Verteilte Systeme – Übung 3
Raft Consensus

Narek Grigoryan

Matr.Nr: 1547616

GitHub Link: https://github.com/Narek7/Verteilte_Systeme_Uebung-_3.git

Inhaltsverzeichnis

Einleitung

In verteilten Systemen ist es wichtig, dass alle Knoten eine einheitliche Entscheidung treffen können, um sicherzustellen, dass der Systemzustand konsistent bleibt. Hier kommen Agreement-Protokolle ins Spiel, die dazu dienen, Konsens zwischen mehreren Knoten in einem Netzwerk zu erreichen. Zwei der bekanntesten Protokolle für diesen Zweck sind Paxos und Raft.

Ich habe mich für Raft entschieden, weil es moderner ist und leichter zu verstehen war als Paxos. Paxos ist schon seit 1989 bekannt und hat sich als sehr robust erwiesen, aber die Komplexität in der Implementierung und im Verständnis ist hoch. Raft hingegen wurde 2014 entwickelt und hat sich das Ziel gesetzt, einfacher verständlich zu sein (genau wie der Title des Papers besagt „In Search of an Understandable Consensus Algorithm “), ohne dabei an Zuverlässigkeit zu verlieren. Dies macht es ideal für eine praktische Implementierung wie meine.

Was machen Agreement-Protokolle?

Agreement-Protokolle sorgen in verteilten Systemen dafür, dass mehrere Knoten gemeinsam eine Entscheidung treffen, auch wenn einige Knoten ausfallen oder nicht synchron sind. Das ist besonders wichtig, wenn man einen zentralen Zustand im System erhalten muss, wie zum Beispiel bei der Replikation von Daten oder der Wahl eines Leaders, der das System steuert. Ohne diese Protokolle könnten Knoten unterschiedliche Ansichten über den Systemzustand haben, was zu Fehlern oder Datenverlust führen kann.

Bekannte Protokolle wie Paxos und Raft haben alle das Ziel, in einem Netzwerk sicherzustellen, dass ein stabiler und einheitlicher Zustand erreicht wird, auch wenn das Netzwerk selbst fehlerhaft ist. Dabei spielt die Wahl eines Leaders eine zentrale Rolle, da dieser die Koordination übernimmt.

Einführung in Raft

Raft basiert auf der Idee, dass es immer einen Leader gibt, der die Verantwortung für das System übernimmt. Jeder Knoten im Raft-Cluster kann eine der folgenden Rollen annehmen:
- Follower: Knoten, die auf Anweisungen des Leaders warten.
- Kandidat: Knoten, die versuchen, durch eine Wahl zum Leader zu werden, wenn kein Leader verfügbar ist.
- Leader: Der Knoten, der gewählt wurde, um das System zu leiten und sicherzustellen, dass alle Knoten denselben Zustand haben.

Im Wesentlichen hat Raft zwei Hauptkomponenten:
1. Führungswahl (Leader Election): Wenn ein Follower keine Nachrichten (Heartbeats) vom Leader empfängt, wird er zum Kandidaten und startet eine Wahl.
2. Log-Replikation: Der Leader ist verantwortlich für die Verteilung von Änderungen an die Follower. Er schreibt diese Änderungen in sein Log und repliziert sie durch AppendEntries-Nachrichten an die Follower.

Das Best-Case-Szenario im Raft-Protokoll

Im Best-Case-Szenario gibt es keine Netzwerkfehler, und ein Knoten wird schneller zum Kandidaten, weil sein Wahlzeitlimit (Election Timeout) zuerst abläuft. Dieser Knoten gewinnt die Wahl und wird der neue Leader. Von diesem Punkt an sendet der Leader regelmäßig AppendEntries-Nachrichten (Heartbeats), um seine Führungsrolle zu bestätigen.

Detaillierte Erklärung der Implementierung

Meine Implementierung des Raft-Protokolls bildet dieses Best-Case-Szenario in einer simulierten Umgebung nach. Die zentrale Logik steckt in den Klassen Cluster und Node. Ich gehe jetzt Schritt für Schritt durch, wie die Wahl des Leaders und die Kommunikation zwischen den Knoten abläuft.

1. Cluster: Vorbereitung des Best-Case-Szenarios

Die Cluster-Klasse stellt das Netzwerk von Knoten dar. Die Methode prepareBestCaseScenario() sorgt dafür, dass ein Knoten ein kürzeres Wahlzeitlimit erhält als die anderen. Dadurch wird dieser Knoten zuerst zum Kandidaten. In meinem Code

wähle ich diesen Kandidaten zufällig aus:

```
private void prepareBestCaseScenario() {  1 usage
    // Randomly select one node to have the shortest election timeout
    Random rand = new Random();
    Node candidateNode = nodes.get(rand.nextInt(nodes.size()));

    long candidateTimeout = 5000; // 5 seconds
    long minOtherTimeout = 8000;   // 8 seconds
    long maxOtherTimeout = 10000;  // 10 seconds
```

Der Kandidat beginnt die Wahl, indem er RequestVote-Nachrichten an die anderen Knoten sendet. Diese Nachricht fordert die Follower auf, für ihn zu stimmen. Der Kandidat wartet dann auf die Antwort der Follower.

2. Node: Durchführung der Wahl

Jeder Knoten wird durch die Klasse Node repräsentiert. Wenn das Wahlzeitlimit eines Knotens abläuft, wird er automatisch zum Kandidaten. Dieser Vorgang geschieht in der Methode nodeTimeoutExpired(). Hier startet der Knoten den Wahlprozess und sendet RequestVote-Nachrichten an alle anderen Knoten.

```
setState("candidate");
votedFor = id;
resetVotesReceived();
log("Node n" + id + " becomes candidate for term " + term + " and requests votes.");

// Send RequestVote messages
for (Node otherNode : cluster.getActiveNodes()) {
    if (otherNode != this) {
        cluster.sendMessage( fromNode: this, otherNode,  messageType: "RequestVote", Color.YELLOW);
    }
}
```

Diese RequestVote-Nachricht wird in gelber Farbe visualisiert, um den Beginn des Wahlprozesses anzuzeigen.

3. Follower: Antworten auf RequestVote

Wenn ein Follower eine RequestVote-Nachricht empfängt, überprüft er, ob er bereits in dieser Wahlperiode für jemanden gestimmt hat. Falls nicht, stimmt er für den Kandidaten und sendet eine Vote-Nachricht zurück.

```
if ((votedFor == null || votedFor == message.getFromId()) && message.getTerm() == term) {
    votedFor = message.getFromId();
    lastHeartbeat = System.currentTimeMillis(); // Reset election timeout
    updateLabel(electionTimeout);
    cluster.sendMessage( fromNode: this, message.getFromNode(),  messageType: "Vote", Color.LIGHTGREEN);
    log("Node n" + id + " votes for Node n" + message.getFromId() + " in term " + term + ".");
} else {
    // Already voted in this term
    log("Node n" + id + " has already voted in term " + term + ".");
}
```

Die Follower senden eine grüne Vote-Nachricht zurück, wenn sie für den Kandidaten gestimmt haben. Der Kandidat zählt dann die Stimmen und prüft, ob er die Mehrheit erreicht hat.

4. Wahl des Leaders

Wenn der Kandidat genug Stimmen (also Mehrheit) erhalten hat, wird er zum Leader, wie in becomeLeader() beschrieben:

```
private void becomeLeader() {  1 usage
    // Attempt to set self as leader in the Cluster
    boolean success = cluster.attemptToSetLeader( node: this);
    if (success) {
        setState("leader");
        cluster.log("Node n" + id + " becomes leader in term " + term + ".");
        cluster.updateMessage("Node n" + id + " becomes leader in term " + term + ".");

        // Leader's election timer is not needed; heartbeats prevent timeouts
        electionTimerRunning = false;

        // Start sending AppendEntries
        cluster.sendAppendEntries();
    } else {
        log("Node n" + id + " detected an existing leader. Aborting leadership.");
        setState("follower");
        electionTimerRunning = true;
        resetElectionTimeout();
    }
}
```

Sobald der Kandidat die Mehrheit erreicht, ändert er seinen Zustand zum Leader. Ab diesem Moment sendet er regelmäßig AppendEntries-Nachrichten an alle Follower.

5. AppendEntries und Ack

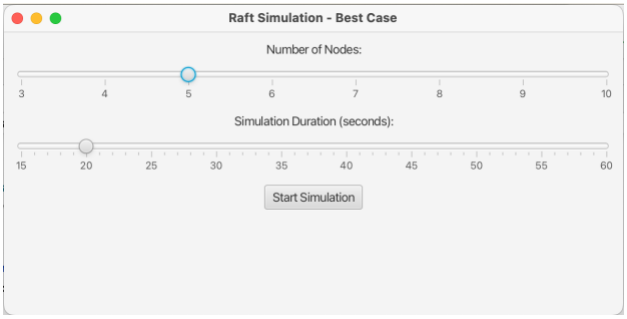
Der Leader sendet regelmäßig AppendEntries-Nachrichten (in rosa Farbe dargestellt), um seine Existenz zu bestätigen und die Follower davon abzuhalten, neue Wahlen zu starten.

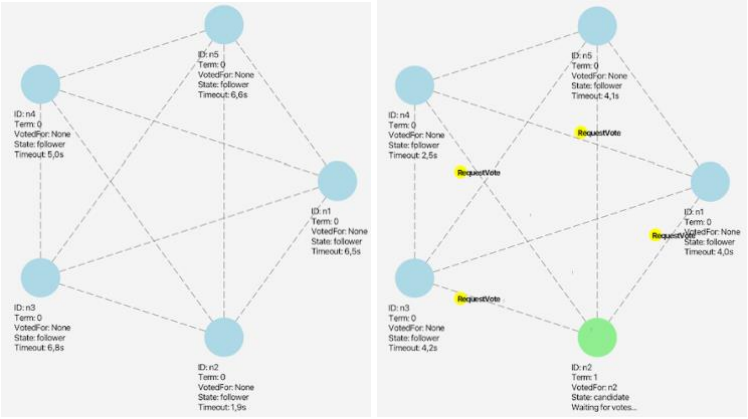
```
public void sendAppendEntries() {  1 usage
    // Leader sends AppendEntries messages periodically
    Timeline timeline = new Timeline(new KeyFrame(Duration.seconds( v: 3), e -> {
        Node currentLeader;
        synchronized (this) {
            currentLeader = leaderNode;
        }
        if (currentLeader != null) {
            currentLeader.log("Leader n" + currentLeader.getId() + " sends AppendEntries to followers.");
            for (Node node : getActiveNodes()) {
                if (node != currentLeader) {
                    sendMessage(currentLeader, node,  messageType: "AppendEntries", Color.PINK);
                }
            }
        }
    }));
    timeline.setCycleCount(Timeline.INDEFINITE);
    timeline.play();

    // Keep track of timelines to pause/resume
    timelines.add(timeline);
}
```

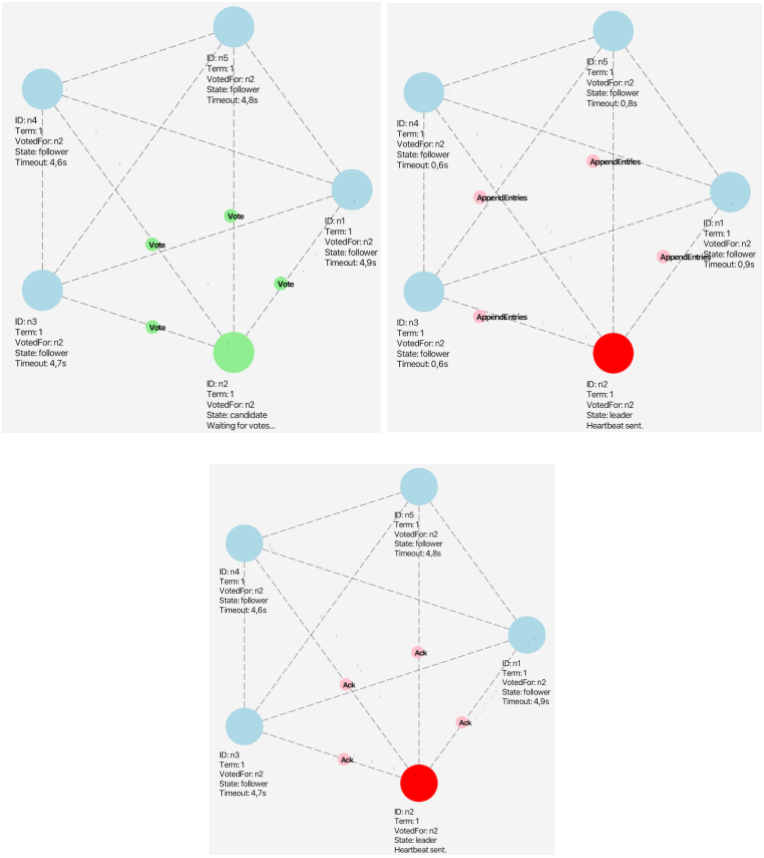
Die Follower empfangen diese AppendEntries-Nachrichten und bestätigen den Erhalt mit einer Ack-Nachricht (Acknowledgment), die dem Leader signalisiert, dass sie noch aktiv sind und keine neuen Wahlen starten:
Die Ack-Nachricht zeigt an, dass der Follower den Leader weiterhin als solchen akzeptiert. Dadurch wird verhindert, dass der Follower aufgrund fehlender Heartbeats zum Kandidaten wird.

Verbindung zwischen Theorie und Praxis

Diese Implementierung bildet die Mechanismen des Raft-Protokolls im Best-Case-Szenario exakt nach. Der Wahlprozess startet mit RequestVote, woraufhin die Follower mit Vote-Nachrichten antworten. Wenn der Kandidat die Mehrheit erreicht, wird er zum Leader und beginnt, AppendEntries-Nachrichten zu senden, um seine Führung aufrechtzuerhalten. Die Follower bestätigen den Erhalt dieser Nachrichten mit Ack, wodurch verhindert wird, dass sie neue Wahlen starten.Dieses System stellt sicher, dass in meiner Simulation stets ein Leader gewählt wird und das Netzwerk in einem stabilen Zustand bleibt. Die Knoten führen ihre Rollen klar aus, und die Visualisierung durch farbige Nachrichten macht den Ablauf verständlich und nachvollziehbar.







Split-Vote-Szenario im Raft-Protokoll

Nachdem wir im ersten Teil des Berichts den Best-Case-Szenario behandelt haben, gehen wir nun auf das sogenannte Split-Vote-Szenario ein, das einen interessanten Sonderfall im Raft-Protokoll darstellt. Während das Raft-Protokoll in der Regel eine klare und schnelle Wahl eines Leaders anstrebt, kann es unter bestimmten Bedingungen zu einem sogenannten Split-Vote kommen, bei dem mehrere Knoten gleichzeitig Kandidaten werden und die Anzahl der erhaltenen Stimmen gleich ist. Dadurch wird verhindert, dass ein Leader gewählt wird. Dieses Szenario verdeutlicht, wie Raft mit Unsicherheiten umgeht und dennoch Konsens erreicht.

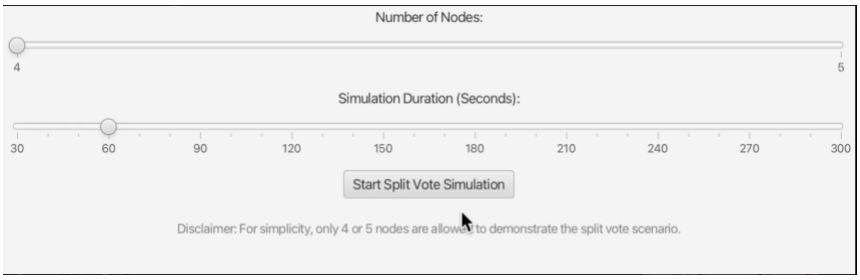
Was passiert im Split-Vote-Szenario?

Das Split-Vote-Szenario tritt auf, wenn zwei oder mehr Knoten in einem Cluster gleichzeitig zu Kandidaten werden, weil ihre Wahl-Timeouts zur nah gleichen Zeit ablaufen. Jeder Kandidat fordert von den anderen Knoten Stimmen an, erhält jedoch nur eine begrenzte Anzahl von Stimmen, da die restlichen Knoten bereits für andere Kandidaten gestimmt haben. Dadurch erreicht keiner der Kandidaten die benötigte Mehrheit der Stimmen, und es wird kein Leader gewählt. Die Kandidaten befinden sich in einer sogenannten Pattsituation, in der keiner genug Stimmen hat, um die Führung zu übernehmen.

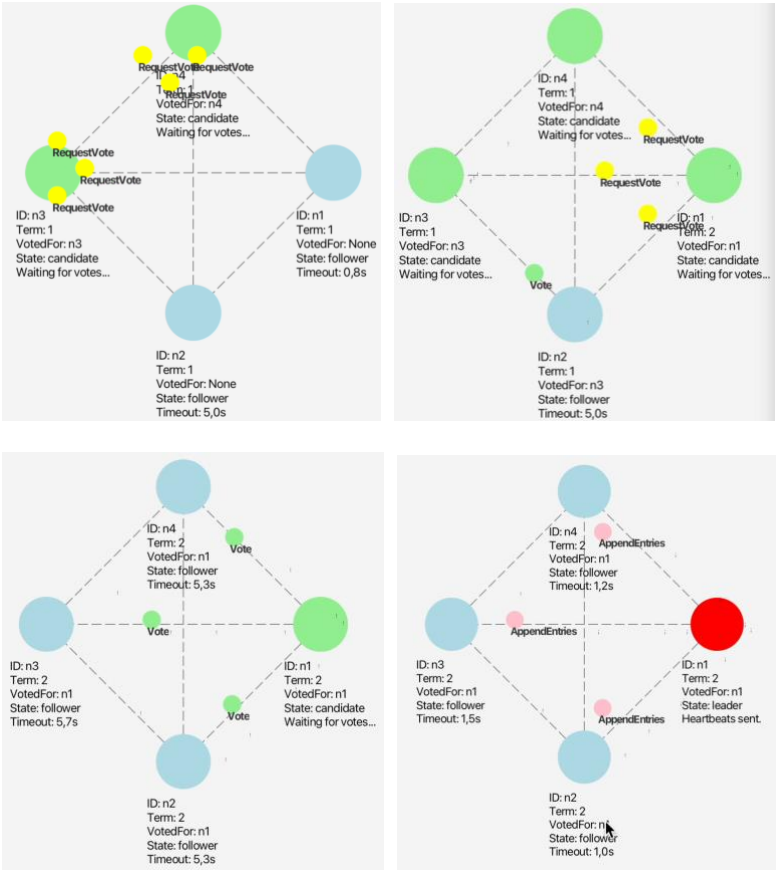
Dieses Szenario wird durch das Raft-Protokoll durch die Randomisierung der Timeout-Werte gemildert. Wenn eine Pattsituation entsteht, warten die alle Knoten einfach auf den Ablauf ihrer Wahl-Timeouts, woraufhin das System in die nächste Wahlperiode übergeht (ein neuer Term beginnt). In diesem neuen Term wird es wahrscheinlicher, dass nur ein Knoten seinen Timeout zuerst erreicht und somit genügend Stimmen erhält, um Leader zu werden. Damit wird die Pattsituation aufgelöst, und der Cluster kehrt in einen stabilen Zustand mit einem Leader und Followern zurück.

Detaillierte Erklärung der Implementierung

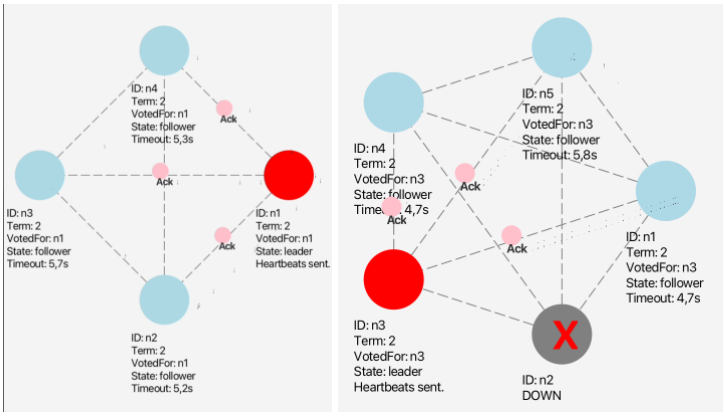
Meine Implementierung erlaubt es dem Nutzer, die Anzahl der Knoten (4 oder 5) und die Simulationsdauer über einen Schieberegler einzustellen. Diese Anzahl wurde der Einfachheit halber gewählt, um die Grundprinzipien des Split-Vote-Szenarios zu demonstrieren.



In größeren Cluster-Systemen mit mehr Knoten kann es selbstverständlich ebenfalls zu Split-Votes kommen, und die Regeln bleiben dabei dieselben. Wichtig ist, dass im Falle eines Split Votes immer auf den nächsten Timeout gewartet wird, bis eine neue Wahl eingeleitet wird. Wie in den 1. Screenshots zu sehen ist, startet die Simulation sofort mit zwei Knoten, die sich um die Wahl zum Leader bewerben. Diese Kandidaten senden RequestVote-Nachrichten an die anderen Knoten. Im ersten Wahlterm (Term 1) gibt es zwei Knoten, die gleichzeitig Kandidaten werden: n3 und n4. Beide warten darauf, Stimmen zu erhalten. Während n3 und n4 noch im ersten Term ihre Stimmen erwarten, tritt der Knoten n1 nahezu gleichzeitig auch in den Kandidaten zustand (Term 2)und fordert ebenfalls Stimmen an, wie im 2. Screenshot zu erkennen ist. Der Knoten n1 ist aufgrund der höheren Term-Zahl (Term 2) im Vorteil: Alle Follower wechseln auf den neuen Term, sobald sie die RequestVote-Nachricht von n1 erhalten (3. Screenshot). Wie der 4. Screenshot zeigt, werden die bisherigen Kandidaten (n3 und n4) wieder zu Followern, und alle Knoten geben ihre Stimmen an n1 ab, da n1 im zweiten Term ist und somit eine höhere Priorität hat. Der Knoten n1 wird schließlich zum Leader gewählt.



Nach der Wahl zum Leader sendet n1 regelmäßig AppendEntries-Nachrichten (Herzschläge) an die Follower, um sie darüber zu informieren, dass er der aktuelle Leader ist, und um neue Einträge im Log zu replizieren. Die Follower bestätigen den Erhalt dieser Nachrichten mit Ack-Antworten (Bestätigungen), wie im 5. Screenshot zu sehen ist. Dadurch wird sichergestellt, dass die Follower keine neuen Wahlen initiieren und die Konsistenz des Systems aufrechterhalten wird. Der 6. Screenshot veranschaulicht den Fall, in dem in einem 5-Knoten-System zufällig ein Knoten ausfällt, was zu einer Split-Vote-Situation führt



Es ist wichtig zu betonen, dass die von mir gewählte Visualisierung nur eine von vielen möglichen Darstellungen des Raft-Protokolls ist. Ziel meiner Implementierung war es, die Grundidee eines Split-Votes und dessen Lösung verständlich darzustellen. In einem echten System kann die Situation komplexer sein, und es gibt mehrere Arten, wie diese Abläufe visualisiert und simuliert werden könnten. Die gewählte Anzahl der Knoten und die Visualisierung der Nachrichten dienten in meiner Implementierung lediglich dazu, die Mechanismen hinter dem Split-Vote klar zu vermitteln. Das Grundprinzip – das Warten auf den nächsten Timeout bei einer Patt-Situation – bleibt jedoch in allen Szenarien unverändert.

Log Replication

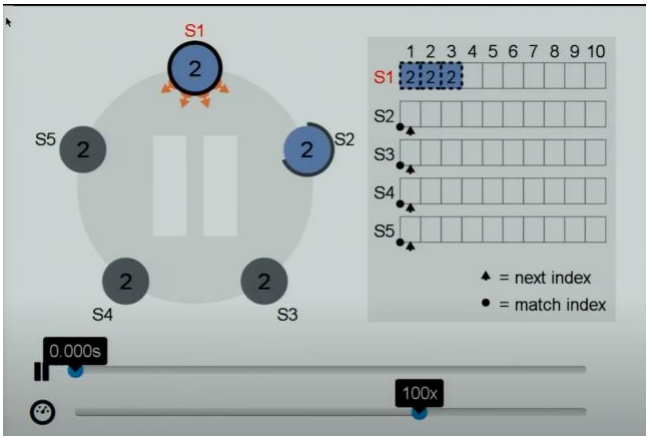
Die Log-Replikation im Raft-Konsensalgorithmus ist ein entscheidender Mechanismus, der Konsistenz über alle Knoten im verteilten System sicherstellt. Sie ermöglicht es dem Leader-Knoten, Log-Einträge, die von Clients empfangen wurden, an die Follower-Knoten zu replizieren, sodass alle Knoten im System den gleichen Log-Zustand beibehalten.

Was ist Log-Replikation?

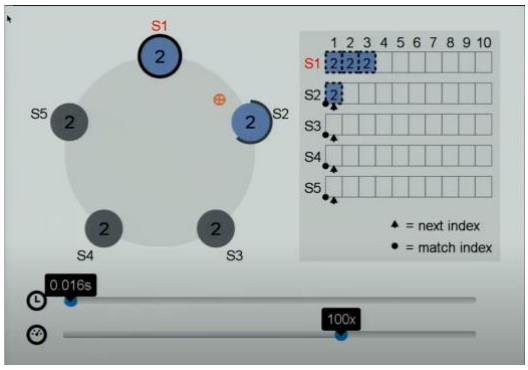
Die Log-Replikation beginnt mit dem Leader, der für das Empfangen von Befehlen von Clients und das Hinzufügen dieser Befehle zu seinem eigenen Log verantwortlich ist. Diese Befehle stellen Änderungen am Zustand des Systems dar (z.B. das Aktualisieren eines Werts in einem Key-Value-Store). Der Leader repliziert diese Log-Einträge dann an die Follower-Knoten, um sicherzustellen, dass sich das Cluster auf dieselbe Reihe von Zustandsänderungen einigt.

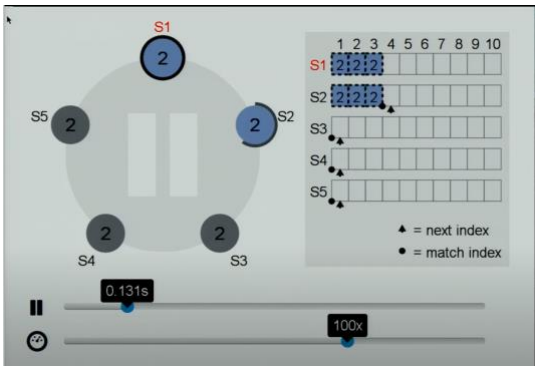
Der Prozess der Log-Replikation

1. Logeinträge im Leader: Der Leader erhält Befehle von Clients und schreibt diese zunächst in sein eigenes Log. Jeder Eintrag wird mit der laufenden Amtszeit (Term) des Leaders markiert.

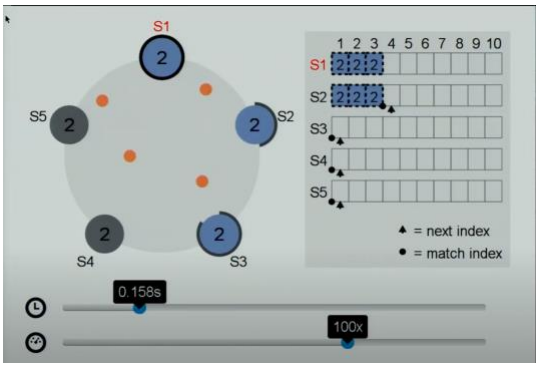


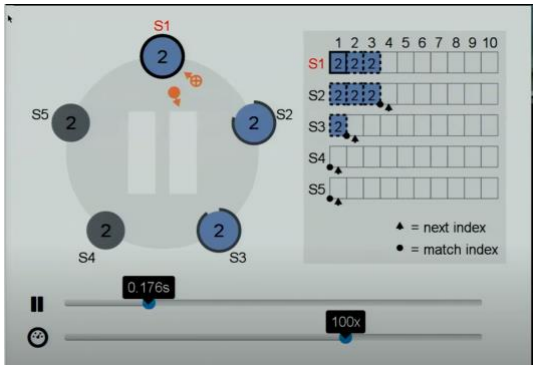
2. Replikation an Follower: Der Leader beginnt dann, diese Einträge an die Follower zu senden (Screenshot 1 zeigt den Anfang des Prozesses). Der Leader verfolgt zwei wichtige Indizes:
    - Next Index: Der Index, der angibt, welcher Logeintrag als nächstes an den Follower gesendet werden muss.
    - Match Index: Der Index, bis zu dem Leader und Follower sich bereits einig sind (d.h. die Einträge stimmen überein).
3. Erfolgreiche Replikation: Sobald der Follower den Eintrag erfolgreich in sein Log übernommen hat, schickt er eine Bestätigung an den Leader (Screenshot 2 und 3 zeigt diesen Fortschritt). Der Match Index wird dann entsprechend aktualisiert.



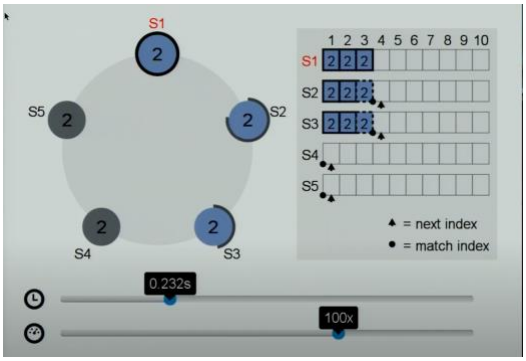


4. Commit der Einträge: Der Leader wartet, bis ein Eintrag von einer Mehrheit der Server im Cluster repliziert wurde, bevor er ihn als „committed“ markiert. Erst dann werden die Einträge für die Clients als dauerhaft bestätigt (Screenshot 4 und 5 zeigen den Fortschritt der Replikation).

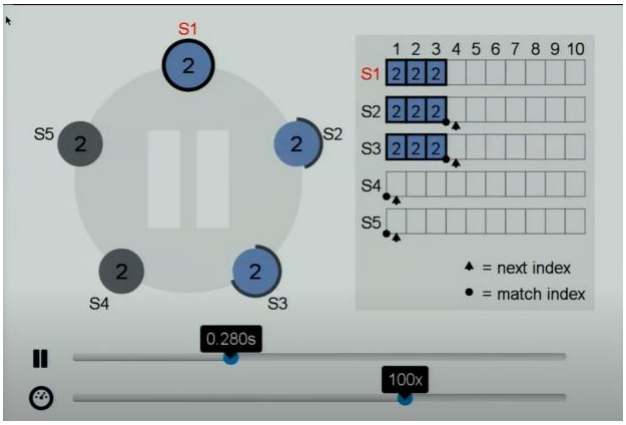




5. Nachrichten zum Commit: Der Leader sendet regelmäßig Nachrichten an die Follower, um sie über den Commit Index zu informieren. Dies stellt sicher, dass die Follower wissen, welche Einträge sicher sind und bei einem möglichen Leader-Wechsel nicht verloren gehen (Screenshot 6 zeigt diese Kommunikation).



6. Vollständige Replikation: Sobald die Einträge erfolgreich in den Logs der Follower repliziert und als „committed“ markiert wurden, ist die Replikation abgeschlossen (Screenshot 7).



Log Replication: Repairing Inconsistencies

Um Inkonsistenzen in der Log-Replikation im Raft-Protokoll zu beheben, wird ein Mechanismus verwendet, der sicherstellt, dass alle Follower die gleichen Einträge im Log wie der Leader haben. Diese Einträge müssen in derselben Reihenfolge vorliegen und dieselben Befehle repräsentieren. Die Inkonsistenzen können auftreten, wenn ein Leader abstürzt und ein neuer Leader gewählt wird, wobei unterschiedliche Logs auf den Followern entstanden sind. Es gibt zwei Hauptarten von Inkonsistenzen, die behoben werden müssen: fehlende Einträge und überflüssige Einträge.

1. Fehlende Einträge

- Wenn ein neuer Leader erkennt, dass ein Follower nicht alle Einträge im Log hat, wird dies durch eine Konsistenzprüfung im AppendEntries-RPC festgestellt. Dabei überprüft der Leader, ob der Follower an der Stelle im Log, wo der nächste Eintrag eingefügt werden soll, einen Eintrag mit derselben Term-Nummer hat.

- Falls der Follower keine Übereinstimmung hat, lehnt er die AppendEntries-Anfrage ab, und der Leader verringert den Next Index, also den Punkt, an dem der nächste Eintrag gesendet wird. Dies passiert solange, bis der Follower und der Leader eine Übereinstimmung im Log finden.
- Sobald eine Übereinstimmung gefunden ist, beginnt der Leader damit, die fehlenden Einträge an den Follower zu senden. Dadurch wird der Follower schrittweise auf denselben Stand wie der Leader gebracht.

2. Überflüssige Einträge

- Manchmal kann es vorkommen, dass ein Follower über Einträge verfügt, die der aktuelle Leader nicht mehr anerkennt, weil sie unter einem vorherigen Leader entstanden sind, der abgestürzt ist, bevor diese Einträge an andere Follower repliziert wurden. Diese Einträge werden als überflüssig betrachtet.
- In diesem Fall sendet der Leader ebenfalls AppendEntries-Anfragen, wobei der Konsistenzcheck sicherstellt, dass die Einträge auf den Followern mit denen des Leaders übereinstimmen.
- Wenn der Follower über Einträge verfügt, die der Leader nicht mehr akzeptiert, wird der Follower seine überflüssigen Einträge löschen und die neuen Einträge vom Leader übernehmen.

Ablauf der Reparatur von Inkonsistenzen

- Der Leader überprüft zunächst, ob die Logs des Followers mit den eigenen übereinstimmen.
- Falls eine Inkonsistenz festgestellt wird (fehlende oder überflüssige Einträge), wird der Next Index des Followers angepasst, bis eine Übereinstimmung im Log gefunden wird.
- Nach der Übereinstimmung werden die fehlenden Einträge repliziert oder die überflüssigen Einträge entfernt, sodass alle Follower und der Leader wieder dasselbe Log haben.

Dieser Mechanismus stellt sicher, dass alle Server im Cluster konsistent bleiben, auch wenn Leaderwechsel oder Abstürze auftreten. Jeder Eintrag wird erst dann als committed markiert, wenn er auf einer Mehrheit der Server repliziert wurde.

Schlussfolgerung und Ausblick

In diesem Bericht wurde der Konsensmechanismus des Raft-Protokolls detailliert analysiert, insbesondere die Leaderwahl, die Log-Replikation und die Reparatur von Inkonsistenzen. Raft zeichnet sich durch seine klare und verständliche Struktur aus, was es einfacher macht, es in verteilten Systemen zu implementieren und zu warten. Die Fähigkeit des Protokolls, durch einfache Mechanismen wie Heartbeats und den Konsistenzcheck robuste Übereinstimmungen zwischen Leader und Followern sicherzustellen, hebt es von anderen Konsensprotokollen wie Paxos ab.

Im Rahmen dieser Arbeit wurde sowohl der Normalfall der Log-Replikation als auch der Umgang mit Abweichungen im Log erfolgreich demonstriert. Insbesondere die effiziente Handhabung von Inkonsistenzen trägt wesentlich zur Stabilität des Systems bei, da verlorene oder überflüssige Einträge zuverlässig behandelt werden.

Literaturverzeichnis

https://raft.github.io/raft.pdf

https://raft.github.io

https://observablehq.com/@stwind/raft-consensus-simulator

https://www.youtube.com/watch?v=ro2fU8_mr2w&t=17s