

# Verteilte Systeme – Übung

## Replikation: Raft

---

Sommersemester 2022

Laura Lawniczak, Tobias Distler

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

[www4.cs.fau.de](http://www4.cs.fau.de)



Lehrstuhl für Verteilte Systeme  
und Betriebssysteme



FRIEDRICH-ALEXANDER  
UNIVERSITÄT  
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

## Replikation

Grundlagen der Replikation

Raft

# Replikation

---

## Grundlagen der 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

## ■ 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
< <i>init</i> >	0
$A_1$	1
$A_2$	2

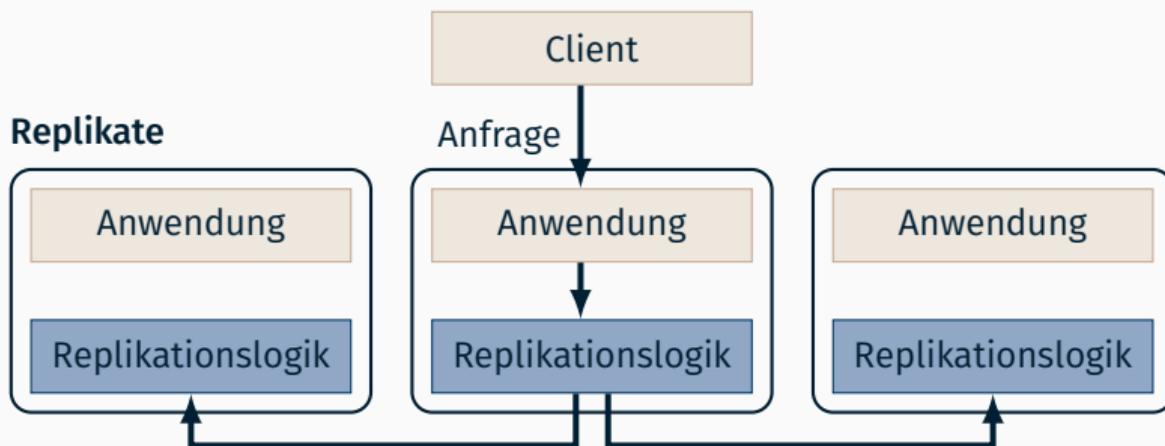
$R_1$	Zähler-Speicher
< <i>init</i> >	0
$A_2$	1
$A_1$	2

→ **Inkonsistente Antworten!**

## ■ Sicherstellung der Replikatkonsistenz

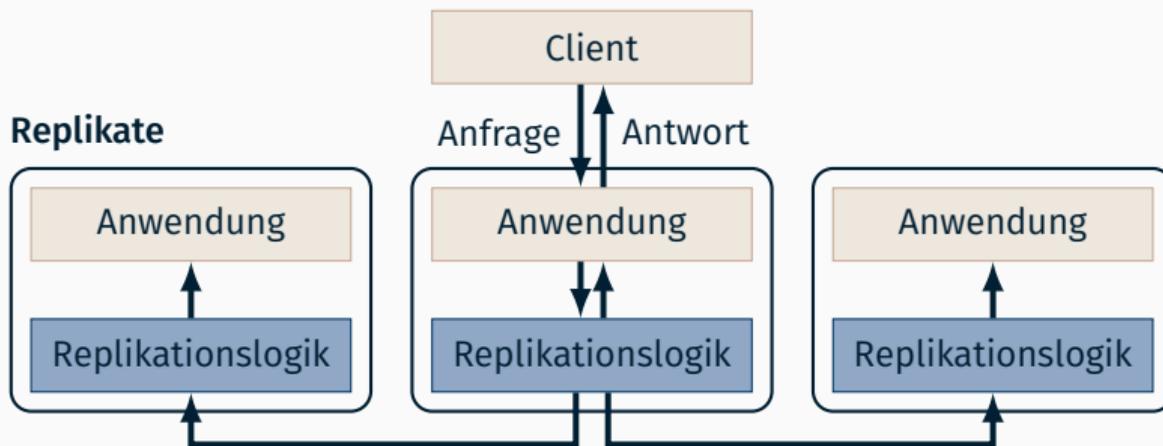
- Alle Replikate müssen Anfragen in derselben Reihenfolge bearbeiten
- Protokoll/Dienst zur Erstellung einer Anfragenreihenfolge nötig

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



## ■ Weg der Antwort

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



## Replikation

---

Raft

- 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

- 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→ 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)

- 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

→ 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

- 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()`

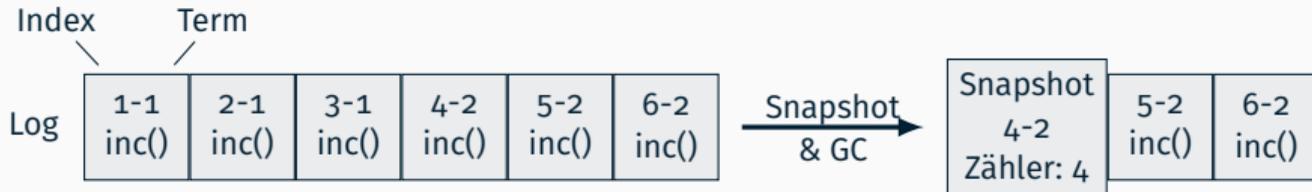
  - Bei Erfolg: `nextIndex[]` und `matchIndex[]` aktualisieren
  - Anführer muss eigenen Eintrag selbst anpassen
  - Bei Anführerwechsel: `nextIndex[]` auf Log-Ende setzen, `matchIndex[]` auf 0

- Anführer passt `commitIndex` nach Änderungen an `matchIndex[]` an

- Committede Log-Einträge ausführen

  - Bereich zwischen `lastApplied` und `commitIndex`
  - Je nach Implementierung genügt der `commitIndex`

- 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