Umgebung der Experimente

Folgende Tabellen beschreiben das System, auf dem die Tests durchgeführt wurden.

Hardware-Spezifikation

Merkmal	**Spezifikation**
CPU-Bezeichnung	AMD Ryzen 5 4500U with Radeon Graphics
CPU-Architektur	x86 (AMD Renoir)
CPU-Fertigungsverfahren	7 nm (TSMC)
Anzahl Kerne / Threads	6 / 6
Basistaktfrequenz	2.3 GHz
Maximale Boost-Taktfrequenz	4.0 GHz
L1-Cache	384 KB
L2-Cache	3 MB
L3-Cache	8 MB
TDP (Thermal Design Power)	15 Watt
Maximale Temperatur	105 °C
Arbeitsspeicher	2x4GB DDR4-3200 (Dual Channel)
Erscheinungsdatum	07.01.2020

Software-Umgebung

Merkmal	**Spezifikation**
Betriebssystem	Arch Linux
Kernel-Version	Linux 6.12.8-arch1-1
Java-Version	Java 21
Java-Implementierung	java-21-openjdk
ZeroMQ-Implementierung	JeroMQ 0.6.0

1 Aufgabe 1 - Latenz bei Kommunikation mit Spinlock

1.1 Erste Ansätze zur Latenzmessung

Bei der Kommunikation über Spinlocks wartet ein Thread auf die Freigabe einer Ressource, indem er fortwährend (z.B. in einer `while`-Schleife) überprüft, ob die Ressource frei ist. Um die Latenz zu messen, habe ich neben dem Main-Thread einen `Reader`-Thread erstellt, der auf die Freigabe einer Ressource (Boolean, der auf `true` gesetzt wird) wartet, und den Wert dann wieder auf `false` setzt. Nachdem der Main-Thread den Wert auf `true` gesetzt hat, wartet er wiederum, bis der Wert wieder auf `false` gesetzt wird.

Um Race Conditions beim Abfragen der Werte aus den While-Schleifen zu vermeiden, wird der Zugriff mithilfe von `synchronized` Setter- und Getter-Methoden geregelt. Jeder Thread trägt immer, wenn er eine Nachricht vom anderen Thread erhält, einen Zeitstempel in Nanosekunden in eine ausreichend große Array-Liste ein, woraus später die Latenz berechnet wird. Nun folgen Auszüge aus dem Source Code, die dies zeigen.

```java
for (int i = 0; i < recursions; i++) {
  // set lock to true -> send message
  setLock(true);;

  // wait for the return message
  while (getLock()) {
    // Do nothing (spinlock)
  }

  // add current time to list
  messageTimes.add(System.nanoTime());
}
```

Listing 1: Spinlocks: Latenzmessung im Main Thread

```java
while (!isInterrupted()) {
  while (!isInterrupted() && experiment.getLock() == false) {
    // do nothing (spinlock)
  }

  // Add the current time to the list
  experiment.getMessageTimes().add(System.nanoTime());

  if (isInterrupted()) break;

  // Reset the lock (main thread is waiting)
  experiment.setLock(false);
}
```

Listing 2: Spinlocks: Latenzmessung im Reader Thread

1.2 Versuchsaufbau / Implementierung

Im Ursprünglichen Versuchsaufbau habe ich für jeden Versuchsdurchlauf den `Reader`-Thread und das `Experiment`-Objekt neu erstellt und jeweils von jedem Experiment die minimale Latenz gespeichert. Das Ergebnis des ersten Durchlaufs ist in Abbildung 1 dargestellt.

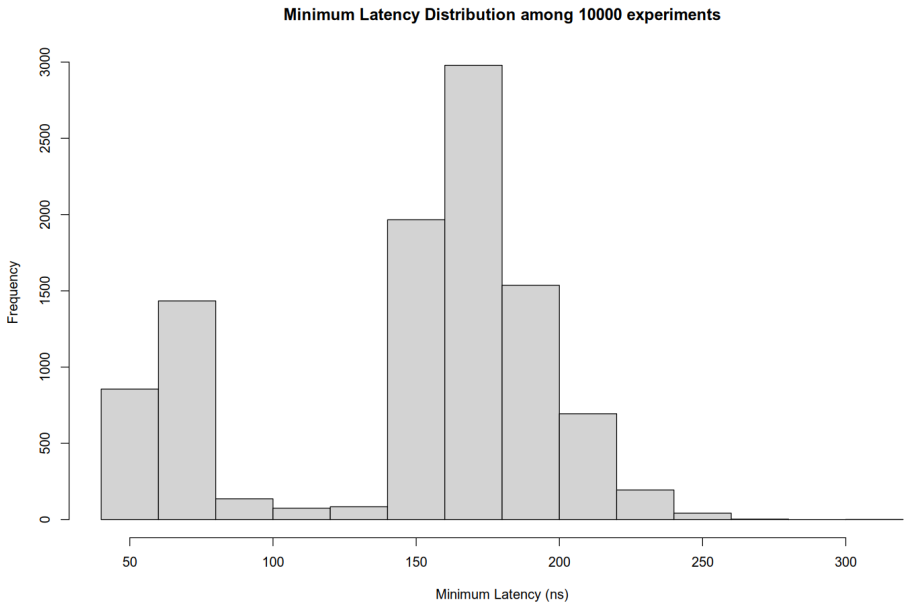


Abbildung 1: Verteilung der Minima im ersten Experiment

Wie man sieht liegt, anders als zu erwarten wäre, keine Normalverteilung der Latenzen vor. Nach reiflicher Überlegung hatte ich den Verdacht, dass ich durch die Instantiierung einer `Experiment`- und `Reader`-Objekts bei jedem Versuch eine große Menge ungenutzter Objekte erzeuge und dadurch ggf. der *Garbage Collector* die Performance zeitweise mindert.

Um einen weiteren Performance-Faktor zu optimieren, habe ich noch die Konstruktion bestehend aus Boolean und synchronized Getter- und Settermethoden gegen einen *AtomicBoolean* ausgetauscht.

Beim Herumexperimentieren wurde mir nun mit der Zeit klar, dass die Latenz derart gering sein muss, dass sie in der gleichen Größenordnung wie die Latenz der Zeitmessung selbst liegt. Nachdem ich den Zeitmessungsprozess so weit ich konnte optimiert habe, sah der Prozess wie folgt aus:

```
1  private void measure(int recursions) {
2    long sendTime;
3    long receiveTime;
4    for (int i = 0; i < recursions; i++) {
5      sendTime = System.nanoTime();
6      lock.set(true);
7      while (lock.get()) {
8        // Do nothing (spinlock)
9      }
10     receiveTime = System.nanoTime();
11
12     sendTimes.add(sendTime);
13     receiveTimes.add(receiveTime);
14   }
15 }
```

Listing 3: Spinlocks: Latenzmessung im Main Thread (optimiert)

```
1  while (!isInterrupted()) {
```

```
2   while (!experiment.lock.get()) {
3     // do nothing (spinlock)
4   }
5
6   // Reset the lock (main thread is waiting)
7   experiment.lock.set(false);
8 }
```

Listing 4: Spinlocks: Reader Thread (optimiert)

So wird jeweils die Zeit für einen Turn-Around gemessen, also die Zeit, die die Information von Main-Thread zum Reader-Thread und wieder zurück braucht. Zum Berechnen der Latenz wird diese Zahl später durch zwei geteilt. Allerdings ist hier immer noch auch die Latenz für das Abfragen und Speichern der aktuellen Zeit enthalten, was das Ergebnis in einer solch niedrigen Größenordnung durchaus verfälschen kann.

1.3 Ergebnis

Schlussendlich habe ich mich mit dem Versuch so zufrieden gegeben. Ich habe 10000 Durchläufe des Experiments mit jeweils 50000 Rekursionen durchgeführt und jeweils die Minimalwerte der einzelnen Experimente zusammengetragen. Von 10000 Durchläufen betrug die minimale Latenz 9854 mal **genau 64 Nanosekunden**. Das spricht dafür, dass wir hier bereits an die Grenzen der Messgenauigkeit stoßen und daher ist die Berechnung eines Konfidenzintervalls eigentlich hinfällig. Hier aber trotzdem die geforderten Werte, auch wenn **ihre Aussagekraft durchaus angezweifelt werden kann**:

- Durchschnittliche Minimallatenz: $63,95ns$
- Untere Schranke des 95%-Konfidenzintervalls: $63,92ns$
- Obere Schranke des 95%-Konfidenzintervalls: $63,98ns$

Hier noch eine Wertetabelle, die alle gemessenen Minimallatenzen aufzeigt.

Latenz (ns)	Häufigkeit
24	7
30	1
35	1
40	12
64	9854
65	124
120	1

2 Aufgabe 2 - Latenz bei Kommunikation durch Semaphore

2.1 Versuchsaufbau / Implementierung

Da ich nun schon sehr viel Zeit mit der Optimierung der Messung in Aufgabe 1 verbracht habe, habe ich mich entschieden, die Lösung zur besseren Vergleichbarkeit nur leicht abzuwandeln und Semaphoren anstelle des `AtomicBoolean` und `while`-Schleifen zu verwenden. Der Ansatz zur Zeitmessung bleibt hier der gleiche: Es gibt zwei Semaphoren, eine `messageSemaphore` und eine `replySemaphore`, die beide zunächst blockiert sind. Der Reader-Thread versucht, sobald er

läuft, den Lock auf der `messageSemaphore` zu bekommen und gibt, wenn er den Lock bekommen hat, die `replySemaphore` frei. Die Zeitmessung läuft im Main-Thread nun wiefolgt ab:

1. Beginn der Zeitmessung
2. `messageSemaphore` freigeben
3. Versuche, Lock auf `replySemaphore` zu erhalten (warte auf Reader-Thread)
4. Wenn der Lock erfolgreich erhalten wurde, stoppe die Zeitmessung

Hier ein Auszug der entsprechenden Stellen aus dem Code:

```
1  private void measure(int recursions) {
2    long sendTime;
3    long receiveTime;
4    for (int i = 0; i < recursions; i++) {
5      sendTime = System.nanoTime();
6      messageSemaphore.release();
7      replySemaphore.acquire();
8      receiveTime = System.nanoTime();
9
10     sendTimes.add(sendTime);
11     receiveTimes.add(receiveTime);
12   }
13 }
```

Listing 5: Semaphores: Latenzmessung im Main Thread

```
1  while (!isInterrupted()) {
2    try {
3      experiment.messageSemaphore.acquire();
4      experiment.replySemaphore.release();
5    } catch (InterruptedException e) {
6      System.out.println("Reader Thread shutting down.");
7      break;
8    }
9  }
```

Listing 6: Freigeben der replySemaphore im Reader Thread

So wird am ende durch Halbierung der gemessenen Zeit die durchschnittliche Latenz der beiden Wege berechnet und schließlich nach einer gegebenen Anzahl von Wiederholungen das Minimum abgespeichert.

2.2 Ergebnis

Bereits beim Testen ist mir anhand der Laufzeit aufgefallen, dass die Latenz bedeutend höher sein muss als in Aufgabe 1. Um die Laufzeit gering zu halten, habe ich daher in dieser Versuchsreihe die Anzahl der Rekursionen auf 10000 reduziert und wie in Aufgabe 1 10000 Wiederholungen des Experiments durchgeführt. Hier die Verteilung der Minimalen Latenzen über die 10000 Experimente:

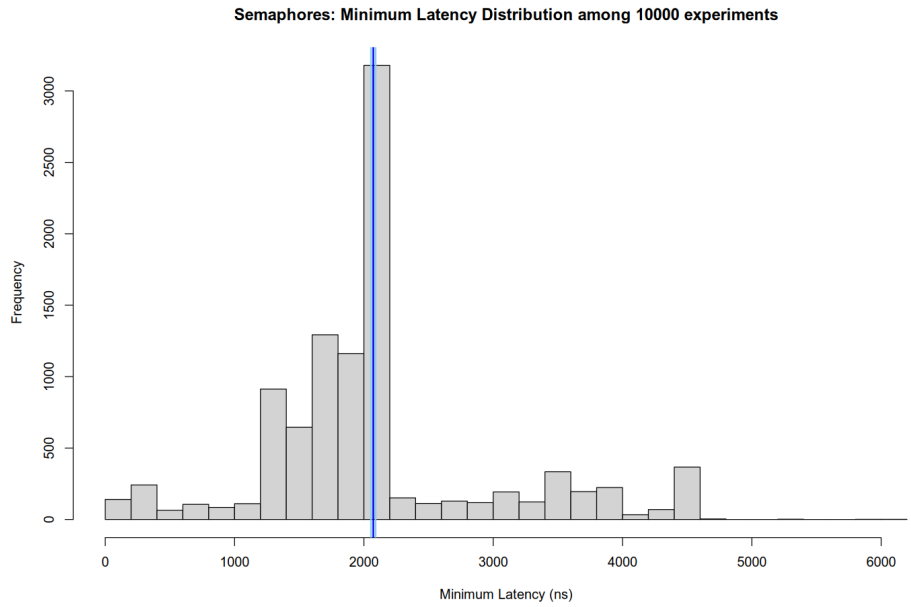


Abbildung 2: Verteilung der Minima in der Versuchsreihe zu Aufgabe 2

Auch wenn sich hier eine Linksschiefe der Verteilung bemerkbar macht, gehe ich bei der Berechnung des Konfidenzintervalls (in Blau dargestellt) von einer Normalverteilung aus. Der Grund für die Linksschiefe des Ergebnisses ist mir nicht ersichtlich. Hier die Daten des 95%-Konfidenzintervalls:

- Durchschnittliche Minimallatenz: $2073ns$
- Untere Schranke des 95%-Konfidenzintervalls: $2055ns$
- Obere Schranke des 95%-Konfidenzintervalls: $2090ns$

Die Latenz ist also um etwa zwei Zehnerpotenzen höher als bei der Verwendung von Spinlocks. Dies ist auch nur logisch, da ein Thread in einem Spinlock alle ihm zur Verfügung stehenden Ressourcen verwendet, um zu überprüfen, ob der Lock freigegeben wurde. Somit wird damit eine nahezu perfekte Latenz erzielt, jedoch bezahlt man den Preis dafür mit der CPU-Last, die ein Spinlock im Vergleich zu einer Semaphore hervorruft. Wird ein Thread durch eine Semaphore blockiert, so wird er schlafen gelegt und verursacht nur eine sehr niedrige CPU-Last.

3 Aufgabe 3 - Latenz bei Kommunikation mit ZeroMQ

3.1 Teil A (In-Process) - Versuchsaufbau / Implementierung

Auch hier habe ich das Messungsverfahren aus den vorherigen Aufgaben nur leicht angepasst. Anstelle der beiden Semaphoren kommen nun zwei PUSH-PULL Message Queues zum Einsatz: Eine befördert Nachrichten vom Main-Thread zum Reader-Thread, die andere in die entgegengesetzte Richtung. Für den Socket wird als Protokoll `inproc` verwendet, welches die geringste Latenz bietet und für die Kommunikation innerhalb eines Prozesses gedacht ist. Das grundsätzliche Vorgehen unterscheidet sich jedoch kaum:

```
1 private void measure(int recursions) throws InterruptedException {
2   long sendTime;
3   long receiveTime;
```

```
4
5   for (int i = 0; i < recursions; i++) {
6     sendTime = System.nanoTime();
7     sender.send(payload);
8     receiver.recv();
9     receiveTime = System.nanoTime();
10
11    sendTimes.add(sendTime);
12    receiveTimes.add(receiveTime);
13  }
14 }
```

Listing 7: ZeroMQ inproc: Latenzmessung im Main Thread

```
1  try {
2    Socket receiver = context.createSocket(SocketType.PULL);
3    receiver.connect("inproc://message");
4
5    Socket sender = context.createSocket(SocketType.PUSH);
6    sender.connect("inproc://response");
7
8    while (!isInterrupted()) {
9      receiver.recv();
10     sender.send(payload);
11   }
12 }
13 catch (Exception e) {
14   System.out.println("Reader thread is shutting down");
15 }
```

Listing 8: Antwort im Reader Thread

Das Setup der ZeroMQ-Sockets im Main-Thread spare ich mir an dieser Stelle, das kann gerne im Code oder der Dokumentation nachgelesen werden. Die `payload`, die hier als Nachricht und Antwort gesendet und empfangen wird, ist ein leeres Byte-Array, um unnötige Verzögerung durch die Nachrichtenübermittlung zu vermeiden und nur die Latenz zu messen. Auch hier wird wieder die Zeit eines Round-Trips gemessen und dann halbiert, um die tatsächliche Latenz einer Nachrichtenübermittlung zu ermitteln.

3.2 Teil A (In-Process) - Ergebnis

Die gemessenene Minimallatenzen sind in der gleichen Größenordnung wie die in Aufgabe 2 gemessenen Latenzen, daher habe ich hier die gleichen Parameter gesetzt wie zuvor: 10000 Wiederholungen pro Experiment und insgesamt 10000 Durchläufe. Hier ein Histogramm der Verteilung:

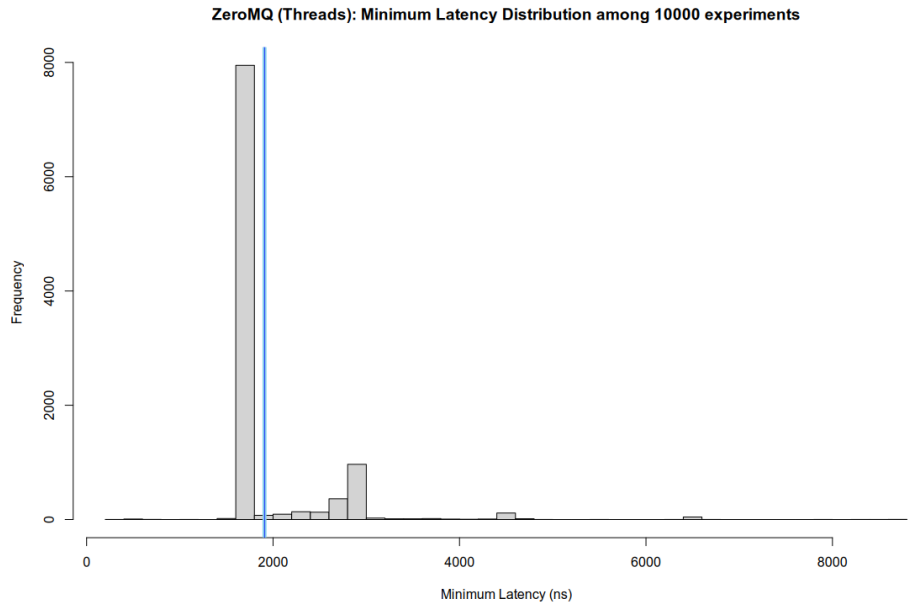


Abbildung 3: Verteilung der Minima in der Versuchsreihe zu Aufgabe 3A

Das Konfidenzintervall ist hier wieder in Blau dargestellt. Hier nochmal die genauen Werte:

- Durchschnittliche Minimallatenz: $1908ns$
- Untere Schranke des 95%-Konfidenzintervalls: $1869ns$
- Obere Schranke des 95%-Konfidenzintervalls: $1921ns$

Wie man sieht, unterbietet ZeroMQ hier die Latenz der Semaphore leicht, was ich durchaus bemerkenswert finde.

3.3 Teil B (Inter-Process) - Versuchsaufbau / Implementierung

Nachdem ZeroMQ schon mal eingebaut war, war der Schritt von In-Process- zu Inter-Process-Kommunikation nicht mehr weit: Ich musste bloß den Worker Thread in ein eigenes Programm umwandeln (Code aus run()-Methode des Threads in eine eigene main-Methode verschieben) und einen eigenen ZeroMQ-Kontext darin verwalten. Zusätzlich wird nun als Protokoll nicht mehr `inproc`, sondern `ipc` verwendet, welches über *UNIX domain sockets* kommuniziert.

```
1  public static void main(String[] args) {
2    try(ZContext context = new ZContext()) {
3      Socket receiver = context.createSocket(SocketType.PULL);
4      receiver.bind("ipc:///tmp/zmq-message.sock");
5
6      Socket sender = context.createSocket(SocketType.PUSH);
7      sender.bind("ipc:///tmp/zmq-response.sock");
8
9      System.out.println("Listening now");
10     while (true) {
11       receiver.recv();
12       sender.send(payload);
13     }
14   }
15   catch (Exception e) {
```

```
16      System.out.println("Reader is shutting down");
17    }
18  }
```

Listing 9: Antwort im Reader Thread

Im Main-Programm musste dafür nur der Connect-String entsprechend angepasst werden bei der Erstellung der Socket-Objekte.

3.4 Teil B (Inter-Process) - Ergebnis

Da die Ergebnisse und somit die Laufzeit nun eine weitere Zehnerpotenz größer wurden, habe ich zum Ausgleich die Anzahl der Versuchsdurchläufe auf 1000 reduziert. Dadurch leidet die Präzision des errechneten Konfidenzintervalls, der Unterschied im Vergleich zu den vorhergegangenen Aufgaben ist aber dennoch offensichtlich. Hier ein Histogramm der Messwerte:

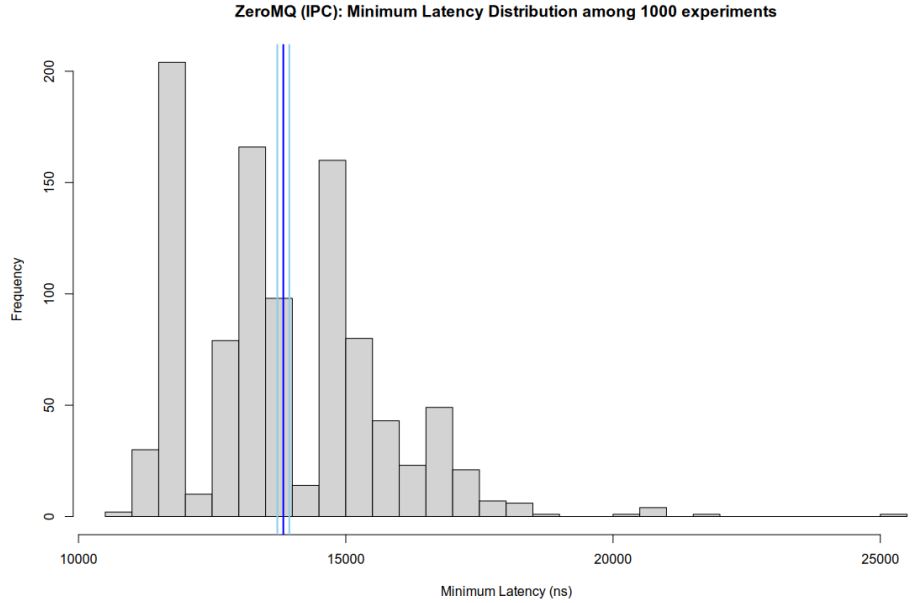


Abbildung 4: Verteilung der Minima in der Versuchsreihe zu Aufgabe 3B

Auch hier ist nur mit etwas Phantasie eine Normalverteilung zu erkennen, da die Ergebnisse recht lückenhaft sind. Trotzdem gehe ich hier wieder von einer Normalverteilung aus und berechne entsprechend das in Blau dargestellte 95%-Konfidenzintervall:

- Durchschnittliche Minimallatenz: $13,83\mu s$
- Untere Schranke des 95%-Konfidenzintervalls: $13,72\mu s$
- Obere Schranke des 95%-Konfidenzintervalls: $13,94\mu s$

4 Aufgabe 4 - Kommunikation zwischen Docker-Containern

4.1 Versuchsaufbau / Implementierung

Auch hier konnte ich wieder auf die Vorarbeit aus den vorherigen Aufgaben zurückgreifen: Da ZeroMQ auch die Kommunikation über TCP unterstützt, ist es sehr gut geeignet, um auch über

ein Docker-Netzwerk zwischen Docker-Containern zu kommunizieren. Diesmal mussten dafür ausschließlich die Socket-Bezeichnungen im Java-Code aus Aufgabe 3B angepasst werden und der Dateiname der Output-Datei. Zusätzlich dazu habe ich je ein Dockerfile für den Reader und den Writer erstellt und schließlich das ganze Projekt mit Netzwerk-Setup und Volume-Einbindung in einem `compose.yaml` zusammengefasst.

4.2 Ergebnis

Die Ergebnisse liegen in einer ähnlichen Größenordnung wie jene aus Aufgabe 3B, daher war keine weitere Verringerung der Wiederholungen notwendig. Hier die Ergebnisse:

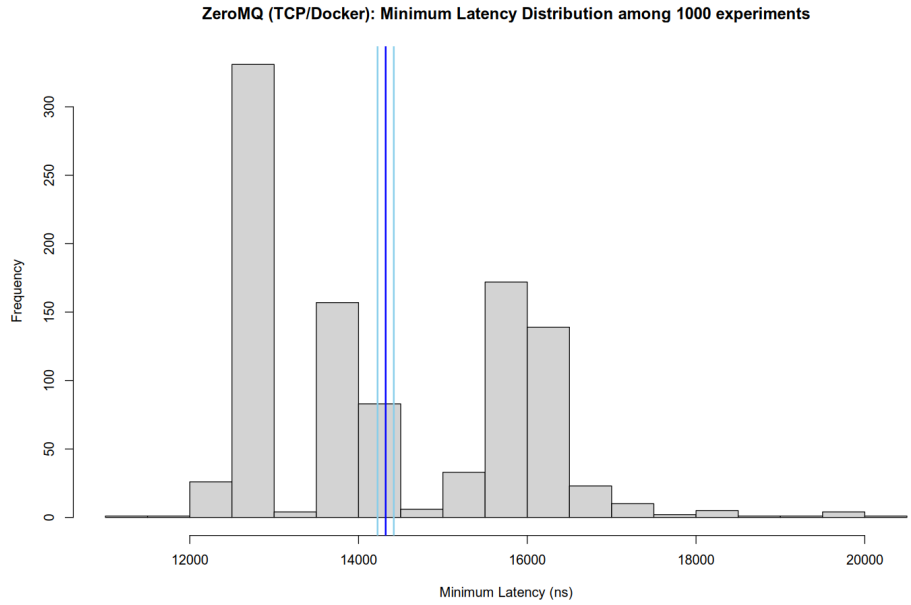


Abbildung 5: Verteilung der Minima in der Versuchsreihe zu Aufgabe 4

Auch hier liegt nicht wirklich eine Normalverteilung vor, trotzdem hier nochmal die Eckdaten des in Blau eingezeichneten Konfidenzintervalls:

- Durchschnittliche Minimallatenz: $14,32\mu s$
- Untere Schranke des 95%-Konfidenzintervalls: $14,23\mu s$
- Obere Schranke des 95%-Konfidenzintervalls: $14,42\mu s$

Somit liegt die minimale Latenz im Durchschnitt etwas höher als die Latenz bei IPC-Kommunikation. Der Overhead durch Docker scheint also sehr gering zu sein, da die Verwendung von TCP an sich bereits einen Overhead mit sich bringt.

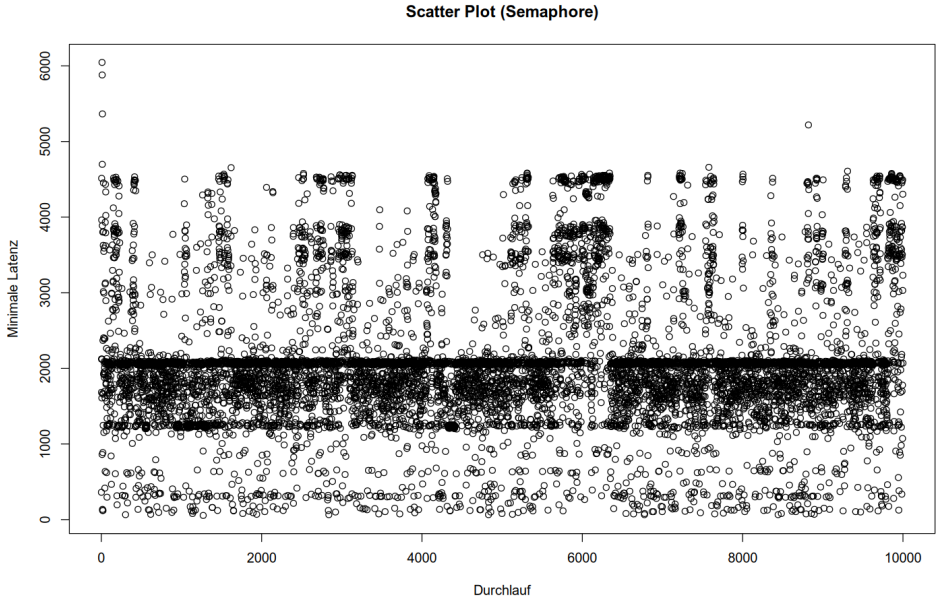
5 Abschließende Gedanken

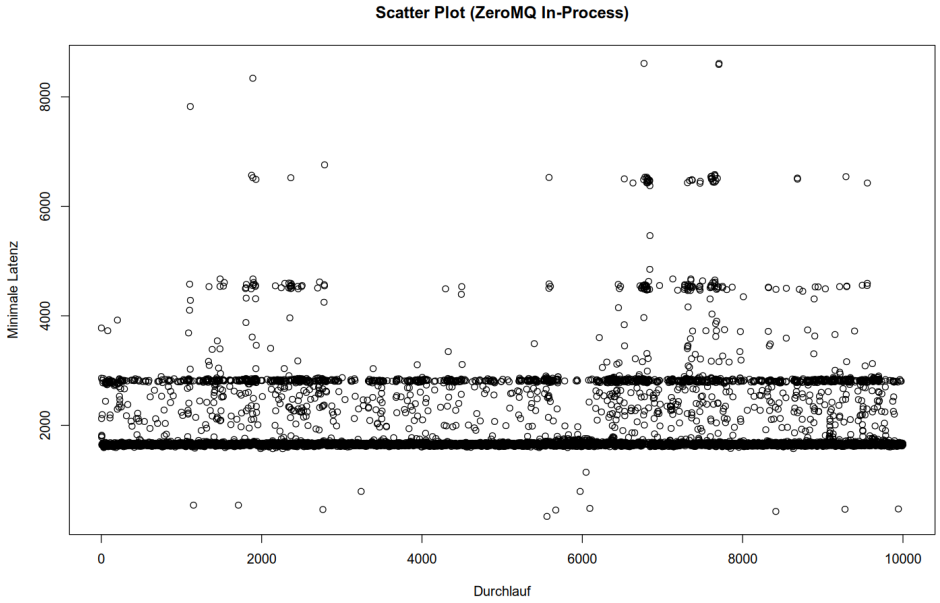
Wie zu erwarten war, ist die Latenz bei Nutzung eines Spinlocks am geringsten, da hier nur wenige Maschineninstruktionen im Prozess involviert sind. Verwendet man Semaphoren, so nimmt die Latenz bereits deutlich zu, jedoch wird dadurch der Overhead an verschwendeter Rechenleistung vermieden, den ein Spinlock erzeugt. Erstaunlich finde ich, dass die Kommunikation über ZeroMQ sogar eine etwas geringere Latenz aufweist als die Kommunikation per

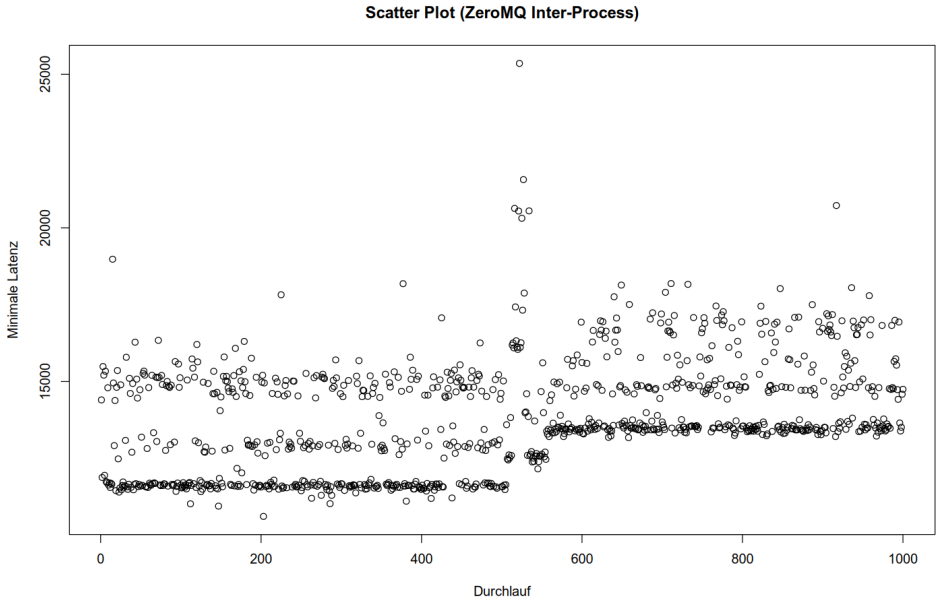
Semaphore, da die Semaphore ja bereits ein Feature auf Betriebssystem-Basis ist. Ad-Hoc kann ich mir diesen Vorteil auch nicht erklären.

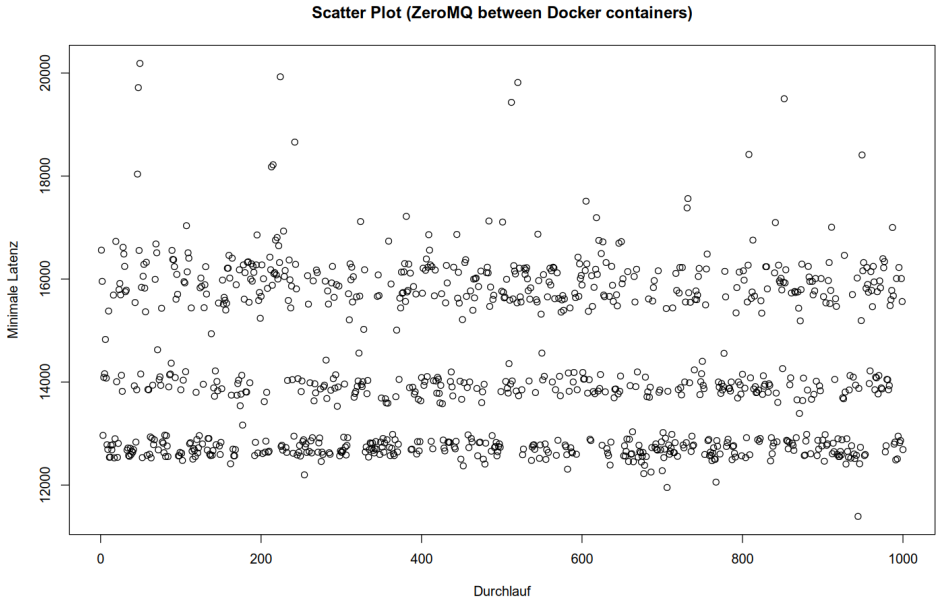
Je weiter man nun die Distanz von Empfänger und Sender erhöht, desto mehr steigt auch die Latenz - so weit, so erwartbar. Hier ist allerdings interessant zu sehen, wie wenig Unterschied es macht, ob Sender und Empfänger nur in verschiedenen Prozessen laufen und über UNIX domain sockets kommunizieren, oder ob sie in verschiedenen Docker-Containern laufen und per TCP miteinander kommunizieren.

Ein Punkt, den ich noch ansprechen möchte, ist die Verteilung der Ergebnisse. Oftmals lässt sich in den Ergebnissen, anders als meiner Ansicht nach zu erwarten wäre, keine Normalverteilung erkennen. Stattdessen sieht man schiefe Verteilungen und sogar Verteilungen mit mehreren lokalen Maxima. Um mir die Sache nochmal besser zu veranschaulichen, habe ich Scatterplots der Versuchsreihen erstellt, in denen diese Tendenz nochmal besser sichtbar wird:




Scatter Plot (ZeroMQ In-Process)
Minimale Latenz
2000
4000
6000
8000
0
2000
4000
6000
8000
10000
Durchlauf


Scatter Plot (ZeroMQ Inter-Process)
Minimale Latenz
15000
20000
25000
0
200
400
600
800
1000
Durchlauf



Besonders bei den Versuchen mit Semaphoren und ZeroMQ mit In-Process-Kommunikation sind hier die Linien sehr deutlich zu sehen, aber auch in den anderen beiden Versuchen wird eine solche Tendenz sichtbar.

Meine Theorie dazu ist, dass auch bei Semaphoren und ZeroMQ eine Art von Polling verwendet wird, um festzustellen, ob die Semaphore nun frei geworden ist bzw. ob eine Nachricht in der Queue anliegt. Wenn also zum Zeitpunkt t_1 gepollt wird und keine Nachricht anliegt, so vergeht mindestens eine konstante Zeit Δt bis erneut gepollt wird und eine Chance besteht, dass die Nachricht gelesen wird, selbst wenn sie eigentlich schon sofort nach dem Verstreichen von t_1 gelesen werden könnte.

Auch fällt ins Auge, dass bei der IPC-Variante (Aufgabe 3B) nach der Hälfte der Versuche die minimalen Latenzen für den Rest der Versuchsreihe stark ansteigen. Dieses Phänomen ist bei mehreren Versuchen immer wieder aufgetreten und mir fällt keine vernünftige Erklärung dafür ein.

6 Anleitung zum Selber-Testen

Damit die Tests auch von motivierten Leser*innen nachgestellt werden können, habe ich in den entsprechenden Ordnern README-Dateien mit Instruktionen platziert.

Die R-Skripte, die für die Auswertung verwendet wurden, liegen im Ordner `statistics`. Dort muss vor der Verwendung das Arbeitsverzeichnis (je erster Befehl) geändert werden auf den tatsächlichen Ort des `statistics`-Ordners. Oder R direkt dort ausführen, das sollte auch gehen. Ich empfehle RStudio, um die Befehle Schritt für Schritt auszuführen.