Ergebnisbericht Übungsblatt 1 SA4E

Simon Haebenbrock, s4sihaeb@uni-trier.de, dc: SimonH (simon_7672)

Aufgabe 1 „Zaghafte erste Glühwürmchen“ (Monolith – Mulit-Threaded)

Die Simulation habe ich in Java realisiert und dabei eine grafische Benutzeroberfläche entwickelt, die den Synchronisationsprozess anschaulich darstellt.

Lösungsansatz

Threads und Nachbarschaft

- Ich habe jedes Glühwürmchen als eigenen Thread implementiert, der autonom seine Phase aktualisiert und gleichzeitig die Nachbarn beeinflusst.
- Die Nachbarn sind nach einer zyklischen Torus-Struktur organisiert. Dadurch hat jedes Glühwürmchen immer vier Nachbarn: oben, unten, links und rechts.

Das Kuramoto-Modell

- Das Kuramoto-Modell, das die Synchronisation von Oszillatoren beschreibt, war die Grundlage der Simulation. Die Phase jedes Glühwürmchens wird dabei durch die Differenzen zu den Phasen der Nachbarn beeinflusst.
- In der Simulation habe ich diese Nachbarschaftsinteraktion regelmäßig berechnet, um den Prozess der Synchronisation nachzubilden.

Visualisierung

- Die grafische Benutzeroberfläche habe ich mit Java Swing umgesetzt. Sie zeigt die Phasen der Glühwürmchen in einem 5x5-Gitter:
    - **Gelb**: Phase im Leuchtzyklus.
    - **Grau**: Phase im Ruhezyklus.
- Ein Timer aktualisiert die Darstellung in kurzen Intervallen, um den Fortschritt kontinuierlich sichtbar zu machen.

Ergebnisse und Beobachtungen

Beschleunigte Synchronisation

- Um die Synchronisation der Glühwürmchen schneller sichtbar zu machen, habe ich die Kopplungsstärke (SYNC_RATE) auf einen hohen Wert von **0.005** gesetzt.
- **Ergebnis**: Die Glühwürmchen synchronisieren sich innerhalb weniger Sekunden, ohne dass dabei signifikante visuelle Störungen auftreten.

Herausforderungen

- **Threading**: Aufgrund der parallelen Threads kam es anfangs zu kleinen Phasenabweichungen, da diese unabhängig voneinander laufen. Ich habe dieses Problem durch eine präzise Regulierung des Phasenfortschritts in jedem Thread minimiert.
- **Visualisierung**: Eine hohe Synchronisationsrate führte bei kurzen Timer-Intervallen zu schnellen Änderungen der Darstellung. Durch die Optimierung des Timer-Intervalls auf **50ms** konnte ich eine flüssige und klare Darstellung sicherstellen.

Quellen A1

- Wikipedia-Autoren. (2024, August 19). *Kuramoto model*. Wikipedia. https://en.wikipedia.org/wiki/Kuramoto_model
- Wikipedia-Autoren. (2003, November 11). *Torus*. https://de.wikipedia.org/wiki/Torus
- GitHub Copilot für Fehleranalyse, Refactoring und Generierung von Boilerplate-Code

Aufgabe 2 „Kommunizierende Glühwürmchen“ (Distributed – Apache Thrift/ gRPC/ Java RMI)“

Für die Realisierung der Aufgabe 2 habe ich Java RMI genutzt, um die Kommunikation zwischen den einzelnen Glühwürmchen zu implementieren. Jedes Glühwürmchen ist ein eigenständiger Prozess/ Thread, der über RMI mit seinen Nachbarn kommuniziert. Die Zustandsinformationen, wie die Phase, werden über ein Interface regelmäßig zwischen den Glühwürmchen ausgetauscht, um den Synchronisationsprozess zu steuern.

Lösungsansatz

Kommunikationsschnittstelle

Als Kommunikationsschnittstelle habe ich ein Java RMI-Interface (FireflyRMI) definiert. Dieses Interface stellt die Grundlage für die Interaktion zwischen den Glühwürmchen dar. Die Schnittstelle enthält die Methoden getPhase, setNeighbors und synchronizeWithNeighbors, welche die essenziellen Aktionen der Glühwürmchen abbilden:

- **getPhase** ermöglicht den Zugriff auf die aktuelle Phase eines Glühwürmchens.
- **setNeighbors** erlaubt das Setzen der Nachbarn, mit denen ein Glühwürmchen interagieren soll.
- **synchronizeWithNeighbors** aktualisiert die Phase basierend auf den Phasen der Nachbarn und implementiert so die Synchronisationslogik.

Durch dieses Interface wird eine klare Trennung zwischen der Implementierung und der Kommunikation erreicht, wodurch der Code modular und flexibel bleibt.

Aufgaben des Servers

Der Server (FireflyServer) übernimmt mehrere zentrale Aufgaben:

1. **Erstellung der RMI-Registry:** Der Server startet die RMI-Registry, die für die Verwaltung der Remote-Objekte zuständig ist.
2. **Instanziierung der Glühwürmchen:** Ich habe 25 Instanzen der Klasse FireflyImplementation erstellt, die jeweils als eigenständige Threads laufen und über RMI registriert werden.
3. **Zuweisung der Nachbarn:** Für jedes Glühwürmchen identifiziert der Server die Nachbarn basierend auf der Torus-Struktur und setzt diese über die Methode setNeighbors.
4. **Start der Threads:** Die Glühwürmchen werden nach ihrer Initialisierung als Threads gestartet, sodass sie unabhängig voneinander laufen und synchronisieren können.

Observer

Um die Zustände der Glühwürmchen zu visualisieren, habe ich einen Observer (FireflyObserver) implementiert. Dieser holt sich über die RMI-Registry Referenzen auf alle Glühwürmchen. Der Observer ordnet die Glühwürmchen in einem 5x5-Gitter an und beobachtet kontinuierlich deren Phasen. Die GUI zeigt die Phasen farblich an:

- Gelb repräsentiert eine Phase, in der das Glühwürmchen leuchtet.
- Grau repräsentiert die Ruhephase.

Die Visualisierung wird alle 50 Millisekunden aktualisiert, um die Synchronisation in Echtzeit darzustellen.

Herausforderungen und Lösungen

1. **Kommunikation über RMI:** Um die Kommunikation zwischen den Glühwürmchen zu ermöglichen, mussten alle Objekte in der Registry eindeutig registriert werden. Die Verwendung von rebind stellte sicher, dass jedes Glühwürmchen unter einem individuellen Namen auffindbar ist.
2. **Nachbarschaftszuweisung:** Die Torus-Struktur der Nachbarschaft erfordert, dass jedes Glühwürmchen korrekt mit seinen vier Nachbarn verbunden ist. Hierbei war darauf zu achten, dass die Nachbarn bei einem Gitterrand auf die gegenüberliegende Seite verweisen.
3. **Thread-Handling:** Obwohl alle Glühwürmchen unabhängig laufen, musste sichergestellt werden, dass der Synchronisationsprozess korrekt abläuft und Threads sich nicht gegenseitig blockieren. Dies habe ich durch Synchronisation (synchronized) gelöst.

Erklärung zu leichten Abweichungen in der Synchronisation
Die Simulation (siehe Firefly_Observer.mp4 in Dropbox) zeigt insgesamt eine korrekte Synchronisation der Glühwürmchen, jedoch treten gelegentlich leichte Abweichungen in den Phasen auf. Ich nehme an, dass diese Abweichungen durch die asynchrone Kommunikation und die parallele Ausführung der Threads verursacht werden. Trotz dieser kleinen Unterschiede bleibt der Synchronisationsprozess insgesamt stabil.

Fazit
Die Nutzung eines Interfaces als Kommunikationsschnittstelle hat die Modularität und Verständlichkeit des Codes gefördert. Der Server übernimmt zentrale Aufgaben wie die Initialisierung und Nachbarschaftszuweisung, während der Observer eine intuitive Visualisierung des Systems ermöglicht. So wird die dezentrale Natur des Systems anschaulich dargestellt.

Um den per zip-Datei bereitgestellten Code auszuführen erst den Server starten und anschließend den Observer, es öffnet sich dann ein „Observer“ Fenster das die Simulation beobachtet.

Quellen A2

- Java Standard: RMI – Wikibooks, Sammlung freier Lehr-, Sach- und Fachbücher. (n.d.). https://de.wikibooks.org/wiki/Java_Standard:_RMI
- JFrame | ScalingBits. (n.d.). http://www.scalingbits.com/java/javakurs1/begleitend/javaapi/swing/jframe
- GitHub Copilot für Fehleranalyse, Refactoring und Generierung von Boilerplate-Code