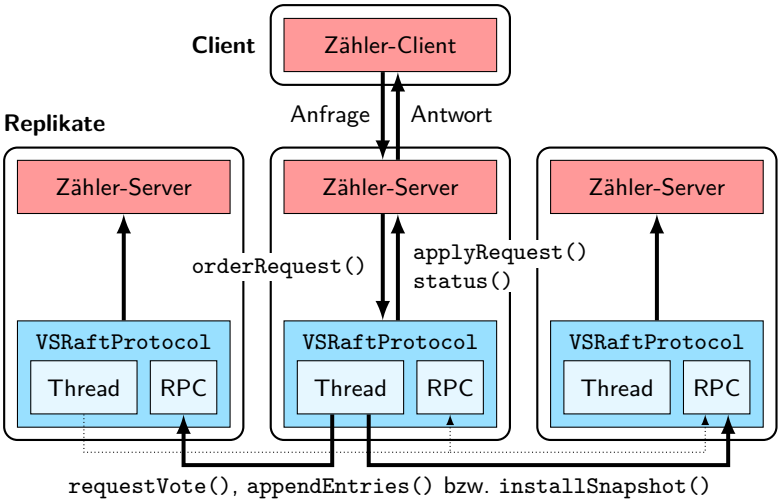
Übungsaufgabe #5: Replikation

Im Rahmen dieser Übungsaufgabe soll ein einfacher Zählerdienst mithilfe des Replikationsprotokolls Raft aktiv repliziert werden. Das Replikationskonzept sieht vor, dass alle Replikate alle Anfragen in derselben Reihenfolge bearbeiten und so ihren Zustand konsistent halten. Im Folgenden werden zunächst einige Aspekte des Gesamtsystems näher beschrieben. Anschließend legt Teilaufgabe 5.1 die Grundlagen für die Kommunikation der Replikate untereinander. Darauf aufbauend realisieren Teilaufgabe 5.2 die Anführerwahl und Teilaufgabe 5.3 die Replikation und Ausführung von Client-Anfragen. In der erweiterten Übung werden schließlich in Teilaufgabe 5.4 Sicherungspunkte dazu genutzt, den Zustand von ausgefallenen Replikaten nach dem Neustart zu aktualisieren.

Überblick Wie in der Abbildung [Folie 5.2 : 3] illustriert setzt sich das Gesamtsystem aus mehreren Komponenten zusammen, von denen ein Großteil jedoch bereits vollständig unter `/proj/i4vs/pub/aufgabe5` bereitgestellt wird. Dies gilt insbesondere für die eigentliche Anwendung, also die Client- und Server-Seite des Zählerdiensts. Dementsprechend liegt der Fokus der Übungsaufgabe auf der Implementierung des Raft-Protokolls in der Klasse `VSRaftProtocol`.

Zählerdienst Der Zähler-Client bietet Nutzenden die Möglichkeit den Wert eines vom Dienst verwalteten Zählers über den Befehl `inc` per Kommandozeile zu inkrementieren. Wird ein solcher Befehl abgesetzt, so schickt der Zähler-Client eine entsprechende Anfrage an eines der Zählerdienstreplikate auf Server-Seite, das sich daraufhin um die Replikation der Anfrage mittels Raft kümmert. Dasselbe gilt für die anschließende Ausführung der Anfrage und das Zurücksenden des neuen Zählerstands in einer Antwortnachricht an den Client.



Replikation Das Raft-Protokoll selbst kommuniziert also nicht direkt mit dem Client, sondern repliziert nur die Anfragen, die es von seinem lokalen Zähler-Server zur Replikation übergeben bekommt; hierzu verwendet der Server die Methode `orderRequest()`. Nach abgeschlossener Replikation, also sobald ein Raft-Knoten die Gewissheit besitzt, dass mehr als die Hälfte aller Replikate eine Anfrage ihrem Log hinzugefügt haben, steht die Anfrage bereit zur Ausführung. Jeder Raft-Knoten signalisiert dies seinem lokalen Zähler-Server, indem er für die betroffenen Anfragen die Methode `applyRequest()` in aufsteigender Reihenfolge der Indizes ihrer Log-Einträge aufruft.

Kontaktreplikat Da es in Raft nur dem amtierenden Anführer-Knoten erlaubt ist Vorschläge für neue Log-Einträge zu unterbreiten, ist es zwingend erforderlich, dass Zähler-Clients ihre Anfragen an genau den Server stellen, auf dem aktuell der Raft-Anführer residiert. Der Mechanismus um diesen zu finden ist bereits implementiert, allerdings erfordert er von Zähler-Servern Kenntnis über den aktuellen Anführerstandort. Hierfür verfügt jeder Zähler-Server über eine Methode `status()` mit deren Hilfe sein lokaler Raft-Knoten ihn über alle Änderungen bei der Rollenverteilung sowie beobachtete Wechsel des Anführerstandorts unmittelbar informiert.

Raft-Implementierung Die Umsetzung des Raft-Protokolls orientiert sich eng an der im Raft-Papier präsentierten Beschreibung und insbesondere der dortigen Übersicht in Abbildung 2. Um die eigene Implementierung eines Raft-Knotens möglichst übersichtlich gestalten zu können, soll diese selbst nur über einen einzelnen aktiven Thread verfügen, der in einer Endlosschleife sämtliche anfallenden Aufgaben sequentiell abarbeitet. Hierzu gehört insbesondere die Interaktion mit den anderen Raft-Knoten im System über die Fernaufrufe `requestVote()`, `appendEntries()` und `installSnapshot()`. Welche Aktionen konkret durchzuführen sind hängt dabei von der aktuellen Rolle des Raft-Knotens ab, also davon ob er gerade Follower, Candidate oder Leader ist. Hat der Protokoll-Thread alle vorliegenden Aufgaben erledigt, so wartet er blockierend darauf, dass neue anfallen.

Nebenläufigkeit Auch wenn jeder Raft-Knoten nur einen eigenen Thread aufweist, so führt die gegenseitige Interaktion über Fernaufrufe trotzdem zu möglichen nebenläufigen Zugriffen auf den Protokollzustand. Da es erfahrungsgemäß alles andere als trivial ist, den benötigten Schutz über feingranulare Synchronisationsmechanismen zu realisieren, ist es im Rahmen dieser Übungsaufgabe ratsam, sämtliche betroffenen Stellen (also unter anderem den Protokoll-Thread und die Rümpfe der oben erwähnten Fernaufrufmethoden) in einem einzelnen kritischen Abschnitt zusammenzufassen [Folien 5.2 : 4–7]. Fernaufrufe anderer Raft-Knoten werden somit nur dann beantwortet wenn der lokale Protokoll-Thread selbst gerade wartet und währenddessen den kritischen Abschnitt bedenkenlos freigeben kann. Um in dieser Konstellation dauerhafte Verklemmungen zu vermeiden ist in den Vorgaben Java RMI per Socket-Timeout so konfiguriert, dass es Fernaufrufe vorzeitig abbricht falls diese (z. B. aufgrund einer Blockade) zu lange dauern. In der Folge kann es daher dazu kommen, dass einzelne Fernaufrufe scheitern, obwohl sowohl das Netzwerk als auch der kontaktierte Raft-Knoten ordnungsgemäß funktionieren.

5.1 Implementierung der Replikatkommunikation (für alle)

In dieser und den folgenden Teilaufgaben soll nach und nach das Replikationsprotokoll Raft in der Klasse `VSRaftProtocol` realisiert werden. Zur besseren Übersicht wird dabei empfohlen bei der Benennung von Variablen und Parametern auf dieselben Begriffe zurückzugreifen, die auch Abbildung 2 des Raft-Papiers verwendet.

Ziel der ersten Teilaufgabe ist es, zunächst die Grundlage dafür zu legen, dass Replikate untereinander mithilfe von Java-RMI-Fernaufrufen kommunizieren können. Die Klasse `VSRaftProtocol` bietet hierfür bereits die Remote-Schnittstelle `VSRaftProtocolService` an, welche die drei von Raft definierten Fernaufrufmethoden `requestVote()`, `appendEntries()` und `installSnapshot()` bündelt. Die eigentliche Implementierung dieser Methoden erfolgt in späteren Teilaufgaben, zuvor ist in dieser Teilaufgabe dafür zu sorgen, dass einem Replikat ein Stub eines anderen Replikats zur Verfügung steht, sobald es mit diesem per Fernaufruf interagieren möchte.

```
public class VSRaftProtocol implements VSRaftProtocolService {
    public void init(VSCounterServer application);
    private VSRaftProtocolService getStub(int replicaId);
    private void discardStub(int replicaId);
    private void testConnection();
}
```

Die Methode `init()` wird beim Start des Replikats vom Zählerdienst aufgerufen, um seine lokale `VSRaftProtocol`-Instanz zu initialisieren. Hierbei soll das Protokollobjekt sich selbst als RMI-Remote-Object für Fernaufrufe verfügbar machen und in einer lokalen Registry bereitstellen. Die Hilfsmethode `getStub()` wird dazu genutzt einen Fernaufruf-Stub für ein bestimmtes Replikat zu erhalten. Aus Effizienzgründen soll `getStub()` nicht bei jedem Aufruf einen neuen Stub erzeugen, sondern existierende Stubs nach Möglichkeit cachen. Mithilfe der Methode `discardStub` lässt sich ein solcher Cache-Eintrag löschen, etwa wenn bei der Kommunikation mit einem anderen Replikat ein Fehler aufgetreten ist (z. B. aufgrund eines temporären Replikatausfalls) und daher beim nächsten Versuch ein neuer Stub angelegt werden soll. Zum Testen der Replikatkommunikation kann zum Abschluss der Protokollinitialisierung die bereits vorhandene Methode `testConnection()` zum Einsatz kommen.

Der Start eines Replikats erfolgt beispielsweise durch folgenden Befehl auf der Kommandozeile:

```
java -cp <classpath> vsue.raft.VSCounterReplica <replica-id> <path-to-config-file>
```

Neben dem Classpath und einer eindeutigen ID ist dem Replikat hierbei der Pfad zu einer Konfigurationsdatei mit allen Replikatadressen zu übergeben [Folie 5.2 : 8]. Selbiger ist auch beim Client-Start erforderlich:

```
java -cp <classpath> vsue.raft.VSCounterClient <path-to-config-file>
```

Aufgaben:

→ Bereitstellen des Protokollobjekts per replikatslokaler Registry in der `init()`-Methode
→ Verwaltung von Replikat-Stubs mittels `getStub()` und `discardStub()`
→ Testen der Replikatkommunikation durch Aufruf von `testConnection()` mit drei Replikaten

Hinweise:

- Die unter `/proj/i4vs/pub/aufgabe5` bereitgestellten Klassen dürfen falls erforderlich beliebig erweitert werden.
- Stubs sollten nur bei Bedarf und nach Kommunikationsproblemen neu abgerufen werden.
- Client(s) und alle Replikate müssen exakt dieselbe Konfigurationsdatei mit Replikatadressen verwenden.

5.2 Anführerwahl (für alle)

Einer der für die anschließende Replikation grundlegenden Mechanismen im Raft-Protokoll ist die Wahl eines Anführerreplikats, bei der ein Follower versucht selbst Leader zu werden, falls er für einen längeren Zeitraum nichts mehr vom bisherigen Anführer gehört hat. Hierfür wechselt ein Replikat in den nächsthöheren Term und wird selbst zum Candidate. Die eigentliche Wahl erfolgt anschließend mittels `requestVote()`-Fernaufrufen.

```
public VSRaftRPCResult requestVote(int term, int candidateId, long lastLogIndex,
                                   int lastLogTerm) throws RemoteException;
```

Die Parameter der `requestVote()`-Methode und ihre Semantik sind identisch mit denen aus der Protokollbeschreibung im Raft-Papier. Als Rückgabewert dient ein Objekt der Hilfsklasse `VSRaftRPCResult`, in dem wie von Raft gefordert der aktuelle Term sowie ein Boolean-Flag zur Erfolgsbestätigung zusammengefasst sind.

Sobald ein Replikat nach Ablauf des Anführerwahl-Timeouts zum Candidate wird, besteht sein Ziel darin, Stimmen von einer Replikatsmehrheit zu sammeln. Hierzu stimmt das Replikat zunächst für sich selbst und ruft anschließend in seinem Protokoll-Thread sequentiell die `requestVote()`-Methoden der anderen Replikate auf. Um die Implementierung zu vereinfachen, sollten diese Fernaufrufe abgesetzt werden, während sich der Protokoll-Thread in dem kritischen Abschnitt befindet, der den Replikatzustand schützt [Folien 5.2 : 4–7].

Während der Ausführung von `requestVote()` prüft der Empfänger des Fernaufrufs, ob er (1) bereits in einem höheren Term ist als vom CANDIDATE angefragt oder (2) zwar denselben Term aufweist, allerdings schon für ein anderes Replikat gestimmt hat. Sollte eines dieser Szenarien zutreffen, so reagiert das angefragte Replikat mit einer Ablehnung, anderenfalls gibt es dem CANDIDATE seine Stimme. Ein CANDIDATE wird zum LEADER, sobald er Stimmen von einer Mehrheit der Replikate gesammelt hat, inklusive seiner eigenen. Wird die Mehrheit der Stimmen dagegen verfehlt, so muss nach Ablauf eines erneuten Timeouts im nächsten Term eine neue Wahl erfolgen.

Im Zuge einer Anführerwahl kann es vorkommen, dass ein Replikat von einem höheren Term erfährt. Dies betrifft sowohl das aufgerufene Replikat (durch den Parameter `term` von `requestVote()`) als auch das aufrufende Replikat (durch die Term-Information im Rückgabewert `VSRaftRPCResult`). In beiden Fällen passt ein betroffenes Replikat daraufhin seinen eigenen Term entsprechend an und wird sofort zum FOLLOWER.

```
public class VSCounterServer {
    public void status(VSRaftRole role, int leaderId);
}
```

Die aktuelle Rolle eines Replikats wird mithilfe einer Enumeration `VSRaftRole` mit den Werten FOLLOWER, CANDIDATE bzw. LEADER verwaltet. Jede Änderung an der eigenen Rolle ist dem lokalen Zähler-Server durch die Methode `status()` mitzuteilen. Dasselbe gilt, falls ein Replikat Informationen über einen neuen Anführer erhält.

Aufgaben:

→ Implementieren der Anführerwahl auf Basis von `requestVote()`-Fernaufrufen
→ Informieren der Anwendung per `status()` über die eigene Rolle und den aktuellen LEADER

Hinweise:

- Wie im Raft-Papier beschrieben ist das Anführerwahl-Timeout jeweils zufällig zu bestimmen. [Folie 5.1 : 7]
- Funktionalitäten zur Log-Replikation werden erst in der nächsten Teilaufgabe relevant.
- Da ein neu gewählter LEADER seinen Pflichten als Anführer bisher nicht nachkommt, besteht das erwartete Systemverhalten am Ende dieser Teilaufgabe darin, dass auf eine Anführerwahl nach Ablauf eines Timeouts sofort die nächste folgt, unabhängig davon ob die vorherige Wahl erfolgreich war oder nicht.

5.3 Replikation von Anfragen (für alle)

Die eigentliche Aufgabe des Raft-Protokolls, nämlich die konsistente und fehlertolerante Replikation von Anfragen, erfolgt mithilfe des `appendEntries()`-Fernaufrufs. Dieser dient zudem als Heartbeat des LEADER, der bei funktionierender Replikatkommunikation den Start neuer Anführerwahlen durch andere Replikate verhindert.

```
public VSRaftRPCResult appendEntries(int term, int leaderId, long prevLogIndex,
                                     int prevLogTerm, VSRaftLogEntry[] entries,
                                     long leaderCommitIndex) throws RemoteException;
```

Der Protokoll-Thread des LEADER realisiert den Heartbeat, indem er per `appendEntries()`-Fernaufruf periodisch Informationen über seinen aktuellen Zustand der Reihe nach an alle anderen Replikate verteilt, und zwar auch wenn zwischenzeitlich keine neuen Log-Einträge hinzugekommen sind. Auf Empfänger-Seite werden die Heartbeats unter anderem dazu genutzt, den eigenen Wissensstand über den gegenwärtige Term und die Identität des Anführers zu aktualisieren, und bei Bedarf per `status()` die Anwendung über Änderungen zu informieren (siehe Teilaufgabe 5.2). Zusätzlich setzen FOLLOWER bei jedem Heartbeat ihr Anführerwahl-Timeout zurück.

```
public class VSRaftProtocol {
    public boolean orderRequest(Serializable request);
}
```

Neu eingetroffene Anfragen werden dem Replikationsprotokoll vom lokalen Zähler-Server durch einzelne Aufrufe der Methode `orderRequest()` übergeben. Nicht-LEADER-Replikate müssen solche Aufrufe durch Rückgabe von `false` zurückweisen ohne weitere Aktionen durchzuführen. Der LEADER reagiert dagegen auf einen `orderRequest()`-Aufruf indem er die übergebene Anfrage ans Ende seines Logs anfügt, die anschließende Replikation durch Aufwecken seines Protokoll-Thread anstößt, und den Erhalt der Anfrage mit `true` bestätigt.

Liegen neue Log-Einträge vor, so repliziert der Protokoll-Thread des LEADER diese per `appendEntries()`-Fernaufruf auf seine FOLLOWER. Der Parameter `entries` dient hierbei zum Versand des Log-Teils, der dem jeweiligen Empfängerreplikat noch fehlt. Auf Basis der Parameter `prevLogIndex` und `prevLogTerm` kann ein Empfänger prüfen, ob er den aktuellen Stand früherer Log-Einträge besitzt. Falls der Empfänger den genannten Log-Eintrag noch nicht hat, lehnt er den `appendEntries()`-Aufruf ab. Der Anführer versucht die Log-Replikation für dieses Replikat daraufhin erneut, nun beginnend bei dem Log-Eintrag mit dem nächstkleineren Index.

Jede erfolgreiche Log-Replikation führt auf dem LEADER zur Aktualisierung seines Commit-Index. Dieser repräsentiert den Log-Index, bis zu dem der Anführer sein Log auf eine Mehrheit von Replikaten repliziert hat. FOLLOWER erfahren den aktuellen Commit-Index über den `appendEntries()`-Parameter `leaderCommitIndex`.

```
public class VSCounterServer {
    public void applyRequest(VSRaftLogEntry entry);
}
```

Anhand des Commit-Index kann jedes Replikat entscheiden, welche Log-Einträge erfolgreich repliziert wurden und daher an die Anwendung ausgeliefert werden dürfen. Der Zähler-Server bietet hierfür die Methode `applyRequest()`, die für alle bestätigten Log-Einträge in aufsteigender Reihenfolge ihrer Indizes aufzurufen ist.

Aufgaben:
- → Implementierung von Leader-Heartbeats durch periodische `appendEntries()`-Aufrufe
- → Entgegennehmen von Anfragen aus der Anwendung in der Methode `orderRequest()`
- → Replizieren des Leader-Logs auf die Follower mithilfe von `appendEntries()`-Aufrufen
- → Testen der Implementierung mit drei Replikaten auf verschiedenen Rechnern
- → Testen, dass der Ausfall eines Replikats toleriert wird

Hinweise:
- Für die Verwaltung der Anfragen kann auf die bereits vorhandene Log-Implementierung `VSRaftLog` zurückgegriffen werden [Folie 5.2 : 9]. Diese beginnt ihr Log zwecks Vermeidung von Sonderfällen immer mit einem leeren Platzhaltereintrag. Zum Aktualisieren des eigenen Logs bietet sich die Methode `storeEntries()` an, die nicht nur die neuen Einträge hinzufügt, sondern bei Bedarf auch bereits vorhandene Einträge überschreibt. Ein explizites Löschen konfligierender Log-Einträge durch das Protokoll ist somit nicht erforderlich.
- Der Zustand eines Replikats soll ausschließlich im Arbeitsspeicher gehalten werden.
- Zur Wahrung der Korrektheit nach einem Anführerwechsel, muss die Behandlung von `requestVote()`-Aufrufen die in Abschnitt 5.4.1 des Raft-Papers beschriebene Anpassung umsetzen: Ein Candidate erhält nur dann eine Stimme, wenn sein Log mindestens genau so aktuell ist wie das des angefragten Replikats.

5.4 Sicherungspunkte (optional für 5,0 ECTS)

Jedes Replikat in Raft erstellt periodisch (= nach einer statisch festgelegten Anzahl von ausgeführten Anfragen) Sicherungspunkte seines aktuellen Anwendungszustands. Diese Sicherungspunkte haben vor allem zwei Funktionen: Zum einen erlauben sie es einem Replikat sein Log regelmäßig zu kürzen und damit zu verhindern, dass ihm der Arbeitsspeicher ausgeht. Zum anderen lassen sich neu gestartete Replikate (beispielsweise nach einem Ausfall) mithilfe von Sicherungspunkten auf einen Stand bringen, der es ihnen ermöglicht (wieder) am regulären Replikationprozess für Log-Einträge teilzunehmen. Die Übertragung eines Sicherungspunkts vom Leader auf ein anderes Replikat erfolgt mittels `installSnapshot()`-Fernaufruf.

```
public int installSnapshot(int term, int leaderId, long lastIncludedIndex,
           int lastIncludedTerm, Serializable snapshotData) throws RemoteException;
```

Eine detaillierte Beschreibung der Funktionsweise und Parameter der `installSnapshot()`-Methode findet sich in Abbildung 13 der erweiterten Version des Raft-Papiers, das unter `/proj/i4vs/pub/aufgabe5` verfügbar ist.

```
public class VSCounterServer {
    public Serializable createSnapshot();
    public void applySnapshot(Serializable snapshot);
}
```

Zur Erstellung eines Sicherungspunkts bietet der Zähler-Server die Methode `createSnapshot()` an. Diese liefert ein vollständiges Abbild des Anwendungszustands in Form eines serialisierbaren Objekts zurück, dessen genauer Inhalt für das Replikationsprotokoll nicht relevant ist. Sobald ein neuer Sicherungspunkt vorliegt, kann das Protokoll durch Aufruf der `VSRaftLog`-Methode `collectGarbage()` alle vorherigen Einträge aus dem Log entfernen.

Die Übertragung eines Sicherungspunkts per `installSnapshot()` erfolgt, sobald der Leader bei der Log-Replikation feststellt, dass das Log eines Follower-Replikats veraltet ist und eigentlich Einträge benötigt, die der Leader bereits verworfen hat. Nach dem Empfang des Sicherungspunkts, spielt der Follower ihn mithilfe der vom Zähler-Server bereitgestellten Methode `applySnapshot()` in die Anwendung ein.

Aufgaben:
- → Periodisches Erstellen von Sicherungspunkten und deren Übertragung bei Bedarf
- → Testen der Implementierung durch wiederholtes manuelles Beenden und Starten von Replikaten

Hinweise:
- Zum Testen sollten häufig neue Sicherungspunkte erstellt werden, beispielsweise nach jeder 10. Anfrage.
- Der `VSCounterClient` erlaubt es per `bench`-Befehl in kurzer Zeit viele Anfragen an Replikate zu senden.
- Anders als im Raft-Papier beschrieben sollen Sicherungspunkte nicht auf die Festplatte gespeichert werden.
- Sicherungspunkte sollten in einem Stück versendet werden.

Abgabe: am Mi., 03.07.2024 in der Rechnerübung

Die für diese Übungsaufgabe erstellten Klassen sind in einem Subpackage `vsue.raft` zusammenzufassen.