Verteilte Systeme
Übungsblatt 3

Kilian Bartz (Mknr.: 1538561)

Hinweis

Der zugehörige Code befindet sich im folgenden Github Repository: `https://github.com/kilianbartz/raft`.

Raft - eine Übersicht

Agreement-Protokolle spielen eine zentrale Rolle dabei, monolithische Anwendungen (in meinem Fall eine simple Key-Value Datenbank) in einem verteilten System einsetzbar zu machen. Konkret bieten Agreement-Protokolle wie Raft oder Paxos ein allgemeines Framework, um Befehle als Logeinträge fehlertolerant an die Server innerhalb eines Clusters zu verteilen. Dies ermöglicht es, nahezu beliebige Anwendungen, deren Zustand sich gänzlich durch eine Abfolge von Befehlen in einem Log auszeichnet (i.e., eine state machine [3]), als fehlertolerante verteilte Anwendung (replicated state machine) umzusetzen. Das Agreement-Protokoll garantiert hier, dass auf allen Rechnern des Clusters der gleiche Zustand (Konsistenz) vorliegt, sodass Erreichbarkeit immer noch gewährleistet werden kann, auch wenn Teile des Clusters ausfallen.
Raft [2] garantiert

- **Sicherheit** in allen Nicht-Byzantinischen Situationen,
- **Erreichbarkeit**, solange der Großteil der Server eines Clusters online ist,
- **Unabhängigkeit von Timing** (das System funktioniert z. B., egal in welcher Reihenfolge die Nachrichten des Leaders die Follower erreicht)
- **schnelle Antwortzeiten** (dadurch, dass im Normalfall für eine Clientanfrage nur die Antworten des Mehrheit des Clusters benötigt werden, bremsen einzelne langsame Server das System nicht aus).

Weshalb Raft?

Obgleich Paxos [1] seit seiner Veröffentlichung 1998 oft diskutiert und erweitert wurde, ist meine Wahl auf Raft [2] gefallen. Dafür sprachen für mich vor allem drei Gründe:

1. Raft ist mit dem Ziel konzipiert worden, einfach verständlich und in tatsächlichen Anwendungen realistisch einsetzbar zu sein. Wie in [2] beschrieben, ist Paxos dahingegen so komplex, dass die ursprüngliche Idee immer wieder neu aufgegriffen wurde, in dem Versuch, sie verständlicher zu machen. Zur Nutzung in praktischen Anwendungen scheint Paxos außerdem nicht unverändert einsetzbar zu sein.
2. Raft nutzt *RPCs* (remote procedure calls), welche ich als praxisnahe und sinnvolle Abstraktion von direkten Nachrichten sehe. Da RPCs (z. B. in ihrer Implementierung in gRPC[1]) mittels einer strukturierten Request-Nachricht und einer zugehörigen Response-Nachricht definiert sind, ermöglicht diese Technologie auch eine zweifelsfreie Zuordnung der Nachrichten ohne UUIDs o. ä. Zusätzlich sind gRPC Aufrufe auch sehr effizient, da mittels *protocolbuffers*[2] Nachrichten in einem effizienten Binärformat übertragen werden, was eine (De-) Serialisierung in / von einem String-Format einspart. Wenngleich ich diese nicht in meinem Projekt nutze (ich greife auf den Simulator sim4da zurück, welcher auf stringbasierte Nachrichten setzt), halte ich diese Punkte für wichtige Vorteile in einer echten Anwendung.

[1] `https://grpc.io/docs/what-is-grpc/introduction/`
[2] `https://protobuf.dev/overview/`

Grundidee von Raft

Raft setzt auf einen **starken Leader**, der Logeinträge von Clients annimmt und für die fehlertolerante Replikation dieser an die anderen Clustermitglieder (Follower) verantwortlich ist. Dies erinnert an einen zentralisierten Ansatz und verringert die Komplexität des verteilten Systems maßgeblich, indem Logeinträge stets nur vom Leader zu Followern fließen. Clients interagieren also im normalen Betrieb nur mit dem Leader und können von der verteilten Natur des Systems abstrahieren. Aufgrund dieser Tatsache können erst Clientanfragen vom System verarbeitet werden, wenn eine erfolgreiche Wahl durchgeführt wurde und so ein Leader feststeht. Zur besseren Verständlichkeit habe ich versucht, die folgenden drei Punkte (Leader Election, Log Replication und Membership Changes) in Abbildung 1 als Schaubild zu visualisieren.

Wahlen

Damit das System trotz der starken Zentralisierung auf den Leader fehlertolerant ist, kann im Falle eines Ausfalls ein neuer Leader für die nächste Amtszeit (*term*) gewählt werden: Anderen Server im Cluster erkennen den Ausfall am **Timeout** einer **Heartbeat**-Nachricht (leerer *appendEntries* RPC). Ein Server, der innerhalb des Timeouts keinen *Heartbeat* erhält, wird zum *Kandidat* und startet eine **Wahl** (*requestElection*), um zum neuen Leader zu werden. Um die Wahrscheinlichkeit zu verhindern, dass viele Server gleichzeitig eine Wahl starten, verwendet Raft **randomisierte** Timeouts. Die anderen Server innerhalb des Clusters vergeben maximal eine Stimme pro Term (an den ersten anfragenden Kandidaten, deren Log mindestens so aktuell ist, wie der eigene). Hat ein Kandidat die Wahl gewonnen, wird sein Zustand (seine Logeinträge) an die anderen Server des Clusters repliziert und eventuell widersprüchliche Einträge überschrieben. Der Zustand des aktuellen Leaders ist also bei Konflikten stets ausschlaggebend. Kann hingegen kein Kandidat eine Mehrheit der Stimmen erreichen, endet die Periode ohne einen Leader und nach erneutem Timeout startet eine neue Wahl in der nächsten Periode.

Log Replikation

Hat der Leader von einem Client einen neuen Logeintrag erhalten, verteilt er diesen mittels *appendEntries* RPC an alle Follower. Sobald er eine positive Antwort von der Mehrheit der Server zurückbekommt (in diesem Fall stimmen diese Server bis einschließlich des neuen Logeintrags überein), gilt der neue Eintrag als **commited** und der Zustand der state machines der Server wird aktualisiert.

Clusteränderungen

Kommt es zu einer Änderung des Clusters (i.e., neue Server treten bei, oder bestehende Server verlassen das Cluster), wird das Cluster in drei Phasen in die neue Konfiguration überführt:

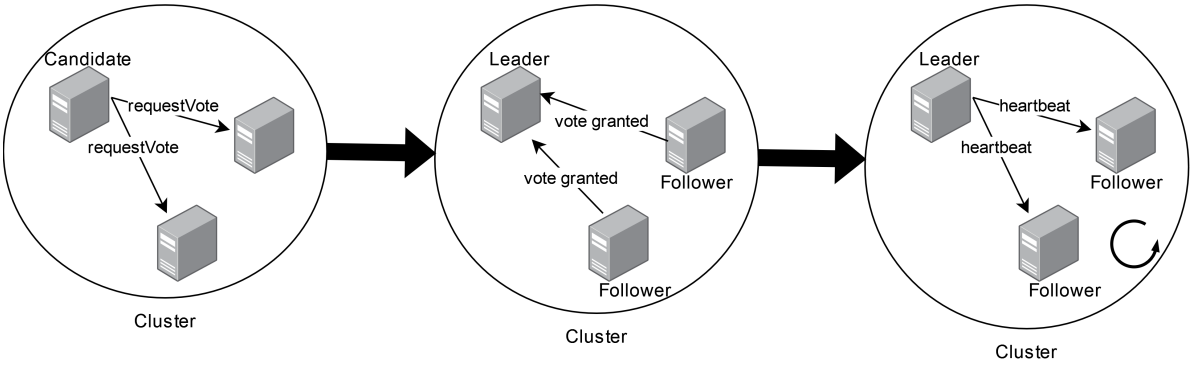
1. Neue Server treten als Follower bei, die jedoch nicht für Mehrheitsentscheidungen beachtet werden (`nonVotingMembers`). Dies ermöglicht es den neuen Servern, auf den Stand der anderen Follower aufzuschließen, während der Rest des Clusters immer noch Clientanfragen verarbeiten kann.
2. **Joint Consensus**: Damit das gesamte Cluster nicht in zwei unabhängige Mehrheiten zerfallen kann (dies ermöglicht die Wahl zweier Leader und Konsistenz kann nicht mehr gewährleistet werden), benötigt Agreement in dieser Phase unabhängige Mehrheiten aus dem alten (`clusterOld` in meiner Implementierung) und dem neuen Cluster (`clusterNew`).
3. Ist ein spezieller Konfigurationslogeintrag (`cOldNew`) erfolgreich commited, kann das System schließlich in die neue Konfiguration wechseln.

Meine Implementierung

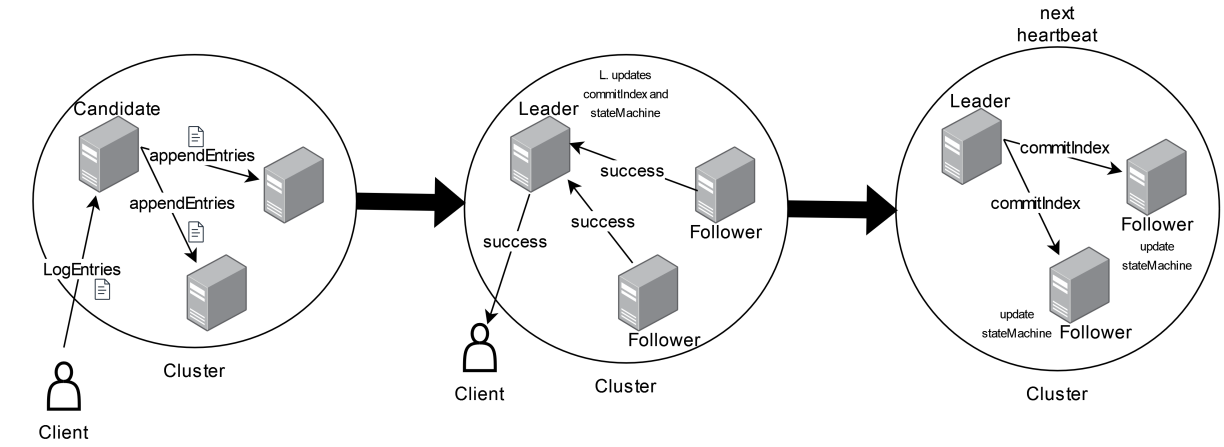
Ich habe mich dafür entschieden, eine verteilte **key-value Datenbank** zu implementieren. Einerseits ist dies eine naheliegende Anwendung, wenn es um Demonstrationen von Agreement-Protokollen geht (drei der Top-5 Raft Implementierungen[3] implementieren solche key-value Datenbanken), da es sich hier um eine ziemlich simple Anwendung handelt, an der sich die Funktionsweise eines Agreement-Protokoll gut veranschaulichen lässt. Andererseits sind solche key-value

[3] `https://raft.github.io/#implementations`

1. Leader Election



2. Log Replication



3. Membership Changes

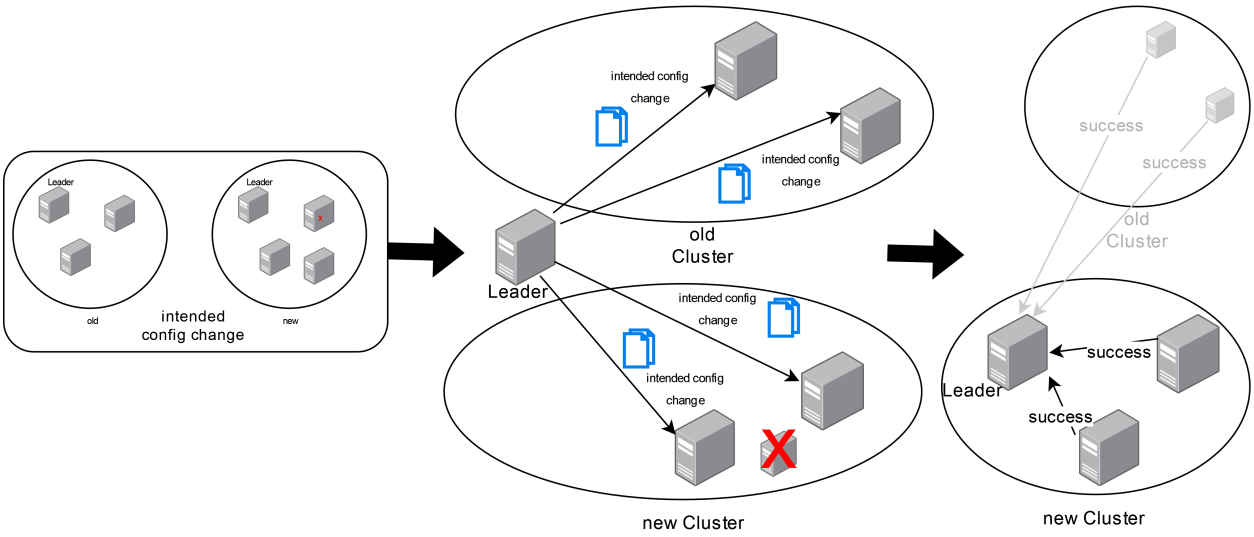


Abbildung 1: Grundlegende Funktionen eines Raft Clusters. Der Server mit dem niedrigsten Timeout startet die erste Wahl (**Leader Election**) in Term 1 und wird Leader. Durch regelmäßige Heartbeats wissen die Follower, dass der Leader noch aktiv ist. Nachdem ein Leader feststeht, können Clients Logeinträge an das Cluster senden, wobei der Leader für die Replizierung verantwortlich ist (**Log Replication**). Nachdem die Mehrheit der Follower einen Erfolg für die Replizierung eines Eintrags meldet, gilt er als commited und kann von der stateMachine verarbeitet werden. Wird ein Server aus dem Cluster entfernt / hinzugefügt (**Membership Changes**), muss zuerst ein spezieller Joint Consensus Logeintrag (cOldNew) commited werden, was separate Mehrheiten vom alten und neuen Cluster benötigt. Danach kann in die neue Konfiguration gewechselt werden.

Datenbanken in der Realität für verschiedenste Zwecke einsetzbar, wie man unschwer an der großen Popularität von Redis (bzw. seinen Open Source Forks) erkennen kann. Hinsichtlich Raft sind alle Mechaniken mit Ausnahme der *Log Compaction* implementiert, da diese nur im Kontext echte Anwendungen sinnvoll ist.

Weshalb Simulator?

Meine ursprüngliche Idee war es, eine tatsächlich nutzbare Datenbank in Rust (mittels *tonic*[4] für RPCs) umzusetzen. Eine Config hätte die IPs und Ports aller beteiligten Server enthalten und Clients hätten via REST API mit dem System interagieren können, um die verteilte Natur des Clusters zu maskieren (solange sich ein Großteil der Server im gleichen Netzwerk befindet, um Latenzen zu minimieren). Allerdings haben mehrere Gründe letztendlich für mich gegen eine tatsächliche Anwendung und für einen simulativen Ansatz mittels *sim4da* gesprochen:

- Ein funktionierendes **Netzwerk** für eine tatsächliche Anwendung aufzubauen, bringt viel **Komplexität** mit sich, sodass das Agreement-Protokoll eventuell nicht mehr im Fokus steht. Jeder Server müsste mittels IP und Port für alle anderen Server im Cluster bekannt sein und für ein tatsächliches Deployment müssten zusätzlich Firewallfreigaben / Portfreigaben beachtet werden. Zur einfacheren (schmerzfreien) Konfiguration des Clusters im selben Netzwerk habe ich mich in mDNS / zeroconf eingelesen - dann jedoch schnell gemerkt, dass die damit verbundene Netzwerkkonfiguration (Multicast etc.) schnell weit von der Agreement-Thematik wegführt.
- **Debugging**. Bereits im Simulator ist dies komplex genug, da es sich um eine Multithreading Anwendung handelt und Timings für Heartbeat-Nachrichten zu beachten sind. Die Netzwerklatenz in einem echten physischen Cluster und getrennte Logs auf den einzelnen Rechnern hätten dies zusätzlich um ein Vielfaches verkompliziert.

Besonderheiten meiner Implementierung

Im Folgenden liste ich Änderungen / Zusätze, die ich in meiner Implementierung für sinnvoll erachtet habe, auch wenn sie im ursprünglichen Raft Paper [2] nicht erwähnt wurden.

1. **matchIndex in Followerantworten**. Angenommen, der Leader sendet zwei *appendEntries* RPCs an einen Follower, im ersten soll der Follower den 0. Logeintrag und im zweiten die Logeinträge 0-2 replizieren (die in der Zwischenzeit dazu kamen). Sendet der Follower nun eine erfolgreiche Antwort (success = 1) an den Leader, so ist nicht klar, welche Logeinträge repliziert wurden, da die RPCs etwa in vertauschter Reihenfolge angekommen sein könnten (dann hätte der Follower bereits alle Einträge 0-2, sonst nur 0). Da es in den Nachrichten ansonsten keine Möglichkeit gibt, die Antwort zur ursprünglichen Anfrage zuzuordnen (z. B. durch eine UUID), habe ich einen *matchIndex* als zusätzliches Attribut in der *AppendEntriesResponse* hinzugefügt, damit der Follower dem Leader mitteilen kann, bis zu welchem Eintrag sein Log nun aktualisiert wurde.

2. In [2] ist für die Followerimplementierung eines *appendEntries* RPC vorgesehen, dass im Falle eines **widersprüchlichen Followereintrags** dieser und alle folgenden Einträge gelöscht werden und nur diejenigen Einträge hinzugefügt werden, die noch nicht im Log des Followers enthalten sind. Dies habe ich dahingegen simplifiziert, dass in jedem Fall alle Einträge ab dem `insertIndex` (prevLogIndex + 1) gelöscht werden und anschließend alle Einträge innerhalb von *entries[]* eingefügt werden. Dies spart einige Fallunterscheidungen und sollte in der Praxis nicht ineffizienter sein, da nicht oft viele Einträge überschrieben werden müssen.

3. Wenn ein Client Logeinträge an den Leader übermittelt (*clientPut*), ist es sinnvoll, dem Client die Einträge in der *clientPutResponse* zurückzugeben, die erfolgreich commited wurden (z. B. um ein teilweisen Erfolg zu melden). Zu diesem Zweck habe ich eine List<Pair> `clientIndices`, implementiert, die für jede Anfrage den **Start- und Endindex** der neuen Logeinträge im Log des Leaders speichert. Falls ein Client z. B. die Logeinträge 10-1000 übermittelt hat und der `leaderCommitIndex` auf 500 erhöht wurde, kann bereits ein teilweiser Erfolg (also der Commit der Einträge 10-500) an den Client übermittelt werden.

4. Um den *no-op* Eintrag (der nach einer erfolgreichen Wahl vom Leader commited wird), sowie Membership Changes zu implementieren, sind LogEinträge mit den Keys "noop", "config:cOldNew" und "config:cNew" **reserviert** und dürfen von Clients nicht verwendet werden.

[4] `https://docs.rs/tonic/latest/tonic/`

5. **Clusterlisten**. Im normalen Betrieb speichert jeder Server eine Liste der IDs der anderen Clustermitglieder (Liste `cluster`). Um Clusteränderungen umzusetzen, gibt es zusätzlich die Listen `nonVotingMembers`, `clusterOld` und `clusterNew`, die in den entsprechenden Phasen eingesetzt werden.

6. **GetQueue**. Damit ein Leader sicher sein kann, dass er nicht als Leader ersetzt wurde und sein Log dementsprechend ausschlaggebend ist, muss er eine Heartbeat-Nachricht mit der Mehrheit des Clusters austauschen. Dazu speichert die `getQueue` für jede "clientGet"-Anfrage, wie viele Follower den Heartbeat bestätigt haben (und somit den Leader noch als legitim ansehen). Übersteigt der Counter die Hälfte des Clusters, antwortet der Leader auf die "clientGet"-Anfrage und entfernt den ersten Eintrag in der `getQueue`.

Aufbau meines Projekts

Der gesamte Code für die Raft-Server befindet sich in der `Server`-Klasse. In der `engage()`-Methode werden für jeden Server 2 zusätzliche Threads (bzw. 3 für den Leader) gestartet, deren Funktion, zusätzlich zu einer Erklärung der wichtigsten Serverattribute, in Abbildung 2 zu sehen ist.
Die Nachrichten, die ein Server erhält, werden in der Methode `processMessage()` verarbeitet, welche den Großteil der Logik umfasst. Sie ist dafür verantwortlich, sämtliche Nachrichten zu verarbeiten und Antworten an den Absender zurückzuschicken. Die Logik für Kandidaten und Follower (Verarbeitung von äppendEntries", "requestVote" und "clientPut") ist in die Methoden `processAppendEntries()`, `appendEntries()`, `requestVote()`, `wonElection()` ausgelagert und selbsterklärend. Der Leader kann zusätzlich auf "clientGet" (`getQueue`), "configChange" (`nonVotingMembers` und `advanceToJointConsensus()`) und äppendEntriesResponse" (Update von `nextIndex`, `matchIndex` und `getQueue`) reagieren.
Schließlich sind selbsterklärende Tests für die drei Unterkategorien in Abschnitt (b) in den Klassen `TestElections`, `TestClientAppends` und `TestMembershipChanges` zu finden.

Hinweis

- Ich habe mich bewusst dafür entschieden, dass der gesamte Aufwand der Replizierung in die Methode `sendHeartbeat()` ausgelagert wird. Dadurch wird der das Interface insofern vereinfacht, dass Logeinträge in Folge von "clientPut", "configChange" und gewonnener Wahl nur an die Queue `log` angehangen werden müssen und die Replizierung automatisch während der periodischen Heartbeats erfolgt. Alternativ könnten *appendEntries* Nachrichten verschickt werden, die unabhängig von den Heartbeats sind, um Verzögerungen bei den Heartbeats zu verhindern. Da das Mitsenden der fehlenden Einträge aber mit relativ wenig Aufwand verbunden ist, gehe ich nicht davon aus, dass dies nötig ist.
- Da der Zustand des Simulators (auch mit `TestInstance.Lifecycle.PER_METHOD`) nicht nach jedem Test zurückgesetzt wird, funktionieren die Tests nur, wenn jeder Test einzeln (z. B. aus der IDE) gestartet wird.

Abschließende Bemerkungen

Wie ich während der Umsetzung dieses Projekts (und der vorherigen Übungen) gemerkt habe, sind verteilte Systeme und vor allem Agreement-Protokolle ziemlich komplex. Eine konkrete Implementierung von Raft, was als verständlichere und praxisnahe Alternative zu Paxos konzipiert wurde, ist durch das Zusammenspiel zwischen Leader und Followern nicht trivial (an welcher Stelle kann welcher Zustandsübergang durchgeführt werden?, wo muss auf welche Bedingung gewartet werden?). Der Praxisfokus und die Verständlichkeit ist den Autoren meiner Ansicht nach vor allem für den Kern des Protokolls gut gelungen, jedoch wird Raft durch die zusätzlichen Implementierungsnotizen und kurz angerissenen Einschränkungen (v. a. §5.4, §6 und §8), die zur Sicherheit des Protokolls getroffen werden müssen, doch verkompliziert. Außerdem setzt Raft (wie vermutlich fast alle Agreement-Protokolle) auf Heartbeat-Nachrichten, um zu verifizieren, dass der Leader noch ordnungsgemäß läuft. Dies sorgt bei einem sehr großen, gut ausgelasteten Cluster (zusammen mit dem Umstand, dass nur der Leader Clientanfragen verarbeiten darf) eventuell für Probleme mit der Bandbreite, da zu jedem Zeitpunkt mehrere Nachrichten unterwegs sein können, welche das Reagieren auf Anfragen verlangsamen. Ein einzelner zentraler Server könnte also auf Kosten der Fehlertoleranz je nach Anwendungsfall durch eine niedrigere Latenz vorteilhaft sein.
Bezüglich meiner konkreten Java Implementierung habe ich außerdem zwei Limitierungen bemerkt: Zum einen verwende ich pro Server 3 (bzw. 4) Threads. Wie in der zweiten HetComp Übung gesehen, führt dies schnell zu einem deutlichen

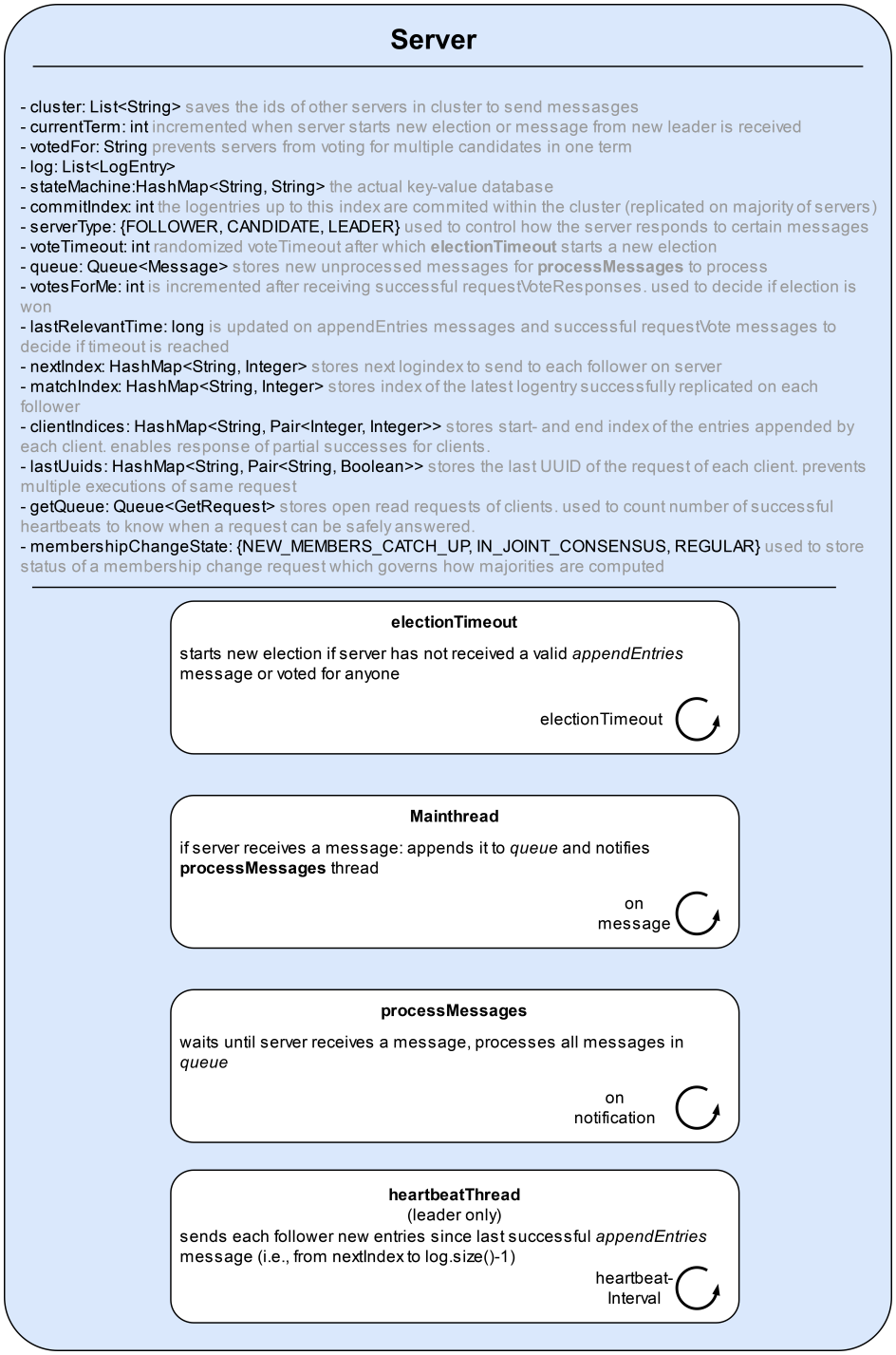


Abbildung 2: Oben: ausgewählte Serverattribute mit ihrer Verwendung (in grau). Unten: Threads pro Server mit ihrem Ausführungsinterval.

Overhead, sodass man den größten Vorteil des Simulators, problemlos Netzwerke mit sehr vielen Servern simulieren zu können, eventuell nur eingeschränkt nutzen kann. Abhilfe könnten hier die neuen virtuellen Threads schaffen, die Java ab Version 21 unterstützt.
Schließlich finde ich etwas problematisch im Raft Protokoll selbst: Ein neuer Server, der erst später bei einem Membership Change zum Cluster hinzugefügt wird, darf nicht zu früh starten. Da er vorher nicht zum Cluster gehört, erhält er keine Heartbeats und startet infolgedessen immer wieder Wahlen ohne Stimmen von den Clustermitgliedern zu erhalten, da sein Log nicht aktuell ist. Dementsprechend kann sein `currentTerm`-Attribut, wenn er schließlich zum Cluster hinzugefügt wurde, so hoch sein, dass er keine *requestVote* oder *appendEntries* RPCs anerkennt und weiterhin ständig neue Wahlen startet. Somit ist dieser Server in einer ständigen Schleife gefangen und wird niemals korrekt als Clustermitglied arbeiten. Aus diesem Grund habe ich im `testNewServer`-Test `currentTerm` des Leaders auf 30 gesetzt, damit der Beitritt des neuen Servers korrekt funktioniert.

Literatur

[1] Lamport, L. The part-time parliament. ACM Trans. Comput. Syst. 16, 2 (1998), 133–169.

[2] Ongaro, D., and Ousterhout, J. In search of an understandable consensus algorithm. In 2014 USENIX annual technical conference (USENIX ATC 14) (2014), pp. 305–319.

[3] Schneider, F. B. Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Computing Surveys (CSUR) 22, 4 (1990), 299–319.