Verteilte Systeme
Übungsblatt 1

Kilian Bartz (Mknr.: 1538561)

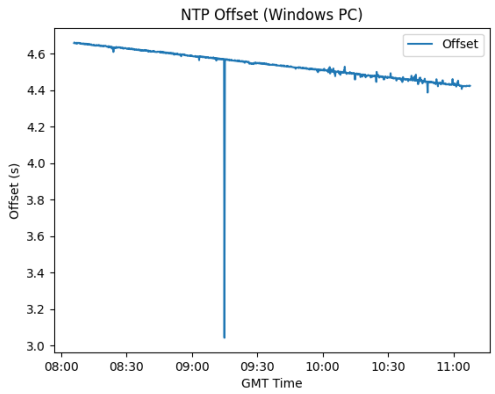
Hinweis. Der zugehörige Code befindet sich im folgenden Github Repository: `https://github.com/kilianbartz/VerteilteSysUeb01`.

Aufgabe 1

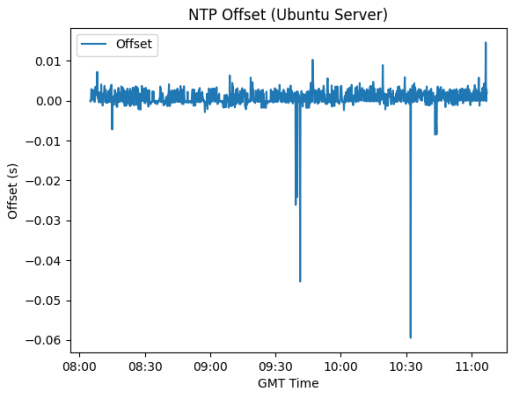
Zur Untersuchung der Genauigkeit der Uhrensynchronisation habe ich ein Python-Skript geschrieben, welches mithilfe der ntplib-Library[1] den Offset der lokalen Uhr zum Zeitstempel, die er bei korrekt synchronisierter Uhr haben sollte, berechnet. Dieser Offset wird konkret wie folgt berechnet

$$\text{offset} = \frac{t_{server.receive} - t_{client.send} + t_{server.send} - t_{client.receive}}{2}$$

Dabei beschreibt $t_{client.send}$ den Zeitpunkt (lokale Zeit), an dem der Client einen NTP-Request versendet, $t_{server.recieve}$ den Zeitpunkt (Serverzeit), zu dem der Server den Request erhält, $t_{server.send}$ den Zeitpunkt (Serverzeit), zu dem der Server die Response verschickt und $t_{client.recieve}$ den Zeitpunkt (lokale Zeit), an dem der Client die Response erhält. Dies berücksichtigt also sowohl die Netzwerklatenz für Request und Response, als auch die Zeit, die der Server zur Bearbeitung der Anfrage benötigt. Die obige Berechnung ist unter Beachtung der Verarbeitungszeit des Servers äquivalent zum in der Vorlesung vorgestellten Angleichen der Uhr auf $t_{server.send} + 0,5 \cdot \text{delay}$.
Den oben beschriebenen offset habe ich in einem Intervall von 10 Sekunden für jeweils 3 unterschiedliche Server berechnet (de.pool.ntp.org, time.google.com, time.windows.com) und dann gemittelt. Insgesamt wurde das Experiment über einen Zeitraum von 3 Stunden durchgeführt. Zum Vergleich zwischen einem Windows PC und einem Ubuntu Server wurde das Experiment zusätzlich zeitgleich auf zwei Rechnern durchgeführt.



(a) Abweichung zwischen lokaler Uhr (Windows PC) und verschiedenen NTP-Servern



(b) Abweichung zwischen lokaler Uhr (Ubuntu Server) und verschiedenen NTP-Servern

Abbildung 1: Vergleich der Abweichung der lokalen Uhr zwischen einem Windows Rechnung und einem Ubuntu Server

Vergleicht man nun die Abweichung NTP-Offsets zwischen einem Windows PC (Abbildung 1a), bei dem zuletzt 4 Tage vor dem Experiment eine Synchronisierung erfolgt ist, mit einem Ubuntu Server (Abbildung 1b), der gemäß Konfiguration alle 32 Minuten eine Synchronisierung vornimmt, so sieht man direkt einen den Effekt der Synchronisierung. Die Offsets beider Rechner zeigen Ausreißer, die sich vermutlich durch Last auf den Rechnern oder NTP-Servern ($\Rightarrow$ ungenaue Zeitstempel) und v. a. durch Unterschiede und temporäre Probleme beim Netzwerkrouting erklären lassen. Während in den Offsets des Ubuntu Servers kein eindeutiges Muster erkennbar ist (die Uhr geht im Mittel weder zu schnell noch zu

[1] `https://github.com/cf-natali/ntplib`

langsam) und die Werte überwiegend zwischen +0,01s und -0,01s schwanken, hat der Windows PC mit über 4s eine viel höhere Ungenauigkeit. Außerdem ist hier ein stetiger Abwärtstrend erkennbar, die lokale Uhr des Windows PC läuft also zu schnell. Würde der Windows PC eine häufigere Synchronisierung durchführen (z.B. alle 10 Minuten), sollte dies nicht mehr auf einem entsprechenden Offset Plot zu erkennen sein.

Aufgabe 2

Zur Durchführung der Experimente habe ich eine Rundendauer von 4 Sekunden und Spielerlatenzen von jeweils 8 Sekunden gewählt. Jedes Experiment umfasst 300 Runden. Erhält ein Client eine STOP Nachricht, verwirft er seine aktuelle Kalkulation und wartet auf den Start der nächsten Runde, um nicht immer weiter zurück zu fallen. Die folgenden drei Systeme wurden verwendet:

1. Name: server. Server und erster Client. Standort: Helsinki. Latenz ca. 0
2. Name: win. Client. Standort: Bangkok*. Latenz ca. 725ms
3. Name: pi. Client. Standort: London*. Latenz ca. 53ms

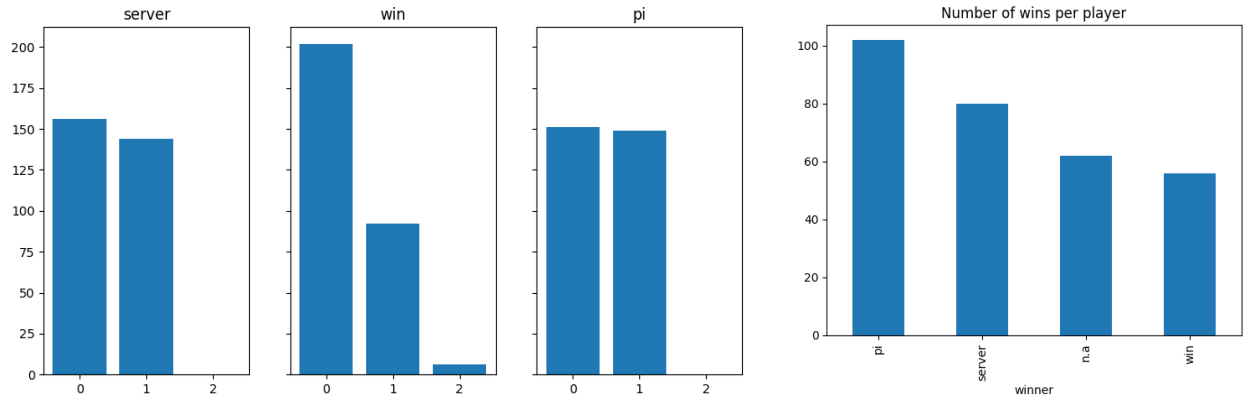
*: Diese Standorte wurden mithilfe von VPN simuliert.

2a

Da es ohne das Austauschen der aktuellen Zeiten zwischen Clients und Servern nicht möglich ist, die Nummer der aktuellen Runde zu synchronisieren, ist es mit diesem Ansatz unmöglich für den Server zu erkennen, ob ein Wurf zu spät eingetroffen ist (also nicht in der aktuellen Runde gewertet werden sollte), oder nicht. Dementsprechend können Systeme an einer Runde

- gar nicht teilnehmen (SPIELER_LATENZ + Übertragungslatenz > DAUER_DER_RUNDE),
- genau 1x teilnehmen (SPIELER_LATENZ+Übertragungslatenz < DAUER_DER_RUNDE),
- 2x an einer Runde teilnehmen (der Wurf aus der letzten Runde trifft erst in der aktuellen Runde ein und SPIELER_LATENZ (aktuelle Runde)+Übertragungslatenz < DAUER_DER_RUNDE).

Da die SPIELER_LATENZ in meinem Versuchsaufbau aus einer Gleichverteilung im Intervall [0,8] gewürfelt wird, sollten die Spieler im Erwartungswert nur an jeder zweiten Runde teilnehmen.



(a) Histogramm der Anzahl der Teilnahmen an einer Runde pro Spieler

(b) Zahl der Siege pro Spieler

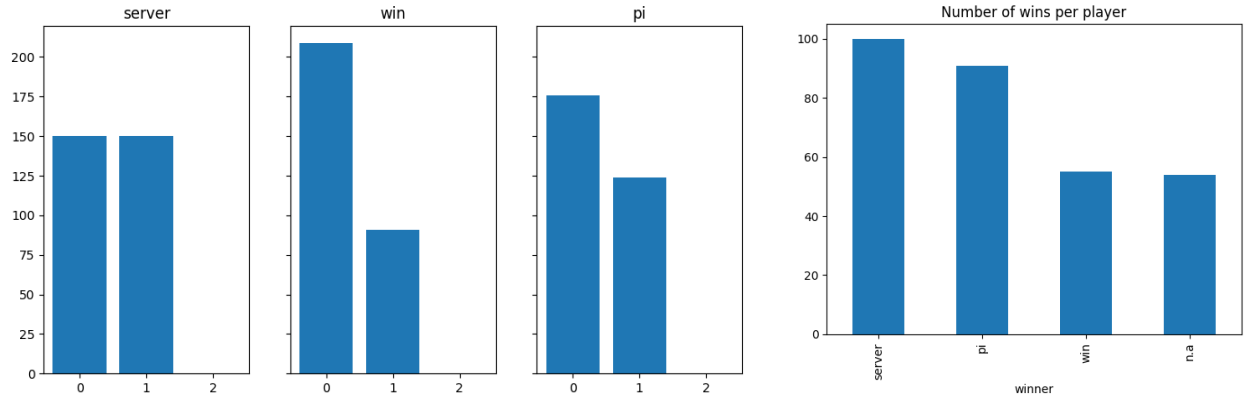
Abbildung 2: Auswertung der Ergebnisse von Experiment 2

Betrachtet man die Ergebnisse (Abbildung 2), so ist dieses erwartete Verhalten auch bei System 1 (Latenz nahe 0) und System 3 zu beobachten. Insbesondere kommt es nie dazu, dass ein System 2x an derselben Runde teilnimmt. Im Gegensatz

führt die hohe Latenz von System 2 dazu, dass Nachrichten des Clients den Server oft erst zu spät erreichen, wodurch es erheblich öfter vorkommt, dass dieses System nicht an einer Runde teilnimmt. Außerdem tritt hier auch der Fall ein, dass die STOP Nachricht des Servers zu spät den Client erreicht, sodass ein verspäteter Wurf bereits auf dem Weg ist und in der nächsten Runde gewertet wird. In besagter Runde würfelt System 2 jedoch eine geringe Wartezeit, sodass auch der reguläre Wurf dieses Systems in dieser Runde ankommt, was in einer doppelten Teilnahme des Systems an dieser Runde resultiert. Diese doppelte Teilnahme verdoppelt zwar die Gewinnchancen in einer entsprechenden Runde für System 2, dies passiert jedoch nicht oft genug, um die große Anzahl an Runden ohne Teilnahme auszugleichen, wodurch System 2 mit Abstand die wenigsten Runden gewinnt.

2b

Durch das Verwenden von logischen Uhren ist es möglich zu erkennen, wenn eine Nachricht verspätet eintrifft, dazu ist bereits die einfache Lamportzeit in der Lage. Um in einer Runde gültig zu sein, muss die Nachricht kausal abhängig von der START Nachricht sein, dementsprechend muss die Lamportzeit, die in der Nachricht mitgesendet wird, in jedem Fall größer sein als die Lamportzeit der START Nachricht. Ebenso muss die Nachricht angekommen sein, bevor die Runde zu Ende war, also muss die Lamportzeit der Nachricht auch kleiner sein als die Lamportzeit der STOP Nachricht. Falls also $lc_{wurf} \leq lc_{START}$, so gehört der Wurf zu einer vorherigen Runde und sollte in der aktuellen nicht beachtet werden. Ebenso sind Würfe mit $lc_{wurf} \geq lc_{STOP}$ zu spät beim Server angekommen und sollten verworfen werden.
Obwohl es durchaus möglich wäre, die genaue Runde zu ermitteln, zu der eine Nachricht gehört, wenn zu jeder Runde die Lamportzeiten von START und STOP Nachricht gespeichert werden, halte ich es in einem Würfelspiel für unfair, nachträglich einen anderen Sieger auszuwählen. Deshalb halte ich es für die beste Option, solche Situationen als vertane Chancen der zu langsamen Spielern zu werten und die entsprechenden Nachrichten komplett zu verwerfen.
Hinweis. Aufgrund VPN verschuldeter Verbindungsabbrüche wurde der Code für diese Teilaufgabe so modifiziert, dass Nachrichten nach einem Delay von 2s/4s erneut versendet werden. Dies sollte die Ergebnisse des Experiments jedoch nicht signifikant ändern.



(a) Histogramm der Anzahl der Teilnahmen an einer Runde pro Spieler

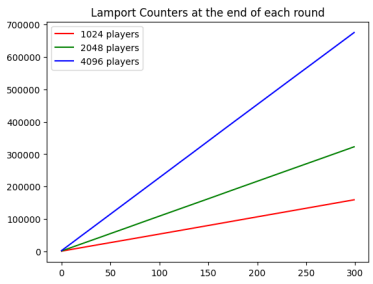
(b) Zahl der Siege pro Spieler

Abbildung 3: Auswertung der Ergebnisse von Experiment 2

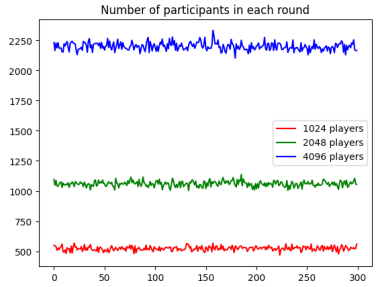
Mit Blick auf die Ergebnisse (Abbildung 3) ist ersichtlich, dass die Verwendung von logischen Uhren das Problem verspäteter Nachrichten nicht beheben kann; System 2 nimmt durch die hohe Latenz an den meisten Runden noch immer nicht teil, wodurch es immer noch eine erheblich schlechtere Siegesquote im Vergleich zu den anderen Teilnehmern hat. Jedoch erlauben sie das Detektieren von Würfen, die nicht innerhalb einer Runde getätigt wurden. Somit können unfaire Runden, in denen einzelne Teilnehmer zwei Würfe einbringen können, ausgeschlossen werden. Mithilfe der logischen Uhren zeigt sich auch der Unterschied zwischen System 1 (keine Latenz) und System 3 (ca. 53ms) deutlich. Während es in Aufgabe 2 wirkte, als ob durch die geringe Latenz von System 3 alle Nachrichten pünktlich beim Server eintrafen, sieht man nun, dass ein Teil der Nachrichten von System 3 tatsächlich zu spät den Server erreichen und fälschlicherweise in der aktuellen Runde gewertet wurden. Erwartungsgemäß steigt dieser Anteil mit steigender Latenz (System 2) weiter.

2c

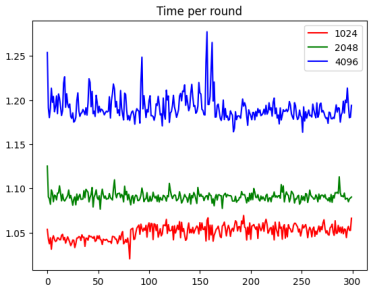
Das Problem, mit einem Server möglichst viele gleichzeitige Netzwerkverbindungen bereitzustellen, ist ein bekanntes Problem mit dem Namen C10k[2]. Meist ist das Problem nicht in der Durchsatz, sondern effizientes Scheduling, da der Server die meiste Zeit damit verbringt, auf Nachrichten der Clients zu warten. Dies ist auch in diesem Fall so, da der Server als Spielleiter nach Versenden der START Nachricht bis zum Versenden der STOP Nachricht nur auf Antworten der Spieler wartet.
Die simple Lösung, die ich für Aufgabe 2b implementiert habe, erzeugt auf dem Server $\#Spieler + 2$ Threads (ein handle_player Thread pro Spieler + start_server Thread + start_game Thread). Zusätzlich erzeugt jeder Client einen zusätzlichen Thread nach dem Empfangen der START Nachricht, dementsprechend liegt die Gesamtzahl der Threads zwischen $2 \times \#Spieler + 2$ und $3 \times \#Spieler + 2$ und überfordert damit schnell das System. Zusätzlich zu diesen Threads, die alle durch das System geplant/verwaltet werden müssen, werden auch $\#Spieler$ Sockets benötigt, die auf Unix-Systemen durch spezielle File Descriptors repräsentiert werden, womit auch die maximale Anzahl gleichzeitig geöffneter Dateien der limitieren kann. Mithilfe von asynchronen Runtimes (rust tokio, Python mit asyncio, Java Virtual Threads, o. ä.) könnte man eine derartige Anwendung deutlich effizienter umsetzen. Wie in Abbildung 4 ersichtlich, steigt der Lamportcounter erwartungsgemäß sehr schnell, wenn viele Nachrichten verschickt werden. Allerdings ist dies unproblematisch, solange ein passender Variblentyp für den Counter (z. B. u64) gewählt wird (bei einer Sekunde pro Runde und 4096 Spielern mit SPIELER_LATENZ = 2s könnten etwa. 9.7×10^{10} Tage gespielt werden). Die Verarbeitungszeit pro Runde bleibt (abgesehen von Ausreißern) stabil, die zunehmende Nachrichtenlänge durch größere Lamportcounter ist also zunächst vernachlässigbar. Zudem liegt die Zahl der Teilnehmer pro Runde nahe am Erwartungswert von $0.5 \times \#Spieler$, wenig überraschend funktioniert die Lamportzeit also auch für sehr viele Clients zuverlässig.



(a) Lamport Counter (Server) am Ende jeder Runde



(b) Anzahl der Teilnehmer pro Runde bei 1024, 2048 und 4096 Spielern



(c) Verarbeitungszeit pro Runde des Servers (s)

Abbildung 4: Auswertung der Ergebnisse von Experiment 2

Zum Aufwand des Einsatzes eines solchen verteilten Systems

Es ist nicht einfach, verteilte Software zu schreiben. Insbesondere der Austausch der Nachrichten führt bei vielen Clients (2c) schnell zu Problemen (Nachricht wird nicht sofort versendet und mehrere Nachrichten erreichen die andere Seite gleichzeitig, Verbindungen können abbrechen, etc.). Der von mir gewählte Ansatz, einen Thread pro Client zu verwenden, erfordert außerdem die Verwendung eines Locks/Mutex, was die Zeit, die Prozesse mit Warten verbringen müssen noch vergrößert und somit die Performance verschlechtert.
Nicht zuletzt ist der bereits Aufwand, dieses simple Programm auf n physischen Rechnern einzurichten, immens. Potenziell muss auf jedem Rechner Python installiert werden (in diesem Fall wurden keine externen Abhängigkeiten verwendet, die ansonsten auch installiert werden müssten). Zudem muss das Programm selbst auf den Rechnern installiert (im Falle eines Python-Programms ist dies eine Kopie des Source Codes) und gestartet werden. Dies bedeutet gleichzeitig einen sehr hohen Wartungsaufwand, da jedes System einzeln aktualisiert werden müsste, sobald es eine neue Version des Programms gibt. Realistisch gesehen ist dieser Aufwand nur zu bewältigen, wenn die Software als Container vorliegt und sämtliche Rechner zusammen in einem kubernetes-Cluster o. ä. (zentral) orchestriert werden können.

[2] `https://en.wikipedia.org/wiki/C10k_problem`