Übung 2

Repository: https://github.com/Raphael90K/BSA2

1 Einleitung

Im zweiten Übungsblatt sollen verschiedene Formen von interprocess communication (IPC) gemessen werden. Hierzu sollen die minimalen Latenzen bei der Verwendung von

1. der Kommunikation über Spinlocks,
2. der Kommunikation über Semaphores,
3. der Kommunikation über die ZeroMQ[1] und
4. der Kommunikation zwischen zwei Anwendungsthreads in verschiedenen Docker Containern

experimentell ermittelt werden.

Als Hardware wurde für die Ausführung der Programme ein Notebook mit Linux Mint und einem Intel i7 6700HQ Prozessor mit einer Grundtaktfrequenz von 2,6 GHz verwendet.

2 Methodik

Für alle zu untersuchenden Methoden wird die gleiche herangehensweise gewählt. Es soll stets eine Komponente (Prozess oder Thread) geben, die eine Messung startet. Die Reihenfolge des Programmablaufs ist dabei durchgehend

1. Messung starten, Genauigkeit in Nanosekunden (ns),
2. mit der anderen Komponente kommunizieren über die jeweilige Methode,
3. auf Antwort der Komponente warten und
4. Messung beenden.

[1] https://zeromq.org/

Mit diesem Ansatz werden 100 Messungen durchgeführt und aus diesen die jeweilige minimale Round-Trip-Time (RTT) gespeichert. Insgesamt wird dies 1.000 Mal wiederholt, sodass bei 100.000 Messungen 1.000 Datenpunkte entstehen, aus denen die mittlere minimale RTT berechnet werden soll. Die Latenz ergibt sich ungefähr aus der Hälfte der RTT [IBH 24]. Die geringe Anzahl von 100 Messungen pro Durchgang soll gewährleisten, dass auch kleinere Varianzen sichtbar gemacht werden können. Die ermittelten 1.000 Messpunkte werden als `csv`-Datei gesichert, sodass sie sich anschließend gezielter auswerten lassen.

Neben der Berechnung des Mittelwertes wird das 95 % Konfidenzintervall für den Mittelwert und die Standardabweichung berechnet. Für das Konfidenzintervall wird folgende Formel verwendet [BaBK22, vgl. p. 131 u. 153 f]:

$$c = (1 - \frac{\alpha}{2})\text{-Fraktil der } N(0;1)\text{-Verteilung} \tag{1}$$

$$s^2 = \frac{1}{n-1} * \sum_{i=1}^{n} (X_i - \hat{X})^2 \tag{2}$$

$$ki = \left[\bar{x} - \frac{sc}{\sqrt{n}}; \bar{x} + \frac{sc}{\sqrt{n}} \right] \tag{3}$$

Die Formel kann für beliebig verteilte Stichproben mit einem Umfang $n > 30$ verwendet werden [BaBK22, vgl. S. 153 f].

Zusätzlich zu den errechneten Kennzahlen werden die Messwerte als Streudiagramm und Histogramm dargestellt und deren RTT verglichen.

3 Implementierung

Um eine möglichst performante Implementierung zu erreichen wurden die Programme zur Lösung dieses Übungsblattes in `C` implementiert. Bei der Erstellung der Programme wurde ChatGPT[2] in der kostenlosen Version als Hilfsmittel verwendet. Der Programmcode befindet sich im Verzeichnis `src/[Aufgabe]`. Sollte die minimalen RTT in mehreren Prozessen gemessen werden, so ist stets

[2] chatgpt.com/

ein Programmcode für den messenden Programmteil A (`*P1.c`) sowie ein Programmcode für den unterstützenden Programmteil B (`*P2.c`) vorhanden. Bei der Ausführung sollte Programmteil B zuerst gestartet werden, da dieser bei den späteren Aufgaben als Server dient und ein Verbindungsversuch über TCP bei Programmteil A nicht möglich ist.

3.1 Aufgabe 1

Die Implementierung erfolgt anhand eines simplen Spinlocks, bei dem die atomare Variable `spinlock` getestet wird, bis der kritische Abschnitt frei ist und von der Komponente (Thread oder Prozess) betreten werden kann. Da sich zwei Komponenten abwechseln sollen, wird der Prozessablauf über die atomare Variable `turn` gesteuert. Durch die Variable `turn` ist sichergestellt, dass beide Komponenten sich während der Zeitmessung abwechseln. Dieser Ansatz wurde von Tanenbaum und Bos beschrieben.[TaBo15, vgl. S. 123 f]
Der Spinlock selbst ist durch die atomare Funktionen `atomic_exchange`[3] implementiert. Das aktive Warten erfolgt somit für die Komponente während des Wartens auf die Variable `turn`.
Die Implementierung erfolgt in `T1single.c` mit zwei Threads sowie in `T1P1.c` und `T1P2.c` über zwei Prozesse, bei denen die o.g. Variablen in einem Shared Memory Bereich gespeichert werden und von beiden Programmen aufgerufen werden können. In beiden Implementierungen wird das Spinlock von Komponente A doppelt so häufig gehalten, dies jedoch außerhalb des Bereichs der Zeitmessung.

3.2 Aufgabe 2

Aufgabe 2 hat einen ähnlichen Programmablauf wie zuvor in Aufgabe 1 beschrieben. Anstatt Spinlocks werden zwei Semaphores (`sem_a` und `sem_b`) zur Kommunikation genutzt. Beide Semaphores werden zunächst mit 0 initialisiert.

Der Programmablauf ist nun in der Messschleife wie folgt:

1. Komponente A wartet, dass `sem_a` frei wird

[3] https://pubs.opengroup.org/onlinepubs/9799919799/functions/atomic_exchange.html

2. Komponente B gibt `sem_a` frei und wartet auf `sem_b`
3. Komponente A läuft weiter und startet die Zeitmessung, im Anschluss gibt es `sem_b` frei und wartet auf `sem_a`
4. Komponente B läuft weiter, gibt `sem_a` frei und wartet auf `sem_b`
5. Komponente A läuft weiter, beendet die Zeitmessung und gibt `sem_b` frei
6. Komponente B läuft weiter

Bei der Implementierung in `T2single.c` werden zwei Threads verwendet. Die Programme `T2P1.c` und `T2P2.c` enthalten den gleichen Programmablauf, jedoch für zwei Prozesse, die separat gestartet werden. Das Semaphore wird hierbei mit `sem_open`[4] geöffnet. Das Semaphore kann dann prozessübergreifend über dessen Namen geöffnet werden [Wolf06]. Ein Shared Memory Bereich ist dann nicht notwendig.

3.3 Aufgabe 3

In Aufgabe 3 wird zur Kommunikation zwischen den Komponenten die ZeroMQ[5] verwendet. Diese kann für die Programmiersprache c in Linux über einen `apt` Befehl installiert werden[6].

Grundlegend misst Komponente A wieder die minimale RTT wie in den beiden vorherigen Aufgaben auch. Nach dem Start der Zeitmessung sendet Komponente A eine Nachricht an Komponente B und wartet auf die Antwort. Nach der Antwort wird die Zeitmessung beendet. Das Programm `T3single.c` enthält die Implementierung für einen Prozess und zwei Threads. Die Programme `T3P1.c` und `T3P2.c` enthalten die Implementierungen für die Kommunikation zwischen zwei Prozessen.

[4] https://pubs.opengroup.org/onlinepubs/9799919799/functions/sem_open.html
[5] https://zeromq.org/
[6] https://zeromq.org/download/

3.4 Aufgabe 4

In den Implementierungen sendet Komponente A wie in der vorherigen Aufgabe wieder eine Nachricht an Komponente B und misst die Zeit bis zur Rückkehr. In den Unterverzeichnissen befinden sich jeweils Programme für die Kommunikation über TCP und TCP unter Verwendung der ZMQ. Zudem sind jeweils Dockerfiles zum Aufbau des Images vorhanden. Die notwendigen Kommandozeilenbefehle befinden sich in der `setup.txt` Datei. Die Programme mit dem Namen `T4P1` enthalten den Code für die messende Komponente und die Programme mit den Namen `T4P2` enthalten den unterstützenden Server, der auf Nachrichten antwortet. Die versendeten Nachrichten haben jeweils eine Größe von 16 Byte, sodass das Ende einer Nachricht bekannt ist. In der TCP Implementierung sorgt die Funktion `read_msg` dafür, dass die 16 Byte zuverlässig gelesen werden. Bei ZeroMQ ist dies nicht notwendig, da Nachrichten ähnlich zu UDP gehandhabt werden [7].

4 Experiment

Die grafischen Ausgaben wurden in einem Jupyter Notebook erzeugt. Für das Finden der richtigen Funktionen, insbesondere im Framework Pandas, wurde ebenfalls ChatGPT in der kostenlosen Version verwendet. Das Jupyter Notebook ist gespeichert als `results.ipynb` und als `results.pdf`. Die Messungen werden für alle Experimente als Streudiagramm und als Histogramm dargestellt. Die 95 % Konfidenzintervalle sind in diesen Graphen ebenfalls enthalten. Am Ende dieses Abschnitts werden die Messungen und die errechneten Kennzahlen zusammengefasst.

4.1 Aufgabe 1

Das Experiment würde für Aufgabe 1 in einem Prozess und mit zwei Prozessen über Shared Memory durchgeführt. Die minimalen RTT sind für die Messung mit einem Prozess in Abbildung 1 dargestellt. Abgesehen von ein einigen Ausreißern zu Beginn der Programmausführung scheint die benötigte Zeit ansonsten homogener zu sein. Im Mittel über 1.000 Wiederholungen beträgt die minimale Prozesszeit 291,79 Nanosekunden (ns). Die Implementierung mit zwei Prozessen führt

[7] https://zguide.zeromq.org/docs/chapter2/

zu den Messungen in Abbildung 2. Die Verteilung der Messwerte ist hier deutlich heterogener und eine konkrete Verteilungsfunktion kann nicht zugeordnet werden, jedoch ist diese Art der Prozesskommunikation mit 270,79 ns im mittel schneller als die Ausführung in einem Prozess.

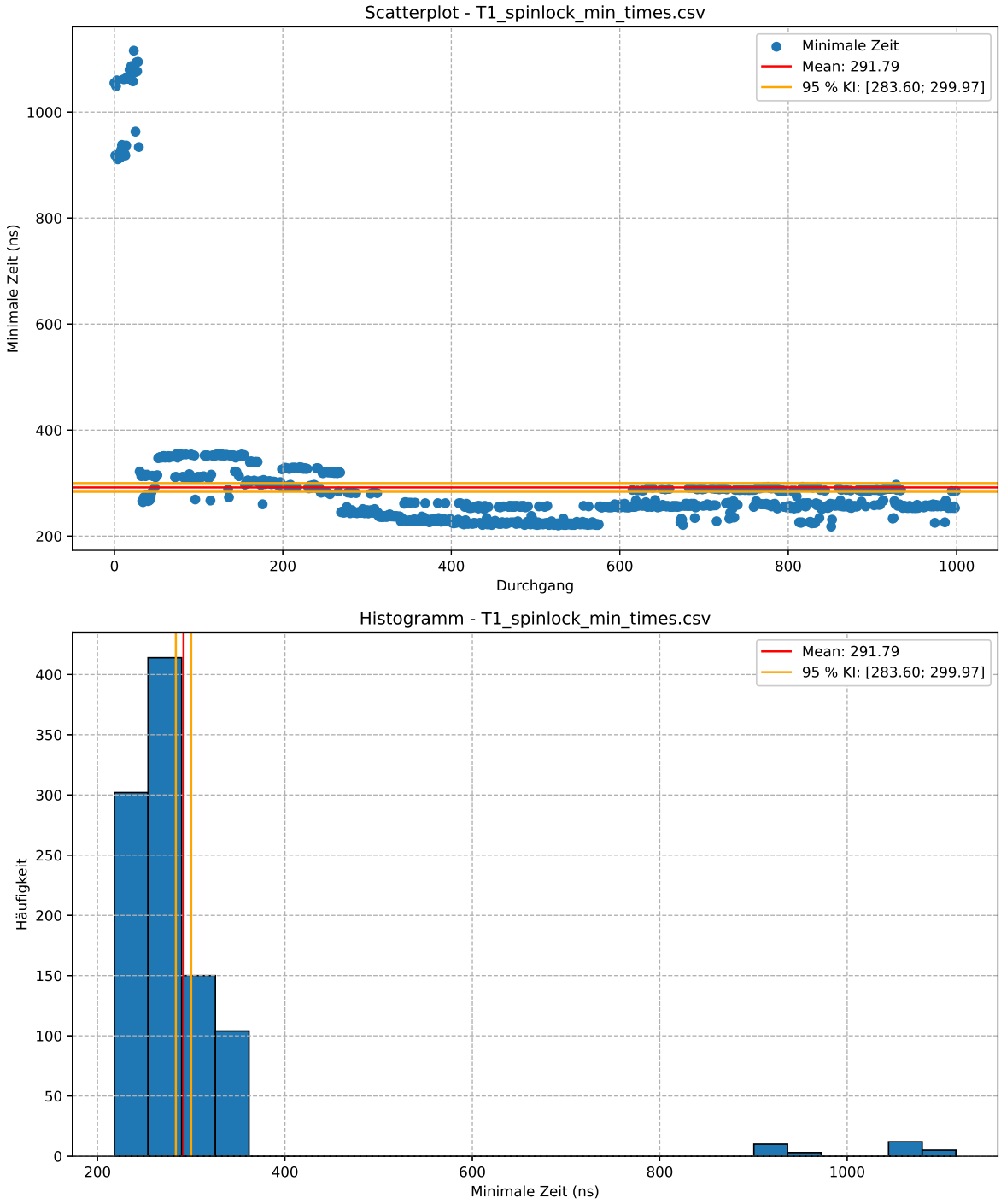

Abbildung 1: Diagramme Aufgabe 1, Spinlock innerhalb eines Prozesses

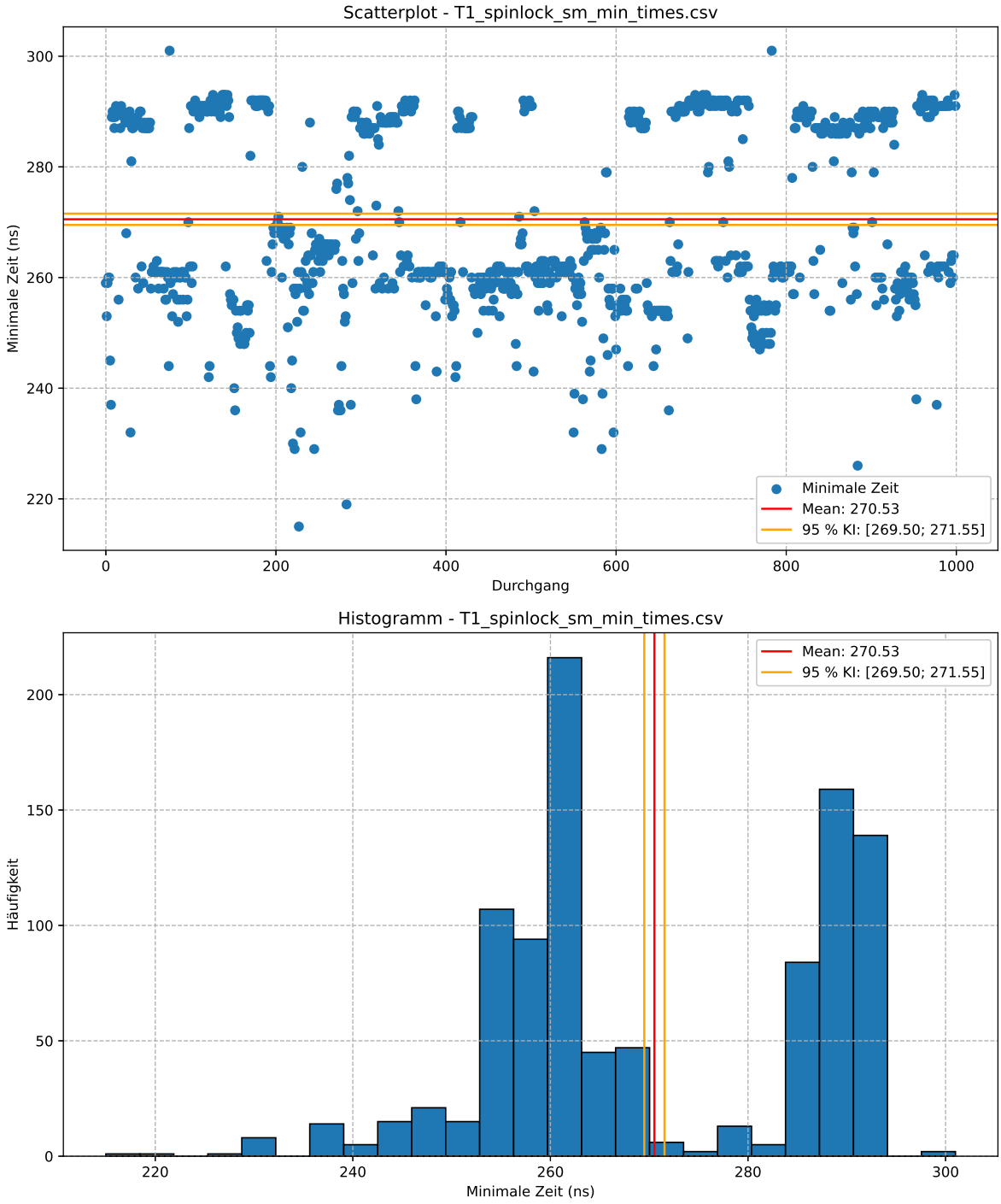

Abbildung 2: Diagramme Aufgabe 1, Spinlock zwischen zwei Prozessen und Shared Memory

4.2 Aufgabe 2

In der zweiten Aufgabe wurde die Kommunikation über Semaphores gemessen. Die Messungen der Ausführung mit einem Prozess sind in Abbildung 3 dargestellt. Die minimale RTT bei der Kommunikation liegt im Mittel bei 6.462,41 ns. In den Diagrammen ist zu erkennen, dass die Messwerte nur über zwei Bereich verteilt sind. Bei der Ausführung in zwei Prozessen kann eine minimale RTT von 3,106,23 ns im Mittel erreicht werden. An Abbildung 4 ist zu erkennen, dass es 4 Zeitslots gibt, in denen sich die Messwerte verteilen. Die schnellere Zeit im Vergleich zu der Implementierung in einem Prozess kann ggfs. damit zutun haben, dass die Prozesse effizienter gescheduled werden und dann häufiger Rechenzeit bekommen als bei einem Prozess und zwei Threads. Wie auch in Aufgabe 2 ist die minimale RTT bei zwei Prozessen kürzer, als bei einem Prozess.

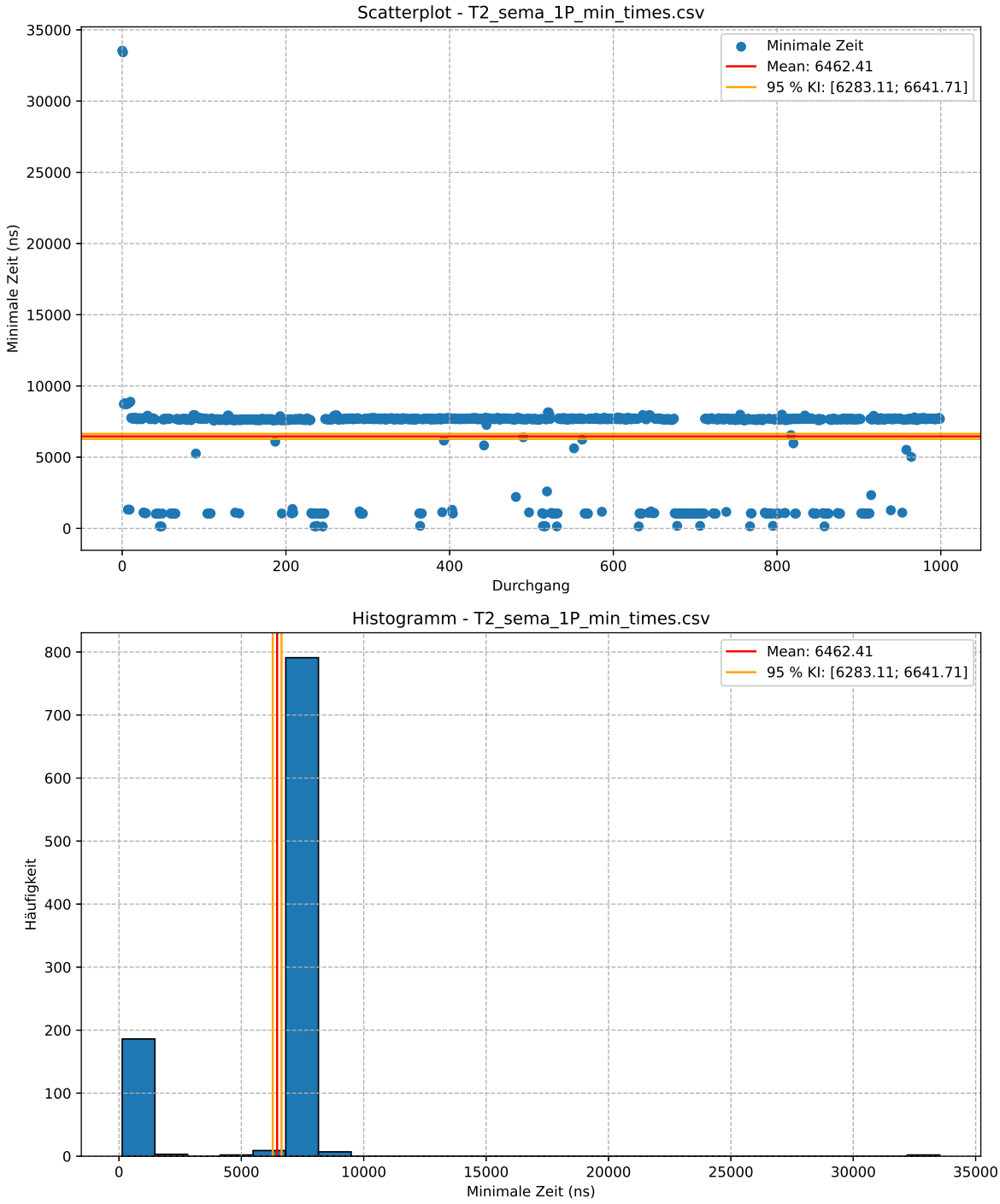

Abbildung 3: Diagramme Aufgabe 2, Semaphore innerhalb eines Prozesses

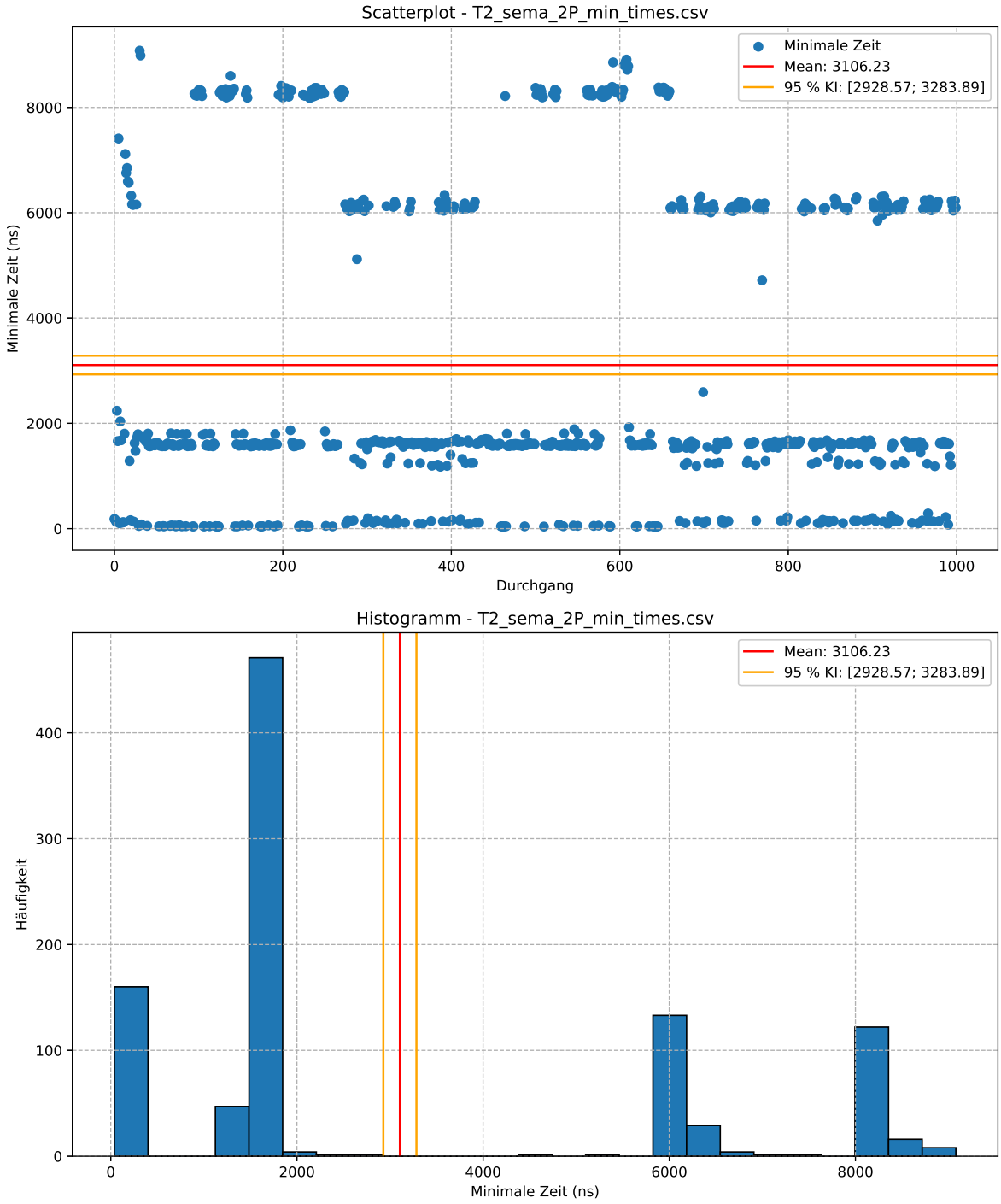

Abbildung 4: Diagramme Aufgabe 2, Semaphore zwischen zwei Prozessen

4.3 Aufgabe 3

In dieser Aufgabe wurde die minimale Kommunikationszeit unter Verwendung von ZeroMQ gemessen. Die Kommunikation innerhalb eines Prozesses benötigt im Mittel mindestens 20.623,13 ns. Das Streudiagramm und das Histogramm in Abbildung 5 zeigen, dass die Messwerte, abgesehen von Ausreißern, in einem Bereich konzentriert liegen. Bei der Kommunikation zwischen zwei Prozessen benötigt dieser Ansatz hingegen 29.339,63 ns und ist damit ca. 42 % mehr Zeit. Die Verteilung der Messwerte ist aus Abbildung 6 zu entehmen.

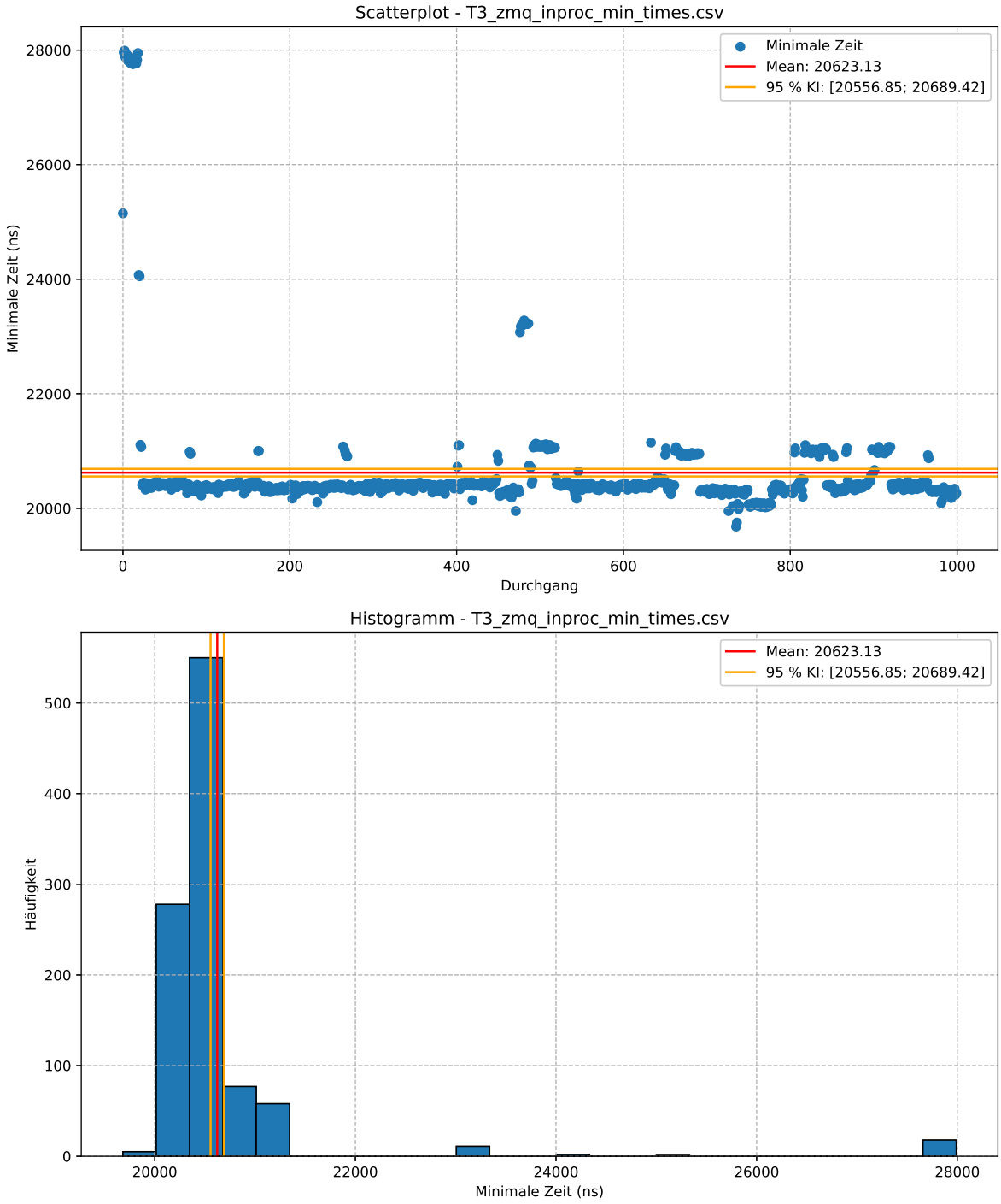

Abbildung 5: Diagramme Aufgabe 3, ZeroMQ innerhalb eines Prozesses

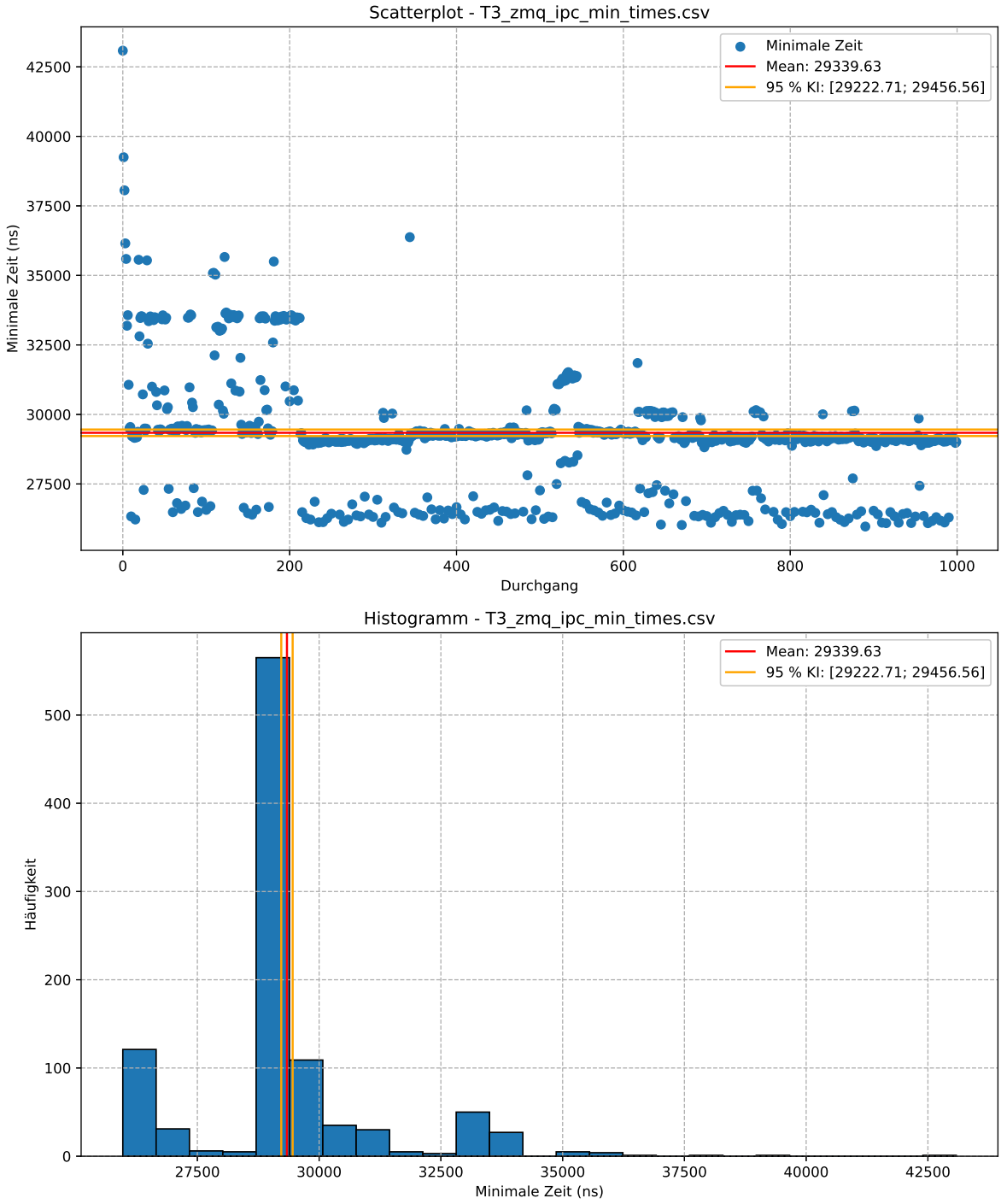

Abbildung 6: Diagramme Aufgabe 3, ZeroMQ zwischen zwei Prozessen

4.4 Aufgabe 4

In der letzten Aufgabe wurde die Kommunikation zwischen Prozessen in zwei Docker Containern gemessen. Bei der Verwendung von TCP ergibt sich eine minimale RTT von 17.645,23 ns zwischen beiden Prozessen. Die Verteilung der Messwerte wird in Abbildung 7 dargestellt. Mit dieser minimalen RTT ist der Ansatz der Kommunikation über das TCP Protokoll zwischen zwei Docker Containern schneller, als beide ZeroMQ Implementierungen aus Aufgabe 3. Der Overhead der ZeroMQ Implementierung wird bei der Verwendung zur Kommunikation zwischen zwei Docker Containern deutlich. Die minimale RTT liegt hier bei 74.127,36 ns. Abbildung 8 zeigt die Verteilung der gemessenen Datenpunkte.

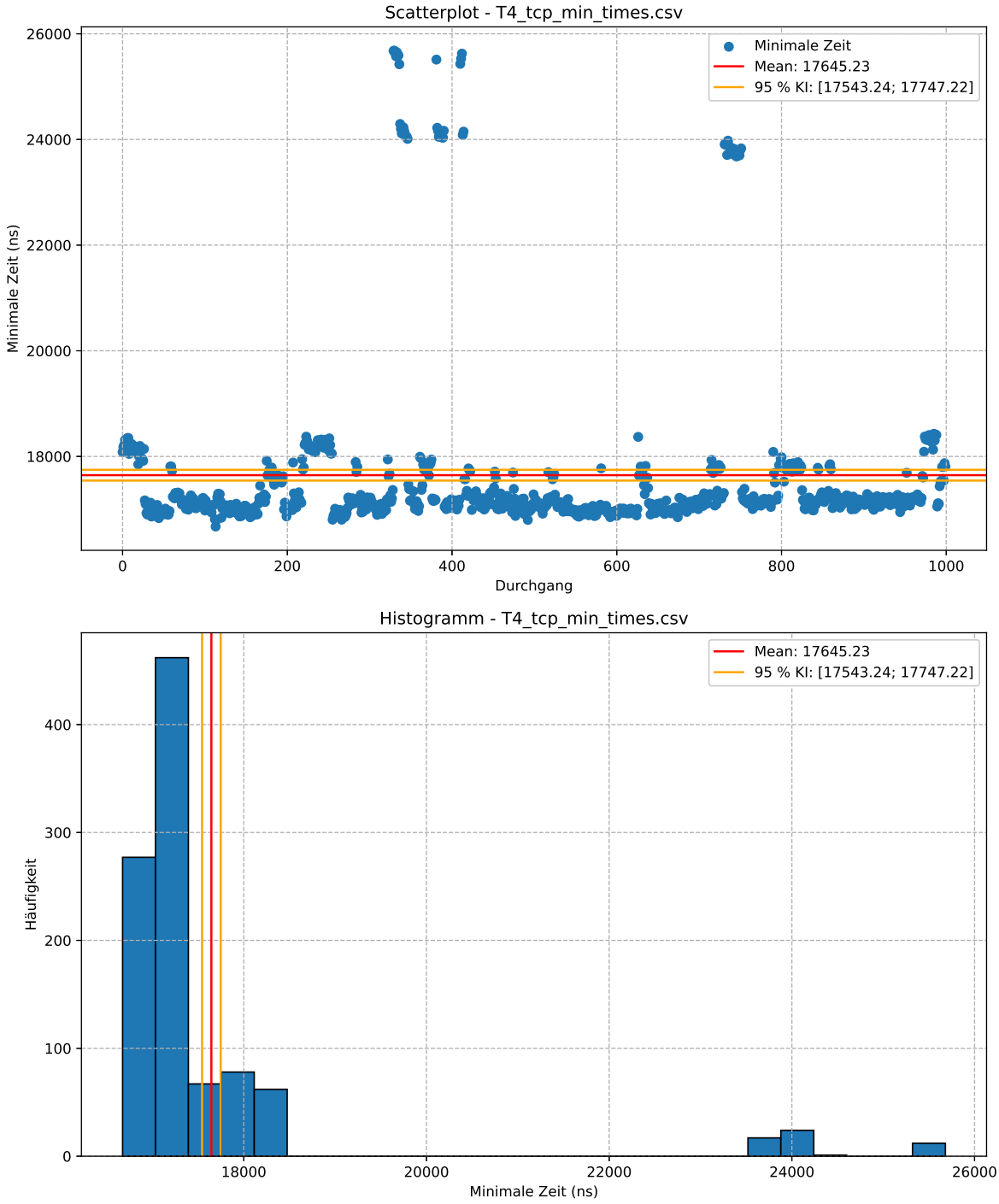

Abbildung 7: Diagramme Aufgabe 4, TCP zwischen zwei Docker Containern

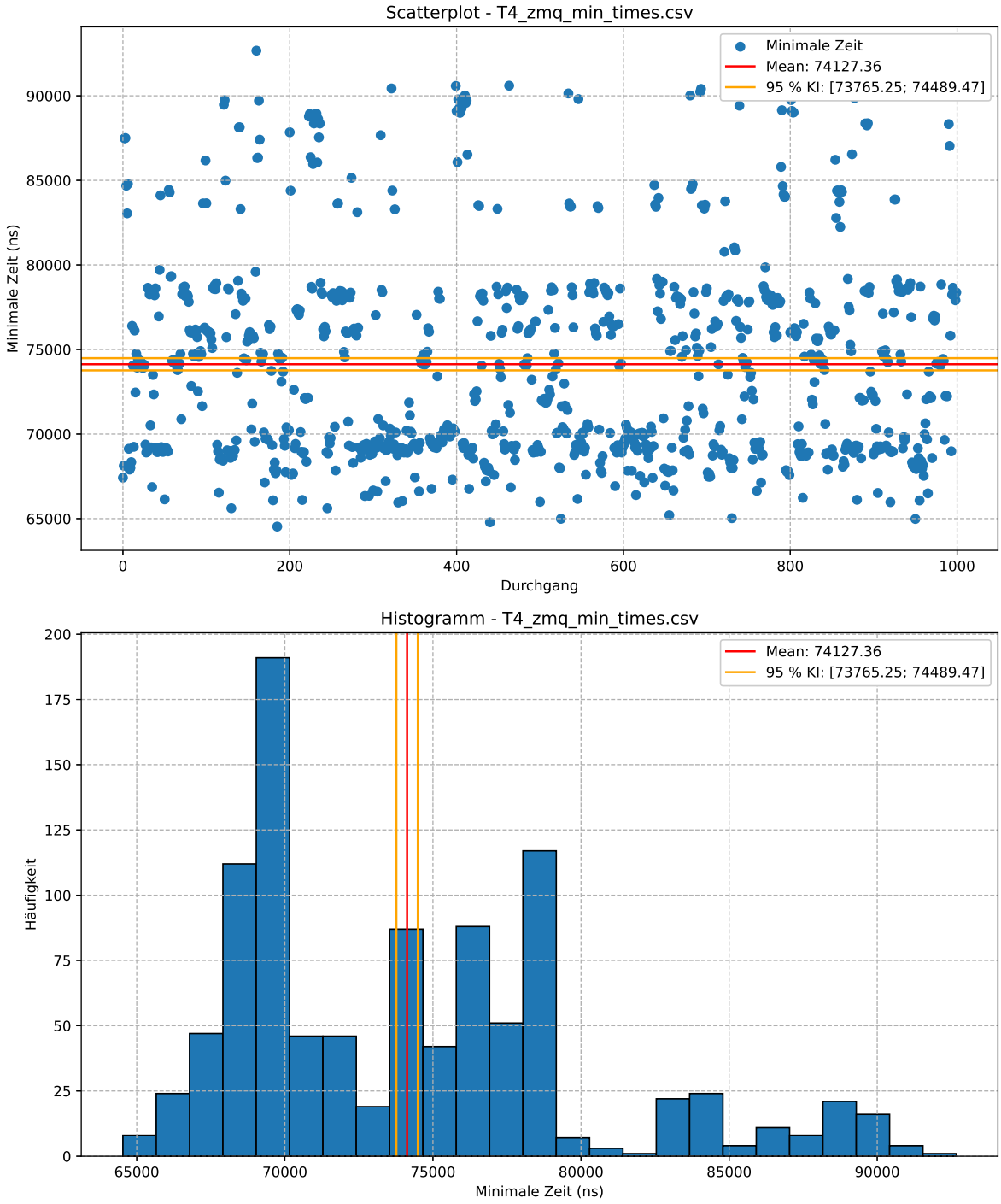

Abbildung 8: Diagramme Aufgabe 4, ZeroMQ mit TCP zwischen zwei Docker Containern

4.5 Zusammenfassung

Tabelle 1 fasst die aus den Messwerten berechneten Kennzahlen zusammen. Die mittleren minimalen Latenzen der Kommunikationsansätze sind in der letzten Spalte der Tabelle dargestellt. Zu erkennen ist, dass die Kommunikation über Spinlocks am schnellsten ist. Dies gilt für beide Implementierungen. Die Kommunikation über Semaphores ist ebenfalls deutlich schneller als die übrigen Ansätze. Die gemessenen Werte der RTT liegen hierbei im unteren μSekunden Bereich. Zudem ist zu erkennen, dass die Kommunikation zwischen zwei Docker Containern über das TCP Protokoll im Minimum schneller ist, als die Verwendung von ZeroMQ innerhalb eines Prozesses. Dies spricht für einen deutlichen Overhead von ZeroMQ.

Log Name	RTT mean	95 % ki low	95 % ki hi	RTT std	Latency
T1_spinlock_min_times.csv	291,79	283,6	299,97	132,08	145,89
T1_spinlock_sm_min_times.csv	270,53	269,5	271,55	16,51	135,26
T2_sema_1P_min_times.csv	6.462,41	6.283,11	6.641,71	2.892,88	3.231,21
T2_sema_2P_min_times.csv	3.106,23	2.928,57	3.283,89	2.866,43	1.553,11
T3_zmq_inproc_min_times.csv	20.623,13	20.556,85	20.689,42	1.069,45	10.311,57
T3_zmq_ipc_min_times.csv	29.339,63	29.222,71	29.456,56	1.886,5	14.669,82
T4_tcp_min_times.csv	17.645,23	17.543,24	17.747,22	1.645,56	8.822,62
T4_zmq_min_times.csv	74.127,36	73.765,25	74.489,47	5.842,38	37.063,68

Tabelle 1: Ermittelte Kennzahlen der Experimente

Die ermittelten minimalen RTT sind abschließend in Abbildung 9 dargestellt.

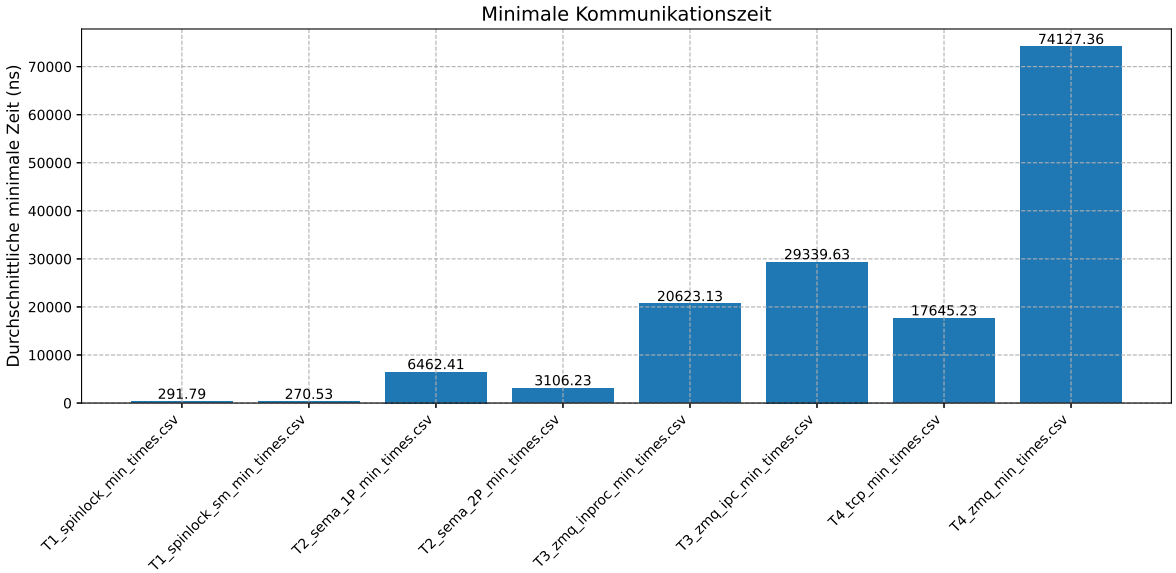

Abbildung 9: Mittlere minimale RTT im Vergleich

5 Fazit und Ausblick

Bei der Bearbeitung dieses Übungsblattes konnte ermittelt werden, dass die Verwendung von Spinlocks zur Kommunikation bei dieser Implementierung die geringste Verzögerung aufweist. Bei einer Verwendung von Shared Memory ist so auch eine Kommunikation zwischen mehreren Prozessen möglich. Dieser Ansatz hat jedoch Nachteile, da die Prozesse in einer Schleife aktiv warten und so Rechenleistung verbrauchen. Bei Anwendungen, bei denen lange Wartezeiten einkalkuliert werden müssen, sollte dieser Ansatz daher vermieden werden [TaBo15, vgl. S. 124]. Der nächste interessante Aspekt ist die deutliche Reduzierung der minimalen Kommunikationszeit bei der Verwendung von Semaphores und zwei Prozessen gegenüber der Implementierung in nur einem Prozess. Da beide Programme den gleichen Ablauf besitzen, dürfte dies auf ein effizienteres Scheduling zurückzuführen sein. Die genaue Ursache sollte jedoch intensiver untersucht werden. Da Semaphores auf Ebene des Betriebssystems angelegt werden können, ist mit ihnen ebenfalls eine einfache und effiziente Kommunikation zwischen Prozessen im μSekunden Bereich möglich.
Die Verwendung von ZeroMQ führt im Vergleich zu den vorherigen Ansätzen zu deutlich längeren Latenzen. Da diese Schnittstelle sich jedoch um den Nachrichtenaustausch kümmert und Standards

wie TCP unterstützt, ist eine deutlich einfachere und flexiblere Anwendungsentwicklung möglich. Der Overhead von ZeroMQ bei der Verwendung von TCP ist jedoch aus den Messwerten von Aufgabe 4 ersichtlich.

In allen Aufgaben wurden die minimalen mittleren Latenzen aus den minimalen Messwerten der jeweils 100 Messungen ermittelt. Dies führt aufgrund der geringen Anzahl an Messungen jedoch zu einer deutlichen Streuung der Messwerte. Durch eine Erhöhung der Messwerte pro Durchgang dürfte sich diese Varianz verringern lassen. Für die Nützlichkeit der Ansätze in realen Anwendungen wäre die durchschnittliche Latenz der verschiedenen Ansätze ebenfalls wichtig. Ob sich hierbei grundlegend andere Verhältnisse der Messergebnisse ergeben, müsste in weiteren Experimenten untersucht werden.

Literatur

[BaBK22] G. Bamberg, F. Baur und M. Krapp. *Statistik Eine Einführung für Wirtschafts- und Sozialwissenschaftler*. De Gruyter Oldenbourg, Berlin, Boston. 19. Auflage, 2022.

[IBH 24] IBH IT-Service. Was bedeutet Latenz und Jitter? https://www.ibh.de/index/was-bedeutet-latenz-und-jitter, 2024. Stand: 12.01.2025, Aufruf am: 12.01.2025.

[TaBo15] A. S. Tanenbaum und H. Bos. *Modern operating systems Elektronische Ressource / Andrew S. Tanenbaum ; Herbert Bos*. Pearson, Boston u.a. 4. ed., global ed.. Auflage, 2015.

[Wolf06] J. Wolf. Linux-UNIX-Programmierung. https://openbook.rheinwerk-verlag.de/linux_unix_programmierung/Kap09-004.htm#RxxKap09004040002D71F017104, 2006. Stand: 12.01.2025, Aufruf am: 12.01.2025.