Ergebnisbericht: Firefly-Synchronisation

Cyril Wagner

November 26, 2024

Disclaimer: Dieser Bericht sowie der zugehörige Code wurden mit Unterstützung einer Künstlichen Intelligenz (KI) erstellt. Alle Inhalte wurden überprüft, angepasst und ergänzt, um den Anforderungen der Aufgabenstellung zu entsprechen.

Abstract

In diesem Bericht werden die Lösungen zu zwei Aufgaben im Rahmen der Firefly-Synchronisation beschrieben. In der ersten Aufgabe wurde eine zentrale Simulation entwickelt, in der alle Fireflies innerhalb eines Programms laufen. In der zweiten Aufgabe wurde die Simulation dezentralisiert: Jede Firefly fungiert als unabhängiger Prozess, der mit seinen Nachbarn über das Netzwerk kommuniziert. Der Bericht beleuchtet die Implementierung, beschreibt Herausforderungen und dokumentiert 'überraschende' Erkenntnisse.

1 Einführung

Die Synchronisation von Fireflies ist ein bekanntes Phänomen in der Natur und ein anschauliches Beispiel für die Selbstorganisation in komplexen Systemen. Ziel der beiden Aufgaben war es, dieses Phänomen zu simulieren: [1]

- **Aufgabe 1: Zentrale Simulation.** Alle Fireflies werden in einem einzigen Programm simuliert. Die Interaktion erfolgt basierend auf dem Kuramoto-Modell.
- **Aufgabe 2: Dezentrale Simulation.** Jede Firefly ist ein eigenständiger Prozess, der Nachbarinformationen über ein Netzwerkprotokoll wie gRPC austauscht.

2 Aufgabe 1: Zentrale Simulation

2.1 Lösungsansatz

Die zentrale Simulation wurde in Python mithilfe der Tkinter-Bibliothek implementiert. Fireflies sind als Objekte in einer 2D-Gitterstruktur organisiert. Jede Firefly hat:

- Eine **Phase**, die ihre aktuelle Position im Synchronisationszyklus beschreibt.
- Eine **Helligkeit**, die basierend auf der Phase berechnet wird:

$$\text{Helligkeit} = \frac{1 + \sin(\text{Phase})}{2}.$$

[1] Der Source Code, sowie die Videos der Simulation sind zu finden unter: `https://github.com/s4cywagn/SA4E_PF1/tree/main/PF1`

Die Interaktion basiert auf dem Kuramoto-Modell. Jede Firefly aktualisiert ihre Phase abhängig von den Phasen ihrer Nachbarn:

$$\theta_i(t+1) = \theta_i(t) + K \sum_{j \in \text{Nachbarn}} \sin(\theta_j - \theta_i),$$

wobei K die Synchronisationsrate darstellt.

Die Visualisierung erfolgt über ein Tkinter-Canvas, wobei jede Firefly als farbiges Rechteck dargestellt wird. Die Farben variieren von Schwarz bis Gelb.

2.2 Ergebnisse

- Fireflies beginnen mit zufälligen Phasen und synchronisieren sich nach einigen Sekunden.
- Die visuelle Animation ist flüssig, und die Farben der Fireflies ändern sich kontinuierlich, bis sie einheitlich werden.

2.3 Implementierung

Die Implementierung war vergleichsweise einfach nachdem ich mir das Kuramoto Modell mit Hilfe von `https://www.youtube.com/watch?v=43OIzJ8OyGM&ab_channel=FlammableMaths` und `https://en.wikipedia.org/wiki/Kuramoto_model` etwas näher gebracht habe, jedoch waren die **Farbübergänge** ein unerwarteter Fokus. Um eine flüssige Animation zu erreichen, mussten Farbänderungen über einen Glättungsfaktor manipuliert werden. Dadurch wurde die Simulation visuell ansprechender (kleines Detail).

3 Aufgabe 2: Dezentrale Simulation

3.1 Lösungsansatz

In der zweiten Aufgabe wurde jede Firefly als eigenständiger Prozess implementiert, der über gRPC kommuniziert. Der gRPC-Ansatz ermöglichte eine effiziente Kommunikation zwischen Fireflies. Die Architektur bestand aus:

- Einem **gRPC-Server** pro Firefly, der Anfragen für Phaseninformationen verarbeitet.
- Einem **gRPC-Client**, der regelmäßig die Phasen der Nachbarn abfragt.

Die Kommunikation wurde durch die folgende Protobuf-Spezifikation definiert:

```
service Firefly {
    rpc SendPhase (PhaseMessage) returns (AckMessage);
    rpc RequestPhase (Empty) returns (PhaseMessage);
}

message PhaseMessage {
    float phase = 1;
}
message AckMessage {
    string status = 1;
}
message Empty {}
```

Die Visualisierung wurde aus Aufgabe 1 übernommen, wobei die Phaseninformationen über das Netzwerk ausgetauscht wurden.

3.2 Ergebnisse

- Fireflies synchronisieren sich zuverlässig, auch wenn sie als separate Prozesse laufen.
- Nach der Synchronisation zeigen alle Fireflies eine synchronisierte Farbe zwischen schwarz und gelb, die global einheitlich ist.
- Das dezentrale System ist robust gegenüber Prozessausfällen.

3.3 Implementierung

Erneut ging es darum, sich zuerst mit den Anforderung vertraut zu machen. gRPC wurde erlernt mit Hilfe von der offiziellen Website `https://grpc.io/` und Videos wie `https://www.youtube.com/watch?v=gnchfOojMk4&ab_channel=ByteByteGo` Während der Implementierung der dezentralen Simulation traten zwei Hauptprobleme auf, die eine Anpassung der Lösung erforderlich machten.

3.3.1 Netzwerkverzögerungen und visuelle Artefakte

Ein überraschend komplexer Teil war die Implementierung der Kommunikation über gRPC. Obwohl gRPC eine leistungsstarke Bibliothek ist, führte die Interprozesskommunikation zu unerwarteten Verzögerungen. Diese Verzögerungen entstehen, weil jede Firefly in regelmäßigen Intervallen die Phaseninformationen ihrer Nachbarn abruft und verarbeitet. Wenn mehrere Prozesse parallel laufen, können folgende Probleme auftreten:

- **Ungleichmäßige Aktualisierungen:** Manche Fireflies empfangen Phasenupdates später als andere, was zu Inkonsistenzen bei der Synchronisation führt.
- **Stockende Animation:** Die Verzögerungen äußerten sich visuell durch „ruckartige“ Farbänderungen, da die Animation nicht kontinuierlich wirkte.

Lösung: Interpolation der Helligkeit Um diese Artefakte zu minimieren, wurde ein Interpolationsmechanismus für die Helligkeit eingeführt. Dieser glättet Übergänge zwischen aufeinanderfolgenden Updates. Konkret wird der Helligkeitswert einer Firefly nicht sofort auf den Zielwert gesetzt, sondern in kleinen Schritten angenähert:

$$\text{Helligkeit}_{\text{neu}} = \text{Helligkeit}_{\text{aktuell}} + (\text{Zielhelligkeit} - \text{Helligkeit}_{\text{aktuell}}) \cdot \alpha,$$

wobei $\alpha = 0.1$ ein Glättungsfaktor ist.

Dies sorgt dafür, dass auch bei unregelmäßigen Netzwerkupdates die Farbänderungen visuell flüssig erscheinen. Durch diese Maßnahme konnte das Problem der stockenden Animation deutlich reduziert werden.

3.3.2 Synchronisationserkennung

Ein weiterer komplexer Punkt war die Erkennung, ob alle Fireflies denselben synchronisierten Zustand erreicht haben. Da keine zentrale Steuerung existiert, die den Status aller Prozesse kennt, musste jede Firefly lokal anhand ihrer Nachbarinformationen abschätzen, ob Synchronisation erreicht wurde. Hierbei ergaben sich folgende Herausforderungen:

- **Inhomogene Phasendifferenzen:** Selbst wenn die meisten Fireflies synchronisiert erscheinen, können kleine Abweichungen zwischen Nachbarn bestehen, die den Zustand als „nicht synchronisiert“ erscheinen lassen.
- **Asynchronität der Kommunikation:** Da Netzwerkpakete mit Verzögerung eintreffen können, ist es schwierig, den Status aller Nachbarn exakt zu einem Zeitpunkt zu erfassen.

Lösung: Schwellenwert für Phasendifferenzen Um dieses Problem zu lösen, wurde ein Schwellenwert $\epsilon = 0.1$ eingeführt. Eine Firefly betrachtet sich selbst als synchronisiert, wenn der absolute Unterschied ihrer Phase zu allen Nachbarn kleiner als ϵ ist:

$$\text{Synchronisiert, wenn: } |\theta_i - \theta_j| < \epsilon \quad \forall j \in \text{Nachbarn}.$$

Dieser Ansatz ist robust gegenüber kleinen Abweichungen und funktioniert gut in Kombination mit regelmäßigen Updates.

Globale Synchronisationserkennung Um sicherzustellen, dass wirklich alle Fireflies synchronisiert sind, wurde zusätzlich überprüft, ob **alle Nachbarn einer Firefly synchronisiert** sind. Dazu propagiert jede Firefly den synchronisierten Zustand (True/False) an ihre Nachbarn. Dieser Zustand wird wiederum propagiert, bis alle Fireflies denselben synchronisierten Zustand melden.

4 Vergleich der Ansätze

Aspekt	**Zentrale Simulation**	**Dezentrale Simulation**
Implementierung	Simpel, alle Fireflies laufen im selben Programm.	Komplex, jede Firefly ist ein unabhängiger Prozess.
Skalierbarkeit	Begrenzte Skalierbarkeit durch zentrale Steuerung.	Sehr skalierbar, da Prozesse unabhängig sind.
Robustheit	Fehler in einer Firefly können das gesamte System beeinflussen.	Robust, da Fireflies unabhängig voneinander arbeiten.
Visualisierung	Flüssige und einfache Animation.	Etwas komplexer wegen Netzwerkverzögerungen.

Table 1: Vergleich der beiden Ansätze

5 Fazit

Die Firefly-Synchronisation ist ein Beispiel für Selbstorganisation. Während die zentrale Simulation einfach zu implementieren war, zeigte die dezentrale Lösung die Herausforderungen verteilter Systeme auf. Überraschende Erkenntnisse waren:

- Die visuelle Interpolation war entscheidend, um eine flüssige Animation zu erreichen.
- Netzwerkverzögerungen in der dezentralen Simulation machten zusätzliche Maßnahmen notwendig.

Die Ergebnisse zeigen, dass lokale Interaktionen globales Verhalten erzeugen können, unabhängig davon, ob die Simulation zentral oder dezentral implementiert ist.