Interprozesskommunikation – Werkstück A
SS2022

Arne Chris Müller (Sockets)

Falko-Hennig Roderich Claus Köning (Message Queues)

Franz Ferdinand Blaschke (Shared Memory)

Jasdip Erhart (Pipes)

Abstract

Interprozesskommunikation erlaubt laufenden Prozessen auf einem Betriebssystem eine sichere Art der Kommunikation. Wie die Synapsen eines Gehirns, bieten sie die Infrastruktur für den Austauschen von Nachrichten. In der folgenden Dokumentation wird dieser Mechanismus anhand von vier Varianten der Prozesskommunikation veranschaulicht.

Problemstellung

Die Herausforderung bei der Implementierung von Interprozesskommunikation ist es, diese mit Python auf Windows bzw. Unix Betriebssystemen zu verwirklichen. Anders als mit den für Betriebssystemen üblichen Programmiersprachen, beispielweise C, gibt es hier keine Dokumentationen oder Lösungsansätze, die studiert werden könnten. Eine Lösung musste somit, mit einfachen Beispielen und der Python Docs Dokumentation, eigenständig entworfen werden. Zudem galt es die Simultanverarbeitung von Multicore-Prozessoren oder symmetrische Mehrfachprozessorarchitektur (SMP) zu unterstützen.

Shared Memory

Um Prozessen auf einem Rechner mit Multicore-Prozessor und Windows Betriebssystem eine Prozesskommunikation zu ermöglichen, gibt es einige Ansätze, die in den folgenden Abschnitten dargestellt werden. Diese Ansätze finden sich unter anderem in dem Python multiprocessing Modul[1], welches seit Python 2.6 eine Lösung für Mehrrechnerverarbeitung von Prozessen bietet[7].

Die Shared Memory Map[7] (Value[8] und Array[9])

Diese Methode gibt durch den Aufruf multiprocessing.Value (typecode_or_type, args) ein ctype-Objekt zurück, welches aus einem gemeinsamen Speichersegment zugewiesen wird. Wobei der Rückgabewert der Methode in einen Wrapper verpackt ist. Es gilt, diese Methode bei der Verwendung von shared state-Variablen zu vermeiden, besonders beim Einsatz von vielen parallellaufenden Prozessen. Objekte dieser Art sind sowohl vor Prozessen geschützt als auch Thread-sicher.

SharedMemory Klasse[2]

Mit dieser Klasse können Prozesse ein neues gemeinsames Speichersegment erzeugen oder sich mit einem bereits existierendes Speichersegment verbinden. Die close()[5] Methode ermöglicht Prozessen sich von dem Speichersegment zu lösen, ohne dieses dabei zu löschen. Ruft ein Prozess die unlink()[6] Methode auf, wird das Segment gelöscht und der Speicher ordnungsgemäß bereinigt.

SharedMemoryManager[3] Klasse und ShareableList[4]

Das Multiprocessing-Modul stellt noch den SharedMemoryManager als auch die ShareableList zur Verfügung. Wobei der Manager über viele Prozesse hinweg die Verwaltung von gemeinsamen Speichersegmenten ermöglicht. Die ShareableList kreiert und gibt ein Objekt zurück, welches aus einem gemeinsamen Speichersegment zugewiesen wird und einer Python List nicht unähnlich ist. Dazu sei gesagt, dass die SharableList nicht ihre Länge verändern kann und kein slicing erlaubt[4].

Für die Prozesskommunikation eines A-/D Converters mit einer Funktion zur Dokumentation und einer statistischen Auswertung werden lediglich natürliche Zahlenwerte übermittelt. Nach der statistischen Auswertung wird wiederrum ein ganzer Zahlenwert und zwei natürlich Zahlenwerte an eine Report-Funktion übermittelt. Da es sich hier um eine einfache und wenig dynamische Anforderung handelt, wird in dem folgenden Projekt das gemeinsame Speichersegment mit der Value Methode erstellt.

Die Main Methode

Das Programm versorgt den Benutzer zunächst mit generellen Informationen zum Ablauf und erstellt eine Textdatei die später für die Berechnung der Summe mit einer ‘0‘ beschrieben wird. Anschließend werden Signal-Handler und Converter-Prozess gestartet.

Der Conv. Prozess

Zur Überwachung der Durchläufe bzw. späteren Berechnung des Mittelwerts initialisieren wir einen Durchlaufzähler lap_counter = 0.

Die while True Endlosschleife stellt sicher, dass Conv. regelmäßig an die empfangenden Prozesse sendet und zählt mit lap_counter += 1 die Durchläufe. Anschließend wird eine zufällige Zahl zwischen 0 und 100 erstellt und in einer Variabel, namens random_integer, gespeichert. Sleep(0.8) gibt der Schleife einen 0.8 Sekunden Takt vor.

In jedem Durchlauf wird eine Variable, mit der zufälligen Zahl, in einem Wrapper verpackt und in einem gemeinsamen Speichersegment erzeugt. Dazu wird der Datentyp der Variable, in diesem Fall ‘i‘ für Integer, angegeben und unter dem Namen shm_numbers gespeichert.

```
shm_numbers = multiprocessing.Value('i', random_integer)
log1 = multiprocessing.Process(target=log, args=(shm_numbers,))
log1.start()
log1.join()
```

Anschließend wird ein Prozess, log1, gestartet, welcher wiederrum die log Funktion aufruft und die Variable aus einem gemeinsamen Speichersegment übergibt.

join() hat die Funktion den Elternprozess zu stoppen[1], bis der Kinderprozess fertig ausgeführt wurde. Dies sorgt in diesem Fall, dass keine Zahl von Conv. erstellt wird, die Log. und Stat. nicht bekommen.

Der Log. Prozess

Die Aufgabe von Log. ist die prozessübergreifende Datensicherung aller zufälligen Zahlen des Converters. Dazu wird beim ersten Durchlauf eine Textdatei erstellt auf die in den folgenden Durchläufen zugegriffen werden kann. Bei jedem Durchlauf schreibt Log. die Zahl in die Textdatei und schließt diese anschließend.

Der Stat. Prozess

Stat. soll statistische Daten errechnen und diese an einen weiteren Prozess übergeben. Ziel ist, jede Zahl, die der Converter verschickt, aufzusummieren und anschließend den Mittelwert zu bestimmen. Auch hier wird zur prozessübergreifenden Datensicherung eine Textdatei verwendet, welche zu Beginn des Programms in der Main Methode mit einer ‘0‘ beschrieben wurde und nach jedem Durchlauf von Stat. wieder mit der neuen Summe beschrieben wird.

Aus dieser Textdatei wird die letzte Summe gelesen und mit der zufälligen Conv. Zahl verrechnet. Der Durchlaufzähler ermöglicht anschließend eine Berechnung des Mittelwerts. Die Summe wird anschließend in dem Dokument gespeichert, welches zuvor mit der Berechtigung ‘w+‘ bereinigt wurde.

Stat. öffnet im weiteren Verlauf ein gemeinsames Speichersegment und übergibt Report den Durchlaufzähler, Mittelwert und Summe. Auch hier wartet Stat. wieder durch den join() Befehl auf die Ausführung von Report.

Report Prozess

Hier werden mit einem Print-Statement die statistischen Daten in der Shell ausgegeben. Um die Lesbarkeit zu steigern, wird hier auch der Durchlaufzähler übermittelt. Somit bleibt der Rechenweg leicht nachzuvollziehen.

Signalhandler Prozess

Zur ordentlichen Beendigung des Programmes wurde ein Signalhandler für das Abfangen des SIGINT Befehls implementiert. Zur Sicherheit, werden hier auch ein letztes Mal die verwendeten Textdateien geöffnet und ihr Inhalt gelöscht. Danach wird das Programm geschlossen.

Bewertung meiner Ergebnisse

In der Lösung übermitteln alle Prozesse ihre Daten zuverlässig und ausschließlich über gemeinsame Speichersegmente, jede Zahl die der Converter produziert wird auch von Stat., Log. und Report verarbeitet. Alle Prozesse werden mit den für Windows verfügbaren Befehlen in der Simultanverarbeitung ausgeführt, ähnlich wie mit Fork für Unix Betriebssysteme. Die Synchronisation wird durch den join() Befehl garantiert. Das Programm lässt sich wie gefordert in der Kommandozeile ausführen und kann mit Ctrl-C abgebrochen werden.

Somit lässt sich, außer der für Windows nichtexistierenden Überwachung der Prozesse (top, ps und pstree) und das Freigeben von gemeinsamem Speicher, jede Anforderung erfüllen.

Shared Memory Fazit

Es wird deutlich, dass Python noch nicht lange zu den für Betriebssystem genutzten Programmiersprachen zählt. Zudem stellt sich heraus, dass Windows nicht die einfachste Plattform für eine solche Aufgabe bietet. So findet man sich an vielen Stellen ohne jegliche Referenz wieder und muss eigene Ideen entwickeln um Problemstellungen zu lösen. Tatsächlich bin ich stolz auf meine Umsetzung obwohl es sicherlich professionellere und dynamische Ansätze gibt.

Würde ich die gleichen Anforderungen ein weiteres Mal umsetzen wollen, wäre ich geneigt dies mit C und Linux zu tun. Generell zeigt mir dieses Projekt auf, wie umständlich die Arbeit mit Windows seien kann, so ist ein probeweiser Betriebssystem-Wechsel eine Konsequenz die ich privat aus dieser Arbeit mitnehmen werde.

Quellen:

1. https://docs.python.org/3/library/multiprocessing.html
2. https://docs.python.org/3/library/multiprocessing.shared_memory.html#multiprocessing.shared_memory.SharedMemory
3. https://docs.python.org/3/library/multiprocessing.shared_memory.html#multiprocessing.managers.SharedMemoryManager
4. https://docs.python.org/3/library/multiprocessing.shared_memory.html#multiprocessing.managers.SharedMemoryManager.ShareableList
5. https://docs.python.org/3/library/multiprocessing.shared_memory.html#multiprocessing.shared_memory.SharedMemory.close
6. https://docs.python.org/3/library/multiprocessing.shared_memory.html#multiprocessing.shared_memory.SharedMemory.unlink
7. https://docs.python.org/2.7/library/multiprocessing.html?highlight=shared%20memory
8. https://docs.python.org/2.7/library/multiprocessing.html?highlight=shared%20memory#multiprocessing.Value
9. https://docs.python.org/2.7/library/multiprocessing.html?highlight=shared%20memory#multiprocessing.Array

Message Queues

Das in der Aufgabenstellung geforderte Programm soll vier Prozesse gleichzeitig ausführen können und diese Prozesse sollen auch noch mit einander kommunizieren können, Python bietet zum Beispiel durch das multiprocessing Modul Prozesskommunikation an. Eine Message Queue in Python kann man durch den Aufruf multiprocessing.Queue() erstellen.

Eine Message Queue kann man sich am besten wie einen Restaurant Betrieb vorstellen, ein Kellner nimmt die Bestellungen auf und hängt sie an ein schwarzes Brett in der Küche, die Bestellungen am schwarzen Brett werden dann der Reihenfolgen entsprechend von den Köchen abgearbeitet. In dem Beispiel ist das schwarze Brett die Message Queue, die Bestellungen die von den Kellnern an das schwarze Brett gehängt werden entsprechen dem Befehl „queue.put()“ und die Köche, welche die Bestellungen dann vom schwarzen Brett entfernen und bearbeiten entsprechen dem Befehl „queue.get()“.

```
schwarzesBrett = Queue()

def Kellner(schwarzesBrett):
    while True:
        BestellNr = randint(1,49)
        schwarzesBrett.put(BestellNr)

def Kueche(schwarzesBrett):
    while True:
        Gericht = schwarzesBrett.get()
        sleep(1)
        print("Gericht Nummer ",Gericht," ist fertig!")
```

Bevor ich verstanden habe wie Message Queues funktionieren hatte ich überlegt meinen Code anders zu strukturieren, da ich keine Idee hatte, wie ich die in conv() generierte Zufallszahl an log() und stat() weitergeben konnte und wollte daher conv() nicht als eigene Methode implentieren, sondern conv() als globale Variable in die Main Methode schreiben und so an log() und stat() weitergeben, aber offensichtlich ist die Variante mit Message Queues schöner und viel sinnvoller, wieso ich meinen Code so und nicht anders entwickelt habe, lässt sich eigentlich mit nur einem Satz beantworten – bei der Ausführung des Codes wurde keine Fehlermeldung ausgegeben.

def conv()

In diesem Prozess wird zunächst eine Endlos Schleife erstellt „while True:“, welche dafür sorgt, dass der nachfolgende Code endlos ausgeführt wird. In der Schleife wird dann zufällig eine Zahl zwischen 1 und 100 erstellt, welche dann durch queue.put() in die Message Queue weitergegeben wird, so das andere Prozesse auf die Zufallszahl zugreifen können, abschliessend wird dem Programm über den Befehl sleep() noch gesagt, dass er 0,8 Sekunden pausieren soll bis er den Code erneut ausführen soll.

def log()

Hier wird ebenfalls zuerst eine Endlos Schleife erstellt (analog zu def conv()). In der Schleife wird zunächst die in der Message Queue befindliche Zufallszahl an eine Variable übergeben und in einen String umgewandelt. Nun wird eine Textdatei erstellt und geöffnet, fortlaufend werden die in conv() erstellten Zufallszahlen in diese Textdatei geschrieben und gespeichert. Nach jedem schreiben und speichern, pausiert das Programm 0,8 Sekunden bevor es den Vorgang erneut durchführt via sleep (analog zu conv()).

def stat()

In diesem Prozess wird zunächst ein Array erstellt und danach analog zu den beiden vorangehenden Prozessen eine Endlosschleife erzeugt. In dieser Schleife wird zunächst wieder die Zufallszahl in einer Variable gespeichert (analog zu log()) und diese Variable wird dann in dem anfangs erstellten Array gespeichert. Daraufhin wird die Summe und der Durchschnitt der Zufallszahlen berechnet. Die Summe und der Durchschnitt wird dann einer zweiten und dritten Message Queue hinzugefügt. Analog zu den zwei ersten Prozessen wird das Programm 0,8 Sekunden via sleep() pausiert, bevor es erneut ausgeführt wird.

def report()

Der letzte Prozess empfängt die Summe und den Durchschnitt, welche in stat() berechnet wurden und über die zweite und dritte Message Queue weitergeschickt wurden. Diesen beiden Werte werden jetzt in der Shell ausgegeben.

Main Methode

In der Main Methode werden zuerst die drei Message Queues erstellt, anschließend werden die vier Funktionen conv(), log(), stat() und report() offiziell als Prozesse deklariert.

Zuletzt werden die Prozesse via „.start()“ gestartet und danach wird durch „.join()“ die Synchronisation sicher gestellt, da erst conv() durchlaufen muss, bevor log() und stat() die Zufallszahl auslesen können, report() startet erst, wenn stat() die Summe und den Durchschnitt berechnet hat.

Der wichtigste Teil meiner Eigenleistung ist die Kommunikation zwischen den Prozessen, die einzelnen Funktionen conv(), log(), stat() und report() kriegt man mit zehn Minuten googlen und ein wenig Stackoverflow hin, aber die Prozesse miteinander kommunizieren zu lassen und die Variablen zwischen den verschiedenen Prozessen zu übergeben ist schwieriger als es aussieht, im Nachhinein ist natürlich einfach.

Abschließend kann ich behaupten die Aufgabenstellung sinngemäß erfüllt zu haben und die Herausforderung der Interprozesskommunikation und der Endlossprozesse gelöst zu haben. Das einzige was ich nicht geschafft habe ist die Implementierung von SIGINT. Zusammengefasst im Großen und Ganzen habe ich einen funktionierendes Programm mit nur einem Makel geschrieben zu haben. Zudem bin ich glücklich darüber den Einstieg in Python gefunden zu haben.

Bei der Entwicklung meiner Lösung habe ich vorwiegend auf folgende Ressourcen zugegriffen, Quellen:

Python Doc (https://docs.python.org/3/)

Hier findet man zu jedem Befehl in Python eine Erklärung und überwiegend auch Beispiele dazu.

Python Discord Server (https://discord.com/invite/python)

Der Python Discord Server ist ein wenig wie Stackoverflow, nur eben Python Spezifisch. Dort wurde mir unteranderem gesagt das ich den Append Mode ‘a‘ und nicht den Write Mode ‘w‘ benutzen sollte, wenn ich Variablen fortlaufend in ein Textdokument schreiben möchte.

Pipes

Einführung Pipes

Was ist eine Pipe?

Grundsätzlich unterscheidet man zwischen anonyme und benannte Pipes. Pipes arbeiten nach dem FiFo-Prinzip (First in-First out). Anonyme Pipes sind ein gepuffter unidirektionaler Kommunikationskanal zwischen zwei Prozessen. Soll eine Kommunikation in beide Richtungen stattfinden, benötigt man zwei Pipes - eine für jede mögliche Kommunikationsrichtung. Anonyme Pipes werden durch den Systemaufruf „pipe()“ angelegt, dabei erzeugt der betriebskern einen Inode und zwei Zugriffskennungen (Handles). Anonyme Pipes ermöglichen eine Prozesskommunikation zwischen eng verwandten Prozessen. Mit benannten Pipes können auch nicht eng miteinander verwandte Prozesse kommunizieren. Mit Hilfe ihrer Namen ist es möglich auf die Pipe zuzugreifen. Jeder Prozess, welcher den Namen einer pipe kennt, kann so eine Verbindung herstellen und mit anderen Prozessen kommunizieren.

Mögliche Wege zur Lösung

Da es zwei verschiedene Arten von Pipes gibt, kann man ein Programm mit Hilfe von anonymen Pipes oder benannten Pipes Implementieren. Beide Möglichkeiten bieten ihre eigenen Vor- und Nachteile.

Grund meiner Lösung

Ich fand den Aufbau von benannten Pipes einfacher umzusetzen, als die Implementierung mit anonymen Pipes. Der Vorteil der benannten Pipes ist, dass jeder Prozess, welcher den Namen der Pipe kennt, eine Verbindung miteinander aufbauen kann. Da log() und stat() beide von conv() „erben“ sollten, war es für mich einfacher der conv()-Pipe einen Namen zu geben und diese mit den anderen beiden Prozessen Kommunizieren zu lassen.

Definition der einzelnen Prozesse

def conv()

Hier wird der Prozess conv() erstellt welche durch den Befehl setproctitle den Titel conv() erhält. Anschließend wird eine Schleife erstellt, welche Zufallszahlen generiert. Durch sys.stdout.write wird eine Eingabe Aufforderung erzeugt, welche durch write in eine Datei geschrieben wird. Durch sleep wird die Zeit angegeben, welche das Programm warten soll, bis die nächste Zahl generiert wird.

def log()

Hier wird der Prozess log() erstellt. Durch „setproctitle“ wird der name des Prozesses auf log() gesetzt. Anschließend wird mit open() die Datei aufgerufen und in den Modus w gesetzt die in eine String-Variable „f“ schreibt. Mit der Funktion f.truncate wird sichergestellt, dass die Variable leer ist bevor die Schleife aufgerufen wird. Daraufhin wird die variable

output = str(sys.stdin.readline()) erstellt u d wartet darauf, dass Werte von außen (später durch die Pipe) übertragen werden. Mit log.write(output) wird der String beschrieben.

def stat()

Hier wird der Prozesstitel in stat() geändert. Anschließend wird eine Liste erstellt namens values In einer Schleife wird die Variable output als integer wert gelesen (Umwandelung von String in Integer). Dann wird das Array mit den Werten aus output befüllt und mit den append()-Befehl wird das Array jeweils um ein einzelnes Element erweitert. In den nächsten zwei Schritten wird der Mittelwert und die Summe berechnet. Anschließend werden diese Werte wieder in einen String gelesen.

def report()

Der Prozesstitel wird hier mit Hilfe von setproctitle auf report() gesetzt. Anschließend wird eine Schleife erstellt, welche Werte über output = str(sys.stdin.readline()) von Output einliest. Ausgegeben werden die Werte durch sys.stdout.write(output). Die

if __name__ == "__main__"-Abfrage prüft, ob das Programm als Hauptprogramm gestartet wurde, falls ja wird nachstehender Codeblock ausgeführt. Anschließend wird geprüft, ob in sys.argv[] Argumente enthalten sind. Daraufhin werden die Argumente benannt und die einzelnen Prozesse übergeben und ausgeführt. Als nächstes werden die Pipes mit popen geöffnet und angelegt. Durch stdin und stdout wird festgelegt, welcher Prozess senden und welcher Empfangen soll. Im Fall von stat_process sogar beides. Der Befehl -u bedeutet, dass der Prozess direkt ausgeben soll was er bekommt. Es soll also nicht im bufer zwischengespeichert werden. Im Beispiel von conv_process wird stdout=Pipe im nachfolgenden Codeblock in der while-true-Schleife definiert. Bufsize heißt, dass zum Beispiel log() nicht eine gewisse Größe abwarten soll, sondern direkt Werte weiter ausgibt. Im letzte Codeblock wird sichergestellt, dass alle Prozesse auch beendet werden. Durch ctrl C im Terminal wird nur der Hauptprozess geschlossen, mithilfe der signal.SIGINT Anweisung werden alle anderen Prozesse beendet.

Pipes Fazit

Für den Aufbau von Programmen mit wenigen Endpunkten eignet sich die Pipe sehr gut, da eine Pipe im Gegensatz zu einer Queue nur zwei Endpunkte hat. In dieser Implementierung war es nur notwendig zwei Prozesse gleichzeitig miteinander kommunizieren zu lassen. Da wir so nur zwei Endpunkte hatten, war das Prinzip einer Pipe hier perfekt für die Implementierung. Der Vorteil der Pipe im Gegensatz zu einer Queue in diesem Programm ist zum Beispiel die schnellere Performance, da weniger Endpunkte vorhanden sind.

Sockets

Was sind Sockets?

Bei Sockets handelt es sich um die Kommunikationsendpunkte einer bidirektionalen Netzwerkverbindung. Es entsteht also ein Datenaustausch zwischen gleichberechtigten Kommunikationspartnern, die ihre Verbindung über das Internet, ein lokales Netzwerk oder auch auf dem gleichen Endgerät zueinander aufbauen können.

Bei den Kommunikationspartnern handelt es sich in der realen Welt klassischerweise um eine sogenannte Client-Server-Architektur. Hierbei läuft beispielsweise ein bestimmter Dienst bzw. ein Programm auf einem Computer, der den Server darstellt, welcher eine Verbindung von einem anderem Computer, der den Client darstellt, erwartet. Server und Client können sich jedoch, wie bereits erwähnt, auch auf dem gleichen Endgerät befinden. Fragt nun ein Client bei besagtem Server an, eine Verbindung zu ihm aufzubauen, so kann die dabei entstehende Netzwerkkommunikation in zwei Typen unterschiedlicher Protokollnutzung unterteilt werden.

Zum einen in Datagram Sockets, welche UDP (User Datagram Protocol) nutzen und nachrichtenorientert arbeiten, und zum anderen in Stream Sockets, welche TCP (Transmission Control Protocol) nutzen und verbindungsorientiert arbeiten. Letzteres ist der Standard, der auch in diesem Projekt verwendet wird. Der Unterschied zwischen beiden Protokollen besteht darin, dass UDP eine Kommunikation über einzelne Nachrichten, dessen Reihenfolge und Zustellung nicht garantiert wird, realisiert und TCP diese Reihenfolge und Zustellung gewährleistet, was TCP zwar verlässlicher aber im vergleich zu UDP auch unflexibler macht.

Beiden Protokollen liegt in diesem Projekt IP (Internet Protocol) zugrunde, welches die Grundlage der bereits beschriebenen Netzwerkkommunikationen darstellt. Beide Kommunikationspartner sind durch eine IP-Adresse identifizier- und erreichbar. Zusätzlich wird für den erfolgreichen Aufbau einer Verbindung ein entsprechender Port benötigt, der auf beiden Kommunikationspartnern für die jeweilige Verbindung identisch sein sollte.

Projektaufbau mit Sockets

Die Interprozesskommunikation für Sockets wurde in diesem Projekt auf dem Betriebssystem Windows mit der Programmiersprache Python auf einem einzigen Computer realisiert. Prinzipiell wäre es jedoch möglich mit angepasster IP-Adresse, eine tatsächliche Client-Server-Architektur, sowohl über ein größeres Netzwerk, als auch mit mehreren Clients zu realisieren. Für den erfolgreichen Aufbau einer Verbindung zwischen dem Server- und Clientprogramm auf dem gleichen Computer wurde die sogenannten Loopback-Adresse als IP-Adresse mit der Kennung 127.0.0.1 verwendet, die speziell für Zwecke dieser Art gedacht ist. Als Port-Kennung wurde die Nummer 51337 verwendet, da diese über 50000 und somit außerhalb der reservierten Portbereiche liegt, um keine Konflikte bei der Netzwerk-kommunikation zu erzeugen.

Prinzipiell erfolgt die Kommunikation zwischen den Prozessen Conv, Log, Stat und Report wie folgt: Zunächst wird das Serverprogramm „socket_server.py“ gestartet. Sobald der Server gestartet wurde, kann das Clientprogramm „socket_client.py“ gestartet werden. Nun sendet der Client über den Conv-Prozess stetig Zufallszahlen an den Server. Auf dem Server schreibt der Log-Prozess diese Zahlen in eine Datei, während der Stat-Prozess stetig Summe und Durchschnitt neu berechnet. Die errechneten Zahlenwerte werden an den Client gesendet, welcher diese über den Report-Prozess ausgibt.

Gewählter Lösungsweg

Es erschien am sinnvollsten den Aufbau so zu gestalten, dass der Server so gesehen Rechenleistung zur Verfügung stellt, die ein Client in Anspruch nimmt. In diesem Zusammenhang dokumentiert der Server die erhaltenen Zahlenwerte und sendet das Endergebnis an den Client, der dieses nur noch ausgeben muss. Überträgt man diesen Projektaufbau auf die reale Welt, so kann dies als plausibler Aufbau für Cloud Computing gesehen werden. Der Client sendet stetig Conv-Zahlenwerte und gibt Report-Ergebnisse aus, während parallel dazu, der Server diese Conv-Werte über Log speichert und daraus über Stat die Summe und den Durchschnitt berechnet. Es findet also ein indirektes Multiprocessing statt, obwohl dieses in Kombination mit Sockets unter Windows und Python normalerweise keine Anwendung findet. Sinnvollerweise wurde hier eine TCP-Verbindung über IP gewählt, da die richtige Reihenfolge, sowie die Gewährleistung der Richtigkeit der übertragenen Daten, zwingend notwendig ist um eine korrekte Funktionsweise zu implementieren. Das Kommentieren des Programmcodes ist in Englisch erfolgt, da dies den Open Source Ansatz von GitHub unterstützt.

Main Methode

Zunächst wird auf dem Server ein Socket mit den Parametern „AF_INET“ für IP und „STREAM“ für TCP geöffnet, der nun darauf wartet bis der Client Socket eine Verbindung anfordert. Parallel dazu wird auf dem Client ein Socket mit den gleichen Parametern geöffnet, der eine Verbindung zum Server Socket anfordert. Wurde die Verbindung über die Loopback-Adresse aufgebaut, so erscheint eine entsprechende Benachrichtigung in der Ausgabe. Anschließend wird auf dem Server die Datei für den Log-Prozess und die Stat-Summe initialisiert. Nun beginnt auf Server und Client die Dauerschleife, welche jeweils den Sende- und Empfangsprozess über Sockets von Conv, Stat und Report sowie SIGINT beinhaltet. Des Weiteren wird auf dem Server der Log-Prozess gestartet und ein Counter für die Durchschnittsberechnung mit Stat implementiert. Drückt man Ctrl+C, so wird SIGINT aktiv, alle Dateien und Verbindungen geschlossen und das Programm gestoppt.

Conv Prozess

Der Client erzeugt nun jede Sekunde Zufallszahlen zwischen 0 und 100 und schickt diese direkt an den Server. Wie sich unter den genannten Bedingungen herausstellte, müssen die Daten zum versenden in einen String umgewandelt und anschließend mit UTF-8 codiert werden, da ein erfolgreiches Empfangen und Decodieren auf Server Seite ansonsten nicht möglich war. Entsprechend wandelt der Server die vom Conv-Prozess empfangenen Zahlen zum weiteren Berechnen wieder in einen Integer um und übergibt diese an die Main und somit an den Log-Prozess. Es erfolgt eine entsprechende Ausgabe auf Server und Client.

Log Prozess

Der Server öffnet nun die zuvor initialisierte und angelegte Log Datei, um jede empfangene Zahl von Conv stetig als String in der Datei untereinander anzuhängen und zu speichern. Anschließend wird die Log Datei wieder geschlossen und es erfolgt eine entsprechende Ausgabe.

Stat Prozess

Die von Conv empfangene Zahl sowie der Schleifenzähler werden nun an den Stat-Prozess übergeben. Anschließend wird die zuvor initialisierte und angelegte Stat Datei geöffnet, um die zwischengespeicherte Summe zu lesen. Beim ersten Durchlauf ist die Datei jedoch noch leer. Dann wird die aktuelle Summe mit der zuletzt empfangenen Zahl aus Conv addiert, um die neue Summe zu bilden. Die neue Summe wird wieder in die Stat Datei geschrieben und überschreibt den alten Summenwert, somit geht beim nächsten Durchlauf die aktuelle Summe für ein erneutes Aufaddieren nicht verloren. Diese Methode war für prozessübergreifendes Verarbeiten der Zahlenwerte zwischen Conv, Stat und Report die einfachere Lösung als die Verwendung eines Arrays. Anschließend erfolgt eine entsprechende Ausgabe und die Übergabe der Werte an die Main und somit an den Report-Prozess.

Report Prozess

Die vom Stat-Prozess berechneten Summen und Druchschnitte für jede Zahl aus Conv werden an den Report-Prozess übergeben und in einen String umgewandelt. Wie auch schon im Conv-Prozess beschrieben, ist die Umwandlung erneut notwendig, um den Datensatz mit UTF-8 zu codieren und diesen anschließend über die Socketverbindung vom Server zum Client zu senden. Der Client empfängt über seinen Report-Prozess nun den Datensatz und decodiert ihn wieder mit UTF-8. Ein Umwandeln von String in Integer ist diesmal nicht notwendig, da Report die von Stat empfangenen Daten direkt auf der Client Shell ausgibt.

Socket Fazit

Die größte Herausforderung am Projekt stellte die sehr geringe bis nicht vorhandene Verfügbarkeit von Code-Beispielen und Sachtexten im Internet dar, die die Kombination aus Windows, Python, Sockets und Multiprocessing mit sich brachte. Dies führte zu einem zeitlich hohen Arbeitsaufwand und dem Entwickeln eigener Lösungsansätze. So wurde beispielsweise über Umwege herausgefunden, dass „fork“ unter Linux aber nicht unter Windows verfügbar ist oder, dass es kein klassisches Multiprocssing in Kombination mit Sockets über Python unter Windows gibt. Insgesamt kann man jedoch sagen, dass die Aufgabenstellung erfolgreich bewältigt wurde. Es laufen mehrere Prozesse auf der Serverseite parallel zu mehreren Prozessen auf der Clientseite, während die Kommunikation der unterschiedlichen Prozesse untereinander über Sockets realisiert wurde. Für einen reibungsloseren Ablauf in zukünftigen Projekten ist es nun motivierend, vermehrt mit Linux zu arbeiten um in weiteren Bereichen der Informatik besser arbeiten zu können. Zudem fand eine erfolgreiche Einführung in Python statt.

Quellen

http://www.christianbaun.de/BSRN22/Skript/bsrn_SS2022_vorlesung_06_de.pdf

https://stackoverflow.com/questions/8545307/multiprocessing-and-sockets-in-python

https://docs.python.org/3/library/multiprocessing.html

https://de.wikipedia.org/wiki/Socket_(Software)

https://www.youtube.com/watch?v=YwWfKitB8aA

https://www.youtube.com/watch?v=Lbfe3-v7yE0