Ergebnisbericht SA4E Übung 1 (Marvin Mokelke)

Präambel

Zur Realisierung der Aufgaben habe ich Python und gRPC verwendet.

Durch ihr dichotomes Weltbild kennen meine Glühwürmchen nur die beiden Zustände leuchten und nicht leuchten…

Ich hatte in (wahrscheinlich falscher) Erinnerung, dass Sie in der ersten Vorlesung eine Demo präsentiert haben, bei denen die Glühwürmchen ebenfalls nur kurz aufgeblinkt sind. Im Nachhinein muss ich zugeben, dass man die Aufgabenstellung auch hätte anders interpretieren können (oszillierend leuchten).

Ich würde argumentieren, dass das eigentliche Ziel Remote Procedure Calls kennenzulernen trotzdem erreicht wurde. 😊

Die Videos zu den Lösungen habe ich über den Upload-Link, den Sie mir per Discord geschickt haben, in Ihre Dropbox geladen. (s4mamoke@uni-trier.de) (Discord: MM)

Aufgabe 1

Das Programm besteht aus den 2 Hauptkomponenten: der Firefly- und der FireflySimulation-Klasse. Eine Instanz der Firefly-Klasse repräsentiert dabei ein einzelnes Glühwürmchen und hat dabei Informationen zu seiner eigenen Frequenz (*flash_interval*) sowie ein Array, das Referenzen auf die Nachbarn beinhaltet (*nb*). Die eigene initiale Frequenz wird bei der Erstellung auf eine zufällige Dauer zwischen 0.5 und 1.5 Sekunden gesetzt. Durch die Methode *update_flash_interval()* wird die eigene Frequenz dann mithilfe des Kuramoto-Modells mit denen der Nachbarn synchronisiert.

```
def update_flash_interval(self):
    # Mittlere Frequenz der Nachbarn berechnen und allmählich anpassen
    neighbor_intervals = [neighbor.flash_interval for neighbor in self.nb]
    average_interval = sum(neighbor_intervals) / len(neighbor_intervals)
    # Anpassung der eigenen Frequenz an die der Nachbarn
    self.flash_interval += self.adjustment_rate* (average_interval - self.flash_interval)
```

Außerdem besitzt die Firefly-Klasse zwei Klassenvariablen: *start_time* und das *sync_event*. Diese stellen sicher, dass alle erzeugten Glühwürmchen zum gleichen Zeitpunkt mit der Simulation starten und Zugang zur Startzeit der Simulation haben, welche zur Berechnung des nächsten Aufleuchtens benötigt wird.

```python
def run(self):

    # Wait for all fireflies to be ready
    Firefly.sync_event.wait()

    while True:
        # Calculate time since start
        current_time = time.time() - Firefly.start_time
        # Wait until next interval
        # Calculate next_flash as follows:
        # 1. Determine how many flash intervals have passed since start (integer division)
        # 2. Add 1 for the next flash interval
        # 3. Multiply by the flash interval to get the next flash time
        next_flash = ((current_time // self.flash_interval) + 1) * self.flash_interval
        time.sleep(max(0, next_flash - (time.time() - Firefly.start_time)))

        self.flash()
        self.update_flash_interval()
```

Alle Fireflies warten nach Erstellung, auf Auslösen des *sync_event* und starten dann in einen Loop aus warten, aufleuchten & nächste Wartezeit bestimmen.

Die Klasse FireflySimulation erstellt dann die Instanzen der Fireflies und bestimmt die Nachbarn (wie im Screenshot zu sehen).

```python
# Nachbarn der Glühwürmchen festlegen
for x in range(n_rows):
    for y in range(n_columns):
        self.grid[x][y].nb = [
        self.grid[(x - 1) % n_rows][y],
        self.grid[(x + 1) % n_rows][y],
        self.grid[x][(y - 1) % n_columns],
        self.grid[x][(y + 1) % n_columns]
        ]
```

Anschließend werden die Threads der Fireflies gestartet. Diese warten zunächst darauf, dass das sync_event ausgelöst wird. Sobald dies geschieht, beginnen alle Fireflies gleichzeitig.

In der main-Methode werden dann einige Parameter wie zum Beispiel Anzahl Reihen, Anzahl Spalten oder Farbe über die Konsole erfragt, eingelesen und dann an eine Instanz der FireflySimulation weitergegeben.

Mit Hilfe moderner Programmierunterstützung ist mir das Erstellen der „Grundarchitektur“ nicht sonderlich schwergefallen. Allerdings trat das Problem auf, dass sich die Leuchtfrequenzen zwar angepasst haben, die Würmchen jedoch durch den Offset (bedingt durch die vorangegangene Blinkhistorie) nicht synchron geblinkt

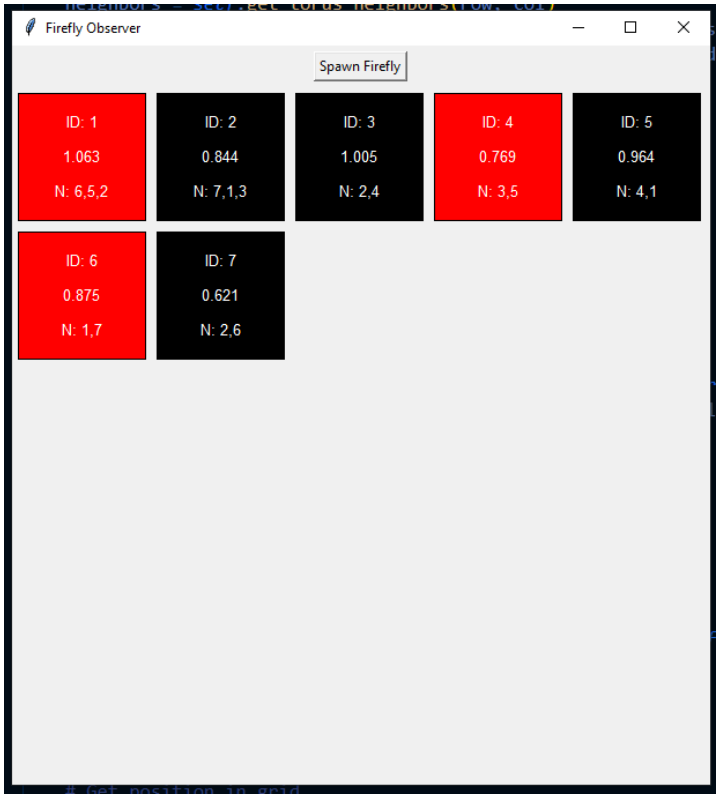
haben. Mit der Implementierung von *next_flash* (siehe Screenshot 2) und dem *sync_event* konnte ich dieses Problem aber nach einigem grübeln beheben.

Aufgabe 2

In Aufgabe 2 habe ich mit einer einfachen Server-Client Architektur angefangen und dann in 3 Etappen auf die Peer-to-Peer Kommunikation zwischen den Glühwürmchen umgestellt.

1. Observer als Server, der von den Fireflies via gRPC deren aktuelle Frequenz erhält und mit der Durschnittsfrequenz aller Fireflies antwortet.
2. Observer als Server, der von den Fireflies via gRPC deren aktuelle Frequenz erhält und mit der Durschnittsfrequenz der benachbarten Fireflies antwortet.
3. Observer und Fireflies als Server. Observer teilt Fireflies die Adressen der Nachbar-Fireflies mit und diese kommunizieren untereinander.

Aufgrund der erhöhten Komplexität gehe ich weniger konkret auf einzelne Variablen ein, sondern versuche im Weiteren eher die Architektur meiner Lösung zu beschreiben.



```
// Observer service for registration and status updates
service ObserverService {
    rpc Register (RegistrationRequest) returns (RegistrationResponse) {}
    rpc UpdateStatus (StatusUpdate) returns (Empty) {}
}

// Firefly service for neighbor communication
service FireflyService {
    rpc ReportFlashInterval (FlashIntervalRequest) returns (Empty) {}
    rpc UpdateNeighbors (NeighborUpdate) returns (Empty) {}
}
```

Observer.py:

Über einen Button im Observer GUI lassen sich neue Fireflies erstellen. Nach deren Erstellung rufen diese über gRPC die Observer-Methode **Register** auf, woraufhin der Observer eine Visualisierung des Glühwürmchens im GUI anlegt und ihm die Adressen seiner Nachbarn mitteilt. **UpdateStatus** verwendet das Glühwürmchen, um dem Observer bei jedem Aufleuchten seine aktuelle Frequenz mitzuteilen.

Firefly.py:

Wird ein neues Würmchen über den Button generiert, werden die in der Nachbarschaft liegenden Würmchen über die **UpdateNeighbors**-Methode informiert. **ReportFlashInterval** nutzen die Würmchen, um sich gegenseitig über Ihre Frequenzen zu informieren. Alle Würmchen besitzen ein Array *neighbor_intervals*, in dem alle Frequenzen der benachbarten Würmchen gespeichert sind. Bei jedem Aufleuchten wird zuerst die eigene Frequenz mithilfe von *neighbor_intervals* angepasst und dann die aktuelle Frequenz an alle Nachbarn gesendet, die mit dieser Information ihr *neighbor_intervals* aktualisieren.

```
# Berechne absolute Zeit seit Start
current_time = time.time()
elapsed_time = current_time - self.observer_start_time

# Berechne nächsten Blinkzeitpunkt
current_phase = elapsed_time % self.flash_interval
next_flash_delay = self.flash_interval - current_phase

# Passe flash_interval basierend auf Nachbarn an
if self.neighbor_intervals:
    avg_interval = sum(self.neighbor_intervals.values()) / len(self.neighbor_intervals)
    adjustment = (avg_interval - self.flash_interval) * 0.1  # 10% Anpassungsrate
    self.flash_interval += adjustment

# Präzises Warten bis zum nächsten Blinken
time.sleep(next_flash_delay)
```

Es entstand wieder ein ähnliches Problem mit dem synchronen Blinken wie in Aufgabe eins. Diesmal bekommen die Würmchen auf Ihre Registrierungsanfrage beim Observer die Startzeit übermittelt und berechnen anhand dieser den nächsten Blinkzeitpunkt. Dabei kommt es besonders oft am Anfang, wenn die Adjustments noch groß sind, zu Fällen, in denen ein Würmchen zweit oder mehrmals schnell hintereinander blinkt. Dies liegt an der Modulo Operation. Ist die *elapsed_time* nur knapp unterhalb eines Vielfachen des *flash_interval*, so ist der *next_flash_delay* klein und das Würmchen blink direkt nochmal. Mit zunehmender Angleichung der Frequenzen nimmt dieses Phänomen ab und es wird trotz Latenz im Takt geblinkt… 😊