

## Replikation

Grundlagen der Replikation

JGroups

Übungsaufgabe 5



## Varianten

- Aktive Replikation („Hot Standby“)
  - Alle Replikate bearbeiten alle Anfragen
  - Vorteil: Schnelles Tolerieren von Ausfällen möglich
  - Nachteil: Vergleichsweise hoher Ressourcenverbrauch
- Passive Replikation
  - Ein Replikat bearbeitet alle Anfragen
  - Aktualisierung der anderen Replikate erfolgt über Sicherungspunkte
  - Unterscheidung: „Warm Standby“ vs. „Cold Standby“
  - Vorteil: Minimierung des Aufwands im fehlerfreien Fall
  - Nachteil: Im Fehlerfall schlechtere Reaktionszeit als bei aktiver Replikation

## Replikationstransparenz

- Nutzer auf Client-Seite merkt nicht, dass der Dienst repliziert ist
- Replikatausfälle werden dem Nutzer verborgen



## Aktive Replikation von Diensten

### ■ Zustandslose Dienste

- Keine Koordination zwischen Replikaten notwendig
- Auswahl des ausführenden Replikats z. B. nach Last- oder Ortskriterien

### ■ Zustandsbehaftete Dienste

- Replikatzustände müssen konsistent gehalten werden
- Beispiel für inkonsistente Zustände zweier Replikate  $R_0$  und  $R_1$ 
  - put()-Anfragen  $A_1$  („MyKey“, „42“) und  $A_2$  („MyKey“, „0“) von verschiedenen Nutzern
  - Annahme:  $A_0$  erreicht  $R_0$  früher als  $A_1$ , bei  $R_1$  ist es umgekehrt

$R_0$	Key-Value-Speicher	$R_1$	Key-Value-Speicher
$\langle init \rangle$	[ ]	$\langle init \rangle$	[ ]
$A_0$	[ („MyKey“, „42“) ]	$A_1$	[ („MyKey“, „0“) ]
$A_1$	[ („MyKey“, „0“) ]	$A_0$	[ („MyKey“, „42“) ]

### ■ Sicherstellung der Replikatkonsistenz

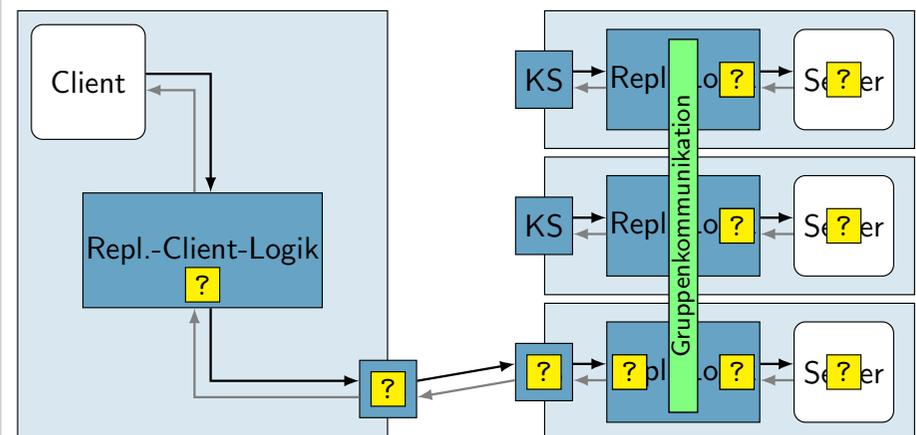
- Alle Replikate müssen Anfragen in derselben Reihenfolge bearbeiten
- Protokoll/Dienst zur Erstellung einer Anfragenreihenfolge nötig



## Aktive Replikation von Diensten

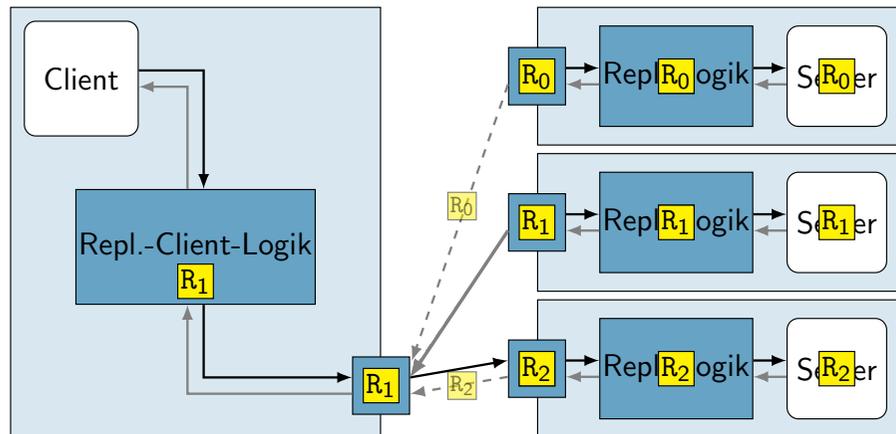
### ■ Weg der Anfrage

- Senden der Anfrage an ein Replikat
- Verteilen der Anfrage (z. B. durch ein Gruppenkommunikationssystem)
- Bearbeitung der Anfrage auf allen Replikaten



## Aktive Replikation von Diensten

- Weg der Antwort
  - Alle Replikate senden ihre Antwort zum Client
  - Im einfachsten Fall: Client akzeptiert schnellste zurückgelieferte Antwort



## Konsistenz von Replikaten

- Voraussetzung für aktive Replikation: Anwendungsreplikate müssen dieselbe *deterministische Zustandsmaschine* realisieren
  - Identische Ausgangszustände
  - Identische Eingaben [→ Anfragen in derselben Reihenfolge.]
  - ⇒ Identische Zustandsänderungen
  - ⇒ Identische Ausgaben
- Herausforderungen
  - Mehrfädige Programme
    - Zutrittsreihenfolge von kritischen Abschnitten kann unterschiedlich sein
    - Ansatz: Zusätzliche Koordinierung zwischen Replikaten notwendig
  - Nichtdeterministische Systemaufrufe
    - Beispiele: `System.currentTimeMillis()`, `Math.random()`
    - Ansatz: Einigung auf gemeinsamen Wert
  - Seiteneffekte bzw. externalisierte Ereignisse
    - Beispiel: Anwendung greift auf externe Dienste zu [z. B. Rückruf am `VS AuctionEventHandler`]
    - Ansatz: Ein Replikat macht Aufruf, Ergebnisweitergabe an die anderen
  - ...



## Überblick

### Replikation

Grundlagen der Replikation

JGroups

Übungsaufgabe 5



## Gruppenkommunikation

## Übersicht

- Bereitstellung von *Virtual Synchrony*
  - Zusammenschluss von Knoten zu Gruppen
  - Senden von Nachrichten an die Gruppe (anstatt an jeden Knoten einzeln)
  - Alle Knoten erhalten jede
    - an die Gruppe versendete Nachricht
    - gruppeninterne Statusmeldungin derselben Reihenfolge
  - Alle Knoten einer Gruppe machen (vermeintlich) synchron Fortschritt
- Grundlegende Dienste
  - Membership-Service
  - Total-Ordering-Multicast
  - Zustandstransfer-Mechanismus
- Beispiele
  - **JGroups** [<http://www.jgroups.org/index.html>]
  - Spread
  - ...



## Membership-Service

- Problemstellungen
  - Zusammensetzung einer Gruppe kann dynamisch variieren
    - Knoten kommen neu hinzu
    - Knoten verlassen die Gruppe
  - Fehlersituationen
    - Verbindungsabbruch zu einzelnen Knoten
    - Gruppenpartitionierung
- Aufgabe des Membership-Service
  - Benachrichtigung aller Gruppenmitglieder über die gegenwärtige Zusammensetzung der Gruppe
- JGroups: Schnittstelle `org.jgroups.MembershipListener`
  - Benachrichtigung über Gruppenänderungen

```
void viewAccepted(View new_view);
```
  - Mitteilung eines Ausfallverdachts

```
void suspect(Address suspected_mbr);
```



## View

- Aktuelle Sicht auf die Gruppe
  - Liste aller aktiven Gruppenmitglieder
- Problem
  - Keine gemeinsame Zeitbasis
  - Was bedeutet also „aktuell“?
- Lösung
  - Änderung der Gruppenzusammensetzung: Erzeugung einer neuen View
  - Aktuelle Teilnehmer einigen sich auf die neue View

⇒ Abfolge von Views fungiert als gemeinsame Zeitbasis
- JGroups: Klasse `org.jgroups.View`
  - Ausgabe der Gruppenmitglieder

```
List<Address> getMembers();
```
  - Ausgabe der Gruppengröße

```
int size();
```



## Total-Ordering-Multicast

- Problemstellung
  - Clients sollen ihre Anfragen an einen beliebigen Server senden können
  - Alle Server müssen alle Anfragen in derselben Reihenfolge bearbeiten
    - Bewahrung konsistenter Server-Zustände
    - Bereitstellung konsistenter Antworten (z. B. für Fehlertoleranz)
- Total-Ordering-Multicast: Alle aktiven Knoten einer Gruppe bekommen alle Nachrichten in derselben Reihenfolge zugestellt
  - Interne Algorithmen, die
    - jeder Nachricht eine eindeutige Sequenznummer zuweisen → totale Ordnung
    - sicherstellen, dass jeder aktive erreichbare Knoten jede Nachricht erhält
    - dafür sorgen, dass jeder Knoten die Nachrichten in der richtigen Reihenfolge an die Anwendung weiter gibt
  - Hinweis
    - Jede Nachricht wird an **alle** Gruppenmitglieder zugestellt; also auch an den Knoten, der die Nachricht ursprünglich gesendet hat



## Nachricht

- Klasse `org.jgroups.Message`
  - Kapselung der eigentlichen Nutzdaten
  - Container für Protokoll-Header
  - Konstruktoren

```
Message(Address dst);
Message(Address dst, Address src, Object obj);
[...]
```

    - `dst` Zieladresse; falls `null` → alle
    - `src` Ursprungsadresse; falls `null` → durch JGroups ausgefüllt
    - `obj` Nutzdaten als Payload
  - Wichtigste Methoden

```
Object getObject();
Message copy();
```

    - `getObject()` Getter-Methode für Payload
    - `copy()` Erzeugung einer Kopie der Nachricht



## Kommunikationskanal

### ■ Klasse org.jgroups.JChannel

#### ■ Konstruktoren

```
JChannel() // Standardkonfiguration
JChannel(File properties) // XML-Datei
JChannel(String properties) // Konfig. als Zeichenkette
```

#### ■ Wichtigste Methoden

- Verbindungsaufbau zur Gruppe `cluster_name`

```
void connect(String cluster_name);
```

- Diverse Getter-Methoden

```
Address getAddress() // Eigene Adresse
String getClusterName() // Gruppenname
View getView() // Aktuelle View
```

- Nachrichtenversand

```
void send(Message msg)
void send(Address dst, Object obj)
```

Hinweis: Senden einer Nachricht an alle → `dst = null` setzen



## Nachrichteneingang: Kombinierte Listener & Adapter

### ■ Kombinierte Schnittstelle: org.jgroups.Receiver

```
public interface Receiver extends MembershipListener,
    MessageListener {}
```

### ■ Erweiterte Adapterklasse: org.jgroups.ReceiverAdapter

- Implementiert (unter anderem) Receiver
- Eigene Receiver-Klasse als Unterklasse von ReceiverAdapter

### ■ Beispiel

```
public class VSReceiver extends ReceiverAdapter {
    public void receive(Message msg) { // <- MessageListener
        System.out.println("received message " + msg);
    }

    public void viewAccepted(View newView) { // <- MembershipListener
        System.out.println("received view " + newView);
    }
}
```

### ■ Registrierung am JChannel

```
void setReceiver(Receiver r);
```



## Zustandstransfer (im Allgemeinen)

### ■ Problem

- Knoten bearbeiten alle Anfragen, um ihre Zustände konsistent zu halten
- Was ist mit Knoten, die
  - später hinzu kommen, also nicht alle Anfragen kennen
  - mit der Bearbeitung der Anfragen nicht hinterher kommen oder
  - aufgrund eines Fehlers über kaputte Zustandsteile verfügen?

### ■ Lösung: Unterstützung von Zustandstransfers

- Mit Hilfe der Gruppenkommunikation wird dafür gesorgt, dass ein Knoten (z. B. beim Gruppenbeitritt) eine Kopie des aktuellen Zustands erhält
- Der aktuelle Zustand stammt von einem Knoten aus der Gruppe



## Zustandstransfer (mit Hilfe von JGroups)

### ■ Zusätzliche Methoden des MessageListener

- Bereitstellung des eigenen Zustands
- ```
void getState(OutputStream output) throws Exception;
```

- Setzen des lokalen Zustands

```
void setState(InputStream input) throws Exception;
```

### ■ Außerdem: Holen des Zustands eines anderen Replikats (JChannel)

```
void getState(Address target, long timeout) throws Exception;
```

### ■ Generelle Schritte

1. Konsistentes Holen des Zustands von existierendem Replikat (`getState()`)
  - ↪ Holen des Zustands am besten vom Koordinator mittels

```
channel.getState(null, 0L);
```
  - Blockiert so lange, bis `setState()` den Zustand zugestellt hat
  - Achtung: `setReceiver()`-Aufruf sollte vorher geschehen sein, da `getState()`-Aufruf sonst unendlich lange blockiert
2. Übertragen des Zustands (→ JGroups)
3. Konsistentes Einspielen des Zustands beim Zielreplikat (`setState()`)



# Überblick

## Replikation

Grundlagen der Replikation

JGroups

Übungsaufgabe 5

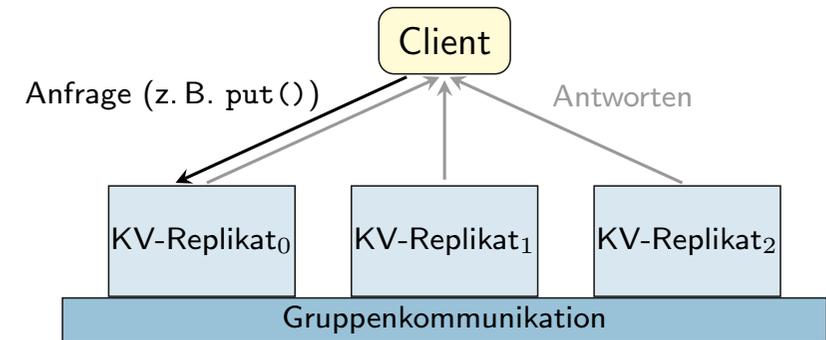
# Speicherdienst für Schlüssel-Wert-Paare

```
public class VSKeyValueClient {  
    // Basisfunktionalitaet  
    public void put(String key, String value) throws RemoteException;  
    public String get(String key) throws VSKeyValueException,  
        RemoteException;  
    public void delete(String key) throws RemoteException;  
    public long exists(String key) throws RemoteException;  
  
    // Erweiterte Variante (optional fuer 5,0 ECTS)  
    public long reliableExists(String key, int threshold)  
        throws RemoteException;  
}
```

- put() (Über-)Schreiben eines Werts value unter Schlüssel key
- get() Abfragen des Datensatzes mit dem Schlüssel key
- delete() Löschen des Datensatzes mit dem Schlüssel key
- exists() Prüfen der Existenz eines Schlüssels und ggf. Rückgabe des Zeitstempels der Erstellung oder letzten Aktualisierung
- reliableExists() Wie exists(), Rückgabe des Ergebnisses aber nur, wenn #threshold Replikate dasselbe Ergebnis liefern

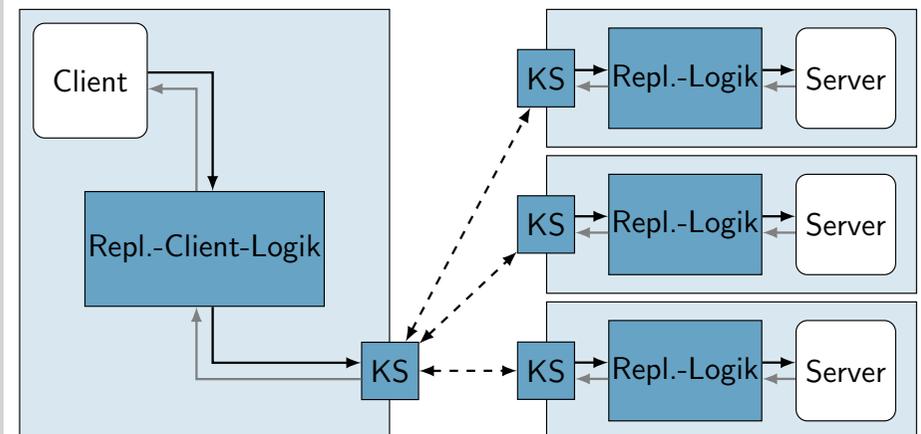
# Übungsaufgabe 5: Überblick

- Basisfunktionalität (für alle)
  - Implementierung eines Schlüssel-Wert-Speichers (engl. key-value store)
  - Replikation des Speicherdiensts
  - Implementierung und Ausfallsicherung der Client-Seite
- Erweiterte Variante (optional für 5,0 ECTS)
  - Verifizierung von Ergebnissen
  - Neustarten eines Replikats nach dessen Ausfall



# Replikation des Speicherdiensts

- Client sendet Anfrage an ein beliebiges Replikat
- Empfang von bis zu #Replikate Antworten



## Replikation des Speicherdiensts

- Aktive Replikation des eigenen Fernaufrufsystems
  - Drei Replikate auf verschiedenen Rechnern
  - Alle Replikate bearbeiten alle Anfragen in derselben Reihenfolge
  - Alle Replikate senden ihre Antwort zum Client
- Replikation mittels JGroups
  - JGroups-Bibliothek im Pub-Verzeichnis (/proj/i4vs/pub/aufgabe5)
  - Konfiguration für Total-Ordering-Multicast: Einsatz eines *Sequencer*
    - Legt Reihenfolge der Nachrichten fest und verteilt diese an alle Replikate
    - Zu verteilende Nachrichten werden (intern) an den Sequencer geschickt

```
JChannel channel = new JChannel(); // Kanal erstellen

// Erweiterung des Standard-Protokoll-Stacks um Sequencer
ProtocolStack protocolStack = channel.getProtocolStack();
protocolStack.addProtocol(new SEQUENCER());

[...] // Receiver registrieren und Verbindung oeffnen

■ Zugriff auf Message-Header (hier: zusätzlicher Zugriff auf Sequenz-Nr.)
long seqNr = ((SequencerHeader) message.getHeader(
    ClassConfigurator.getProtocolId(SEQUENCER.class))).getSeqno();
```



## Client-Server-Kommunikation

- Alle Replikate müssen in der Lage sein, den Client zu erreichen
- Antwortübermittlung von Replikaten zu Clients mittels Rückrufen
  - Replikat implementiert `handleRequest()`-Methode, die vom Client (fern-)aufgerufen werden kann

```
public interface VSKeyValueRequestHandler extends Remote {
    public void handleRequest(VSKeyValueRequest request)
        throws RemoteException;
}
```

↪ Mögliche Implementierung:  
Remote-Referenz des Client ist Bestandteil der Anfragenachricht

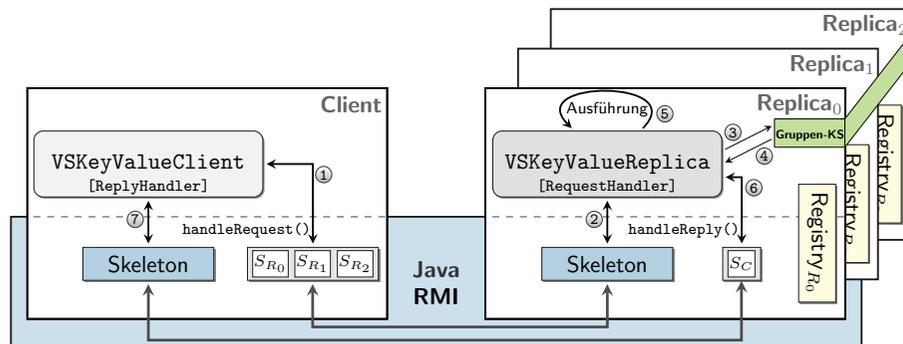
- Client implementiert `handleReply()`-Methode, die von den Replikaten (fern-)aufgerufen werden kann

```
public interface VSKeyValueReplyHandler extends Remote {
    public void handleReply(VSKeyValueReply reply)
        throws RemoteException;
}
```



## Client-Server-Kommunikation: Gesamtüberblick

- Methodenfernaufrufe erfolgen über Java RMI
- Jedes Replikat verfügt über eigene (RMI-)Registry
  - Clients erhalten Replikat-Stubs ( $S_{R_0}$ ,  $S_{R_1}$ ,  $S_{R_2}$ ) über jeweilige Registry
- Replikate nutzen jeweiligen Client-Stub ( $S_C$ ) zur Antwortrückgabe



## Referenzierung von Diensten/Replikaten

- Problem: Clients müssen Replikatreferenzen bekanntgemacht werden
  - Bekanntmachen und Festlegen der Adressen (Hostname:Port) der einzelnen Replikat-Registries über dieselbe Datei
- Beispieldatei (Dateiname: `replica.addresses`)

```
faii00a:12345
faii00b:12346
faii00c:12347
```

→ 1. Zeile korrespondiert zu 1. Replikat, 2. Zeile zu 2. Replikat usw.

- Beispielkommandozeilenaufruf

- Client

```
java -cp <classpath> vsue.replica.VSKeyValueClient replica.addresses
```

- Server (Starten des 1. Replikats → ID: 0 ⇔ 1. Zeile)

```
java -cp <classpath> vsue.replica.VSKeyValueReplica 0 replica.addresses
```



## Logging in JGroups

### Konfiguration

- Granularitätsstufen: OFF, SEVERE, WARNING, INFO, FINE, FINER, ALL,...
- Konfiguration in Datei
- Programmstart

```
java -Djava.util.logging.config.file=<Datei> <Programm>
```

### Beispiele für Konfigurationsdateien

- Ausgabe der Log-Meldungen auf der Konsole (Stufe: FINE)

```
handlers=java.util.logging.ConsoleHandler  
.level=FINE
```

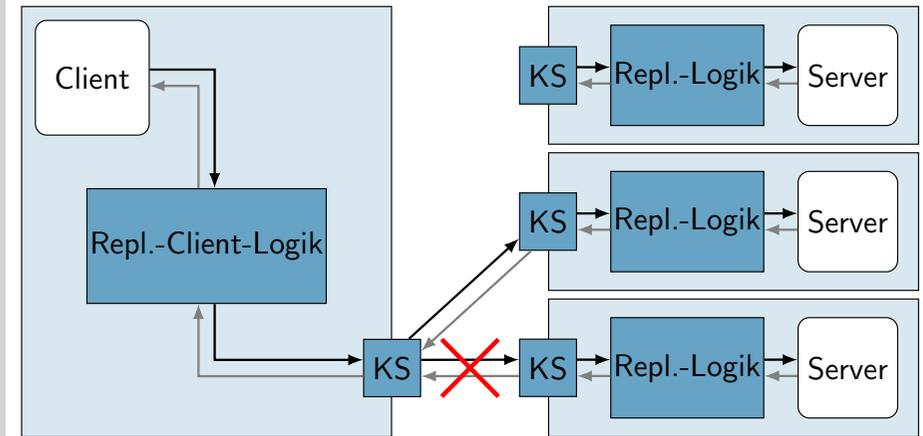
- Ausgabe der Log-Meldungen in einer Datei vs.log (Stufe: INFO)

```
handlers=java.util.logging.FileHandler  
.level=INFO  
java.util.logging.FileHandler.pattern=vs.log
```



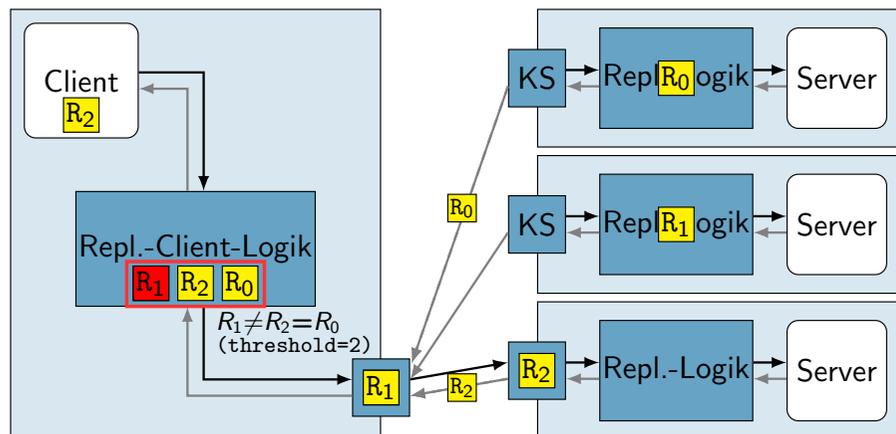
## Ausfallsicherung der Client-Seite

- Wechsel des Replikats bei Verbindungsabbrüchen
- Client implementiert selbst verschiedene Strategien zum Wechsel eines Replikats



## Verifizierung von Ergebnissen

- exists() liefert stets schnellste, aber evtl. fehlerhafte Antwort
- Bei Verwendung von reliableExists() erfolgt Vergleich mehrerer Antworten, um falsche Antworten zu erkennen



## Neustart nach Replikatausfall

- Problem
  - VSKeYValueReplica-Implementierung verwaltet ihren Zustand im Hauptspeicher
  - Datenverlust bei Ausfall eines Replikats
  - Kein Neustart des Replikats möglich
- Lösung
  - Verwendung von JGroups für Zustandstransfer (siehe Folie 7-15)
    - Annahme: Zu jeder Zeit ist mindestens ein Replikat verfügbar
    - Neustart: Holen des Zustands von einem anderen Replikat
  - Implementieren von getState()/setState() zum Auslesen/Serialisieren und Setzen/Deserialisieren des Zustands
  - Zustandstransfer und Nachrichtenempfang sind voneinander entkoppelt, d. h. Zustellung totalgeordneter Nachrichten ist asynchron zu Zustandstransfer
    - Nachrichten können bereits vor initialem getState()-Aufruf eintreffen und sich auch während des Zustandstransfers weiter anhäufen
    - {get, set}State()- und receive()-Aufrufe müssen synchronisiert werden
    - Konsistenzwahrung beim Zustandstransfer ist Teil der Replikatlogik

