Jakob Schouren

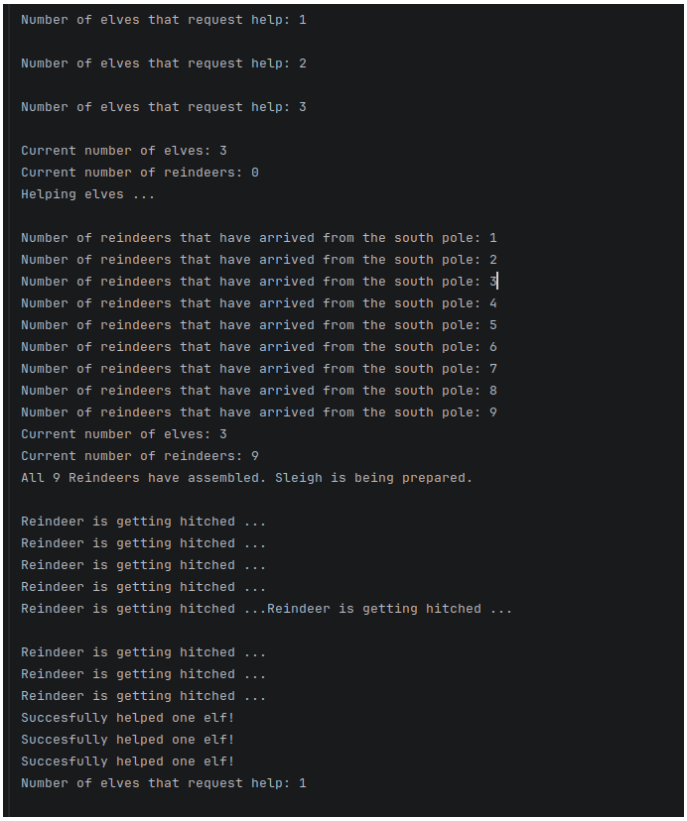
Betriebssysteme, Übungsblatt 2, Winter 2025

Für die Umsetzung beider Aufgaben wurde Ubuntu 24.04.3 LTS als Betriebssystem sowie PyCharm als IDE verwendet. Der Programmcode wurde entsprechend in Python verfasst.

Aufgabe 1: „Santa Claus Problem“ – Semaphor-basierte Implementierung

Analog zu der Umsetzung des Problems im Buch „The Little Book of Semaphores“ wurden für die Implementierung der ersten Aufgabe insgesamt vier Semaphore verwendet, drei davon für Santa, die Rentiere und die Elfen sowie ein Semaphor für Mutual Exclusion. Semaphore sind eine Datenstruktur für die koordinierte Ausführung mehrerer Prozesse/Threads auf beschränkten Ressourcen. Alle Semaphore wurden dabei entweder mit dem Wert 0 (Santa, Rentiere) oder 1 (Elfen, Mutex) initialisiert. Neben den Semaphoren wurde zudem eine Klasse „ChristmasSync“ eingeführt, um die Anzahl an Elfen und Rentieren zentralisiert zu definieren und mithilfe von Getter/Setter-Funktionen auch global zu verwalten. In der Main-Methode werden ein Thread für Santa gestartet sowie jeweils eine Liste von Elf Elfen-Threads und Neun Rentier-Threads. Jeder Thread erhält die „ChristmasSync“-Klasse bei der Initialisierung als Argument. Um die Logik für Santa, Elfen und Rentiere umzusetzen, wurde jeweils eine Funktion (*run_santa(), run_elves(), run_reindeers()*) definiert, die eine Endlosschleife durchläuft, wodurch eine länger anhaltende Kommunikation simuliert werden soll (mehr als nur ein Weihnachten). *run_reindeers()* nimmt zuerst das Mutex auf und erhöht den globalen Rentier-Counter um Eins. Sobald der Rentier-Counter Neun erreicht, wird das Santa-Semaphor auch das Mutex freigegeben. Um die eigene Endlosschleife nun durchlaufen zu können, muss Santa noch auf das Mutex warten. Anschließend wird bei Neun Rentieren die Hilfs-Funktion prepare_sleigh() aufgerufen sowie das Rentier-Semaphor Neun mal releast. Entsprechend wird auch der globale Rentier-Counter verringert. Das ist notwendig, um jedes Rentier abarbeiten zu können und auch den Zustand konsistent zu halten. Jedes Rentier nimmt nun das Rentier-Semaphor auf und führt die Hilfs-Funktion get_hitched() aus. Anders als die Rentiere, müssen Elfen zu Beginn der Endlosschleife nicht nur das Mutex aufnehmen, sondern auch das Elfen-Semaphor, welches zu Beginn einen Wert von Eins innehat. Analog zu den Rentieren wird der Elfen-Counter um Eins erhöht und sobald dieser einen Wert von drei erreicht, wird das Santa-Semaphor freigegeben. Falls der Counter noch kleiner als drei ist, werden das Elfen-Semaphor sowie das Mutex wieder freigegeben, um anderen Elfen die wieder die Möglichkeit zu geben, ein Problem vorzubringen. Zudem wird über *get_help()* die aktuelle Anzahl wartender Elfen ausgegeben. In *run_santa()* wird anschließend für einen Elfen-Counter von Drei oder höher die Hilfs-Funktion *help_elves()* aufgerufen. Da dieser Abgleich nach der zuvor angeführten Logik für die Rentiere stattfindet, haben die Rentiere immer Vorrang vor den Elfen. Sind alle Rentiere zurückgekehrt, müssen die Elfen bis nach Weihnachten auf ihr Anliegen warten. In *run_elves()* wird nachfolgend ein zweites Mal das Mutex aufgenommen und den Elfen wird einzeln geholfen. Für jedes gelöste Problem wird der Elfen-Counter um Eins verringert, bis er einen Wert von Null erreicht. Erst dann wird das zuvor noch blockierte Elfen-Semaphor wieder freigegeben. Um im Rahmen der semaphor- und threadbasierten Umsetzung eine Starvation der Elfen/Rentiere zu vermeiden und die Kommunikation möglichst „realistisch“ zu gestalten, wurden an geeigneten Stellen time.sleep()-Befehle integriert. Diese bewirken ein temporäres Pausieren des aktuellen Threads, wodurch anderen Threads die Möglichkeit gegeben wird, ihren eigenen Code auszuführen. Es wird also

ein „künstlicher“ Kontextwechsel erzwungen. Da sich alle Funktionen im gleichen Programm abspielen, findet keine „echte“ Kommunikation statt. Wie man in der folgenden Graphik erkennen kann, werden die Arbeitsschritte in logisch korrekter Reihenfolge ausgeführt (Drei Elfen haben ein Problem, aber bevor ihnen geholfen werden kann, kommen die 9 Rentiere an und die Elfen müssen dementsprechend noch warten, bis die Rentiere wieder zum Südpol verschwunden sind).



Aufgabe 2: Implementierung mittels Docker-Container und ZeroMQ

Um das Santa-Claus-Problem über 1+r+e Docker-Container abzubilden, wird das ursprüngliche Programm aus Aufgabe 1 in drei Teilprogramme aufgeteilt, welche die entsprechende Logik für Santa, die Rentiere sowie Elfen enthalten. Die zuvor für die Abbildung der einzelnen Entitäten verwendeten Threads werden hier über einzelne Docker-Container realisiert, wobei die Gesamtheit aller Container über den Befehl *docker compose up –build* in der Kommandozeile gestartet werden kann. Dafür werden einerseits ein *Dockerfile* und andererseits eine *docker-compose.yaml* verwendet, welche für Dependencies, die Anzahl der Container sowie den Start jener Container verantwortlich sind.

Dockerfile:

Das Dockerfile orientiert sich dabei sehr stark an dem Beispiel, das im offiziellen Python-Repository des DockerHubs bereitgestellt wird und gibt für jeden Container vor, wie er gebaut werden soll. In diesem Kontext wird *python:3* als Base-Image verwendet. Zudem wird in jedem Container ein neues Arbeitsverzeichnis erstellt, in das eine requirements.txt-Datei kopiert und anschließend alle enthaltenen Dependencies mit *pip* installiert werden. Im Fall dieser Aufgabe

handelt es sich lediglich um *pyzmq*, da diese Library benötigt wird, um die Kommunikation der Container über ZeroMQ zu realisieren.

docker-compose.yaml:

In der Datei docker-compose.yaml werden die Container orchestriert und gebaut. Dafür werden drei Services – jeweils einer für Santa, die Rentiere und die Elfen – definiert. Über den Ausdruck *build:* . wird jedem dieser Services mitgeteilt, dass er für den Build im aktuellen Arbeitsverzeichnis nach dem Dockerfile suchen soll. Alle Container haben dadurch die gleiche Grundstruktur. Worin sich die drei Services jedoch unterscheiden, ist der Command der im Rahmen des Builds ausgeführt wird. Für Santa wird *santa.py* ausgeführt, für die Rentiere *reindeer.py* und für die Elfen *elf.py*. Alle Teilprogramme laufen mit der Flag *-u*, wodurch evtl. Ausgaben in stdout und stderr nicht erst in einen Buffer geschrieben werden, sondern sofort in der Konsole angezeigt werden. Da im Santa-Claus-Problem mehrere Rentiere und Elfen vorhanden sind und diese nicht mehr über Threads abgebildet werden, müssen für den Rentier- und Elfen-Service noch eine beliebige Anzahl an *replicas* definiert werden. Im Falle der vorliegenden Implementierung sind das Neun Rentier-Replicas und Acht Elfen-Replicas, was sich in einer entsprechenden Anzahl an Containern widerspiegelt. Bei der Ausführung von *docker compose up –build* wird im aktuellen Arbeitsverzeichnis nach einer Datei namens docker-compose.yaml gesucht. Sollte die Datei nicht im aktuellen Arbeitsverzeichnis liegen, kann ihr über die *-f* Flag explizit angegeben werden.

reindeer.py:

Jedes der Teilprogramme startet, indem ein ZeroMQ-Kontext initialisiert wird. Außerdem wird für jedes Rentier anhand des Hostnames des Docker-Containers eine einzigartige ID zugewiesen. Diese wird im Anschluss für die Identifikation der Rentiere im Rahmen der Kommunikation genutzt. Da für Letzteres das Dealer-Router-Pattern aus ZeroMQ zum Einsatz kommt, wird jedem Rentier mit *context.socket(zmq.DEALER)* die Rolle eines Dealers zugeteilt. Das Dealer-Router-Pattern erlaubt ein flexibles Routing von Nachrichten zwischen einer beliebigen Anzahl von Dealern und einem Router. In dieser Implementierung übernimmt Santa die Rolle des zentralen Routers und die Rentiere sowie Elfen agieren als Dealer. Jeder Dealer verbindet sich mit dem TCP-Port 2222 von Santa. Wie auch schon in Aufgabe 1 wird in jedem Rentier-Container eine Endlosschleife durchlaufen. Um ein mehrfaches Senden des gleichen Rentiers zu vermeiden, wird eine boolesche Variable *sent* eingeführt. Falls diese einen Wert von False hat, sendet der Dealer mit *send_multipart()* eine Nachricht an Santa in der Form *list(bytes)*. Diese Liste enthält ein json-Objekt mit dem Key-Value-Paar *„type“: „reindeer“* und einem zweiten Key-Value-Paar *„id“: „rID“*, welches als Bytes nach UTF-8 kodiert werden. Dadurch erfährt Santa einerseits, ob es sich um ein Rentier/einen Elfen handelt, andererseits kann er das jeweilige Rentier aber auch genau identifizieren. Anschließend wird die sent-Flag auf True gesetzt. Nachfolgend wartet der jeweilige Container in jedem Schleifendurchlauf auf eine Antwort des Santa-Services und sobald alle Neun Rentiere eingetroffen sind, erhält er eine Nachricht mit dem Key-Value-Paar *„action“: „reindeer_go“* erhält, sendet er eine Nachricht zur Bestätigung an Santa zurück, die das Key-Value-Paar *„type“: „ackReindeer“*, aber auch wieder Rentier-ID als auch die aktuelle Sequenznummer. Nach dem Ausliefern der Geschenke ruhen die Rentiere eine kurze Weile von Fünf Sekunden, was über *time.sleep()* abgebildet wird. Danach sind die Rentiere wieder fit und können für das darauffolgende Weihnachten erneut ihre Ankunft vom Südpol bei Santa ankündigen.

elf.py:

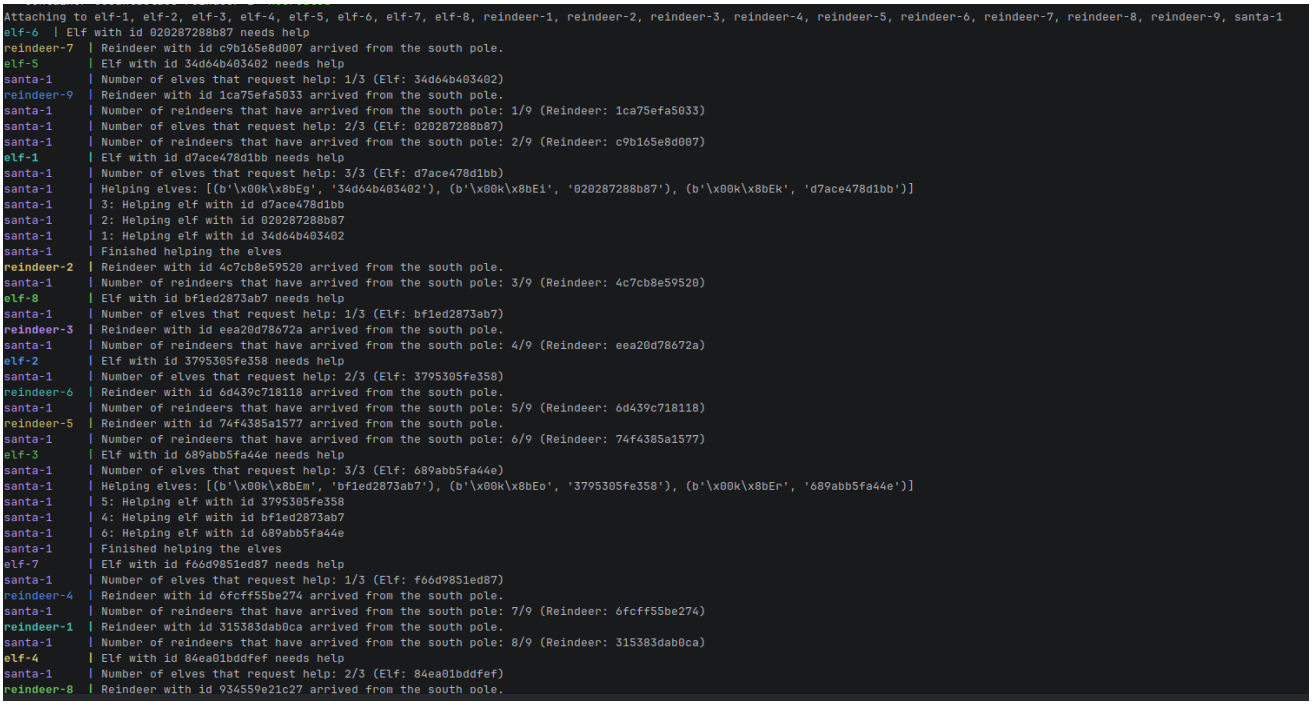
Analog zur Umsetzung der Rentiere, agieren auch die Elfen als Dealer und connecten zum TCP-Port 2222 von Santa. Ebenfalls ist die jeweilige ID des Elfen gleich dem Hostname des Docker-Containers. Die Nachricht, die der Dealer sendet, hat in diesem Falle aber nicht mehr das Key-Value-Paar *„type“: „reindeer“*, sondern *„type“: „elf“* und als id, die jeweilige Elfen-ID. Sobald drei Elfen Hilfe bei Santa angefordert haben, erhalten die Elfen eine Nachricht mit dem Key-Value-Paar *„type“: „elves_help“*, woraufhin erneut eine Bestätigungsnachricht an Santa gesendet wird, die das Key-Value-Paar „type“: „ackElf“ enthält, sowie die aktuelle Sequenznummer. Auch die Elfen ruhen sich über *time.sleep()* anschließend Fünf Sekunden aus, bevor sie sich neuen Problemen widmen.

santa.py:

Santa übernimmt in der Implementierung die Rolle des Routers. Um die die Nachrichten der anderen Container aufnehmen zu können binded er seinen Kontext auf den eigenen TCP-Port 2222, auf den die verschiedenen Elfen und Rentiere connecten. Anschließend werden für die ankommenden Elfen und Rentiere zwei Queues initialisiert, um die Reihenfolge der Abarbeitung gemäß dem Zeitpunkt des Eintreffens zu gewährleisten. Queues eignen sich hierzu, da sie nach dem FIFO-Prinzip agieren. Zusätzlich wird für zusätzliche Sichtbarkeit eine Sequenznummer eingeführt sowie Counter für die Bestätigungsnachrichten der Elfen/Rentiere. Des Weiteren werden zwei boolesche Variablen eingeführt, die dafür sorgen, dass Santa nicht gestört wird, während er die Rentiere vorbereitet bzw. den Elfen hilft. In der Endlosschleife wird nun zu Beginn immer der *„type“* der Nachricht abgefragt. Wie bereits in *reindeer.py* und *elf.py* angeführt, gibt es die Möglichkeiten *„reindeer“, „elf“, „ackReindeer“* und *„ackElf“*. Handelt es sich um einen der ersten beiden, werden die IDs der Elfen/Rentiere in die jeweilige Queue hinzugefügt. Handelt es sich um eine Bestätigungsnachricht, wird der entsprechende Counter hochgezählt. Falls gerade keine Rentiere vorbereitet werden und die Anzahl der in der Queue wartenden Rentiere Neun betägt, wird die boolesche Variable auf True gesetzt und der Counter für die Bestätigungsnachrichten der Rentiere auf Null. Nachfolgend werden die Neun nächsten Objekte in der Rentier-Queue „gepopt“ und an jedes dieser Rentiere wird eine Nachricht mit dem Key-Value-Paar *„action“: „reindeer_go“* sowie der entsprechenden Container-ID und der aktuellen Sequenznummer geschickt. Letztere wird bei jeder Nachricht um Eins hochgezählt. Anhand einer Kommunikations-/Session-ID für alle Dealer, die Santa beim Empfangen der Nachrichten zusammen mit der Elfen-/Rentier-ID gespeichert hat, erreichen alle Nachrichten den korrekten Empfänger. Die entsprechende ID ist immer der erste Frame in der Liste von Bytes, die mit *send_multipart()* übermittelt wird. Im Anschluss wartet Santa, bis alle Rentiere sich zurückgemeldet haben, der Bestätigungscounter also 9 erreicht hat. Danach wird die boolesche Variable wieder freigegeben und die Rentiere kehren zum Südpol zurück. Genau die gleiche Logik wird nachfolgend ebenfalls für die Elfen angewendet. Allerdings müssen hier nur Drei Elfen zusammenkommen. Da die Logik zur Abarbeitung der Elfen auf Logik für die Rentiere folgt, haben Rentiere gemäß dem Santa-Claus-Problem immer Vorrang vor den Elfen.

Wie man in der nachfolgenden Graphik erkennen kann, werden die Elfen/Rentiere in korrekter Reihenfolge von Santa abgearbeitet und auch die Reihenfolge der Gruppen ist korrekt (Gruppe mit Sequenzen 4,5 und 6 erfolgt nach Sequenzen 1, 2 und 3), jedoch melden sich die Container der Elfen/Rentiere unterschiedlich schnell bei Santa zurück, nachdem sie von Santa bearbeitet wurden, was dazu führt, dass die Ausgabe der Print-Befehle für die erfolgte Hilfe eine

unterschiedliche Reihenfolge hat, die chronologisch nicht korrekt ist. Des Weiteren kommt es bei der vorliegenden Implementierung ohne die „Ruhephase“ der Rentiere/Elfen nach der erfolgreichen Abarbeitung durch Santa zu Inkonsistenzen bzw. einer Starvation der Rentiere zugunsten der Elfen, da die betroffenen Rentiere/Elfen im nächsten Schleifendurchlauf sofort wieder neue Anfragen stellen und es nur Drei Elfen, aber Neun Rentiere für eine Bearbeitung durch den Weihnachtsmann braucht. Eine letzte Auffälligkeit ist bereits zu Beginn in der Graphik zu sehen. Obwohl der Print-Befehl in Container *elf-6* vor *elf-5* ausgeführt wurde, hat die Nachricht von *elf-5* Santa früher erreicht, wodurch er vor *elf-6* in der Queue steht und ihm somit früher geholfen wird. Letztlich zählt nur der Zustand, der in Santa vorliegt, auch wenn dieser global betrachtet vielleicht nicht chronologisch korrekt ist.



Zusammenfassend lässt sich sagen, dass Docker-Container in Kombination mit ZeroMQ besser für die Umsetzung dieses Synchronisations-Problems geeignet sind als mehrere Threads in einem einzelnen Programm, aber auch hier zusätzliche Annahmen getroffen werden müssen, um eine korrekte Ausführung zu gewährleisten.