Ergebnisbericht Übungsblatt 2 Betriebssysteme

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

In dieser Analyse habe ich Python verwendet, um die Kommunikationslatenz zwischen Threads mit verschiedenen Mechanismen zu messen. Obwohl C aufgrund seiner kompilierten Natur und des geringeren Overheads niedrigere Latenzen bieten würde, habe ich Python aus Gründen der Praktikabilität und der einfachen Handhabung bei der schnellen Entwicklung und Datenanalyse gewählt.

Teilaufgabe 1: Latenz bei Spinlock-Kommunikation

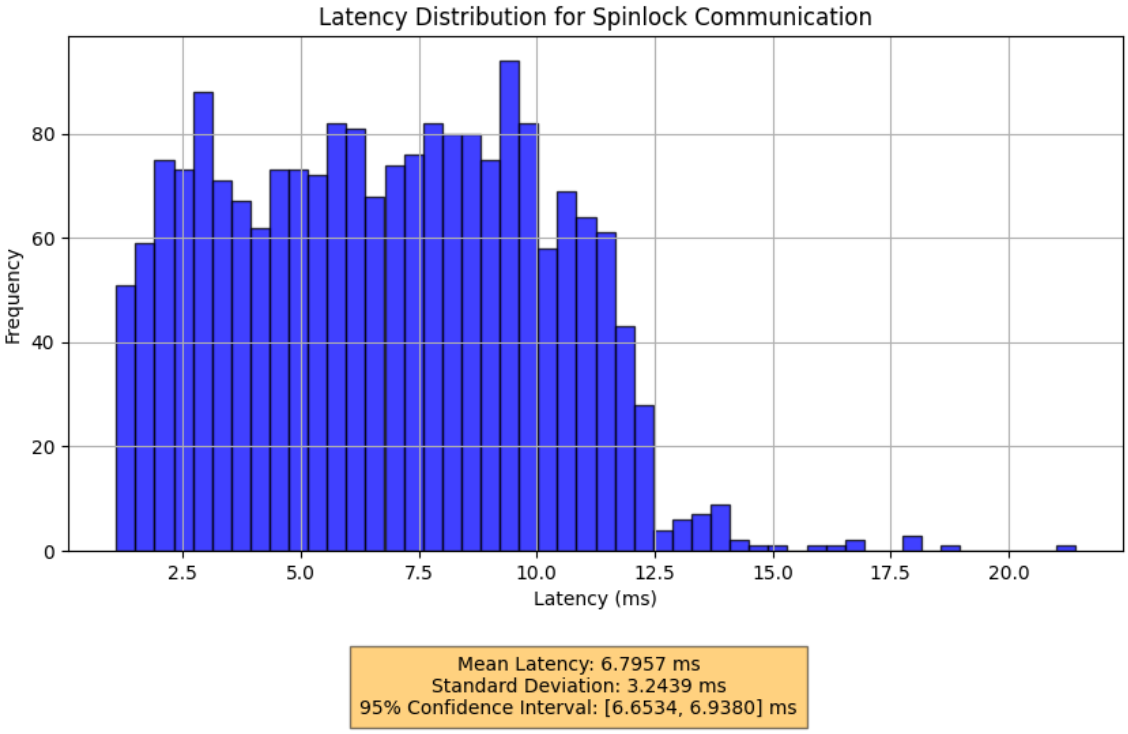
In dieser Aufgabe habe ich die Kommunikationslatenz zwischen zwei Threads mit einem Spinlock-Mechanismus gemessen. Das Experiment wurde 1000 Mal durchgeführt, um genügend Daten für eine statistische Analyse zu sammeln.

Um die gesammelten Latenzdaten zu analysieren, habe ich die mittlere Latenz, die Standardabweichung und das 95%-Konfidenzintervall berechnet.

Die **t-Verteilung** wurde aus folgenden Gründen verwendet:
Kleine Stichprobengröße: Mit 1000 Stichproben ist die t-Verteilung angemessener als die Normalverteilung, die besser für größere Stichproben geeignet ist.
Unbekannte Populationsstandardabweichung: Die Standardabweichung der Population ist unbekannt, und die t-Verteilung berücksichtigt diese Unsicherheit.

Ergebnisse:
Mittlere Latenz: 6.7957 ms
Standardabweichung: 3.2439 ms
95%-Konfidenzintervall: [6.6534, 6.9380] ms

Das folgende Histogramm zeigt die Verteilung der Latenzen für die Spinlock-Kommunikation:



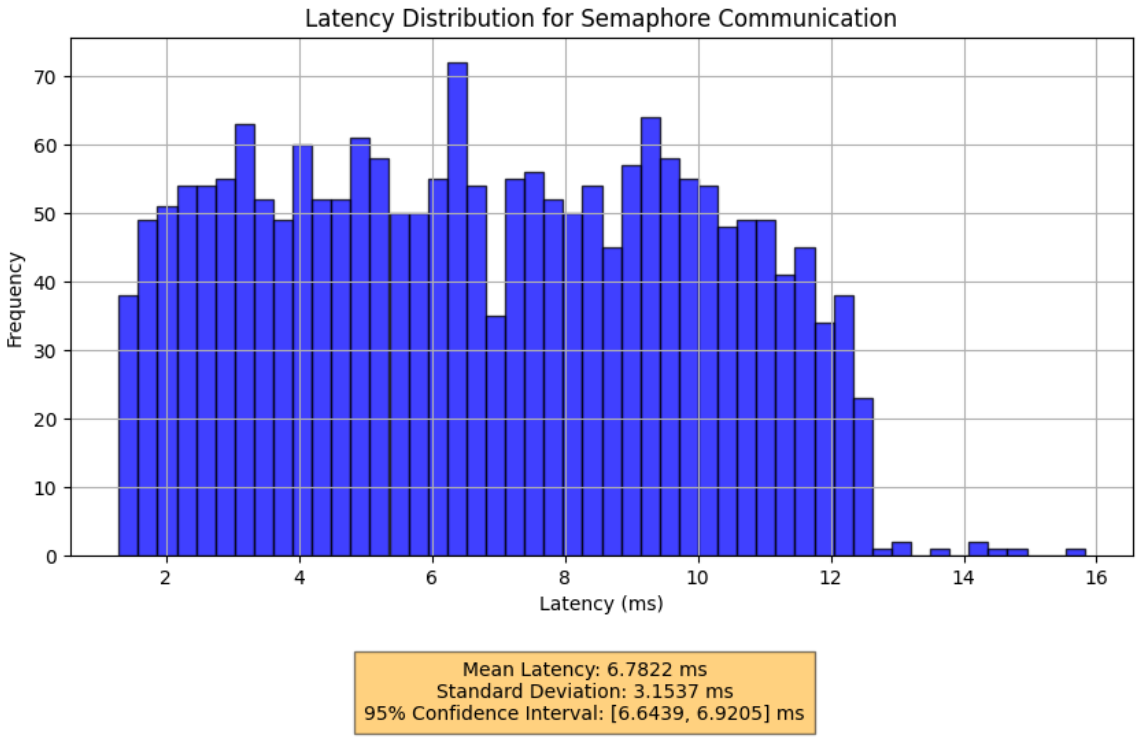
Fazit
Der Spinlock-Mechanismus zeigte eine mittlere Latenz von 6.7957 ms mit einer Standardabweichung von 3.2439 ms. Das 95%-Konfidenzintervall für die mittlere Latenz beträgt **[6.6534, 6.9380] ms**. Diese Analyse liefert eine zuverlässige Schätzung der Kommunikationslatenz unter Verwendung von Spinlocks.

Teilaufgabe 2: Latenz bei Semaphore-Kommunikation

In dieser Teilaufgabe habe ich die Semaphore-Kommunikation zwischen zwei Threads mithilfe der Python-Bibliothek *threading* umgesetzt. Dabei verwendete ich *threading.Semaphore(initial),* um eine Semaphore mit einem initialen Wert von 1 zu erstellen, was bedeutet, dass nur ein Thread gleichzeitig auf die Ressource zugreifen kann. Zwei Threads erwarben die Semaphore, führten eine zufällige Verzögerung ein und gaben sie wieder frei. Das Experiment wurde 1000 Mal wiederholt, um eine ausreichende Datenbasis für die statistische Auswertung zu erhalten.

Ergebnisse:
Mittlere Latenz: 6.7822 ms
Standardabweichung: 3.1537 ms
95%-Konfidenzintervall: [6.6439 ms, 6.9205 ms]

Das folgende Histogramm zeigt die Verteilung der Latenzen für die Semaphore-Kommunikation:



Fazit
Die Verwendung der Semaphore führte zu einer mittleren Latenz von 6.7822 ms, mit einer Standardabweichung von 3.1537 ms. Das 95%-Konfidenzintervall liegt zwischen 6.6439 ms und 6.9205 ms. Dies zeigt eine konsistente Latenz, die durch den blockierenden Charakter der Semaphore verursacht wird. Im Vergleich zu Spinlocks führt die Semaphore zu einer messbaren Verzögerung, eignet sich jedoch gut für Szenarien mit moderater Thread-Konkurrenz, wo eine zuverlässige Synchronisation erforderlich ist.

Teilaufgabe 3: Latenz bei Kommunikation über Message-Queue ZeroMQ

Für diese Teilaufgabe habe ich ZeroMQ verwendet, um die Kommunikation zwischen zwei Threads zu untersuchen. Die Python-Bibliothek *zmq* wurde eingesetzt, um eine universelle Message-Queue zu erstellen, die asynchrone Kommunikation ermöglicht. ZeroMQ bietet verschiedene Kommunikationsmuster, darunter das *PAIR*-Pattern, welches speziell für die direkte Punkt-zu-Punkt-Kommunikation geeignet ist.

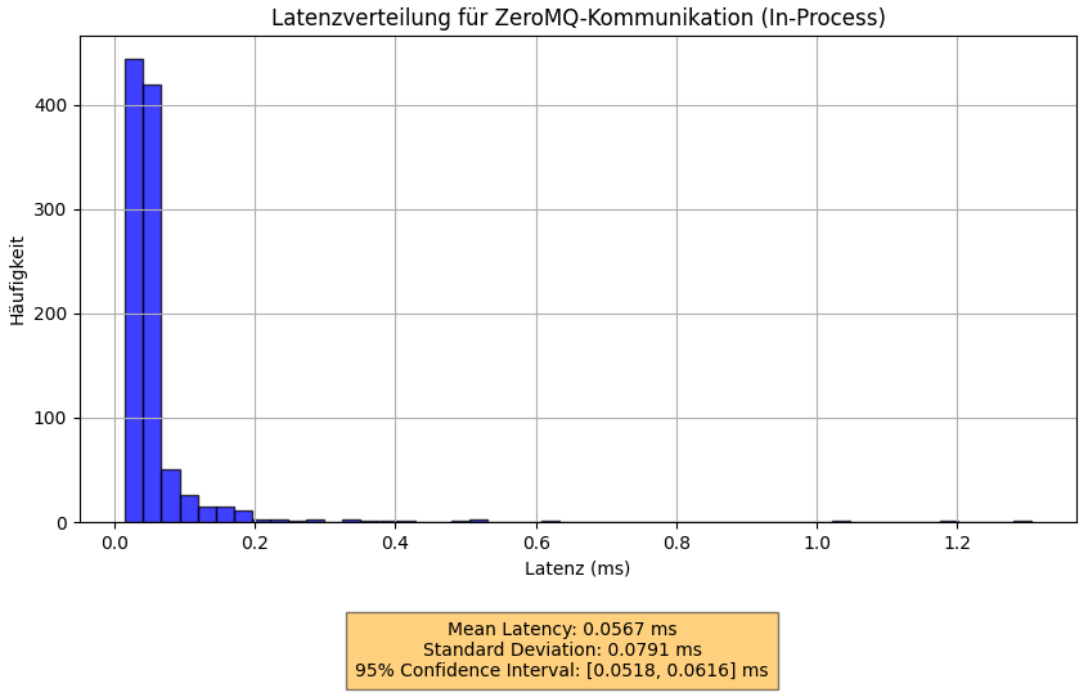
InProcess-Kommunikation: Zwei Threads kommunizieren innerhalb eines Prozesses über eine *inproc://-Adresse*.

InterProcess-Kommunikation: Zwei Threads kommunizieren zwischen zwei Prozessen über eine *ipc://-Adresse.*

Um eine statistisch belastbare Auswertung zu gewährleisten, wurde auch hier jedes Experiment 1000 Mal wiederholt.

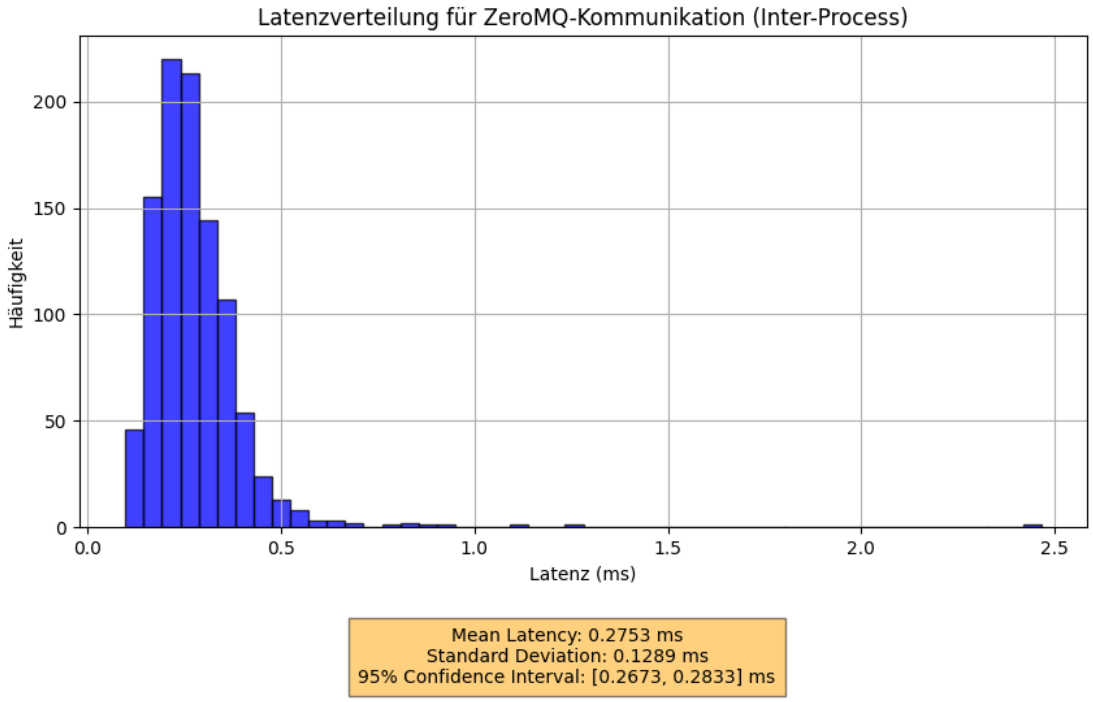
Ergebnisse
InProcess-Kommunikation
Mittlere Latenz: 0.0567 ms
Standardabweichung: 0.0791 ms
95%-Konfidenzintervall: [0.0518, 0.0616] ms

Das folgende Histogramm zeigt die Verteilung der Latenzen für die InProcess-Kommunikation:



InterProcess-Kommunikation
Mittlere Latenz: 0.2753 ms
Standardabweichung: 0.1289 ms
95%-Konfidenzintervall: [0.2673, 0.2833] ms

Das folgende Histogramm zeigt die Verteilung der Latenzen für die InterProcess-Kommunikation:



Unerwartetes Problem: Adresskonflikte bei InProcess-Kommunikation
Während der Experimente zur InProcess-Kommunikation trat ein unerwarteter Fehler auf: *Address already in use*. Dies geschah, weil die inproc://-Adresse nicht korrekt freigegeben wurde, bevor sie erneut verwendet wurde. ZeroMQ speichert aktive Adressen im Kontext, und fehlende Freigaben führen zu Konflikten.

Um das Problem zu lösen, habe ich sichergestellt, dass:
Sockets und der Kontext nach jedem Durchlauf ordnungsgemäß geschlossen werden.
Ein **Kontextmanager** (with-Anweisung) verwendet wurde, um den Lebenszyklus der Ressourcen effizient zu verwalten.

Fazit
Die Ergebnisse zeigen deutlich, dass die InProcess-Kommunikation mit einer mittleren Latenz von 0.0567 ms und einer engen Standardabweichung von 0.0791 ms wesentlich schneller und konsistenter ist als die InterProcess-Kommunikation, bei der eine mittlere Latenz von 0.2753 ms und eine höhere Standardabweichung von 0.1289 ms gemessen wurden.

Die größere Latenz und Variabilität bei der InterProcess-Kommunikation lassen sich auf den zusätzlichen Overhead durch den Betriebssystem-Kernel zurückführen, während die InProcess-Kommunikation durch direkte Thread-Interaktion innerhalb eines Prozesses effizienter arbeitet.

Teilaufgabe 4: Latenz bei Kommunikation über Docker-Containern

In dieser Aufgabe habe ich verschiedene Docker-Dateien verwendet, um eine verteilte Anwendung zu simulieren und die Latenzzeiten zu messen. Die Docker-Umgebung besteht aus mehreren Komponenten:

Dockerfile: Definiert das Basis-Image und die notwendigen Abhängigkeiten für die Anwendung.

docker-compose.yml: Konfiguriert die verschiedenen Dienste (z.B. Client und Server) und deren Interaktionen.

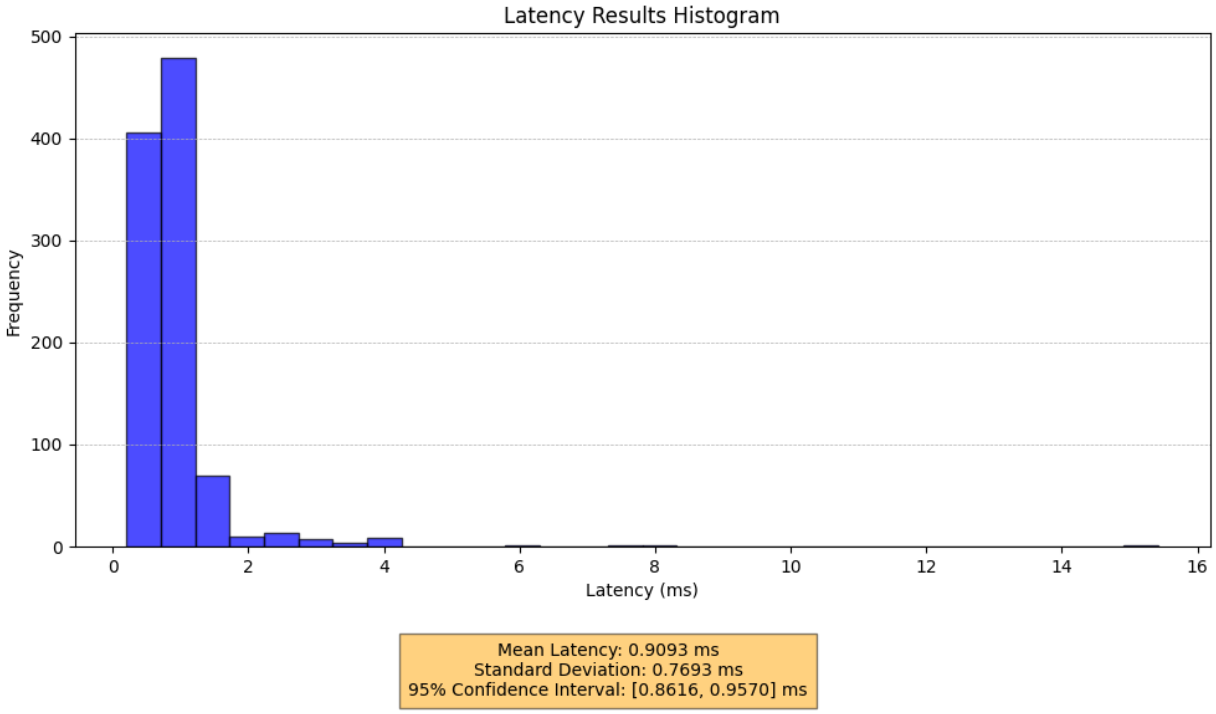
client.py: Implementiert den Client, der Nachrichten an den Server sendet und die Antwortzeiten misst.

server.py: Implementiert den Server, der Nachrichten vom Client empfängt und beantwortet.

Diese Dateien arbeiten zusammen, um die Latenzzeiten der Kommunikation zwischen Client und Server zu messen und die Ergebnisse zu visualisieren. Ergebnisse.

Ergebnisse:
Mean Latency: 0.9093 ms
Standard Deviation: 0.7693 ms
95% Confidence Interval: [0.8616, 0.9570] ms

Das folgende Histogramm zeigt die Verteilung der Latenzen für die Kommunikation über Docker Container:



Fazit
Die durchschnittliche Latenzzeit beträgt 0.9093 ms, was auf eine schnelle Kommunikation zwischen Client und Server hinweist. Die Standardabweichung von 0.7693 ms zeigt, dass die Latenzzeiten relativ konsistent sind.

Vergleich der Ergebnisse aller Teilaufgaben
Das Skript plot_results.py vergleicht die Ergebnisse aller Teilaufgaben, indem es die Latenzzeiten aus verschiedenen Experimenten lädt und in einem gemeinsamen Plot darstellt.

Zusammenfassend zeigt die Analyse, dass ZeroMQ mit InProcess-Kommunikation die geringste Latenz (0.0567 ms) und höchste Konsistenz bietet, während die InterProcess-Variante (0.2753 ms) aufgrund von Kernel-Overhead langsamer ist. Docker-basierte Kommunikation (0.9093 ms) eignet sich für verteilte Systeme, zeigt jedoch eine höhere Latenz durch Virtualisierung. Spinlocks und Semaphore (jeweils ca. 6.8 ms) sind vergleichsweise langsam, was auf die Synchronisationsmethoden und den höheren Overhead von Python zurückzuführen ist. ZeroMQ erweist sich somit als effizientester Mechanismus für niedrige Latenz und hohe Performance.

plot_a4_results.py: Analysiert und visualisiert die gesammelten Latenzzeiten und gibt folgenden Plot: