Verteilte Systeme – Übung

Replikation

Sommersemester 2026

Paul Bergmann, Christian Berger

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

Replikation

- Grundlagen der Replikation
- Raft

Übungsaufgabe 5

Replikation

Grundlagen der Replikation

Replikation

- Aktive Replikation
  - 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 passive replication“ vs. „Cold passive replication“
  - 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 vor 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 Inkonsistenzen zweier Replikate R_0 und R_1
    - `incrementAndGet()`-Anfragen A_1 und A_2 von verschiedenen Nutzern
    - Annahme: A_1 erreicht R_0 früher als A_2, bei R_1 ist es umgekehrt

R_0	Zähler-Speicher
$< init >$	0
A_1	1
A_2	2

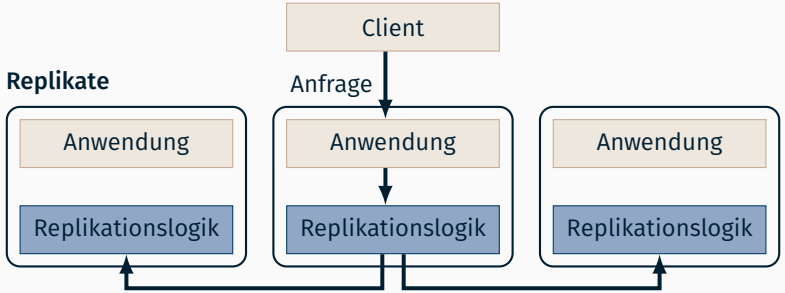
R_1	Zähler-Speicher
$< init >$	0
A_2	1
A_1	2

→ Inkonsistente Antworten!

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

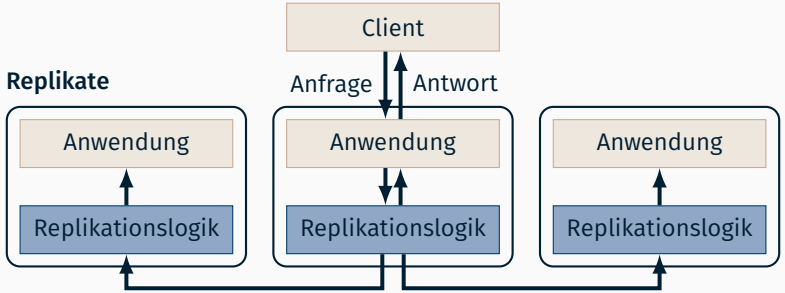
Aktive Replikation von Diensten

- Weg der Anfrage
  - Senden der Anfrage an das Anführer-/Kontaktreplikat
  - Verteilen der Anfrage (z. B. durch ein Replikationsprotokoll)



Aktive Replikation von Diensten

- Weg der Antwort
  - Kontaktreplikat: Rückgabe der Antwort
  - Bearbeitung der Anfrage auf allen Replikaten
  - Alle anderen Replikate: Speichern/Verwerfen der Antwort, abhängig von der Semantik



Replikation

Raft

Raft-Protokoll

- Aktive Replikation einer Anwendung
  - Einigung auf Ausführungsreihenfolge für alle Replikate
  - Zuverlässige, stark konsistente Replikation der entsprechenden Log-Einträge
  - Benötigt $2f + 1$ Replikate, bei bis zu f Ausfällen
- Starker Anführer
  - Im Normalfall
    - Anführer erstellt Log-Einträge anhand von Client-Anfragen
    - Anführer verteilt Log-Einträge per `appendEntries()`-Fernaufruf
    - Anführer gibt replizierte Log-Einträge zur Ausführung frei
    - Anführer beantwortet Anfragen
  - Im Fehlerfall
    - Kandidaten versuchen per `requestVote()` gewählt zu werden
    - Fehlerhafter Anführer muss ersetzt werden, bevor neue Anfragen verarbeitet werden können
- Im Folgenden werden nur ausgewählte Aspekte von Raft betrachtet

D. Ongaro and J. Ousterhout

In Search of an Understandable Consensus Algorithm

Proceedings of the USENIX Annual Technical Conference (USENIX ATC'14), p. 305–319, 2014

Anführer(ab)wahl

- Fragestellung: Wie setzt man einen Anführer ab?
  - Alter Anführer soll nach dem Absetzen keinen Einfluss mehr haben
  - Keine gemeinsame Zeitbasis zwischen Replikaten
    - Replikate können dem alten oder neuen Anführer folgen
    - Anführer könnte noch nicht von eigener Absetzung erfahren haben

- Alle Fernaufrufe und deren Rückgabewerte in Raft enthalten Term
  - Term = „Regentschaft“
  - Neuer Anführer hat höheren Term als alle vorherigen Anführer

  $\rightarrow$ Hochzählen bei jeder Anführerwahl

- Term nutzen, um alte Fernaufrufe auszusortieren
  - Fernaufruf-Empfänger erhält Aufruf mit **neuem** Term
    - *Empfänger wechselt* in neuen Term und wird zum Follower (Anführer ggf. noch unbekannt)
    - Fernaufruf mit dem neuen Term ausführen
  - Fernaufruf-Empfänger erhält Aufruf mit **altem** Term
    - Empfänger lehnt Fernaufruf ab und gibt neuen Term zurück
    - *Absender wechselt* in den neuen Term und wird zum Follower (Anführer ggf. noch unbekannt)

Zeitpunkt der Anführerwahl und Bekanntgabe des Wahlergebnisses

- Randomisiertes Timeout für Anführerwahl
  - Zufälliger Wert zwischen $t_{wahl}/2$ und t_{wahl}, wobei t_{wahl} = Election Timeout
  - Möglichst nur ein Replikat soll auf einmal ins Timeout laufen
  - **Wichtig:** Timeout muss nach jeder Anführerwahl neu gewürfelt werden
- Anführerwahl-Timeout löst immer wieder aus, solange kein Anführer dies verhindert
  - Timeout zurücksetzen durch `requestVote()` bzw. Heartbeats mittels `appendEntries()`
  - Anführer muss Heartbeats an alle Replikate innerhalb von Timeout senden
  - Abgetrennte Replikate wechseln laufend in höheren Term

  $\rightarrow$ Bei Wiederbeitritt springen alle anderen Replikate in den höheren Term

- Ein Replikat kann Follower werden, ohne zu wissen wer aktuell der Leader ist
  - Beispiel: Replikat erhält `requestVote()`-Fernaufruf für neueren Term
  - Replikat wechselt als Follower in neuen Term
  - Hier gibt es noch **keinen** Anführer
- `appendEntries()` informiert über aktuellen Anführer

Committen von Log-Einträgen

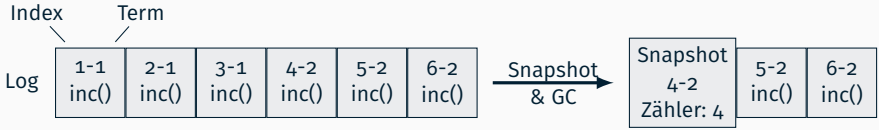
- Relevanter Teil des Replikatzustands

`nextIndex[]`	Index des nächsten an Replikat zu übertragenden Log-Eintrags
`matchIndex[]`	Index des höchsten erfolgreich replizierten Log-Eintrags
	Log des Anführers und Replikat *i* identisch bis inklusive `matchIndex[i]`
`commitIndex`	Index des höchsten zur Ausführung freigegebenen Log-Eintrags
`lastApplied`	Index des höchsten ausgeführten Log-Eintrags

- Replikation von Log-Einträgen entsprechend `nextIndex[]` mittels `appendEntries()`
  - Bei Erfolg: `nextIndex[]` und `matchIndex[]` aktualisieren
  - Anführer muss eigenen Eintrag selbst anpassen
  - Bei Anführerwechsel: `nextIndex[]` auf Log-Ende setzen, `matchIndex[]` auf 0
- Anführer passt `commitIndex` nach Änderungen an `matchIndex[]` an
- Committete Log-Einträge ausführen
  - Bereich zwischen `lastApplied` und `commitIndex`
  - Je nach Implementierung genügt der `commitIndex`

Sicherungspunkte

- Problem: Ausgefallenes / Zurückhängendes Replikat aktualisieren
  - Replikat muss fehlende Log-Einträge erhalten und verarbeiten
  - → **Hoher Aufwand**: Schlimmstenfalls notwendig alle Log-Einträge seit Systemstart zu übertragen
  - → **Hoher Speicherverbrauch**: Log wird beliebig groß
- Sicherungspunkt
  - Enthält Kopie des Anwendungszustands nach Ausführen eines Log-Eintrags
  - Enthält Log-Index des zuletzt verarbeiteten Log-Eintrags
  - → Zusammenfassung aller vom Sicherungspunkt abgedeckten Log-Einträge



- Zustand im Sicherungspunkt entspricht exakt dem Zustand nach Ausführen aller Log-Einträge bis zum Sicherungspunkt
  - **Begrenzter Aufwand**: Replikate können mit Sicherungspunkt aktualisiert werden und weite Teile des Logs überspringen
  - **Begrenzter Speicherverbrauch**: Frühere Log-Einträge können gelöscht werden

Snapshot-Erstellung und -Übertragung in Raft

- Analog zu erweiterter Version des Raft-Papiers, siehe `/proj/i4vs/pub/aufgabe5`
- Snapshot-Erzeugung
  - Jedes Replikat erstellt Snapshot wenn Log groß genug (z.B. nach jeweils 10 verarbeiteten Anfragen)
  - Snapshot enthält Anwendungszustand, Log-Index und -Term
  - Alle früheren Log-Einträge und Snapshots löschen
- Snapshot-Übertragung
  - Versand an zurückhängendes Replikat per `installSnapshot`-Fernaufruf

```
int installSnapshot(int term, int leaderId,
        long lastIncludedIndex, int lastIncludedTerm, Serializable data)
```

`term`, `leaderId`	aktueller Term und Anführer
`lastIncluded{Index,Term}`	neuster im Snapshot enthaltener Log-Eintrag
`data`	Anwendungszustand im Snapshot
Rückgabewert	neuster dem Empfänger bekannter Term

↪ In der Übung: Anders als im Papier soll der Anwendungszustand **auf einmal** übertragen werden

- Ablauf
  - Leader überträgt Snapshot, wenn ein bereits gelöschter Log-Eintrag benötigt würde
  - Empfänger speichert Snapshot und spielt diesen in Anwendung ein

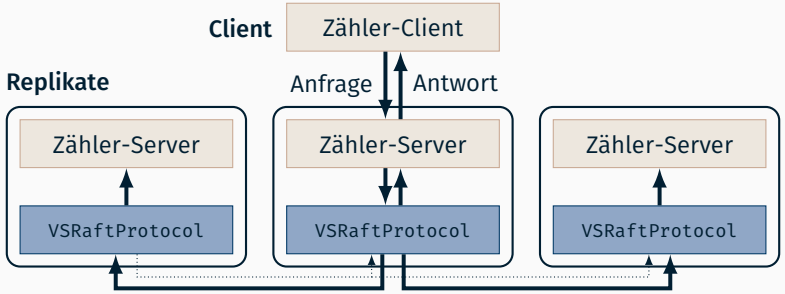
→ Snapshot speichern für den Fall, dass Empfänger zum Anführer wird

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



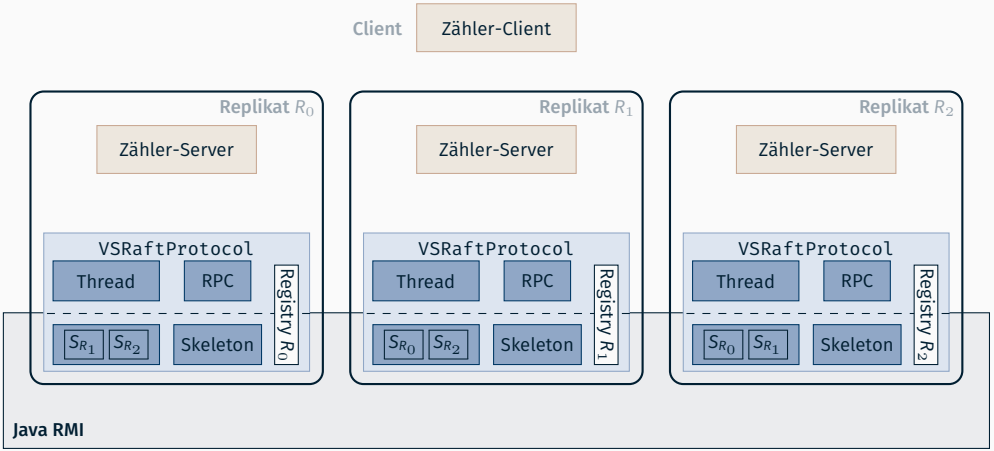
Schnittstelle zwischen Anwendung und Raft-Protokoll

```
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

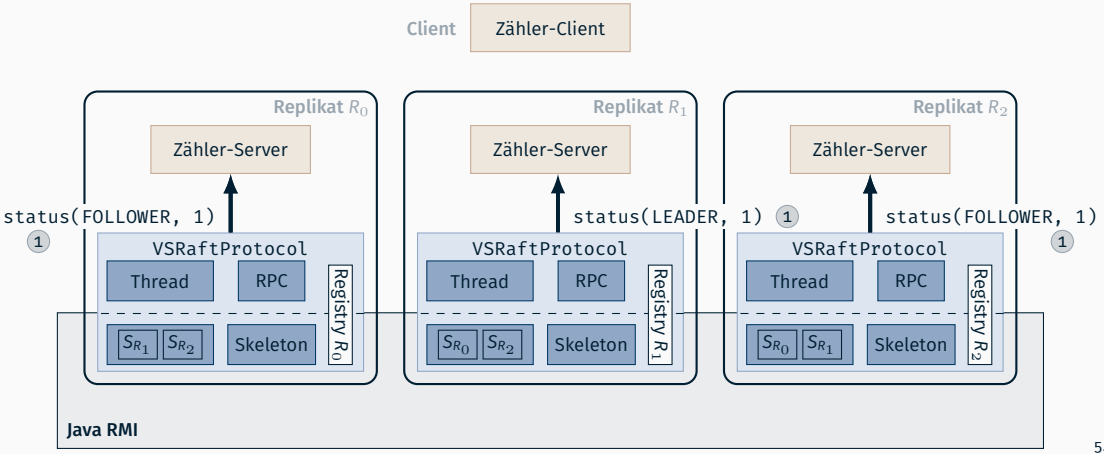
Anfragereplikation: Gesamtüberblick

- 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



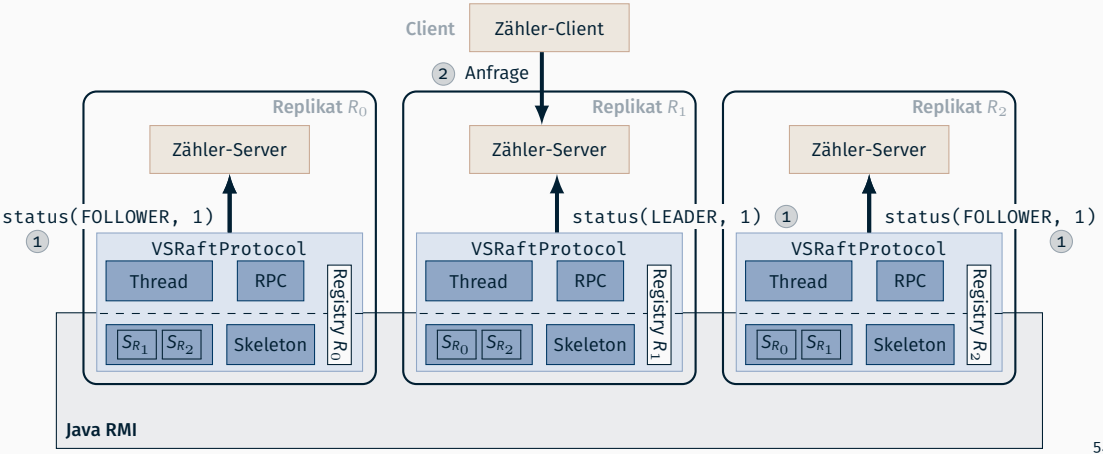
Anfragereplikation: Gesamtüberblick

- 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



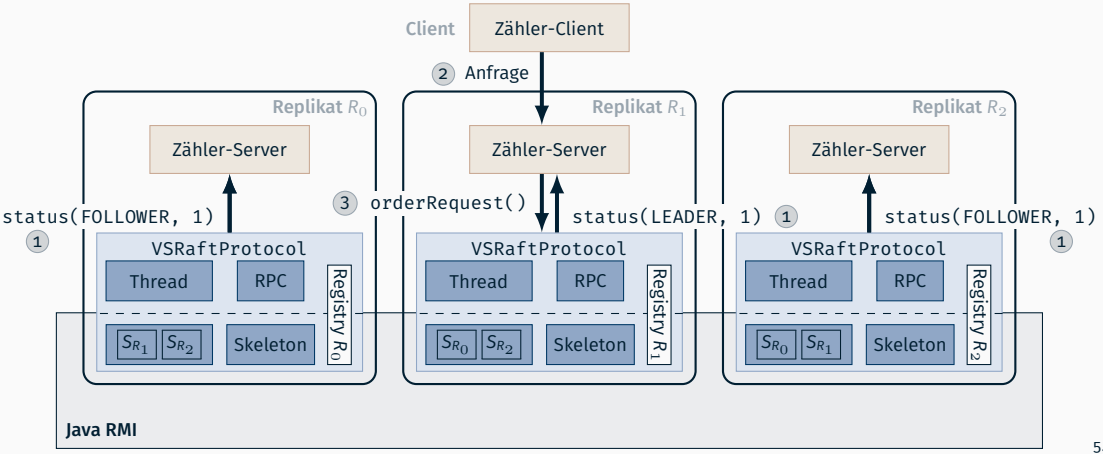
Anfragereplikation: Gesamtüberblick

- 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



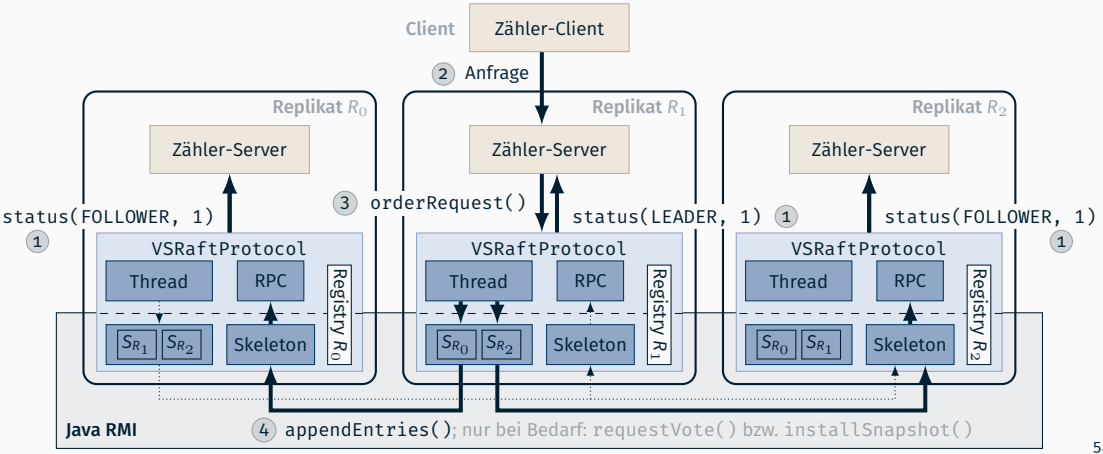
Anfragereplikation: Gesamtüberblick

- 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



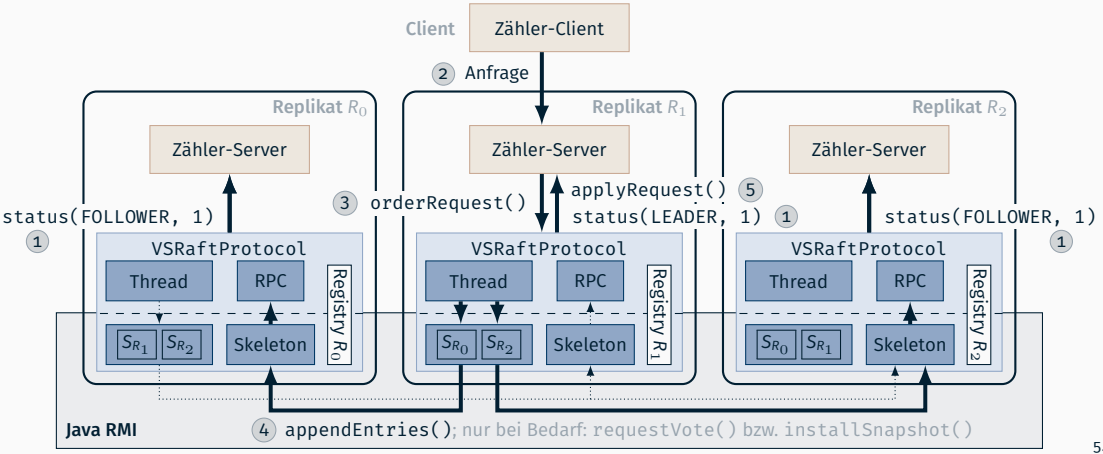
Anfragereplikation: Gesamtüberblick

- 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



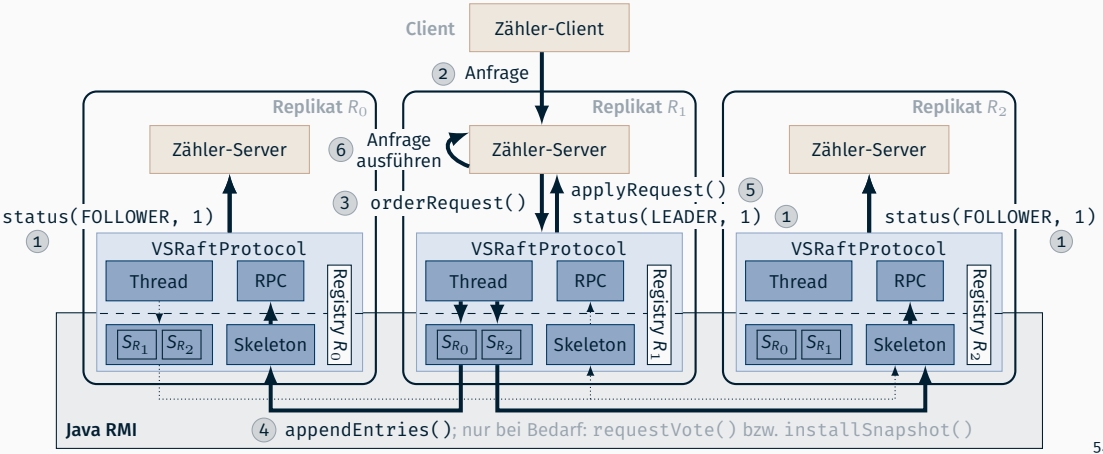
Anfragereplikation: Gesamtüberblick

- 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



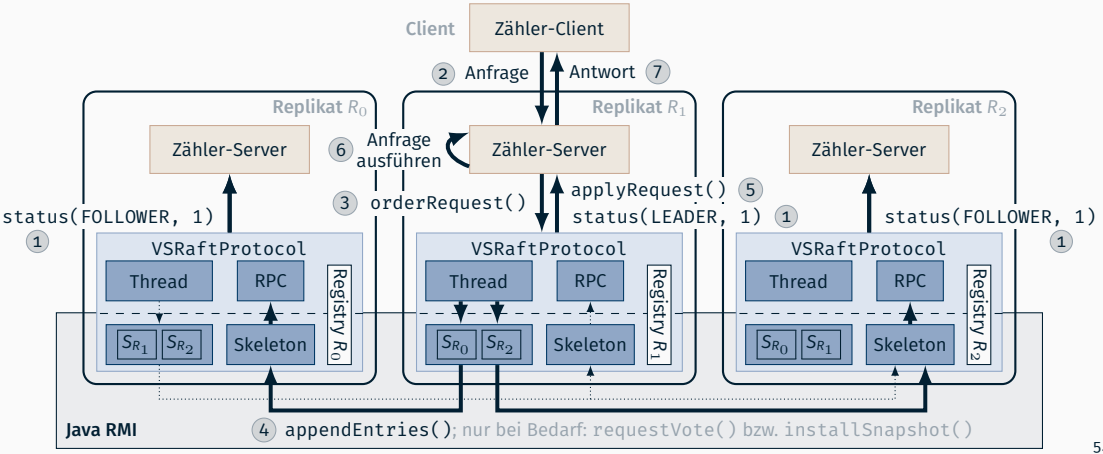
Anfragereplikation: Gesamtüberblick

- 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



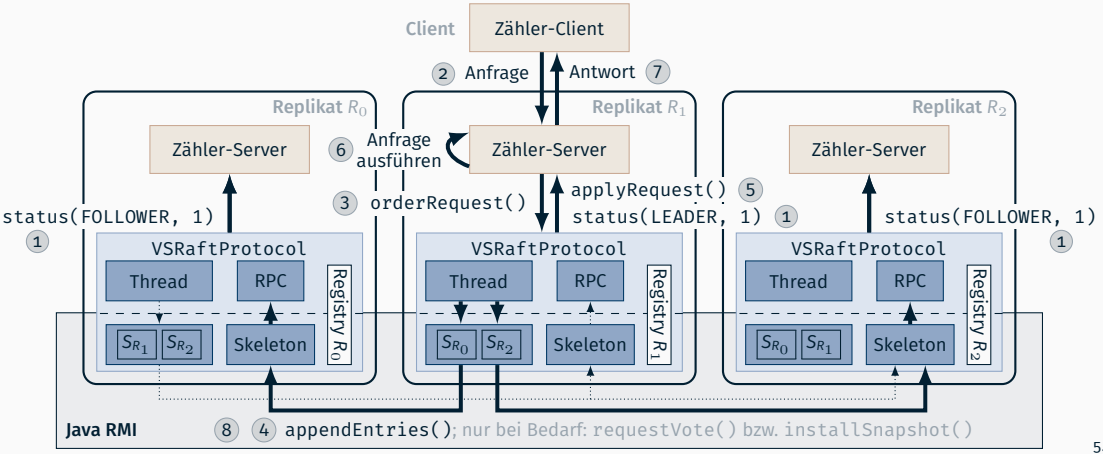
Anfragereplikation: Gesamtüberblick

- 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



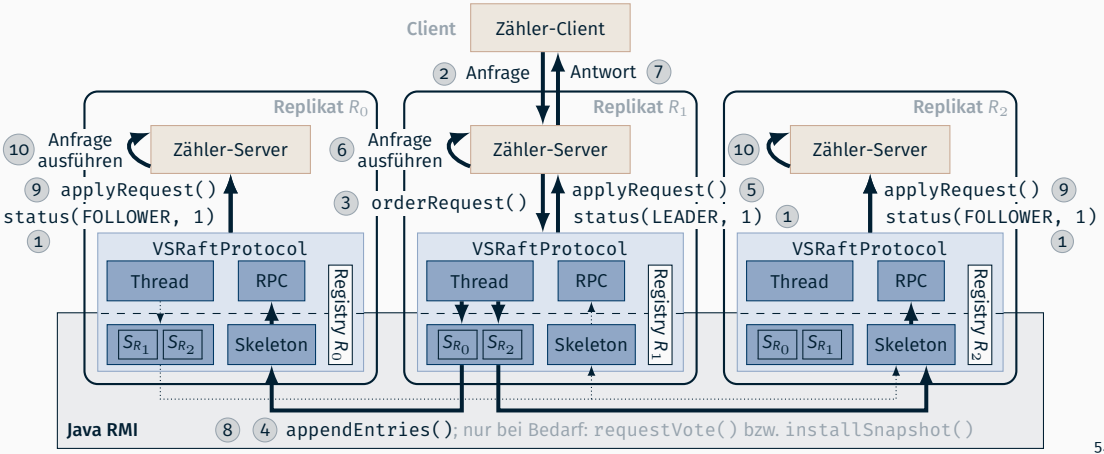
Anfragereplikation: Gesamtüberblick

- 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();
}
```

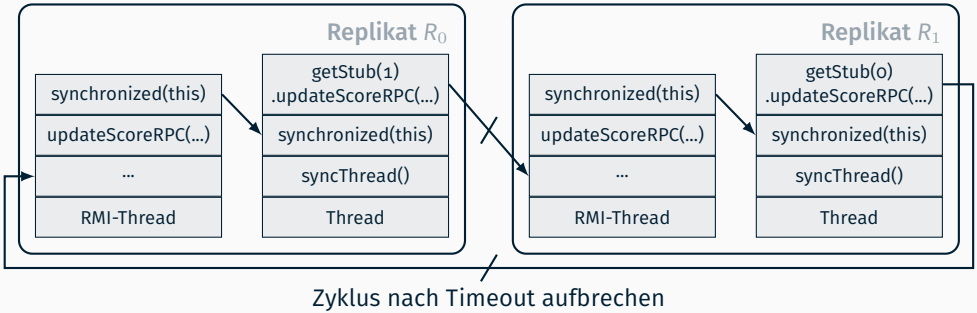
RPC erfolgt innerhalb des `synchronized`-Blocks

⚠ 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!)