1 Einleitung

Nebenläufigkeit beschreibt in der Informatik die Eigenschaft eines Softwaresystems, mehrere Aufgaben, Berechnungen oder Anweisungen gleichzeitig ausführen zu können. In solchen Systemen können sogenannte Synchronisationsprobleme, sobald mehrere Ausführungseinheiten auf gemeinsame Ressourcen zugreifen oder bestimmte Ereignisse in einer festgelegten Reihenfolge stattfinden müssen. Ziel von Synchronisation ist es, solche Abhängigkeiten korrekt durchzusetzen und dabei Fehler wie Race Conditions, Deadlocks oder unnötige Blockierungen zu vermeiden.

Eine Einführung in diese Problemklasse sowie klassische Lösungsansätze finden sich im *Little Book of Semaphores* von Allen B. Downey.[1] Zur Veranschaulichung dieser Konzepte stellt Downey anschauliche abstrahierte Szenarien vor, die reale Synchronisationsanforderungen modellhaft abbilden. Neben den gängigen Problemen wie bspw. das *Producer Consumer Problem* ist ein weiteres solches Beispiel ist das *Santa Claus Problem*:

Das Problem modelliert die Interaktion zwischen drei Arten von Akteuren: einem Weihnachtsmann, einer festen Anzahl von Rentieren sowie einer Gruppe von Elfen. Die Rentiere kehren einmal jährlich rechtzeitig zum Nordpol zurück, damit der Weihnachtsmann mit der Auslieferung der Geschenke beginnen kann. Der Weihnachtsmann darf jedoch erst dann aktiv werden, wenn alle Rentiere anwesend sind. Parallel dazu arbeiten die Elfen kontinuierlich an der Produktion der Geschenke. Treten dabei Probleme auf, dürfen die Elfen den Weihnachtsmann nicht einzeln stören, sondern erst dann, wenn mindestens eine vorgegebene Anzahl von Elfen gleichzeitig Hilfe benötigt. In diesem Fall kümmert sich der Weihnachtsmann nacheinander um die wartenden Elfen und kehrt anschließend in seinen Ruhezustand zurück.

In der von Downey vorgeschlagenen Lösung fehlten mir die tatsächliche Auslieferung der Geschenke und die damit verbundenen Vorbereitungen, weshalb ich das Problem in meiner Implementierung bewusst erweitert habe: *Nach der Rückkehr aller Rentiere wird nicht unmittelbar mit Weihnachten begonnen, sondern zunächst der Schlitten vorbereitet und alle Rentiere einzeln angespannt. Erst wenn diese Vorbereitungen vollständig abgeschlossen sind, wird der Beginn der Geschenkverteilung ausgelöst. Während dieser gesamten Phase widmet sich der Weihnachtsmann ausschließlich den Weihnachtsvorbereitungen und eine Betreuung von Elfen findet in diesem Zeitraum nicht statt.*

Ziel dieser Arbeit ist die praktische Umsetzung des Santa-Claus-Problems in zwei unterschiedlichen Ausprägungen. In der ersten Implementierung wird das Problem mithilfe von Threads modelliert, wobei die Synchronisation ausschließlich über Semaphore erfolgt. Diese Lösung orientiert sich eng an der im *Little Book of Semaphores* beschriebenen Vorgehensweise, wird aber durch meine eigene Erweiterung des Problems ergänzt, und dient als Referenz für das grundlegende Synchronisationsverhalten.

Die zweite Implementierung überträgt das Problem in eine verteilte Umgebung. Hier wird jeder beteiligte Akteur durch einen eigenen Docker-Container repräsentiert. Die Synchronisation erfolgt nicht mehr über gemeinsame Speicherstrukturen, sondern nachrichtenbasiert mithilfe von ZeroMQ. Die Container werden dabei automatisiert orchestriert, sodass die Simulation über einen einzigen Startbefehl ausgeführt werden kann.

[1] Allen B Downey. *The little book of semaphores*. Green Tea Press, 2016.

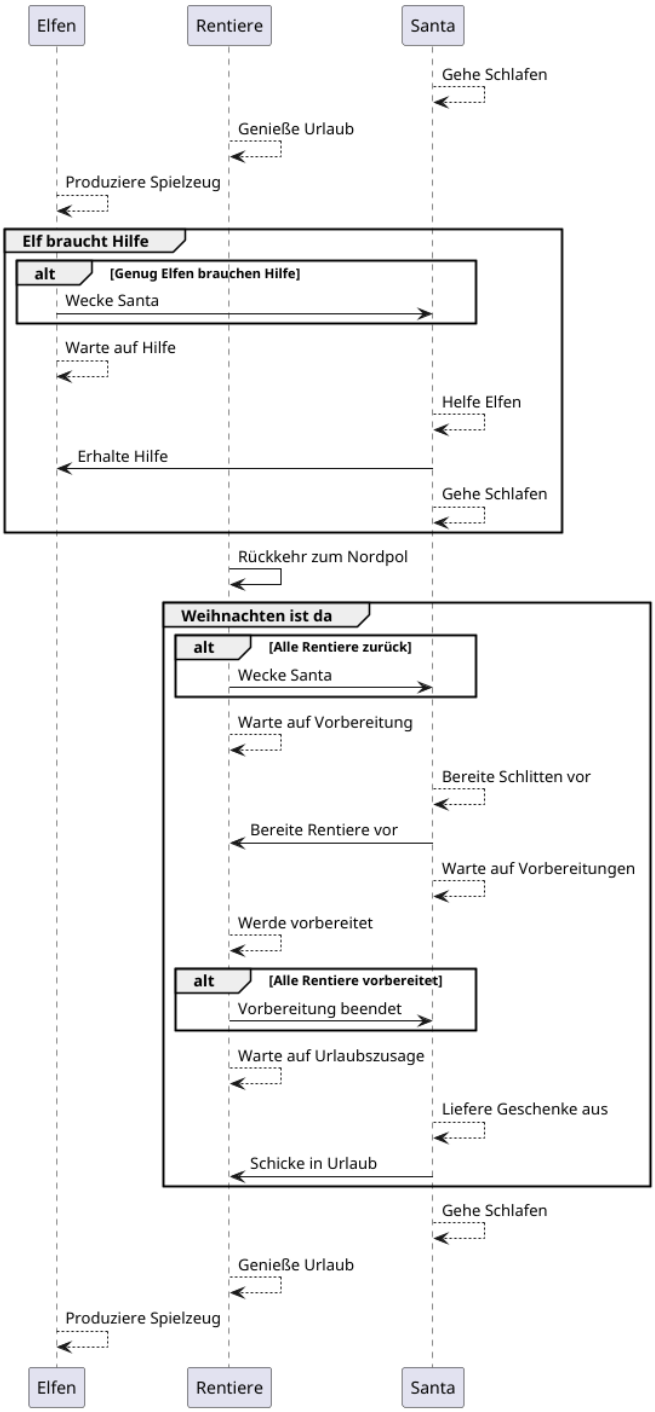

Abbildung 1: Vereinfachtes UML-Sequenzdiagramm, welches die Interaktionen zwischen den Elfen, Rentieren und dem Weihnachtsmann veranschaulicht.

2 Implementierung mit Threads

Die erste Lösung des Problems habe ich so umgesetzt, dass alle beteiligten Akteure jeweils als eigenständiger Thread modelliert werden. Obwohl es sich konzeptionell um Threads handelt, habe ich auf die Verwendung von Pythons `threading`-Bibliothek verzichtet und stattdessen das `multiprocessing`-Modul eingesetzt. Ausschlaggebend für diese Entscheidung war der Global Interpreter Lock (GIL), der in dem Standard-Modul von Python

verhindert, dass mehrere Threads gleichzeitig Bytecode ausführen.[2] Weil das behandelte Problem ein echtes Synchronisationsproblem darstellt, hätte eine solche Umsetzung die Parallelität künstlich eingeschchränkt und damit das beobachtbare Verhalten verfälscht.

Durch die Verwendung von `multiprocessing` wird jeder Akteur als eigener Betriebssystemprozess ausgeführt. Diese Prozesse werden beim Start durch in dieser Implementierung durch Forking des Hauptprozesses erzeugt. Jeder Prozess verfügt dadurch über einen eigenen Python-Interpreter und umgeht dadurch den GIL vollständig. Da Prozesse keinen implizit gemeinsamen Speicher besitzen, werden gemeinsam genutzte Zustände nicht automatisch geteilt. Stattdessen stellt das `multiprocessing`-Modul eigene Mechanismen bereit, um Synchronisation und Datenaustausch sinnvoll zu ermöglichen. Hierzu zählen unter anderem Semaphore sowie explizit durch Shared-Memory geteilte Datenobjekte.

Die gesamte Koordination zwischen den Prozessen von dieser Implementierung erfolgt ausschließlich über Semaphore. Kritische Abschnitte, in denen gemeinsam genutzte Zustandsvariablen verändert werden, sind durch einen globalen Mutex-Semaphore geschützt, um Race Conditions zu vermeiden. Zusätzlich werden ereignisbasierte Semaphore eingesetzt, um Prozesse gezielt zu blockieren oder freizugeben. Zur Strukturierung der Synchronisationslogik wurde eine eigene `HR`-Klasse eingeführt. Diese fungiert als zentrale Koordinationsinstanz und kapselt sämtliche gemeinsam genutzten Semaphore sowie alle relevanten Zustandsvariablen, wie die Anzahl wartender Elfen oder zurückgekehrter und vorbereiteter Rentiere. Die Einführung dieser Klasse vermeidet direkte Abhängigkeiten zwischen den Prozessen und führt eine ausdefinierte gemeinsame Schnittstelle ein.

2.1 Elfen

Zunächst produziert jeder Elf selbstständig Spielzeug und wartet anschließend für eine zufällige Zeit. Diese Phase stellt den Normalbetrieb dar und erfordert keine Synchronisation mit anderen Akteuren. Mit einer festgelegten Wahrscheinlichkeit benötigt ein Elf Hilfe vom Weihnachtsmann. In diesem Fall betritt er einen kritischen Abschnitt, der durch das globale Mutex-Semaphore `glob_mutex` geschützt ist, und erhöht dort die gemeinsame Zustandsvariable `elves_needing_help`. Erreicht dieser Zähler exakt die konfigurierte Toleranzgrenze `problem_tolerance`, weckt der entsprechende Elf den Weihnachtsmann das freigeben des Semaphores `santa_sem` auf. Nach dieser Anmeldung blockiert der Elf auf dem Semaphore `help_elves_sem` und wartet darauf, vom Weihnachtsmann Hilfe zu erhalten. Erst wenn Santa dieses Semaphore freigibt, wird der Elf entblockt. Anschließend reduziert er, erneut geschützt durch `glob_mutex`, den Zähler `elves_needing_help` und kehrt in die Produktionsschleife zurück.

[2] Pascal Lindner. *Multithreading und Multiprocessing in Python*. Techn. Ber. Universität Hamburg, Fakultät für Mathematik, Informatik und Naturwissenschaften, Fachbereich Informatik, Arbeitsbereich Wissenschaftliches Rechnen. URL: `https://wr.informatik.uni-hamburg.de/_media/teaching/sommersemester_2019/pih-19-lindner-ausarbeitung.pdf`.

2.2 Rentiere

Zu Beginn wartet jedes Rentier blockierend auf dem Semaphore `holiday_approval_sem`. Erst wenn dieses Semaphore vom Weihnachtsmann freigegeben wird, darf das Rentier seinen Urlaub antreten. Nach der Freigabe verbringt das Rentier eine fest definierte Zeit im Urlaub, die durch eine Wartephase simuliert wird. Anschließend kehrt es zum Nordpol zurück und erhöht in einem durch `glob_mutex` geschützten Abschnitt die gemeinsame Zustandsvariable `returned_reindeers`. Ist das Rentier das letzte zurückgekehrte, signalisiert es dies durch die Freigabe des Semaphores `santa_sem` und weckt damit den Weihnachtsmann. Im Anschluss blockiert jedes Rentier auf dem Semaphore `hitch_reindeer_sem` und wartet darauf, vom Weihnachtsmann angespannt zu werden. Nach dem Anspannen erhöht das Rentier, erneut geschützt durch `glob_mutex`, den Zähler `prepared_reindeers`. Das letzte vorbereitete Rentier gibt schließlich das Semaphore `christmas_sem` frei und signalisiert damit, dass alle Vorbereitungen abgeschlossen sind. Anschließend wartet das Rentier wieder darauf, in den Urlaub gehen zu dürfen.

2.3 Weihnachtsmann

Der Weihnachtsmann befindet sich grundsätzlich in einem Ruhezustand und wartet blockierend auf dem Semaphore `santa_sem`. Dieses Semaphore wird entweder von Elfen freigegeben, sobald `problem_tolerance` Elfen gleichzeitig Hilfe benötigen, oder von den Rentieren, wenn alle aus dem Urlaub zurückgekehrt sind. Nach dem Aufwachen betritt Santa einen durch `glob_mutex` geschützten Abschnitt und entscheidet anhand von `returned_reindeers` und `elves_needing_help`, welche Aufgabe Priorität hat.

Haben alle Rentiere den Nordpol erreicht, so werden die Weihnachtsvorbereitungen priorisiert. In diesem Fall bereitet Santa zunächst den Schlitten vor und gibt anschließend für jedes Rentier einmal das Semaphore `hitch_reindeer_sem` frei, sodass alle Rentiere angespannt werden können. Danach wartet er blockierend auf dem Semaphore `christmas_sem`, bis das letzte Rentier seine Bereitschaft signalisiert hat. Erst dann beginnt die eigentliche Geschenkverteilung. Nach Abschluss der Weihnachtsphase setzt der Weihnachtsmann die Zustandsvariablen `returned_reindeers` und `prepared_reindeers` zurück und gibt für jedes Rentier das Semaphore `holiday_approval_sem` frei, sodass ein neuer Zyklus beginnen kann.

Liegt hingegen keine vollständige Rückkehr aller Rentiere vor und beträgt die Anzahl wartender Elfen mindestens `problem_tolerance`, kümmert sich der Weihnachtsmann um die Elfen. In diesem Fall gibt er das Semaphore `help_elves_sem` genau so oft frei, wie aktuell Elfen Hilfe benötigen. Dadurch werden ausschließlich die wartenden Elfen entblockt und können ihre Arbeit fortsetzen. Anschließend verlässt Santa den kritischen Abschnitt und kehrt erneut in seinen Ruhezustand zurück.

3 Implementierung mit Containern

In der zweiten Implementierung des Santa-Claus-Problems habe ich das System als eine verteilte Anwendung umgesetzt, in der jeder Akteur in einem separaten Docker-Container simuliert wird. Die gesamte Koordination erfolgt nachrichtenbasiert mithilfe von ZeroMQ. Jeder Container startet dasselbe Python-Programm, übernimmt jedoch abhängig von einem übergebenen Kommandozeilenparameter eine unterschiedliche Rolle (Weihnachtsmann, Elf, Rentier oder HR). Die zentrale Orchestrierung der Container erfolgt über `docker-compose`, sodass das vollständige System mit einem einzigen Befehl gestartet werden kann. Die notwendigen Netzwerkverbindungen zwischen den Containern werden dabei automatisch bereitgestellt.

Gemeinsame Zustände, wie die Anzahl wartender Elfen oder zurückgekehrter Rentiere, werden nicht verteilt gespeichert, sondern ausschließlich zentral durch eine eigene HR-Komponente verwaltet. Die HR-Instanz fungiert als Vermittlungsstelle zwischen allen Akteuren. Sie nimmt Bewerbungen entgegen, vergibt eindeutige Identifikatoren, verwaltet den globalen Systemzustand und übersetzt eingehende Ereignisse in entsprechende Anweisungen an den Weihnachtsmann oder Broadcast-Nachrichten an Elfen und Rentiere. Anders als in der ersten Implementierung treffen Elfen, Rentiere und der Weihnachtsmann selbst keine globalen Entscheidungen mehr, sondern reagieren ausschließlich auf konkrete Anweisungen der HR-Abteilung.

ZeroMQ stellt unterschiedliche Kommunikationsschemata zur verfügung, die sich für unterschiedliche Interaktionsformen eignen:

1. **`REQ/REP`** Realisiert eine dialogorientierte Anfrage-Antwort-Kommunikation. Jeder gesendeten Anfrage folgt genau eine Antwort, wodurch ein synchroner und kontrollierter Nachrichtenaustausch ermöglicht wird.

2. **`PUSH/PULL`** Dient der unidirektionalen Übertragung von Nachrichten, bei der Sender und Empfänger zeitlich und logisch entkoppelt sind, sodass Ereignisse asynchron verarbeitet werden können.

3. **`PUB/SUB`** Ermöglicht die Broadcast-basierte Verteilung von Nachrichten an mehrere Empfänger. Publisher senden Nachrichten ohne Kenntnis der Empfänger, während Subscriber alle veröffentlichten Nachrichten erhalten.

In dieser Implementierung werden die Kommunikationsschemata wie folgt eingesetzt: `REQ/REP`-Sockets werden für die Registrierung der Elfen und Rentiere bei der HR-Abteilung sowie für die direkte Kommunikation zwischen HR und dem Weihnachtsmann verwendet. Zustandsmeldungen der Elfen und Rentiere, insbesondere Hilfsanfragen der Elfen und Rückkehrmeldungen der Rentiere, werden über `PUSH`-Sockets an die HR-Abteilung gesendet und dort über einen zentralen `PULL`-Socket entgegengenommen. Anweisungen der HR-Abteilung an Elfen und Rentiere, wie die Freigabe der Hilfe oder das Anspannen der Rentiere, werden über `PUB/SUB`-Sockets als Broadcast verteilt.

3.1 Logging

Zur besseren Nachvollziehbarkeit des Systemverhaltens habe ich zusätzlich ein zentrales, zeitlich geordnetes Logging implementiert. Dazu habe ich eine Hilfsklasse eingeführt und eine angepasste `print`-Methode fzur Verfügung stellt. Dabei wird jede Log-Nachricht mit einem Zeitstempel versehen und entweder lokal gepuffert oder über ZeroMQ an die HR-Abteilung gesendet. Die HR-Abteilung fungiert als zentrale Log-Sammelstelle. Sie betreibt einen `PULL`-Socket, über den alle anderen Akteure ihre Log-Nachrichten per `PUSH`-Socket übermitteln. Eingehende Log-Nachrichten werden in einem thread-sicheren Puffer gesammelt, der als Prioritätswarteschlange nach Zeitstempeln organisiert ist. Dadurch können Nachrichten aus unterschiedlichen Containern korrekt zeitlich eingeordnet werden.

Die eigentliche Ausgabe erfolgt verzögert in einem separaten Thread. In regelmäßigen Abständen werden alle Log-Einträge ausgegeben, deren Zeitstempel eine definierte Verzögerung unterschreiten. Diese Verzögerung stellt sicher, dass auch leicht verspätet eintreffende Nachrichten korrekt einsortiert werden, bevor sie ausgegeben werden. Auf diese Weise entsteht eine konsistente, chronologisch sortierte Log-Ausgabe, obwohl die Ereignisse in unterschiedlichen Prozessen und Containern erzeugt werden.

3.2 Elfen

Jeder Elf wmeldet sich beim Start des Containers zunächst bei der HR-Abteilung, um eine eindeutige Identifikationsnummer zu erhalten. Anschließend beginnt der Elf mit seiner regulären Arbeit, die in der Simulation durch das Produzieren von Spielzeug und zufällige Wartezeiten simmuliert wird.Mit einer festgelegten Wahrscheinlichkeit benötigt ein Elf Hilfe vom Weihnachtsmann. In diesem Fall sendet er eine entsprechende Nachricht an die HR-Abteilung. Der Elf selbst blockiert dabei nicht aktiv, sondern wartet passiv auf eine Broadcast-Nachricht der HR-Abteilung, die signalisiert, dass der Weihnachtsmann Hilfe gewährt. Sobald der Weihnachtsmann seine Sprechstunde eröffnet, sendet die HR-Abteilung eine Broadcast-Nachricht an alle Elfen. Jeder wartende Elf empfängt diese Nachricht, nimmt die Hilfe in Anspruch und setzt seine Arbeit anschließend fort. Eine individuelle Zuordnung zwischen Anfrage und Antwort ist dabei nicht notwendig, da alle wartenden Elfen gleichzeitig freigegeben werden.

Ein Problem hat sich durch die Verwendung eines `PUB/SUB`-Sockets für die Benachrichtigung der Elfen ergeben. Broadcast-Nachrichten werden von ZeroMQ von jedem Subscriber empfangen, der zum Zeitpunkt des Sendens aktiv verbunden ist. Da Elfen kontinuierlich arbeiten und nicht permanent auf Hilfe warten, kann es vorkommen, dass ältere Broadcast-Nachrichten noch im Empfangspuffer eines Elfen liegen. Um sicherzustellen, dass ein Elf ausschließlich auf eine aktuell gültige Hilfszusage reagiert, wird der Empfangspuffer vor dem eigentlichen Warten explizit geleert. Hierzu liest der Elf alle noch vorhandenen Nachrichten nicht-blockierend aus dem `SUB`-Socket, bevor er anschließend blockierend auf eine neue Nachricht wartet. Erst diese nach dem Senden der eigenen Hilfsanfrage empfangene Nachricht wird dann als gültige Hilfe interpretiert.

3.3 Rentiere

Auch die Rentiere bewerben sich beim Start bei der HR-Abteilung, um eine Identifikationsnummer zu erhalten. Zu Beginn befinden sich alle Rentiere im Urlaub, der durch eine feste Wartezeit simuliert wird. Nach Ablauf dieser Zeit kehrt jedes Rentier selbstständig zurück und meldet seine Rückkehr an die HR-Abteilung. Die HR-Instanz zählt die eingehenden Rückmeldungen und informiert den Weihnachtsmann, sobald alle Rentiere zurückgekehrt sind.

Nach der Rückkehr warten die Rentiere auf weitere Anweisungen. Sobald der Weihnachtsmann mit den Weihnachtsvorbereitungen beginnt, sendet die HR-Abteilung eine Broadcast-Nachricht an alle Rentiere, die das Anspannen an den Schlitten signalisiert. Jedes Rentier bestätigt nach dem Anspannen seine Bereitschaft durch eine weitere Nachricht an die HR-Abteilung. Nach Abschluss der Geschenkverteilung erhalten die Rentiere schließlich eine erneute Broadcast-Nachricht, die ihnen den nächsten Urlaub genehmigt. Anschließend beginnt der Zyklus von vorne.

3.4 Weihnachtsmann

Der Weihnachtsmann stellt einen `REP`-Socket bereit, über den er ausschließlich mit der HR-Abteilung kommuniziert. Er befindet sich standardmäßig in einem passiven Wartezustand und reagiert nur auf explizite Anweisungen. Erhält der Weihnachtsmann die Anweisung, den Elfen zu helfen, öffnet er seine Sprechstunde und signalisiert dies der HR-Abteilung. Diese informiert daraufhin alle wartenden Elfen per Broadcast. Sobald alle Rentiere zurückgekehrt sind, wird der Weihnachtsmann angewiesen, die Weihnachtsvorbereitungen zu beginnen. In diesem Fall bereitet er zunächst den Schlitten vor und fordert über die HR-Abteilung das Anspannen der Rentiere an. Anschließend wartet er, bis ihm gemeldet wird, dass alle Rentiere vorbereitet sind, und beginnt dann mit der Geschenkverteilung. Während der Weihnachtsvorbereitungen eingehende Hilfsanfragen von Elfen werden bewusst zurückgestellt. Falls nach Abschluss von Weihnachten noch Hilfe benötigt wird, öffnet der Weihnachtsmann seine Sprechstunde erneut.

3.5 HR-Abteilung

Die HR-Abteilung stellt die zentrale Koordinationsinstanz des verteilten Systems dar. Sie verwaltet alle globalen Zustandsinformationen und ist der einzige Akteur, der den vollständigen Überblick über den aktuellen Systemzustand besitzt. Beim Start nimmt die HR-Abteilung Bewerbungen von Elfen und Rentieren entgegen und vergibt fortlaufende Identifikationsnummern. Darüber hinaus sammelt sie alle Ereignisse der Arbeiter, wie Hilfsanfragen von Elfen oder Rückmeldungen der Rentiere, über einen `PULL`-Socket. Auf Basis dieser Ereignisse entscheidet die HR-Abteilung, wann der Weihnachtsmann informiert werden muss und löst Hilfsanfragen der Elfen erst dann aus, wenn eine definierte Toleranzschwelle erreicht ist. Nach Abschluss der Weihnachtsauslieferung setzt die HR-Abteilung die relevanten Zustände zurück und schickt die Rentiere wieder in den Urlaub.

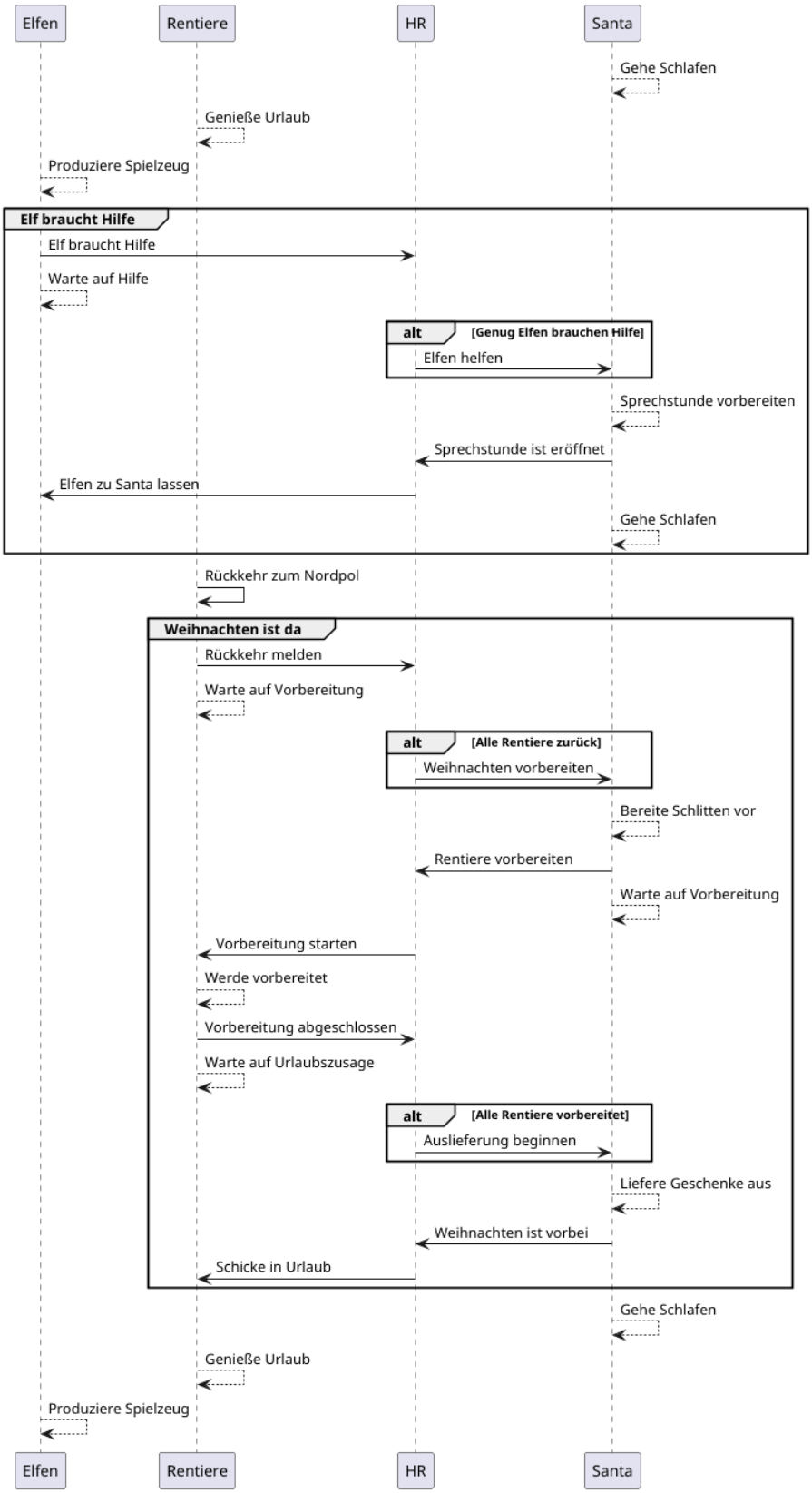

Abbildung 2: Vereinfachtes UML-Sequenzdiagramm, welches die Interaktionen zwischen den Elfen, Rentieren, den Weihnachtsmann und einer HR Abteilung veranschaulicht.