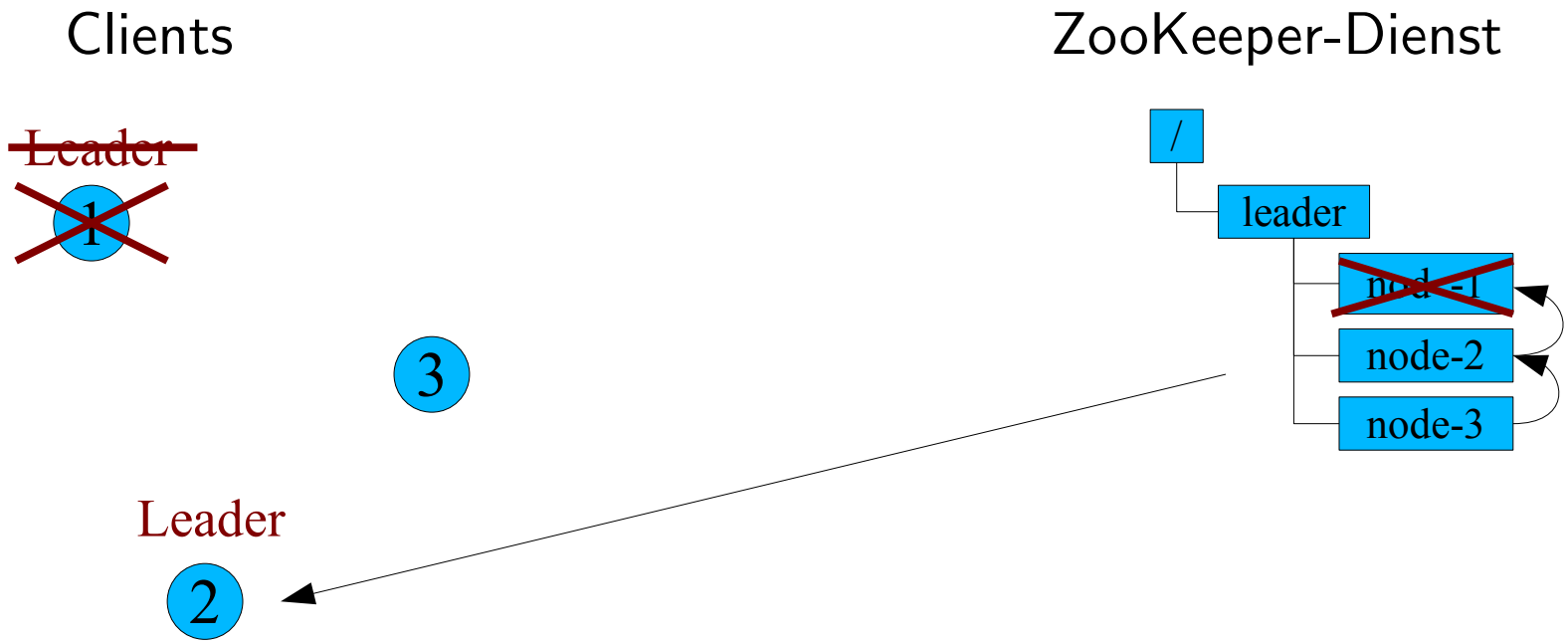
Beispielablauf

- Client 3 kommt neu zur Gruppe hinzu
  - Erstellen eines flüchtigen Kindknotens `/leader/node-3`
  - Client 3 wird zum Follower
  - Client 3 setzt Watch auf Kindknoten mit nächstkleinerer Sequenznummer ($\rightarrow$ `/leader/node-2`)

Anwendungsbeispiel: Wahl eines Anführers

Beispielablauf

- Ausfall des Leader-Knotens Client 1
  - Abbruch der Verbindung zum ZooKeeper-Dienst
  - Automatische Löschung des Kindknotens `/leader/node-1`
  - Client 2 wird per Watch über den Ausfall benachrichtigt und steigt damit zum neuen Leader auf



Überblick

ZooKeeper

- Einführung
- Konsistenzwahrung in ZooKeeper
- Aufgabe 6

Konsistenzwahrung in ZooKeeper

- Problemstellung
  - Replikation einer zustandsbehafteten Anwendung
  - Replikatzustände müssen konsistent gehalten werden
  - Beispiel für inkonsistente Zustände zweier Replikate R_1 und R_2
    - Zwei Anfragen A_1 und A_2, die einem Knoten `/node` neue Daten zuweisen

A_1:
```
setData("/node", new byte[] { 47 }, -1);
```

A_2:
```
setData("/node", new byte[] { 48 }, -1);
```

    - Annahme: A_1 erreicht R_1 früher als A_2, bei R_2 ist es umgekehrt

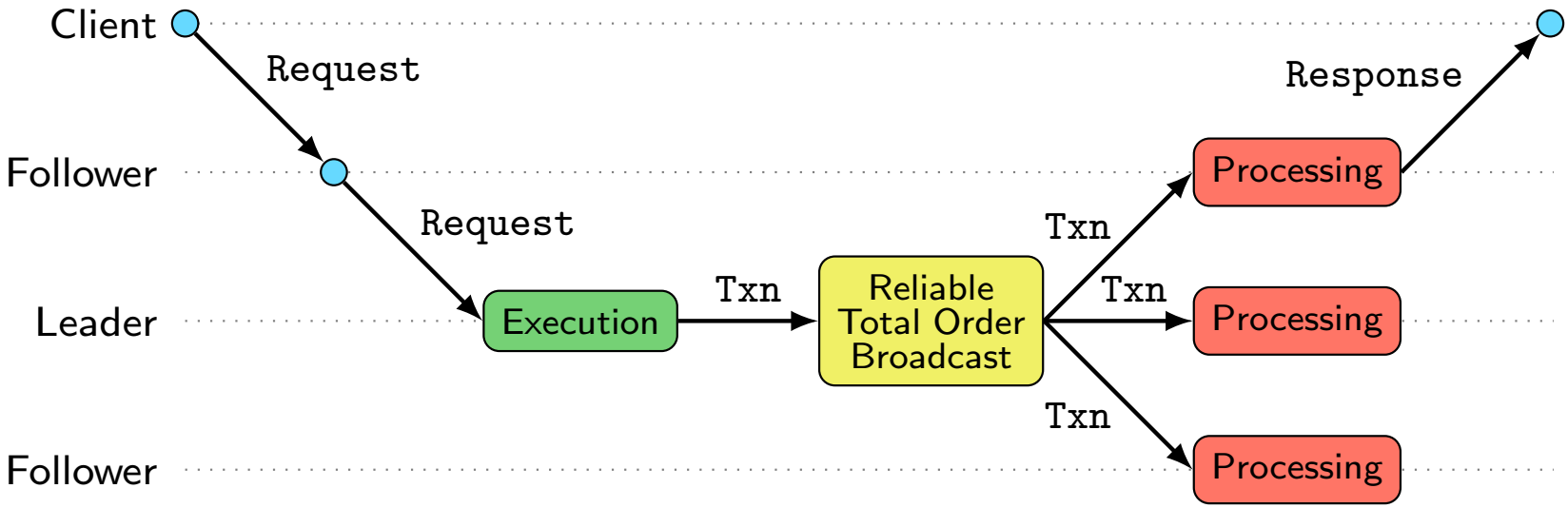
R_1	/node-Daten
$< init >$	null
A_1	[47]
A_2	[48]

R_2	/node-Daten
$< init >$	null
A_2	[48]
A_1	[47]

- Sicherstellung der Replikatkonsistenz in ZooKeeper
  - Alle Replikate vollziehen Zustandsänderungen in derselben Reihenfolge
  - Protokoll zur Erstellung und Bestätigung einer eindeutigen Reihenfolge

Replikation in ZooKeeper

- Gruppe von ZooKeeper-Replikaten
  - $2f + 1$ Replikate zur Tolerierung von höchstens f Fehlern bzw. Ausfällen
  - Jedes Replikat nimmt Verbindungen von Clients an
- Leader-Follower-Ansatz für schreibende Anfragen
  - Follower leitet Anfrage an den Leader weiter
  - Leader bearbeitet Anfrage und erzeugt dazu gehörige *Zustandstransaktion*
  - Fehlerfall: Erstellung einer *Fehlertransaktion* [Bsp.: Zu löschender Knoten existiert nicht.]
  - Konsistente Ausführung bestätigter Transaktionen auf allen Replikaten



Optimierung für lesende Anfragen

- Einsicht: Leseanfragen haben keinen Einfluss auf Replikatkonsistenz
- Optimierte Bearbeitung lesender Anfragen in ZooKeeper
  - Ausschließlich durch direkt mit Client verbundenem Replikat
  - Sofort nach Erhalt, d. h. unabhängig von schreibenden Anfragen
  - Aber: Unter Garantie von FIFO für sämtliche Anfragen eines Clients
- Vorteile
  - Einsparung von Ressourcen
  - Kürzere Antwortzeiten
- Konsequenzen
  - Antworten auf Leseanfragen sind abhängig vom bearbeitenden Replikat
  - Rückgabe von „veralteten“ Daten und Versionsnummern möglich
- Aufruf der `sync()`-Methode
  - Erzwingen eines Synchronisationspunkts
  - Warten, bis alle vor dem `sync()` empfangenen Anfragen bearbeitet wurden

Zustandsverwaltung im Anführer

- Problemstellung
  - Leseanfragen dürfen nur konsistenten, bestätigten Zustand zurückgeben
  - Schreibanfragen müssen auf aktuellem, unbestätigtem Zustand arbeiten

  $\Rightarrow$ Anführer muss beide Zustände gleichzeitig verwalten
- Effizienter Lösungsansatz
  - Bestätigter Zustand Z_B
    - Verwaltung des vollständigen Baumes von Datenknoten
    - Aktualisierung durch Einspielen bestätigter, total geordneter Transaktionen
    - Grundlage für die Bearbeitung rein lesender Anfragen
  - Aktueller Zustand Z_A
    - Verwaltung in Form einer Sammlung von Änderungen gegenüber Zustand Z_B
    - Modifikation durch Bearbeitung von schreibenden Anfragen
    - Basis für die Erstellung von Zustandstransaktionen
- Mechanismus zur Garbage-Collection
  - Vergabe eindeutiger IDs (`zxids`) an Zustandsänderungen/-transaktionen
  - Einspielen einer Transaktion $\rightarrow$ Löschen der unbestätigten Änderung

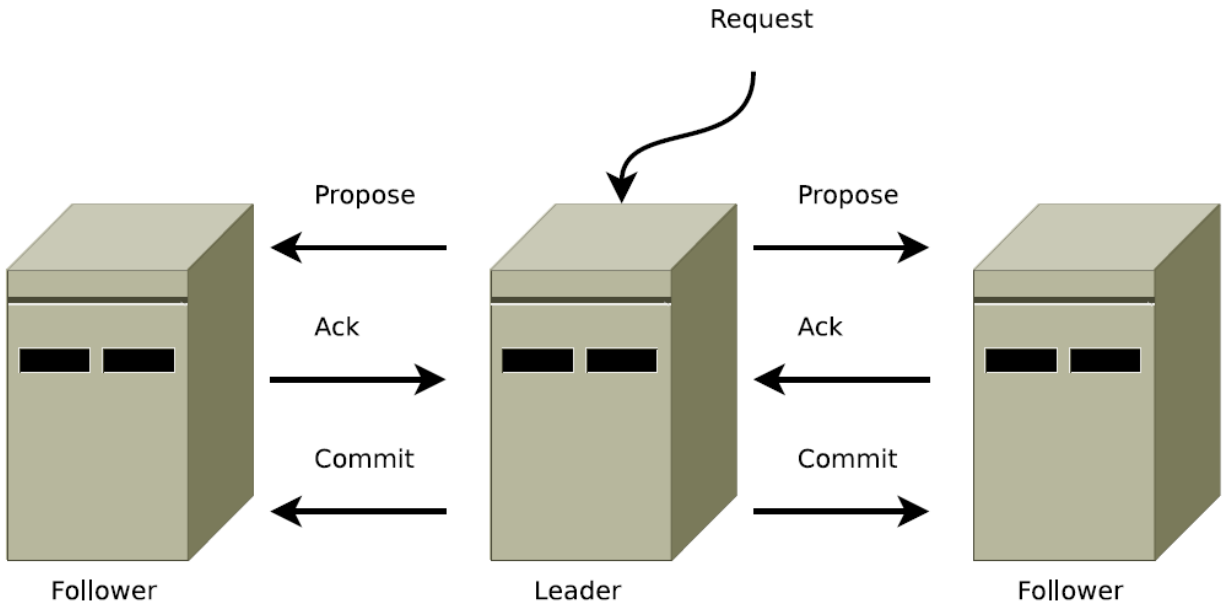
- Protokoll für zuverlässigen und geordneten Nachrichtenaustausch
  - Von Apache ZooKeeper verwendet, aber nicht modular integriert
  - Nachträgliche eigenständige Implementierung als *Zab*
  - Modifikation zur Anpassung an die Übungsaufgabe
- *Totally Ordered Broadcast Protocol*
  - Leader-Follower-Ansatz
  - Zwei Protokoll-Modi
    - *Broadcast* Normalbetrieb
    - *Recovery* Wahl eines neuen Anführers
- Literatur

Benjamin Reed and Flavio P. Junqueira
A simple totally ordered broadcast protocol
Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware, pages 1-6, 2008.

- Ziel

  Herstellung einer einheitlichen Reihenfolge aller Zustandstransaktionen

- Vorgehensweise
  - `PROPOSE` Anführer schlägt Sequenznummer (`zxid`) für Transaktion vor
  - `ACK` Follower akzeptieren den Vorschlag
  - `COMMIT` Anführer bestätigt die Sequenznummer der Transaktion



[Abbildung aus Reed et al. „A simple totally ordered broadcast protocol“]

- Abbruch des Broadcast-Modus
  - Ausfall des Anführers
  - Anführer hat keine Mehrheit mehr
- Eigenschaften des Recovery-Protokolls
  - Eine Transaktion, die auf einem Replikat bestätigt wurde (`COMMIT`), wird (gegebenenfalls nachträglich) auf allen Replikaten bestätigt
  - Nichtbestätigte Vorschläge werden verworfen
- Wahl eines neuen Anführers
  - Ziel: Neuer Anführer wird derjenige, dem der Vorschlag mit der höchsten `zxid` bekannt ist; bei Gleichstand entscheidet die höhere Replikat-ID
  - Rundenbasierte Abstimmung, in der jedes Replikat jedem anderen seinen aktuellen Kenntnisstand mitteilt
  - Bei Fehlern während der Wahl: Neustart des Vorgangs nach Timeout
- Nach erfolgreicher Wahl
  - Anführer stellt verloren gegangene Vorschläge und Bestätigungen bereit
  - Wiederaufnahme des Broadcast-Modus

- Repräsentation eines Zab-Knotens in der abstrakten Basisklasse `Zab`
- Varianten von Zab-Teilnehmern
  - `SingleZab` Einzelne (lokale) Instanz
  - `MultiZab` Teil einer verteilten Gruppe von Replikaten
- Methoden

```
public void startup();
public void shutdown();
public void forwardRequest(Serializable request);
public long createZXID();
public void proposeTxn(Serializable txn, long zxid);
```

  - `startup()` Starten eines Zab-Knotens
  - `shutdown()` Stoppen eines Zab-Knotens
  - `forwardRequest()` Weiterleiten einer Anfrage an den Anführer
  - `createZXID()` Erstellen einer neuen `zxid`
  - `proposeTxn()` Vorschlagen einer zu ordnenden Transaktion

[Hinweis: Da Zab in den ersten 4 Bytes einer `zxid` eine Epochennummer codiert, führt eine Neuwahl des Anführers zu einem Sprung in den von `createZXID()` erzeugten `zxid`-Werten.]

- Empfang von Nachrichten über die Schnittstelle `ZabCallback`
- Methoden

```
public void deliverRequest(Serializable request);
public void deliverTxn(Serializable txn, long zxid);
public void status(ZabStatus status, String leader);
```

  - `deliverRequest()` Übergabe einer dem Anführer weitergeleiteten Anfrage
  - `deliverTxn()` Zustellung der nächsten geordneten Transaktion
  - `status()` Benachrichtigung über Änderungen des Status
- Status eines Zab-Knotens (`ZabStatus`)
  - `LOOKING` Temporärer Zustand während der Anführerwahl
  - `FOLLOWING` Lokales Replikat ist Follower
  - `LEADING` Lokales Replikat ist Anführer
- Hinweise
  - Geordnete Transaktionen werden durch Zab sequentiell zugestellt
  - Aufrufe von `deliverRequest()` können dagegen nebenläufig erfolgen

- Übergabe eines `Properties`-Objekts an den `Zab`-Konstruktor
- Parameter
  - `myid` ID des lokalen Replikats
  - `peer<i>` Zab-Adresse des Replikats *i*
  - ...
- Beispielkonfiguration eines `MultiZab`-Knotens (insgesamt 3 Replikate)
  - Zusammenstellung der Konfiguration

```
Properties zabProperties = new Properties();
zabProperties.setProperty("myid", String.valueOf(1));
zabProperties.setProperty("peer1", "localhost:12345");
zabProperties.setProperty("peer2", "localhost:12346");
zabProperties.setProperty("peer3", "localhost:12347");
```

  - Initialisierung eines Zab-Knotens

```
ZabCallback zabListener = [...];
Zab zabNode = new MultiZab(zabProperties, zabListener);
```

Überblick

ZooKeeper

- Einführung
- Konsistenzwahrung in ZooKeeper
- Aufgabe 6

Aufgabe 6

- Umsetzung eines Koordinierungsdienstes
  - Funktionen zum Erstellen, Löschen, Schreiben und Lesen von Knoten
  - ZooKeeper-Implementierung von Apache als Vorbild
- Teilaufgaben
  - Implementierung als Client-Server-Anwendung
  - Replikation unter Zuhilfenahme von Zab
  - Unterstützung flüchtiger Knoten
- 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);
```

- Fokus der Übungsaufgabe
  - Konsistente Replikation eines zustandsbehafteten Diensts
  - Unterschiedliche Behandlung von schreibenden und lesenden Anfragen

Ausgabeparameter in Java

- Problem
  - Methode 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
}
```

- Hinweis zum Einsatz von Object-Streams in Verbindung mit Sockets
  - Der Konstruktor des `ObjectInputStream` blockiert so lange, bis auf der anderen Seite der Verbindung ein `ObjectOutputStream` geöffnet wurde

⇒ Zuerst den `ObjectOutputStream` öffnen, dann den `ObjectInputStream`

Logging mit log4j

- Zab verwendet intern die Logging-API *log4j*
  - Konfiguration mittels einer Datei `log4j.properties`, die im Classpath der Java-Anwendung abgelegt sein muss
  - Granularitätsstufen: `OFF`, `ERROR`, `WARN`, `DEBUG`, `ALL`, ...
- Beispiele für log4j-Konfigurationen
  - Ausgabe der Log-Meldungen auf der Konsole (Stufe: `DEBUG`)

```
log4j.rootLogger=DEBUG, CONSOLE
log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
log4j.appender.CONSOLE.layout=org.apache.log4j.PatternLayout
```

  - Ausgabe der Log-Meldungen in der Datei `zab.log` (Stufe: `INFO`)

```
log4j.rootLogger=INFO, FILE
log4j.appender.FILE=org.apache.log4j.FileAppender
log4j.appender.FILE.File=zab.log
log4j.appender.FILE.layout=org.apache.log4j.PatternLayout
```