

# Middleware – Cloud Computing – Übung

## Aufgabe 6: ZooKeeper

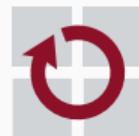
---

Wintersemester 2025/26

Paul Bergmann, Christian Berger

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

Aufgabe 6

Replikation

Konsistenzwahrung

Zab

**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.*

- 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]

- 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

- 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 ⇒ 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()` geleert wird.

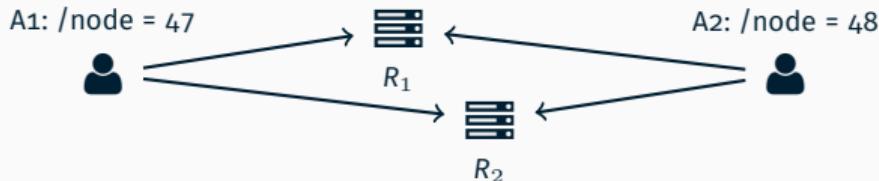
**Replikation**

---

**Konsistenzwahrung**

# Konsistenzwahrung

- 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



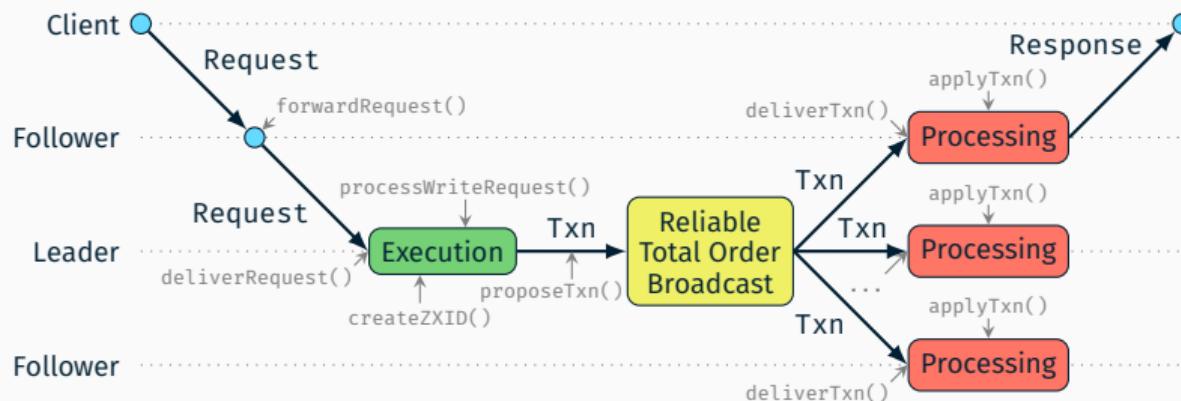
- Annahme:  $A_1$  erreicht  $R_1$  früher als  $A_2$ , bei  $R_2$  ist es umgekehrt

$R_1$	/node-Daten	$R_2$	/node-Daten
< init >	$\emptyset$	< init >	$\emptyset$
$A_1$	47	$A_2$	48
$A_2$	48 ⚡	$A_1$	47 ⚡

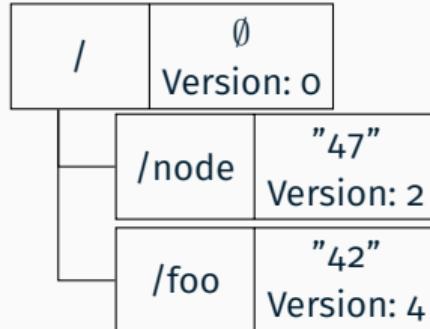
- Sicherstellung der **Replikatkonsistenz**: Alle Replikate vollziehen Zustandsänderungen in derselben Reihenfolge
- Replikationsvarianten
  - Aktiv: Anfragen an alle Replikate verteilen und dort ausführen
  - Passiv (**Zookeeper**): Anführer bearbeitet Anfragen und verteilt Zustandsänderungen

# Replikation in ZooKeeper

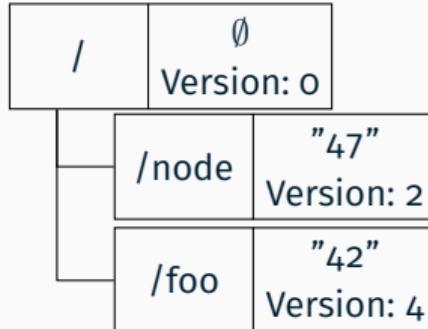
- Gruppe von ZooKeeper-Replikaten
  - $2f + 1$  Replikate zur Tolerierung von höchstens  $f$  Fehlern bzw. Ausfällen
  - Jedes Replikat nimmt Verbindungen von Clients an
- Leader-Follower-Ansatz für stark konsistente Schreibanfragen
  - Follower leitet Anfrage an den Leader weiter
  - Leader bearbeitet Anfrage und schreibt Änderungen in Zustandstransaktion
  - Fehlerfall: Erstellung einer Fehlertransaktion [Bsp.: Zu löschernder Knoten existiert nicht.]
  - Total Order Broadcast verteilt Transaktionen in vom Leader vorgegebener Reihenfolge
  - Transaktionsauslieferung erst nach Bestätigung durch Mehrheit der Replikate
  - Konsistente Ausführung ausgelieferter Transaktionen auf allen Replikaten



Zustand des Anführers



Zustand eines Followers



Anfrage

Client A: setData("/node", "47", 1)

Transaktion

/node	"47"
	Version: 2

Antwort an Client



Client B: setData("/node", "48", 1)

/node	⚡
-------	---



⚠ Das Beispiel wird im zugehörigen Video besprochen

- 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
- sync()-Methode
  - Erzwingen eines Synchronisationspunkts
  - Wartet bis alle vor dem sync() empfangenen Anfragen bearbeitet wurden

## ■ Problemstellung

- Leseanfragen dürfen nur konsistenten, bestätigten Zustand zurückgeben
  - ⇒ Unbestätigte Zustandsänderungen könnten im Fehlerfall noch verloren gehen
  - Schreibanfragen müssen aber auf aktuellem, unbestätigtem Zustand arbeiten
  - ⇒ 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

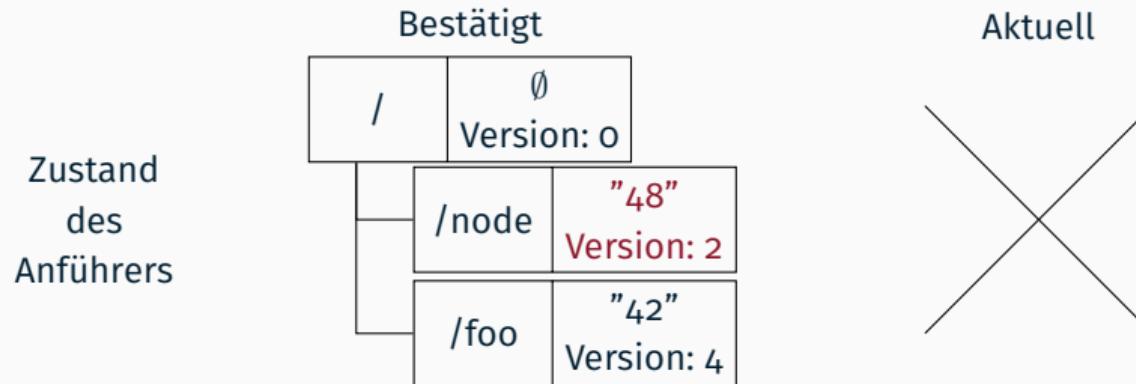
### ▪ Aktueller Zustand $Z_A$

- Verwaltung in Form einer Sammlung von gegenüber Zustand  $Z_B$  geänderten Knoten
- 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 → Löschen der unbestätigten Änderung

# Anfrageverarbeitung ohne aktuellen Zustand

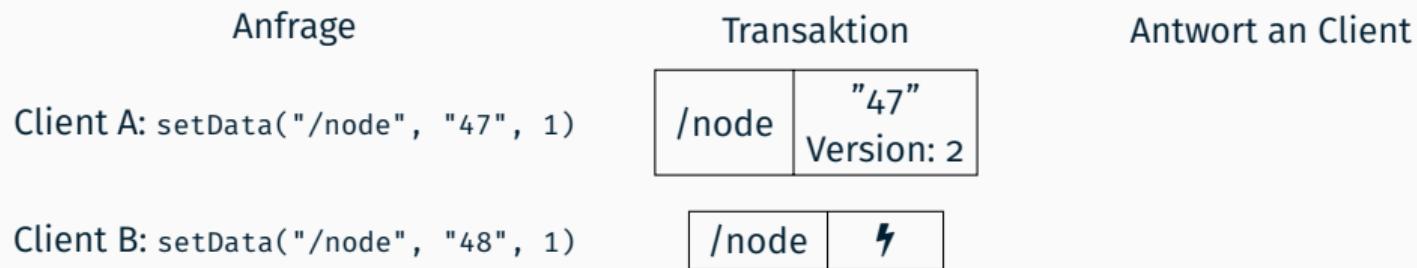
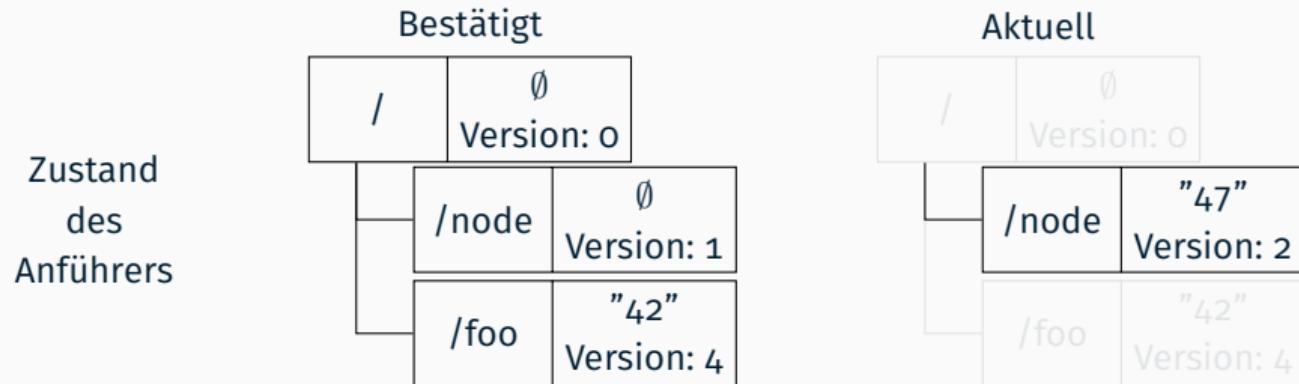


---

Anfrage	Transaktion	Antwort an Client		
Client A: setData("/node", "47", 1)	<table border="1"><tr><td>/node</td><td>"47" Version: 2</td></tr></table>	/node	"47" Version: 2	✓
/node	"47" Version: 2			
Client B: setData("/node", "48", 1)	<table border="1"><tr><td>/node</td><td>"48" Version: 2</td></tr></table>	/node	"48" Version: 2	✓ 
/node	"48" Version: 2			

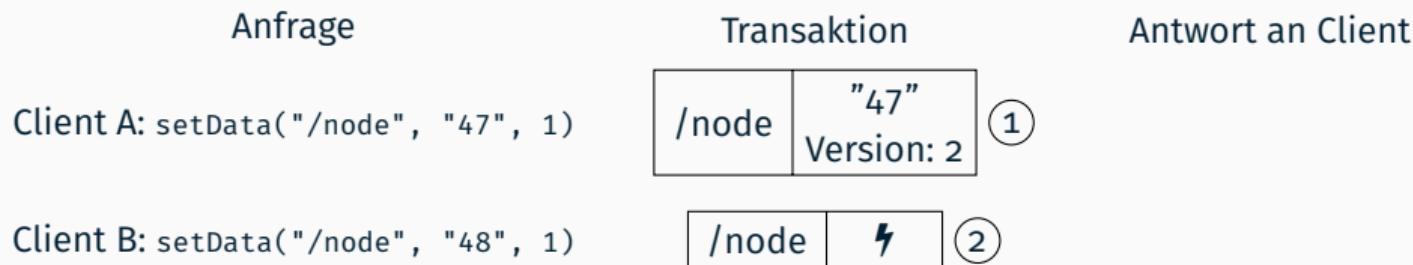
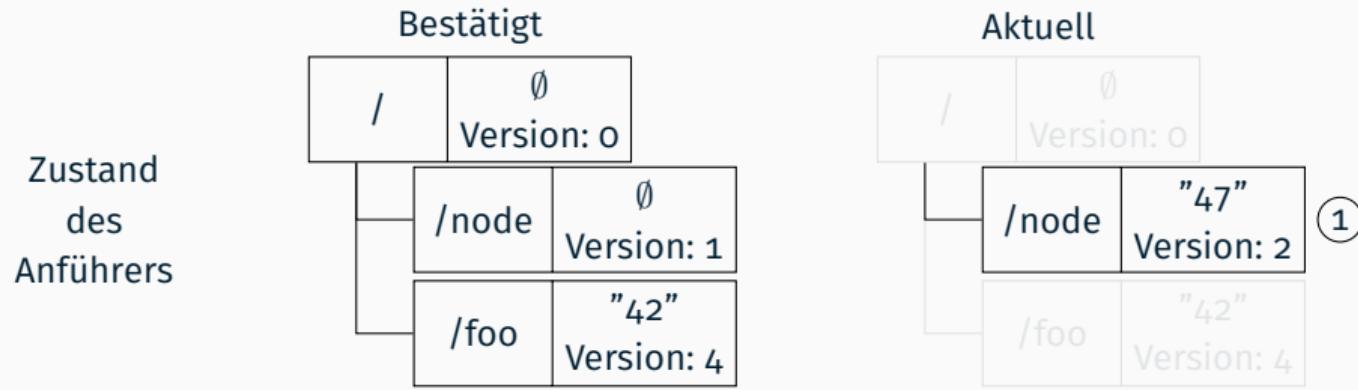
⚠ Das Beispiel wird im zugehörigen Video besprochen

# Anfrageverarbeitung mit aktuellem Zustand



⚠ Das Beispiel wird im zugehörigen Video besprochen

# Garbage-Collection von Transaktionen



⚠ Das Beispiel wird im zugehörigen Video besprochen

## **Replikation**

---

**Zab**

- 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
  - Übungsfolien sind Dokumentation der modifizierten Bibliothek
- *Totally Ordered Broadcast Protocol* mit zwei Betriebsmodi
  - **Normalbetrieb (Broadcast)**
    - Bereitstellen einer eindeutigen **Sequenznummer** (zxid) für jede Transaktion
    - Zuverlässige Verteilung aller Zustandstransaktionen in Reihenfolge der Sequenznummern
  - **Wahl eines neuen Anführers (Recovery)**
    - Szenarien: Ausfall des Anführers, Anführer hat keine Mehrheit mehr
    - Sicherstellung der Eindeutigkeit von Sequenznummern
- 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.*

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

- `startup()` Starten eines Zab-Knotens
  - `shutdown()` Stoppen eines Zab-Knotens
  - `forwardRequest()` Weiterleiten einer Anfrage an den Anführer
  - `createZXID()` Anfordern der nächsten Sequenznummer (zxid)
  - `proposeTxn()` Vorschlagen einer zu ordnenden Transaktion
    - Aufruf muss in Reihenfolge der zxids erfolgen
    - `createZXID()` und `proposeTXN()` immer als Paar aufrufen

[Hinweis: Da Zab in den ersten 4 Bytes einer zxid eine Epochennummer codiert, führt eine Neuwahl des Anführers zu einem Sprung in den von `createZID()` 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

- Aufrufe von deliverRequest() können nebenläufig erfolgen
- Geordnete Transaktionen werden dagegen durch Zab sequentiell zugestellt
- Alle von einer Mehrheit ( $f + 1$ ) der  $2f + 1$  Replikate bestätigten Transaktionen werden auf allen korrekten Replikaten zugestellt

- Übergabe eines Properties-Objekts an den zab-Konstruktor
- Parameter
  - myid ID des lokalen Replikats
  - peer<*i*> Zab-Adresse des Replikats *i*
  - ...
- Identische Konfiguration der peer<*i*>-Adressen auf allen Replikaten nötig
- Beispielkonfiguration eines MultiZab-Knotens (insgesamt 3 Replikate)
  - Zusammenstellung der Konfiguration für ein Replikat mit der ID 1

```
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);
```

- Zab verwendet intern die Logging-API *log4j*
  - Konfiguration z.B. durch eine Datei `log4j2.properties`, die im Classpath abgelegt sein muss
  - Granularitätsstufen: OFF, ERROR, WARN, INFO, DEBUG, ALL, ...
  - Dokumentation unter:  
<https://logging.apache.org/log4j/2.x/manual/configuration.html>
- Beispiele für log4j-Konfigurationen
  - Ausgabe der Log-Meldungen auf der Konsole (Stufe: DEBUG)

```
rootLogger = DEBUG, CONSOLE
appender.CONSOLE.name = CONSOLE
appender.CONSOLE.type = Console
appender.CONSOLE.layout.type = PatternLayout
```

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

```
rootLogger=INFO, FILE
appender.FILE.name = FILE
appender.FILE.type = File
appender.FILE.fileName = zab.log
appender.FILE.layout.type = PatternLayout
```