Verteilte Systeme – Übung

Replikation: Aufgabe

Sommersemester 2025

Harald Böhm, Christian Berger, Tobias Distler

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

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Technische Fakultät

Überblick

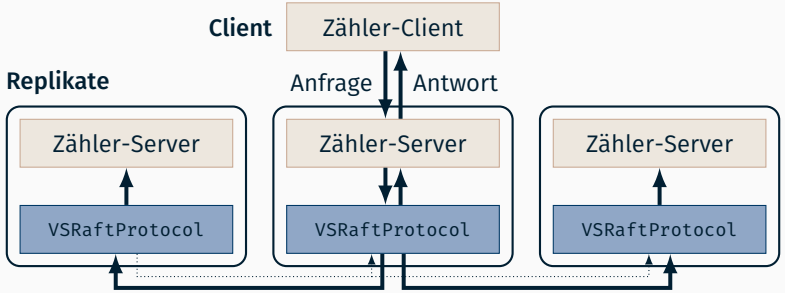
Übungsaufgabe 5

Übungsaufgabe 5

Übungsaufgabe 5: Überblick

Replikation eines einfachen Zählerdiensts mithilfe des Replikationsprotokolls Raft

- Basisfunktionalität (für alle)
  - Implementierung der Anführerwahl
  - Implementierung der Replikation von Anfragen
- Erweiterte Variante (optional für 5,0 ECTS)
  - Übertragung von Snapshots zwischen Replikaten
  - Neustarten eines Replikats nach dessen Ausfall



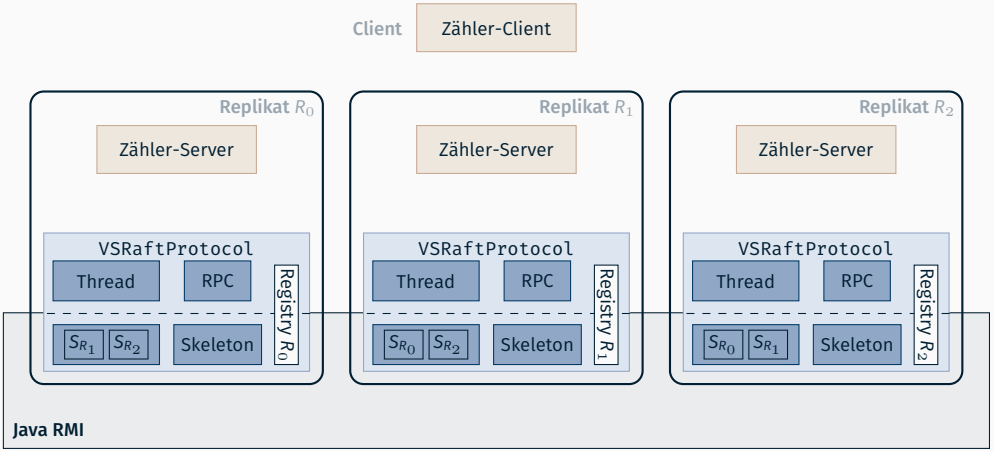
Schnittstelle zwischen Anwendung und Raft-Protokoll

```
public class VSRaftProtocol {
  public void init(VSCounterServer application);
  public boolean orderRequest(Serializable request);
}
public class VSCounterServer {
  // Basisfunktionalitaet
  public void status(VSRaftRole role, int leaderId);
  public void applyRequest(VSRaftLogEntry entry);
  // Erweitere Funktionalitaet (optional fuer 5 ECTS)
  public Serializable createSnapshot();
  public void applySnapshot(Serializable snapshot);
}
```

- Replikationsprotokoll: Raft `VSRaftProtocol`
  - `init()` Replikatkommunikation aufsetzen und Protokoll-Thread starten
  - `orderRequest()` Anfrage zum Replizieren übergeben
- Anwendung: Zählerdienst `VSCounterServer`
  - `status()` Rolle dieses Replikats und aktuellen Anführer der Anwendung mitteilen
  - `applyRequest()` Fertig geordnete Anfrage ausführen
  - `createSnapshot()` Snapshot des Anwendungszustands erstellen
  - `applySnapshot()` Snapshot einspielen, um Anwendungszustand zu aktualisieren

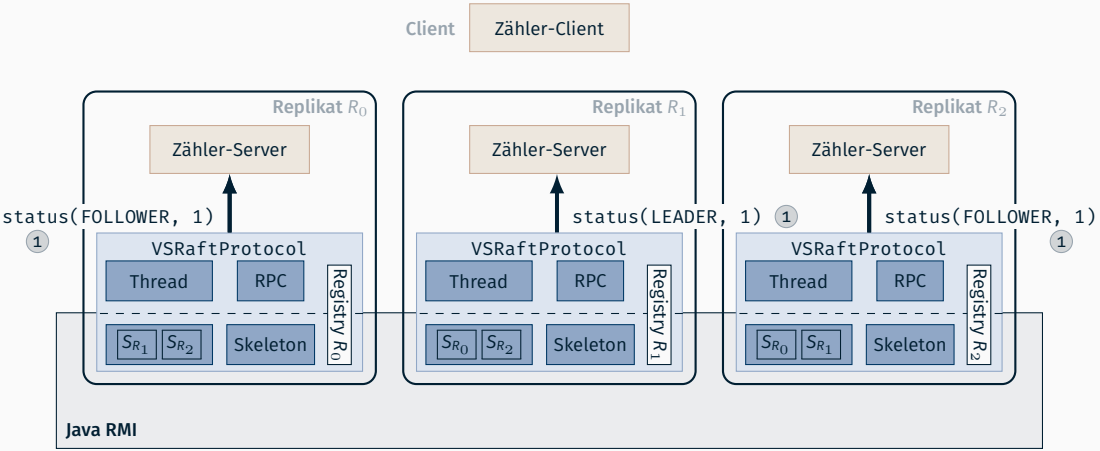
Anfragereplikation: Gesamtüberblick

- Methodenfernaufrufe erfolgen per Java RMI
  - Jedes Replikat R_i verfügt über (RMI-)Registry für eigenen Stub S_{R_i}
  - Stub bei Kommunikationsproblemen erneut abfragen
- Bereitgestellter Client wiederholt Anfrage im Fehlerfall



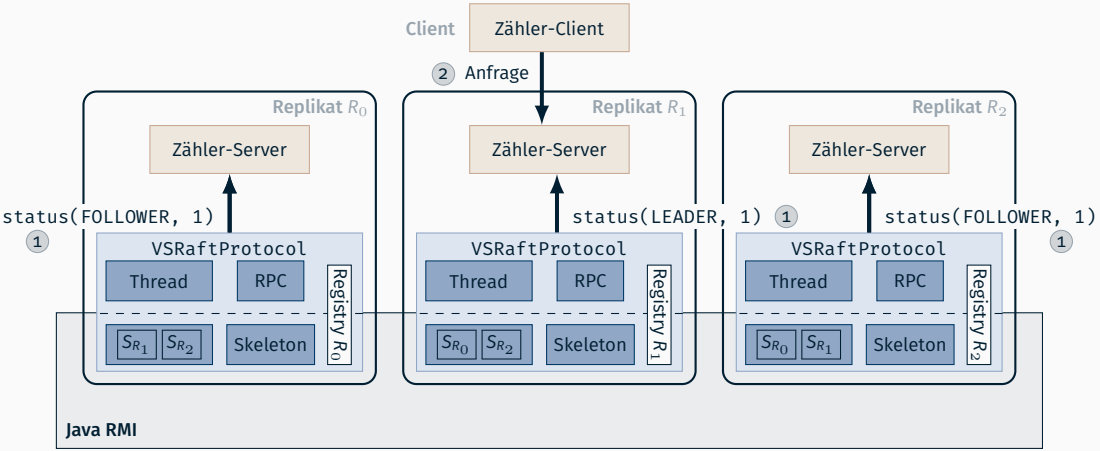
Anfragereplikation: Gesamtüberblick

- Methodenfernaufrufe erfolgen per Java RMI
  - Jedes Replikat R_i verfügt über (RMI-)Registry für eigenen Stub S_{R_i}
  - Stub bei Kommunikationsproblemen erneut abfragen
- Bereitgestellter Client wiederholt Anfrage im Fehlerfall



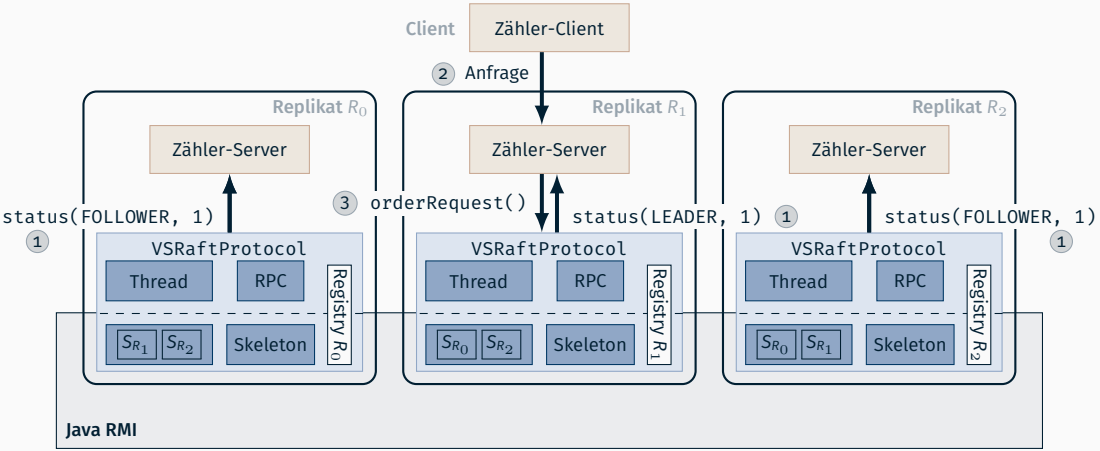
Anfragereplikation: Gesamtüberblick

- Methodenfernaufrufe erfolgen per Java RMI
  - Jedes Replikat R_i verfügt über (RMI-)Registry für eigenen Stub S_{R_i}
  - Stub bei Kommunikationsproblemen erneut abfragen
- Bereitgestellter Client wiederholt Anfrage im Fehlerfall



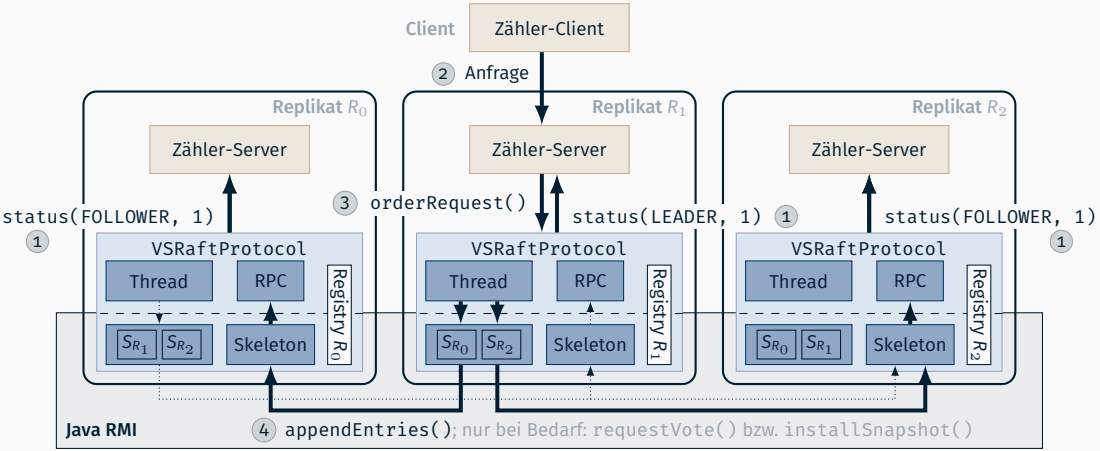
Anfragereplikation: Gesamtüberblick

- Methodenfernaufrufe erfolgen per Java RMI
  - Jedes Replikat R_i verfügt über (RMI-)Registry für eigenen Stub S_{R_i}
  - Stub bei Kommunikationsproblemen erneut abfragen
- Bereitgestellter Client wiederholt Anfrage im Fehlerfall



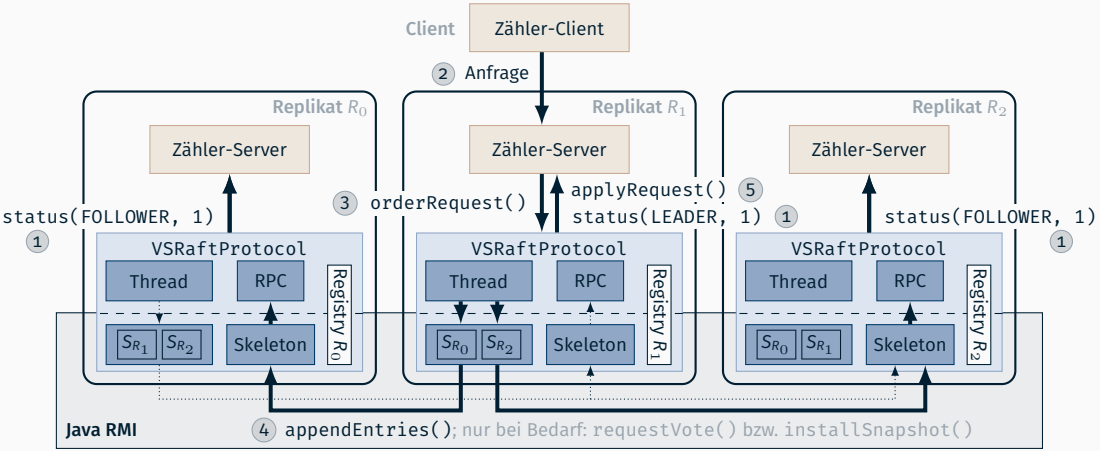
Anfragereplikation: Gesamtüberblick

- Methodenfernaufrufe erfolgen per Java RMI
  - Jedes Replikat R_i verfügt über (RMI-)Registry für eigenen Stub S_{R_i}
  - Stub bei Kommunikationsproblemen erneut abfragen
- Bereitgestellter Client wiederholt Anfrage im Fehlerfall



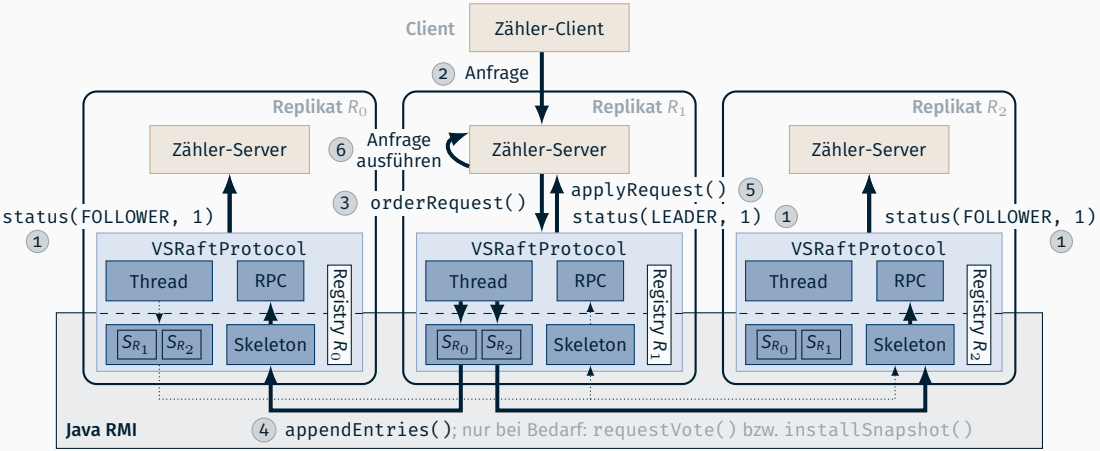
Anfragereplikation: Gesamtüberblick

- Methodenfernaufrufe erfolgen per Java RMI
  - Jedes Replikat R_i verfügt über (RMI-)Registry für eigenen Stub S_{R_i}
  - Stub bei Kommunikationsproblemen erneut abfragen
- Bereitgestellter Client wiederholt Anfrage im Fehlerfall



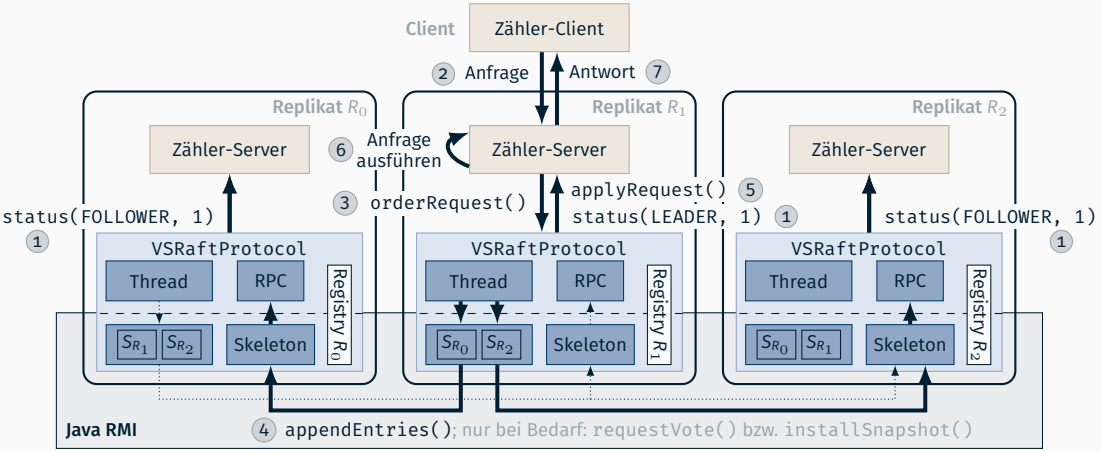
Anfragereplikation: Gesamtüberblick

- Methodenfernaufrufe erfolgen per Java RMI
  - Jedes Replikat R_i verfügt über (RMI-)Registry für eigenen Stub S_{R_i}
  - Stub bei Kommunikationsproblemen erneut abfragen
- Bereitgestellter Client wiederholt Anfrage im Fehlerfall



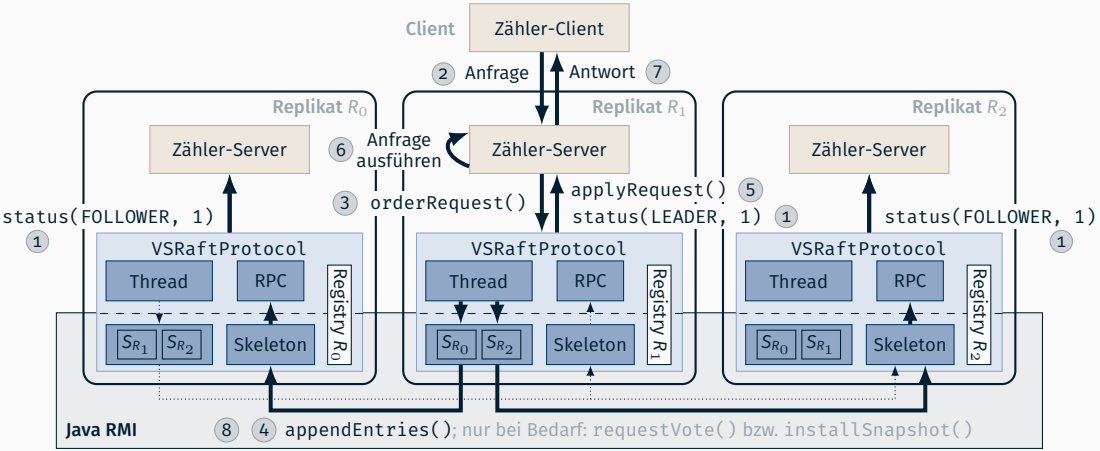
Anfragereplikation: Gesamtüberblick

- Methodenfernaufrufe erfolgen per Java RMI
  - Jedes Replikat R_i verfügt über (RMI-)Registry für eigenen Stub S_{R_i}
  - Stub bei Kommunikationsproblemen erneut abfragen
- Bereitgestellter Client wiederholt Anfrage im Fehlerfall



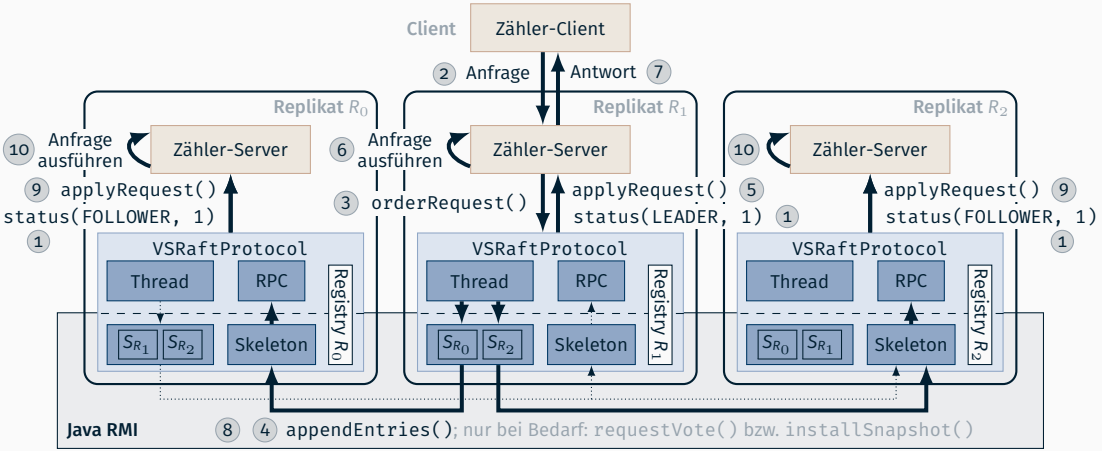
Anfragereplikation: Gesamtüberblick

- Methodenfernaufrufe erfolgen per Java RMI
  - Jedes Replikat R_i verfügt über (RMI-)Registry für eigenen Stub S_{R_i}
  - Stub bei Kommunikationsproblemen erneut abfragen
- Bereitgestellter Client wiederholt Anfrage im Fehlerfall



Anfragereplikation: Gesamtüberblick

- Methodenfernaufrufe erfolgen per Java RMI
  - Jedes Replikat R_i verfügt über (RMI-)Registry für eigenen Stub S_{R_i}
  - Stub bei Kommunikationsproblemen erneut abfragen
- Bereitgestellter Client wiederholt Anfrage im Fehlerfall



Protokoll-Thread - Beispiel

- Protokollimplementierung
  - Aktiver Teil: **Protokoll-Thread** sendet Fernaufrufe an andere Replikate
    - Führt anfallende Aufgaben nacheinander aus
    - Wartet blockierend auf neue Aufgaben
    - Bearbeitet periodische / rollen-spezifische Aufgaben
  - Passiver Teil: Empfangene Fernaufrufe abarbeiten

- **Beispiel-Anwendung: Leaderboard**
  - Highscore mithilfe von Java RMI zwischen Rechnern verteilen

```
void updateScoreRPC(String name, int score);
```

  - Highscore besteht aus Name und erzielten Punkten
- Nur Inhaber des Highscore soll diesen verteilen
  - Wenn sich der eigene Score ändert $\Rightarrow$ sofort verteilen
  - Sonst $\Rightarrow$ periodisch wiederholen
- Nur ein einziger Protokoll-Thread
  - Einfache Implementierung
  - Sequentielle Fernaufrufe

Einfache, aber **fehlerhafte** Implementierung eines Leaderboards

```
void syncThread() { // Nur ein Thread
    while(true) {
        synchronized(this) {wait(10_000);} //  [...] InterruptedException behandeln
        if (!highscoreName.equals(myName)) continue; // Pruefen, ob eigener Highscore
        for (int i = 0; i < replicaCount; ++i) { // RPCs der Reihe nach absetzen
            if (i == myId) continue;
            getStub(i).updateScoreRPC(myName, highscore); //  [...] RemoteException behandeln
        }
    }
}

void updateScoreRPC(String name, int score) { // Highscore aktualisieren
    if (score > highscore) {
        highscore = score;
        highscoreName = name;
    }
}

void newScore(int score) {
    highscore = score;
    highscoreName = myName;
    synchronized(this) {notify();} // Neuen Highscore sofort verteilen
}
```

Verlust von Highscore während Verteilung möglich

Zustand nicht atomar aktualisiert

Einfachste Lösung: Alles mit `synchronized` versehen

```
synchronized void syncThread() {
    while(true) {
        wait(10_000); // InterruptedException behandeln
        if (!highscoreName.equals(myName)) continue;
        for (int i = 0; i < replicaCount; ++i) {
            if (i == myId) continue;
            getStub(i).updateScoreRPC(myName, highscore);  // RemoteException behandeln
        }
    }
}

synchronized void updateScoreRPC(String name, int score) {
    if (score > highscore) {
        highscore = score;
        highscoreName = name;
    }
}

synchronized void newScore(int score) {
    highscore = score;
    highscoreName = myName;
    notify();
}
```

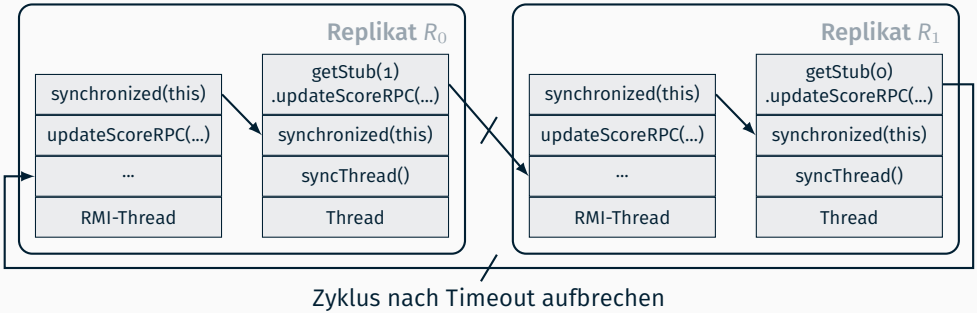
RPC erfolgt innerhalb des `synchronized`-Blocks

⚠ Deadlock-Gefahr

✔ `synchronized` verhindert nebenläufig Highscore-Änderung

✔ Zustand atomar aktualisiert

- Möglicher Deadlock, wenn zwei Replikate aktiv sind:



- Dauer von Fernaufrufen in RMI begrenzen

```
// Antwortzeit fuer RPC begrenzen
System.setProperty("sun.rmi.transport.tcp.responseTimeout", "100");
```

Referenzierung von Diensten/Replikaten

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

```
replica0=faui00a:12345
replica1=faui00b:12346
replica2=faui00c:12347
```

  → 1. Zeile korrespondiert zu Replikat 0, 2. Zeile zu Replikat 1 usw.
- Beispielkommandozeilenaufruf
  - Client

```
java -cp <classpath> vsue.raft.VSCounterClient replica.addresses
```

  - Server (Starten von Replikat 0)

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

VSRaftLog

Bereitgestellte Klasse zum Verwalten des Logs von Raft

```
public class VSRaftLog {
    public void addEntry(VSRaftLogEntry entry);                                  // Schreiboperationen
    public void storeEntries(VSRaftLogEntry[] entries);
    public VSRaftLogEntry getEntry(long index);                                  // Leseoperationen
    public VSRaftLogEntry[] getEntriesSince(long startIndex);
    public VSRaftLogEntry getLatestEntry();
    public long getLatestIndex();
    public void collectGarbage(long lastSnapshotIndex, int lastSnapshotTerm);    // Garbage Collection
    public long getStartIndex();
}
```

`addEntry()`	Log-Eintrag hinzufügen
`storeEntries()`	Log-Bereich abspeichern, ersetzt Log-Einträge bei Überschneidung
`getEntry()`	Log-Eintrag abrufen
`getEntriesSince()`	Log-Bereich ab Index abrufen
`getLatestEntry()`	Neusten Log-Eintrag abrufen
`getLatestIndex()`	Index des neusten Log-Eintrags abrufen
`collectGarbage()`	Einträge löschen, die bereits in Snapshot enthalten
`getStartIndex()`	Index des ältesten noch verfügbaren Log-Eintrags abrufen

Tipps zum Debugging in Replizierten Systemen

- Bugs sind häufig abhängig vom Timing und/oder treten über Rechnergrenzen hinweg auf
  → Debugger o.Ä. nur begrenzt einsetzbar

- **Bessere Alternative:** Logs bzw. `System.out.println()`-Debugging zum Nachvollziehen der Ereignisse im System
  - Problem: Logs werden durch viele Ereignisse schnell sehr lang und unübersichtlich

  → Reduktion der Ereignisse durch künstliche Verlangsamung des Systems, z.B. durch
    - Hochsetzen der Replikat-Timeouts (z.B. Heartbeat-Timeout) für weniger *unnötige* Nachrichten
    - Verringerung der Anfragelast durch Senden von einzelnen Anfragen bzw. `sleep()`-Aufrufe zwischen Anfragen

Aber Achtung!

- Gut zum Lokalisieren und Beheben von bereits bekannten Bugs geeignet, **aber**
- Einige Fehler (insbesondere Nebenläufigkeitsprobleme) treten nur unter hoher Last auf!

→ Ausgiebiges Testen immer zusätzlich unter **Lastsituation**
(insbesondere Randfälle wie z.B. Replikatausfälle!)