Ergebnisbericht Übungsblatt 1 Verteilte Systeme (SoSe 2025)

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

Verwendete Entwicklungsumgebung

Die Entwicklung fand in PyCharm Professional auf einem MacBook Pro M1 (macOS 15.5) statt. PyCharm lief in einer venv mit bis zu **3 GB** Arbeitsspeicher, um parallele UDP-Prozesse zuverlässig zu unterstützen.

Aufgabe 1 – Ein Feuerwerk an UDP-Nachrichten (Pseudo-Verteilt)

1.1 Implementierung

Das Repository gliedert sich in drei zentrale Skripte + plots:

- **process.py**
  - Enthält die Prozesslogik: Erzeugen und Weiterreichen des Tokens, Multicast-„Feuerwerk“, Halbierung von *p*, Zähler für aufeinanderfolgende Null-Runden und geordneter Shutdown.
  - Schreibt am Ende die gesammelten Kennzahlen (total_rounds, total_fireworks, min/avg/max_round_time) in summary.csv.
- **run_experiments.py**
  - Fährt zunächst die **Fixed Tests** über vordefinierte *(n, p, k)*-Kombinationen und protokolliert alle Resultate in data/results.csv.
  - Führt im Anschluss den **Max-n-Scan (p=0.5, k=3)** durch: grobe Suche ab n_0=140 in Zehnerschritten bis zum ersten Fehlversuch, gefolgt von einem Refinement zwischen letztem Erfolg L und erstem Fehlschlag U. Das finale max_n wird in data/max_n_summary.csv abgelegt.
- **config_utils.py**
  - Fasst alle globalen Konstanten (BASE_PORT, Default-Werte) sowie Logging-Funktionen log(…) und log_stat(…) zusammen.
- **plot_results.py**
  - Lädt data/results.csv mit pandas, erstellt drei Diagramme (Ø Rundenzeit vs. n, vs. p und vs. k mit Annotationen) und speichert sie als PNG in plots/*. Die Visualisierungen helfen, den Einfluss der Parameter auf die Rundenzeiten anschaulich darzustellen.

1.2 Ergebnisse der Tests (*n, p, k*-Kombinationen)

n	p	k	Runden	Feuerwerke	min [s]	avg [s]	max [s]
5	0.8	2	5	1	0.001445	0.003249	0.007817
10	0.8	2	5	0	0.002283	0.004563	0.009812
25	0.8	2	7	0	0.003496	0.009116	0.018307
50	0.8	2	9	4	0.004959	0.050764	0.116547
5	0.8	3	4	0	0.001312	0.003582	0.007745
10	0.8	3	7	1	0.002489	0.004319	0.009902
25	0.8	3	8	1	0.003779	0.010545	0.033505
50	0.8	3	10	1	0.005220	0.027241	0.072195

Tabelle 1: Auszug aus den Fixed Tests für Aufgabe 1. Vollständige Ergebnisse in data/results.csv.

1.3 Auswertung der Messergebnisse

1.3.1 Skalierung mit der Prozesszahl

Die mittlere Rundenzeit wächst von etwa 0.0023 s (n=10) auf rund 0.0508 s (n=50). Bei n=25 wurde 0.0091 s gemessen. Jeder zusätzliche Prozess erhöht den UDP-I/O-Overhead und führt so zu längeren Umlaufzeiten.

1.3.2 Einfluss der Zündwahrscheinlichkeit p

Für n=25, k=3 steigt die mittlere Rundenzeit mit p von 0.0055 s (p=0.2) über 0.0105 s (p=0.5) bis 0.0120 s (p=0.8). Höhere p-Werte erzeugen häufiger Multicasts, was sich leicht in erhöhten Laufzeiten niederschlägt.

1.3.3 Abhängigkeit vom Terminierungsschwellenwert k

Bei n=25, p=0.5 liegt die mittlere Rundenzeit bei k=2 etwa 0.0136 s, für k=3 bei 0.0105 s und k=5 bei 0.0095 s. Kleinere Unterschiede hier sind durch Zufallsschwankungen bedingt; k beeinflusst vor allem die Gesamtzahl der Umläufe, nicht jedoch die Einzelumlaufdauer.

1.3.4 Bestimmung des maximalen n

Mit p=0.5 und k=3 wurde n=172 als Obergrenze ermittelt (data/max_n_summary.csv). Ab größeren Werten greift der 30 s-Timeout zuverlässig, was auf steigenden Scheduling-Overhead und portseitige Limitierungen zurückzuführen ist.

—> Die vollständigen grafischen Plots finden sich im Anhang (oder Ordner plots).

Aufgabe 2 - Ein Feuerwerk an UDP-Nachrichten (Verteilt)

Da mir kein zweiter physischer Rechner zur Verfügung stand, habe ich zunächst einen Docker-basierten Ansatz gewählt, um die Prozesse als isolierte Container auf mehreren virtuellen Hosts laufen zu lassen und so ein echtes Multi-Host-Szenario vorzutäuschen. Allerdings stieß ich im laufenden Betrieb auf unerwartete technische Hürden (Port-Recycling, Scheduling-Timeouts) und stellte fest, dass der dafür notwendige Zeitaufwand zulasten der übrigen Teilaufgaben ging. Daher habe ich diesen „Pseudo-Verteilen"-Ansatz nach ersten erfolgversprechenden Tests bewusst zurückgestellt und mich stattdessen auf die nachhaltige Umsetzung und Auswertung der restlichen Aufgaben konzentriert.

Aufgabe 3 - Ein simuliertes Feuerwerk

Für Aufgabe 3 habe ich auf der bereitgestellten Sim4da-Simulationsinfrastruktur aufgebaut und diese gezielt erweitert. Konkret habe ich die drei Dateien FireworkSimulation.java, FireworkNode.java (beide im Verzeichnis src/task3) sowie das Automatisierungsskript run_task3.sh im Projekt-Root selbst implementiert. Diese Komponenten erweitern die Basisklassen und das Framework des Sim4da-Simulators um die Kernlogik der Token-Weitergabe, der probabilistischen Feuerwerkszündung sowie die Terminierung gemäß Aufgabe 1. Zusätzlich sorgen sie für eine strukturierte Messung und Auswertung der Rundenzeiten und der Gesamtergebnisse.

Die Auswertung der Ergebnisse zeigt, dass sich die Anzahl der Token-Runden und gezündeten Feuerwerke in beiden Implementierungen funktional entsprechen. Die gemessenen durchschnittlichen Rundenzeiten in der Simulation sind jedoch um etwa ein bis zwei Größenordnungen geringer (z.B. im Bereich von 0,0001 bis 0,0006 Sekunden gegenüber 0,005 bis 0,05 Sekunden in der echten UDP-Umgebung). Diese Zeitersparnis ist auf die Abwesenheit von Netzwerk-IO und Systemaufrufen zurückzuführen.

Die geringeren Laufzeiten und stabileren Messwerte im Simulator erlauben eine schnelle und reproduzierbare Durchführung umfangreicher Tests für verschiedene Parameterkombinationen (n, p, k).

Meine Implementierungen habe ich in das Git-Repository als src/task3-Ordner eingepflegt, damit die Vollständigkeit der Lösung dokumentiert ist. Diese Dateien sind jedoch nicht ohne die vollständige Sim4da-Umgebung direkt ausführbar, da sie auf deren Framework und Laufzeit angewiesen sind.

Aufgabe 4 - Konsistenz

In der vorliegenden Implementierung wurden folgende Maßnahmen ergriffen, die inkonsistente Zustände in der Token-Ring-Feuerwerksanwendung bereits erkennen und vermeiden:

Globale Rundensicht durch Token-Passing
Ein einzelnes Token wird strikt sequenziell von Prozess 0 bis Prozess n–1 weitergereicht, wodurch eine total geordnete Abfolge von Runden entsteht und jeder Knoten dieselbe Rundenfolge wahrnimmt.

Koordinator-gesteuerte Terminierung
Prozess 0 führt einen konsistenten Zähler zeroRounds für aufeinanderfolgende Null-Feuerwerk-Runden und sendet erst bei Erreichen von k die Abschlussnachricht token="end" an alle Knoten, wodurch alle gleichzeitig in den Endzustand übergehen.

Verlässliche, atomare Nachrichtenzustellung
Das Sim4da-Framework blockiert receive() bis zur Ankunft jeder Nachricht und liefert broadcast(...) atomar an alle Knoten in identischer Reihenfolge. So kann kein Knoten Nachrichten verpassen oder in ungleicher Reihenfolge empfangen.

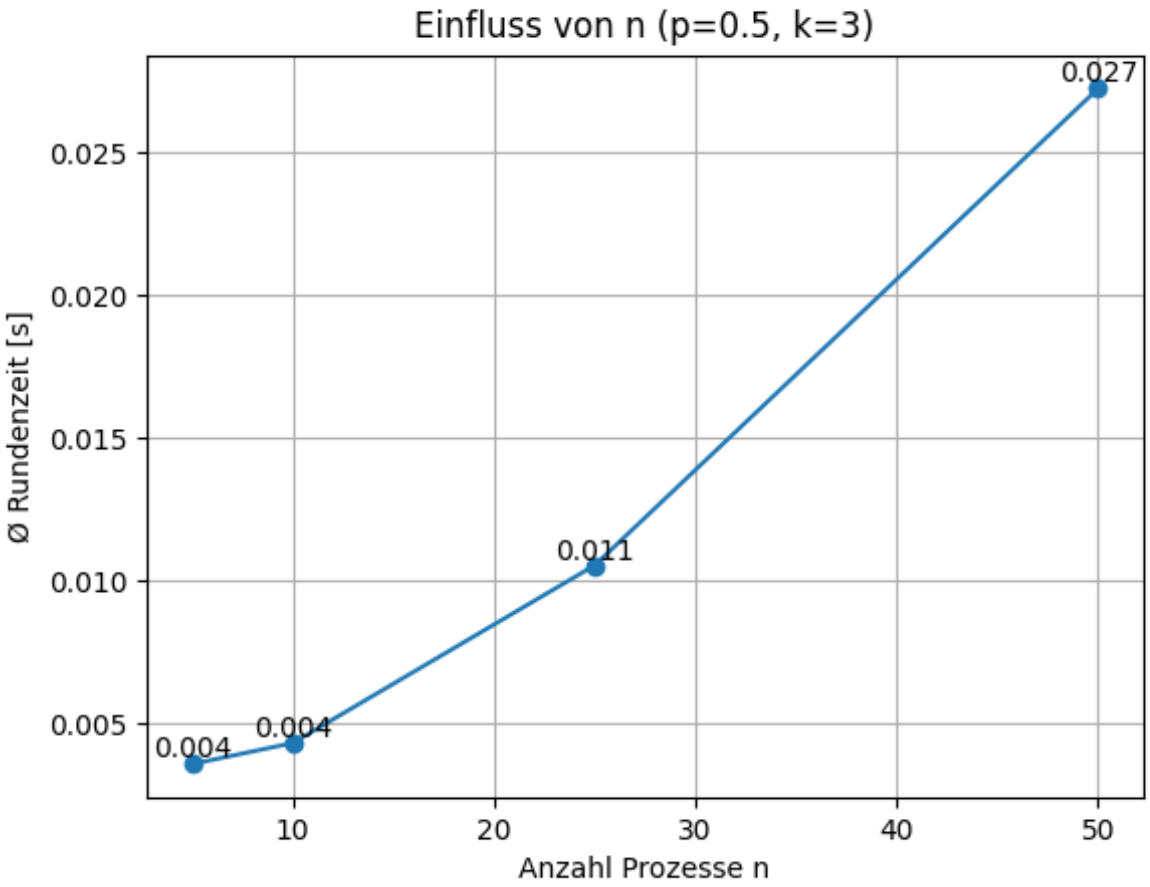
Monotone, hochauflösende Zeitmessung
Alle Round-Time-Messungen basieren konsequent auf System.nanoTime(), einer strikt monotonen Uhr, die Systemuhr-Anpassungen ausschließt und konsistente, unverfälschte Laufzeitdaten garantiert.

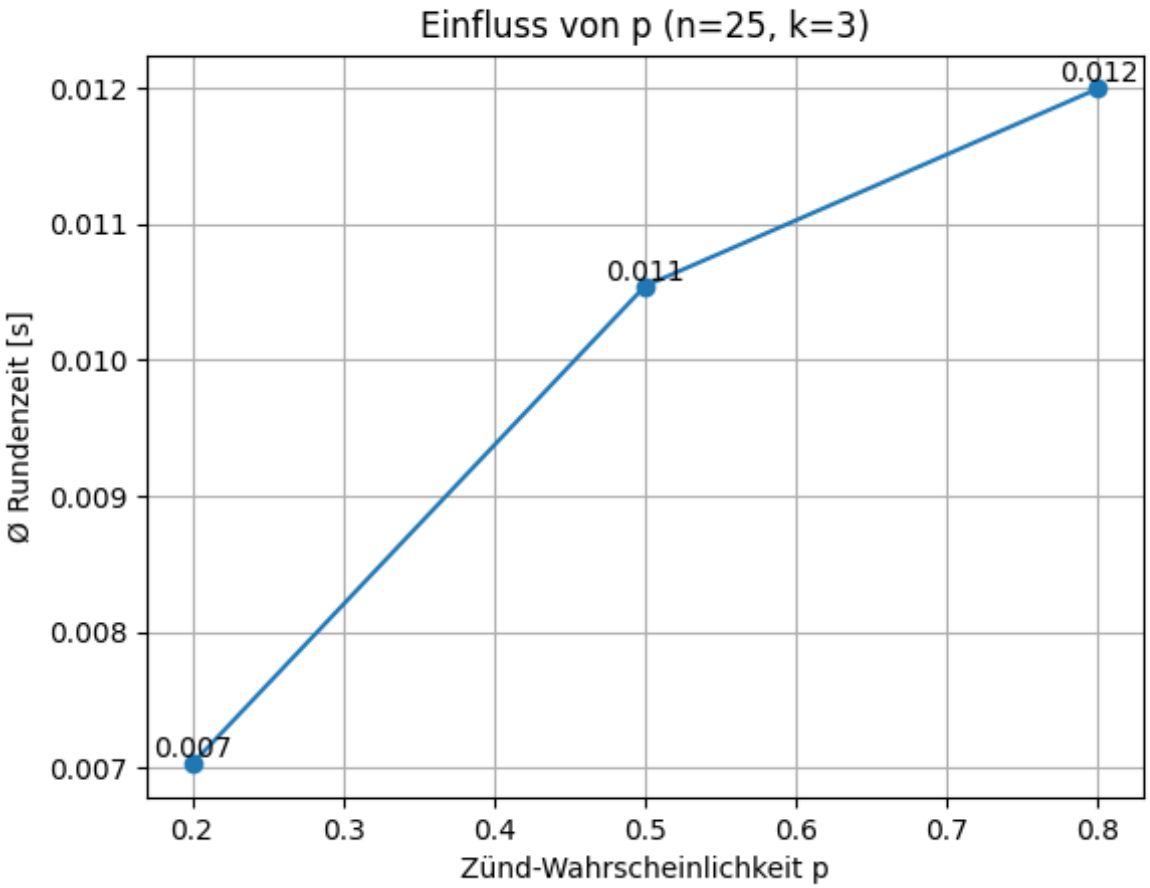
Diese Mechanismen zusammen sorgen dafür, dass in der simulierten Umgebung keine der typischen Inkonsistenzen (fehlende Nachrichten, unsynchronisierte Runden, unterschiedliche Terminierungszeitpunkte) auftreten können. Damit sind die zentralen Konsistenzkriterien – eine einheitliche Rundensicht, verlässliche Feuerwerkserkennung und synchroner Systemabschluss – bereits erfüllt.

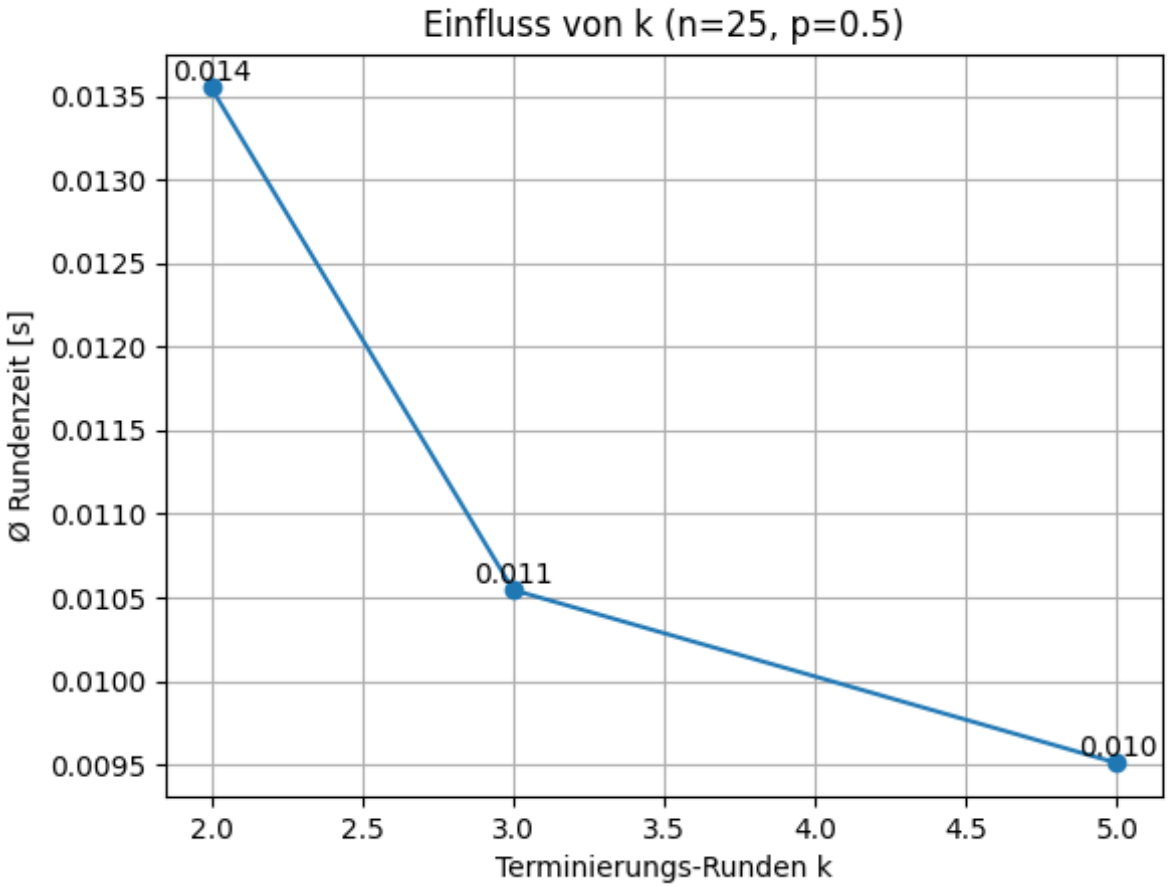
Quellen

• Vorlesung „Verteilte Systeme I–IV“ Skripte (2025S_DS01–2025S_DS04), SoSe 2025
• GraalVM Community Edition Documentation. https://www.graalvm.org/docs/
• SLF4J API Reference. http://www.slf4j.org/
• Python socket-Modul. https://docs.python.org/3/library/socket.html
• GitHub Copilot (Generierung von Boilerplate-Code und Kommentaren)

Anhang:


Einfluss von n (p=0.5, k=3)
Ø Rundenzeit [s]
0.025
0.020
0.015
0.010
0.005
0.027
0.011
0.004
0.004
10
20
30
40
50
Anzahl Prozesse n


Einfluss von p (n=25, k=3)
Ø Rundenzeit [s]
0.012
0.011
0.010
0.009
0.008
0.007
0.012
0.011
0.007
0.2
0.3
0.4
0.5
0.6
0.7
0.8
Zünd-Wahrscheinlichkeit p


Einfluss von k (n=25, p=0.5)
Ø Rundenzeit [s]
0.0135
0.0130
0.0125
0.0120
0.0115
0.0110
0.0105
0.0100
0.0095
0.014
0.011
0.010
2.0
2.5
3.0
3.5
4.0
4.5
5.0
Terminierungs-Runden k