

Verkehrslsleitsystem

Thomas Alpert & Alexander Lutz

28.04.2023

Inhalt

Analyse	2
Aufgabenanalyse	2
Abgrenzung	4
Grobkonzept.....	5
Architektur.....	5
Verteilte Aspekte	6
Kommunikation	6
Logik/Lastverteilung	7
Transparenz	11
Nebenläufigkeit	11
Synchronisierung.....	12
Datenhaltung.....	12
Zusammenfassung CAP Theorem:.....	15
Schnittstellen.....	16
Beschreibung der Anpassung	17
Programm.....	17
Voraussetzungen	17
Quellcode	18
Knoten	18
Client.....	22
Startanweisung.....	25
Testplan	26
Nachbetrachtung.....	30

Analyse

Aufgabenanalyse

Die Aufgabe besteht darin, ein verteiltes System mit einem Leitsystem zu erstellen, das ein intelligentes Routing von Fahrzeugen ermöglicht. Dabei sollen mindestens 1, 10, und 100 simulierte Fahrzeuge als Clients unterstützt werden. Das System soll aus mindestens 3 Knoten bestehen, die Anfragen beantworten und eine konsistente Sicht auf den aktuellen Stand im Gebiet haben. Das Leitsystem soll alternative Routen über eine intelligente Verkehrssteuerung bereitstellen und dem Fahrzeug den verkehrsgünstigsten Weg zum Ziel bereitstellen. Die Position des Testclients kann innerhalb eines 1000x1000-Punkte-Koordinatensystems gewählt werden und die maximale Verkehrslast pro Knoten soll 2 betragen.

Das System muss den Ausfall von 1, 2 oder allen Knoten unterstützen und beim Wiederhochfahren bei einem Komplettausfall auf dem letzten konsistenten Stand weiterarbeiten können. Der Testclient soll auf Totalausfall reagieren können, während Teilausfälle innerhalb des Systems kompensiert werden müssen.

Die Aufgabe soll als Konsolenprogramm mit Startparametern und Konfiguration erstellt werden. Das Betriebssystem für die Knoten soll auf 3 Raspberry PI 4 mit je 4 GB RAM lauffähig sein, während der Client ein Windows-Client sein kann.

Grundsätzlich ist zu sagen, dass alle Anforderungen in 4 verschiedenen Kategorien aufgeteilt werden können.

- Infrastrukturelle Anforderungen (Infrastruktur)
- Kommunikation der Nodes/Verteile-System (Kommunikation)
- Programmlogik (vorgegeben) (Logik)
- Datenhaltung (Daten)

Die Anforderungen werden in folgender Anforderungsmatrix dargestellt.

Anforderungen	Priorität	Kategorie	Zuständigkeit
Unterstützung von 1, 10, 100 Fahrzeugen als Clients	Hoch	Funktional Infrastruktur	Alex
Theoretische Unterstützung von 1000 oder 10000 Fahrzeugen als Clients	Niedrig	Funktional Infrastruktur	Gruppe

Mindestens 3 Knoten im System	Hoch	Funktional Infrastruktur	Thomas
Konsistente Sicht auf den aktuellen Stand im Gebiet	Mittel	Funktional Daten	Alex
Verkehrssteuerung über alternative Routen durch ein intelligentes Leitsystem	Hoch	Funktional Logik	Alex
Fahrzeug (simuliert durch den Testclient) wird der verkehrsgünstigste Weg zum Ziel bereitgestellt	Mittel	Funktional Logik	Thomas
Maximale Verkehrslast von 2 pro Knoten	Niedrig	Funktional Datenhaltung	Alex
Ausfall von 1, 2 oder allen Knoten wird unterstützt	Hoch	Funktional Kommunikation	Thomas
Wiederhochfahren nach Komplettausfall soll auf dem letzten konsistenten Stand weiterarbeiten können	Mittel	Funktional Datenhaltung, Kommunikation	Alex
Totalausfall soll vom Client erkannt und kompensiert werden können	Mittel	Funktional Kommunikation	Thomas
Teilausfall soll innerhalb des Systems kompensiert werden können	Mittel	Funktional Kommunikation	Alex
Testclient sendet Position und Ziel an beliebigen Knoten	Hoch	Funktional Logik	Thomas
Testclient erhält als Antwort die nächste Position, die er ansteuern soll	Hoch	Funktional Logik	Thomas
Testclient wiederholt Anfrage, bis er am Ziel ist	Hoch	Funktional Logik	Thomas
Position des Testclients innerhalb eines 1000x1000-Punkte-Koordinatensystems wählbar	Niedrig	Funktional Logik	Thomas
Zeitmessung und Ausgabe der Zeit, die der Testclient benötigt hat, um das Ziel zu erreichen	Mittel	Funktional Logik	Thomas

Konsolenprogramm mit Startparametern und Konfiguration	Hoch	Nicht-funktional	Gruppe
Betriebssystem: Raspberry PI 4 mit 4 GB RAM für Knoten, (Windows)-Client für Testclient	Hoch	Nicht-funktional	Alex
Lauffähigkeit des Systems auf den vorgegebenen Betriebssystemen	Hoch	Nicht-funktional	Gruppe
Dokumentation mit Anleitung zum Starten und Vorbedingungen	Mittel	Nicht-funktional	Gruppe
Compile-Anweisung, falls notwendig	Mittel	Nicht-funktional	Alex
Skalierbarkeit des verteilten Systems	Mittel	Nicht-funktional	Alex

Abgrenzung

Es wird keine grafische Benutzeroberfläche oder GUI entwickelt. Das System soll als Konsolenprogramm umgesetzt werden.

Die Lastverteilung erfolgt nicht auf Basis der Auslastung von CPU, Arbeitsspeicher oder Netzwerklast am Node. Die Knoten werden gleichmäßig belastet und Anfragen werden anhand eines Load-Balancing-Algorithmus verteilt. Es wird keine "echte" Lastverteilung umgesetzt, die auf der aktuellen Auslastung des Systems basiert.

Der Client hat keinen Überblick über das Feld, wie es im verteilten System aussieht. Die Berechnung/Simulation des Fahrweges erfolgt vollständig auf dem Wissen des verteilten Systems. Es handelt sich um ein theoretisches Konzept. Auf dem Client wird nicht überprüft, ob die vom verteilten System vorgegebene Route tatsächlich befahrbar ist. In der Realität kann es vorkommen, dass der gewählte Weg aufgrund anderer Umstände nicht befahrbar ist. Ein solcher Fall kann vom Client nicht als Information an den Server zurückgegeben werden, um eine mögliche Alternativroute zu berechnen.

Es wird davon ausgegangen, dass der vom verteilten System vorgegebene Weg immer funktioniert und der Client diesem stur folgt.

In der Aufgabenstellung ist nicht angegeben, was mit den Clients/Autos passiert, die ihr Ziel erreicht haben. Wir gehen davon aus, dass Autos, die ihr Ziel erreicht haben, am Zielort bleiben und parken

und nicht verschwinden. Dies könnte jedoch zu Problemen führen, da parkende Autos während der Ausführung "im Weg stehen" könnten, da die maximale Verkehrsbelastung pro Feld 2 beträgt.

Grobkonzept

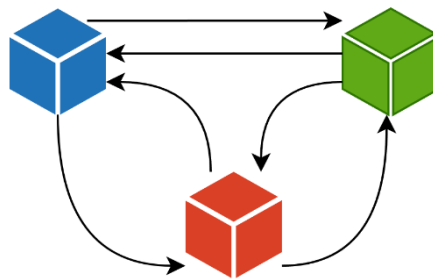
Das Grobkonzept basiert zunächst auf 3 Nodes, mit denen entwickelt und getestet wird. Die Zielumgebung soll auf 3 Raspberry 4 Pi realisiert werden. Wir gehen davon aus, dass das Betriebssystem auf dem Pi, das auf Debian 11 basierende Raspberry Pi OS sein wird.

Für die Entwicklung und den Test der Aufgaben stehen uns keine physischen Raspberry Pis zur Verfügung. Wir haben uns für die Verwendung von lokalen VMs und die Verwendung von VMs in der BaWU-Cloud entschieden. Diese VMs laufen ebenfalls unter Debian 11 und wurden ebenfalls mit jeweils 4GB Arbeitsspeicher konfiguriert.

Architektur

Für die Architektur wurden 3 Knoten gewählt, die alle das Gleiche können und tun. Es ist kein Master-Slave-System. Jeder Knoten ist sozusagen ein Master Knoten. Die Knoten bilden das verteilte System, das alle Verbindungen und Aufgaben, die vom Client angefordert werden, gleichmäßig verteilen soll.

Die Architektur kann daher als "verteiltes Gleichgewichtssystem" beschrieben werden. In einem verteilten Gleichgewichtssystem gibt es mehrere Knoten, die alle die gleiche Logik ausführen und keine spezielle Master- oder Slave-Rolle haben. Jede von einem Client an das System gesendete Anfrage wird gleichmäßig auf die verfügbaren Knoten verteilt, um die Last im System gleichmäßig zu verteilen und eine höhere Skalierbarkeit und Verfügbarkeit zu ermöglichen.



Jeder Knoten im verteilten System kennt immer alle Knoten und versucht, eine Verbindung zu ihnen herzustellen. Jeder Knoten versucht also beim Hochfahren immer, alle Knoten zu erreichen, so dass es $(n-1)$ eingehende und ausgehende Verbindungen zu den Knoten gibt.

Bei $n=3$ Knoten hat jeder Knoten $(3-1)$ zwei ausgehende Verbindungen zu den anderen Knoten und $(3-1)$ zwei eingehende Verbindungen von den verbleibenden 2 Knoten. Dies ist auch in der Abbildung zu sehen. Warum es in unserem Fall wichtig ist, dass jeder Knoten eine ausgehende und eine eingehende Verbindung zu jedem anderen Knoten hat, wird im nächsten Kapitel erläutert.

Verteilte Aspekte

Kommunikation

Für die grundlegende Kommunikation zwischen den einzelnen Knoten und die Verbindung von Client zu Knoten/verteilterm Testsystem werden Sockets verwendet. Die Verwendung von Sockets ist u.a. dadurch begründet, dass die ausgetauschten Datenmenge pro Anfrage (zwischen den Knoten und von Client zu Knoten) sehr gering sein wird. Mit Sockets können kleine Datenmengen schnell und effizient zwischen den Knoten des verteilten Systems übertragen werden.

Außerdem kann das verteilte System dadurch skalierbarer werden: Wenn ein verteiltes System auf Sockets basiert, können mehrere Knoten einfach hinzugefügt werden, um die Systemleistung zu erhöhen. Dies ermöglicht eine einfache Skalierbarkeit des Systems, da jeder Knoten die gleiche Logik ausführt und die Last gleichmäßig verteilt werden kann.

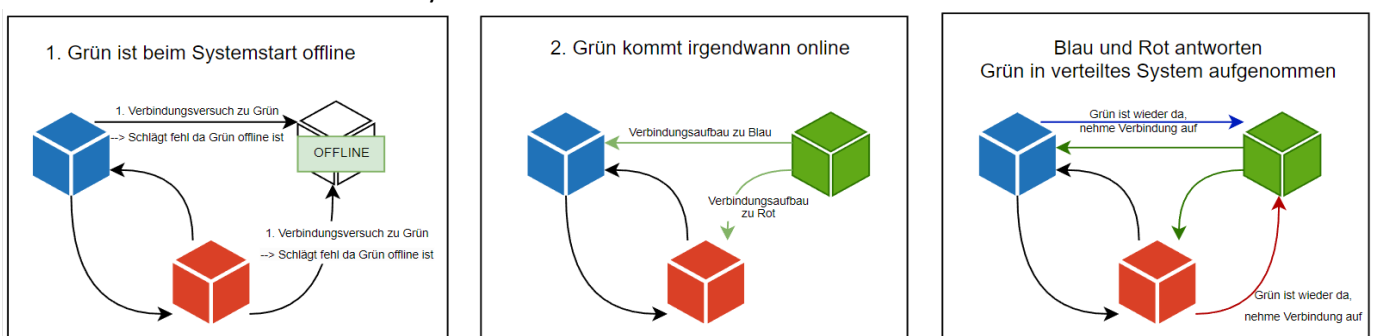
Ein weiterer wichtiger Punkt ist die Plattformunabhängigkeit: Sockets sind plattformunabhängig und können auf verschiedenen Betriebssystemen und Programmiersprachen eingesetzt werden. Dies ermöglicht eine höhere Flexibilität bei der Entwicklung und Implementierung des verteilten Systems. Dies ist wichtig, da der Client theoretisch auch ein Windows-Programm sein kann.

Verbindung Knoten zu Knoten:

Wie bereits erwähnt und in der Abbildung auf der vorherigen Seite zu sehen, hat jeder Knoten insgesamt zwei Socket-Verbindungen zu allen anderen Knoten. Diese auf den ersten Blick möglicherweise redundant erscheinenden Verbindungen haben jedoch zwei wesentliche Vorteile.

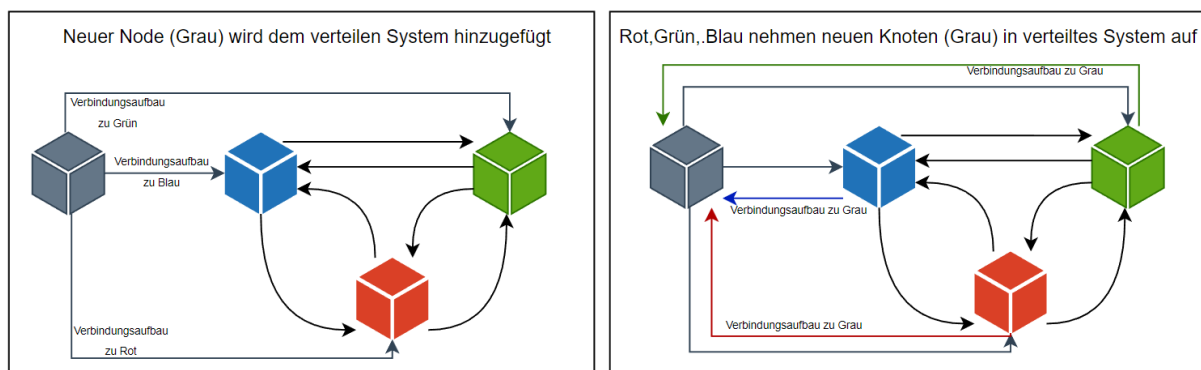
Wenn z.B. ein Knoten beim Start des verteilten Systems nicht online ist, versuchen die anderen Knoten, die online sind, nicht aktiv eine Verbindung zu diesem Knoten aufzubauen. Es wird einmalig versucht eine Verbindung zu diesem/alle Knoten aufzubauen, schlägt dies fehl, wird kein weiterer Versuch unternommen.

Wenn der zu spät gestartete Node dann hochgefahren wird, versucht er sich bei allen anderen Nodes anzumelden und eine Verbindung aufzubauen. Die anderen Nodes im System bemerken nun eine eingehende Verbindung eines Nodes und bauen automatisch eine Verbindung zu diesem auf und nehmen ihn in das verteilte System auf.



Dies geschieht auch, wenn ein Knoten während des Betriebs zufällig oder geplant offline geht, die anderen Knoten bemerken, dass ein Knoten offline ist, versuchen aber nicht von sich aus aktiv, den verlorenen Knoten wieder zu erreichen, erst wenn der Knoten von sich aus wieder online geht, meldet er sich im verteilten System an und wird in dieses aufgenommen.

Der zweite Punkt betrifft die Skalierbarkeit des Systems. Im laufenden Betrieb können neue Knoten einfach zum bestehenden System hinzugefügt werden. Der neu hinzugefügte Knoten meldet sich bei allen verfügbaren Knoten des aktuellen Systems, die ihn dann in das verteilte System aufnehmen und eine Verbindung zu ihm aufbauen.



Die Vorteile dieses Konzeptes sind also, dass Wartung in Form von Rolling Updates durchgeführt werden oder das System einfach erweitert/skaliert werden kann, indem neue Knoten zur Laufzeit hinzugefügt werden.

Verbindung von Client zu Knoten:

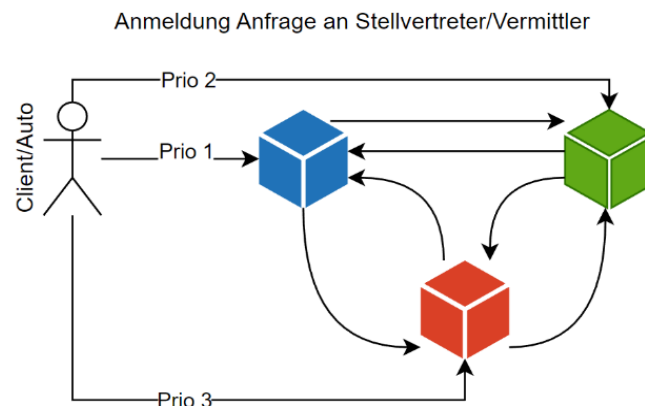
Der Client, der die Anfragen der Fahrzeuge simulieren soll, sendet die Anfrage ebenfalls in Form von Sockets an das verteilte System. Jeder einzelne Client baut eine Socket-Verbindung zum verteilten System (einem bestimmten Knoten) auf, die erst dann getrennt wird, wenn die Berechnung abgeschlossen und der Client am Ziel angekommen ist.

Logik/Lastverteilung

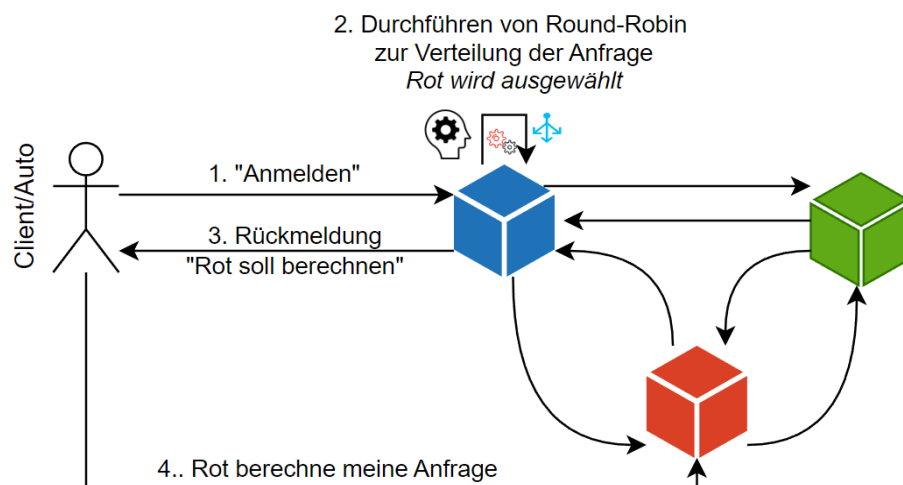
Die Lastverteilung, d.h. die Verteilung der vom Client eingehenden Anfragen, wird vollständig und ausschließlich vom verteilten System übernommen.

Ein Client, der eine Anfrage an das verteilte System stellen möchte, muss seine Anfrage zunächst "Anmelden", bevor diese bearbeitet werden kann. Dabei meldet sich der Client immer beim gleichen Knoten an. Der Client muss jederzeit über alle Knoten im System informiert sein, diese Information (die IP-Adressen der Knoten im verteilten System) muss dem Client zur Verfügung gestellt werden. Für die „Anmeldung“ gibt es aus Sicht des Clients einen Vermittler oder einen Stellvertreter, der die

Anfragen an das verteilte System entgegennimmt. Ist dieser Stellvertreter aufgrund eines Netzwerk- oder Softwarefehlers nicht erreichbar, wird einfach der nächste Knoten in der Liste der bekannten Knoten im verteilten System angefragt.



Der Knoten, der die "Anmeldung" akzeptiert, führt als nächstes einen Round-Robin-Lastverteilungsalgorithmus aus, um den Knoten innerhalb des verteilten Systems auszuwählen, der die Client-Anfrage letztendlich bearbeiten/berechnen soll. Das bedeutet, dass der Client, der die "Anmeldung" beim Stellvertreterknoten anfordert, als Antwort die IP-Adresse des vom Round Robin Algorithmus ausgewählten Knotens erhält. Die Socket-Verbindung zum Stellvertreter wird daraufhin getrennt und der Client meldet sich anschließend bei dem Knoten, der die Berechnung durchführen soll, diesmal nicht als Anmeldung, sondern mit der Bitte, die Berechnung durchzuführen.



In diesem Beispiel meldet sich der Client beim Stellvertreterknoten Blau, der intern den Round-Robin-Lastverteilungsalgorithmus ausführt und Rot als passenden Kandidaten erhält. Die IP-Adresse des roten Knotens wird dann dem Client mitgeteilt und die Socket-Verbindung wird getrennt. Der Client baut nun eine Socket-Verbindung zum roten Knoten auf und sendet seine Berechnungsanfrage an diesen Knoten, wie sein Auto die Ziel-Position erreichen kann. Der Knoten berechnet nun intern immer den nächsten Schritt im Feld und gibt diesen an den Client zurück. Der Client stellt sein Fahrzeug auf

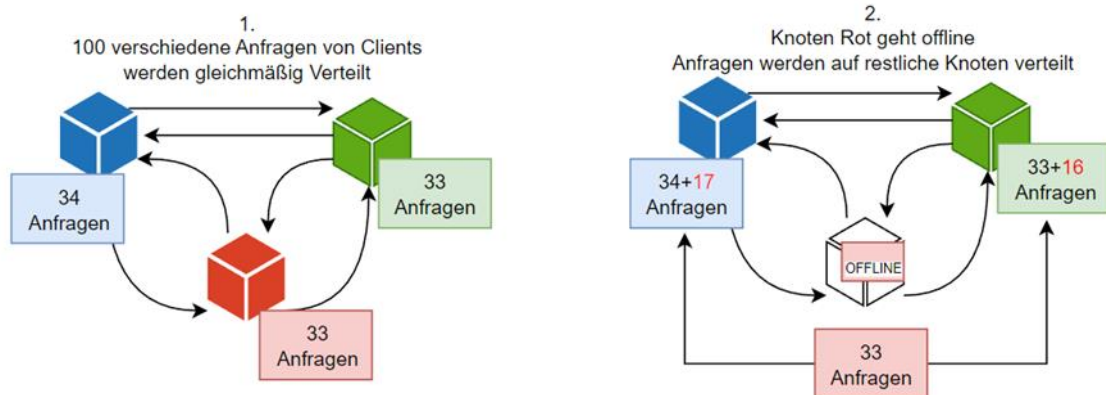
den erhaltenen Schritt und wiederholt die Anfrage mit der neuen Position. Dieser Vorgang wird so lange wiederholt, bis der Client sein Ziel erreicht hat. Erst dann wird die Socket-Verbindung getrennt.

Bei einem Verbindungsabbruch geht die Socket-Verbindung verloren und der Client muss die Anfrage erneut an das verteilte System stellen. Dabei muss der Weg wieder von vorne beginnen und die Berechnung beim Vertreter "anmelden" und die Anfrage wird wieder mit Round Robin verteilt.

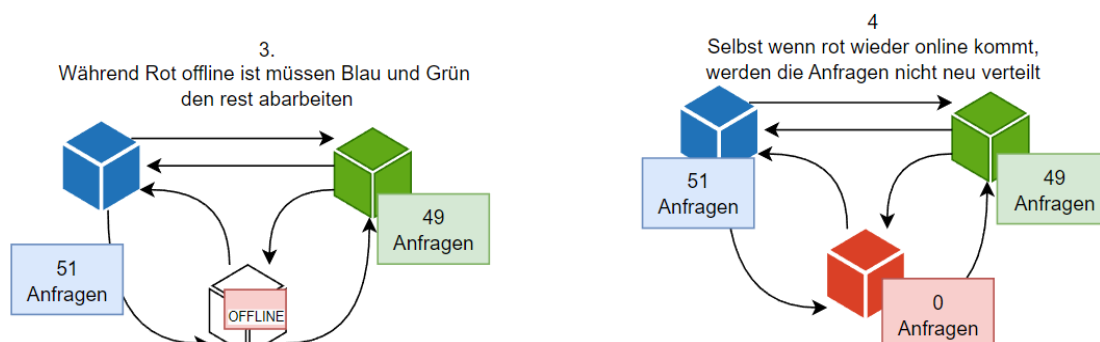
Probleme einfacher Lastverteilung mit Round Robin :

Die einfache Verteilung mit Round Robin bei der Antragstellung hat jedoch einen kleinen Nachteil. Dazu muss ein bestimmtes Szenario betrachtet werden.

Es sind jeweils 100 Client-Anfragen zur Berechnung des Weges von einer Startposition und einer Zielposition vorhanden. Diese wurden bereits gleichmäßig mit Round Robin verteilt. 34 Anfragen landen bei Blau und werden dort berechnet. 33 Anfragen landen bei Grün und die restlichen bei Rot. Wenn nun der rote Knoten ausfällt, werden die 33 Anfragen, die von den Clients gesendet wurden, an diesen zurückgesendet. Der Client sendet diese 33 Anfragen erneut an das verteilte System. Der einfache Round Robin Algorithmus verteilt nun nach und nach die 33 Anfragen an die verbleibenden 2 Knoten im Verbund. 17 gehen an Blau, 16 an Grün.



Das System als solches arbeitet weiter und die verbleibenden Knoten bearbeiten nun die 51 Anfragen an Blau und die 49 Anfragen an Grün. Wenn der rote Knoten wieder online ist, hat er 0 Anfragen und wird keine weiteren mehr erhalten. Das System ist in diesem Zustand nicht balanciert. Noch schlimmer wird es, wenn 2 Knoten gleichzeitig ausfallen und wieder online gehen, dann muss ein Knoten die gesamte Berechnung durchführen.

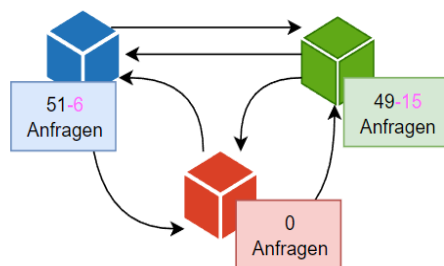


Um dieses Problem der unausgewogenen Arbeitsbelastung zu lösen, ist ein weiterer Schritt in der Anwendungslogik erforderlich. Es sollte ein Mechanismus vorhanden sein, der zu lange laufende Berechnungen an Knoten nach und nach unterbricht und die Verbindung trennt.

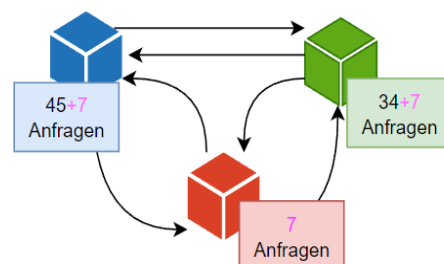
Verbindungen von einem Client zu einem Knoten im System sollten nur für eine bestimmte Anzahl von Berechnungen/Schritten im Feld für das Auto bestehen. Sobald der Client mehr als die definierte Anzahl von Schritten benötigt, wird er vom Knoten getrennt. Das Fahrzeug bzw. der Client hat in diesem Fall die definierte Strecke zurückgelegt, ist aber noch nicht am Ziel angekommen und muss daher die Anfrage zum Ziel mit der Berechnung erneut an das verteilte System senden. Der Knackpunkt ist, dass die erneute Anfrage dann wieder beim Stellvertreter "angemeldet" und neu balanciert werden muss.

Diese "**erweiterte Lastverteilung**" wird am gleichen Beispiel nochmals verdeutlicht. Der rote Knoten war offline und ist wieder online, er hat 0 Anfragen, die restlichen 100 Anfragen gehen an Blau und Grün. Berechnungen / Feldbewegungen, die länger als die definierte Anzahl dauern, werden dann vom verteilten System getrennt.

5.
Nach einer definierten Anzahl an Feldbewegungen pro Anfrage wird der Client getrennt



6.
Die getrennten Verbindungen (21) werden neu "Angemeldet" und entsprechen mit Round Robin wieder gleichmäßig +7 verteilt



In diesem Fall haben 21 Fahrzeuge die maximale Weglänge erreicht und werden vom System getrennt. Diese 21 Anfragen werden nun wieder "an den Stellvertreter gemeldet" und intern mit Robin-Robin so verteilt, dass die 21 Anfragen im Client gleichmäßig auf die 3 Knoten verteilt werden. Der rote Knoten, der gerade online gegangen ist, erhält nun 7 Anfragen von verschiedenen Clients, die ihre Berechnung zum Ziel an genau der Stelle fortsetzen wollen, an der sie zuvor vom verteilten System getrennt wurden.

Diese erweiterte Lastverteilung ermöglicht es dem System, sich im Laufe der Zeit selbst auszugleichen. Dieses Verfahren ermöglicht es, die Last der Knoten nach einem Ausfall so weit wie möglich zu verteilen, nimmt aber in Kauf, dass die Berechnungen insgesamt länger dauern können, da Verbindungen abgetrennt und neu angefordert werden müssen.

Bei der Umsetzung ist dann der Grenzwert der maximalen Weglänge hinsichtlich der Simulationsperformance zu untersuchen. Es ist abzuwägen, ob der Schwellwert niedrig gewählt wird, man damit aber riskiert, dass die Gesamtzeit der Berechnung länger wird, oder ob der Schwellwert hoch gewählt wird, man dann aber möglicherweise im Ausfallfall einer Knotens die Last nicht mehr richtig verteilt bekommt.

Ob dies einen großen Unterschied macht, ist bei der endgültigen Implementierung zu prüfen, wobei dann auch eine Schwelle ausgetestet werden kann, ab welcher maximalen Feldbewegung eine Verbindung schließlich getrennt wird.

Transparenz

Bei dieser Art von verteiltem System kann der Punkt Transparenz nicht erfüllt werden. Aus Sicht des Clients ist es in unserem Fall nicht zu verbergen, dass es sich um mehrere Rechnersysteme und nicht um "ein" logisches System handelt. Der Client muss alle Knoten kennen, um bei der ersten Verbindung mit einem Vertreter zu kommunizieren. Dieser Vertreter teilt ihm dann mit, zu welchem Knoten er sich verbinden muss. Der Vertreter kann zwar indirekt als Middleware betrachtet werden, er leitet aber keine Anfragen weiter (wie ein Proxy), sondern teilt dem Client nur den nächsten Kontaktpunkt mit.

Nebenläufigkeit

Ein weiterer wichtiger Aspekt in unserem verteilten System ist die Nutzung von Nebenläufigkeit. In unserem System spielt die Nebenläufigkeit eine wichtige Rolle, um ein effizientes und reaktionsschnelles Kommunikationsmodell zwischen den Knoten zu ermöglichen. Ohne Nebenläufigkeit müssten alle Verbindungen und Anfragen nacheinander bearbeitet werden, was zu schlechter Leistung und langsamen Antwortzeiten führen würde. Durch Nebenläufigkeit können mehrere Aufgaben parallel bearbeitet werden. Dies kann durch die Verwendung von Tasks und asynchronen Methoden erreicht werden, die es dem System ermöglichen, mehrere Verbindungen parallel zu verarbeiten, ohne sich gegenseitig zu blockieren.

Die Nebenläufigkeit kann durch das Threading gewährleistet werden. Das Programm, das für die Knoten des verteilten Systems zuständig ist, muss in jedem Fall einen Main-Thread haben, der nur für den Start des Programms vorgesehen ist. Der Main Thread läuft ständig im Hintergrund und hält das Programm am Leben.

Außerdem sollte es für jede eingehende Socket-Verbindung einen eigenen Thread geben. Dies stellt sicher, dass eingehende Nachrichten von Clients und Knoten parallel verarbeitet werden, ohne sich gegenseitig zu blockieren. Ein eigener Thread zum Starten des eigenen Sockets wäre in diesem Fall nicht notwendig, da das Öffnen und Binden des Sockets keine zeitintensive Aufgabe ist und keine

kontinuierliche Verarbeitung erfordert. Die Hauptaufgabe des Programms besteht darin, auf eingehende Verbindungen und Anfragen zu warten und darauf zu reagieren.

Wie in den vorhergehenden Abbildungen zu sehen ist, wird für jeden eingehenden Pfeil auf jedem Knoten ein eigener Thread erstellt, in dem die Verbindungen behandelt werden.

Synchronisierung

In verteilten Systemen ist die Synchronisierung wichtig, um die Datenkonsistenz und das korrekte Verhalten der Anwendung zu gewährleisten. Da mehrere Tasks oder Threads gleichzeitig auf gemeinsame Ressourcen zugreifen können, kann es zu unerwarteten Nebeneffekten kommen, wenn die Synchronisation nicht korrekt verwaltet wird. Dies bedeutet, dass das Programm bestimmte Ressourcen/Variablen im laufenden System vor gleichzeitigem Zugriff schützt. Dies kann in den meisten Multi-Core-Programmiersprachen durch einen Lock-Mechanismus sichergestellt werden. Ein Lock stellt sicher, dass nur eine Task oder ein Thread gleichzeitig auf den geschützten Codeblock zugreifen kann. Andere Tasks oder Threads, die den gleichen Codeblock ausführen wollen, müssen warten, bis der erste Task oder Thread den geschützten Bereich verlassen hat.

In unserem verteilten System müssen folgende Ressourcen gesperrt werden

- Liste der aktuellen Knoten:

Die Liste der aktuellen Knoten muss mit lock gesperrt werden, um gleichzeitige Änderungen an der Liste, wie das Hinzufügen oder Entfernen von Knoten, zu verhindern.

- Verkehrssteuerungslogik:

Verhindert gleichzeitige Änderungen der Bewegung und der Fahrzeugpositionen, die zu inkonsistenten Berechnungen führen könnten. Verhindert dadurch gleichzeitiges Schreiben und Ändern der Datenhaltung)

- Round-Robin-Lastverteilung:

Bei der Berechnung der Lastverteilung muss verhindert werden, dass ein anderer Threads diese parallel ausführen kann, was zu einer inkonsistenten Verteilung führen könnte.

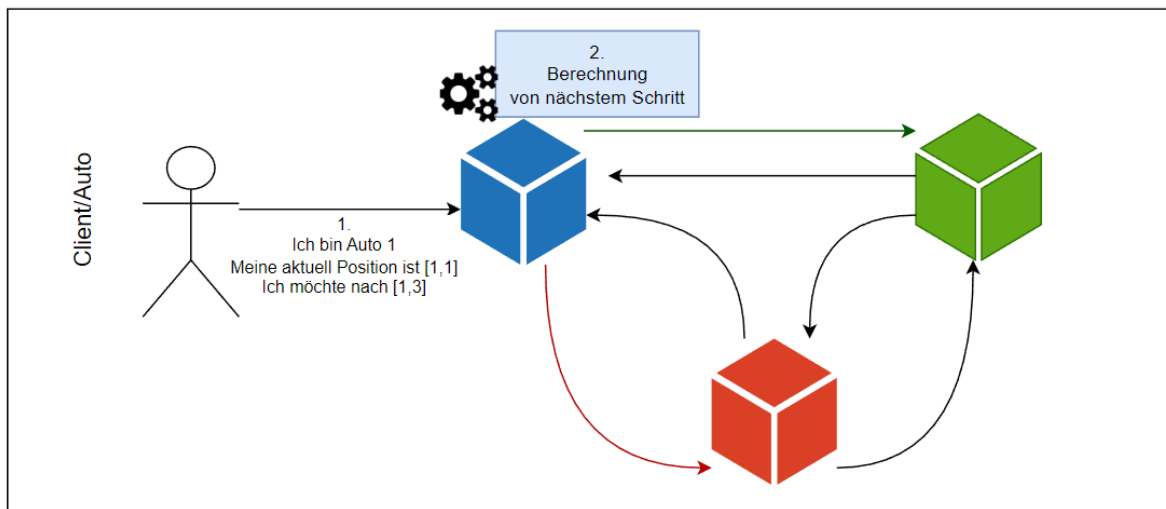
Datenhaltung

Die Datenhaltung auf jedem Knoten erfolgt vollständig und separat im Arbeitsspeicher. Es werden keine Textdateien oder (verteilte) Datenbanken verwendet. Die Datenhaltung im Arbeitsspeicher anstelle einer Datenbank oder Textdatei bietet einen enormen Vorteil, da die Daten im Arbeitsspeicher

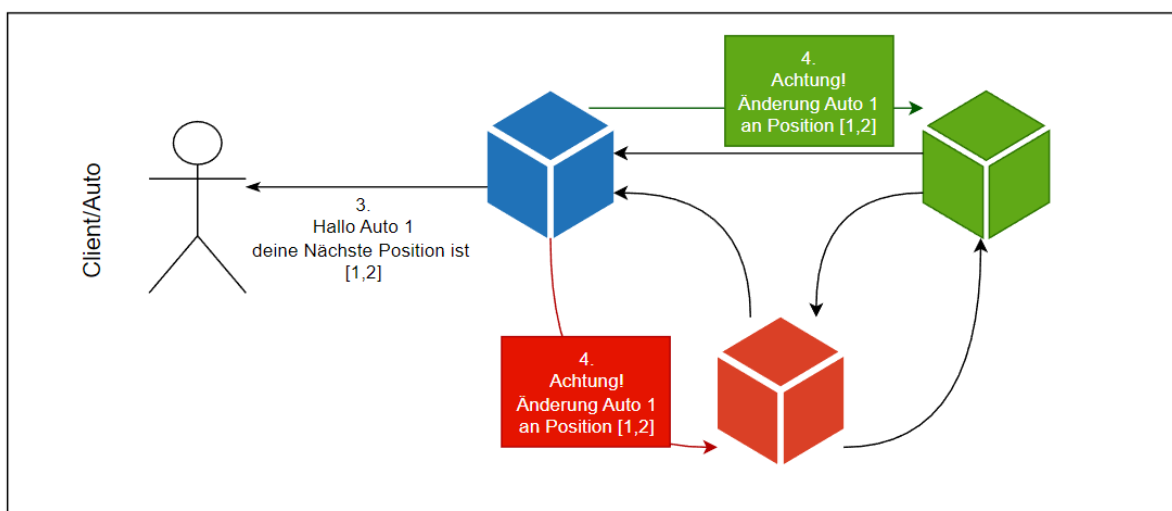
wesentlich schneller gelesen und geschrieben werden können als eine Schreib- oder Leseoperation auf der Festplatte oder einer Datenbank.

Die Daten, die gespeichert werden, sind nur das Feld und die Positionen der Fahrzeuge auf dem Feld. Die Knoten informieren sich gegenseitig, wenn sich die Daten ändern, und versuchen so, den Datenbestand konsistent zu halten.

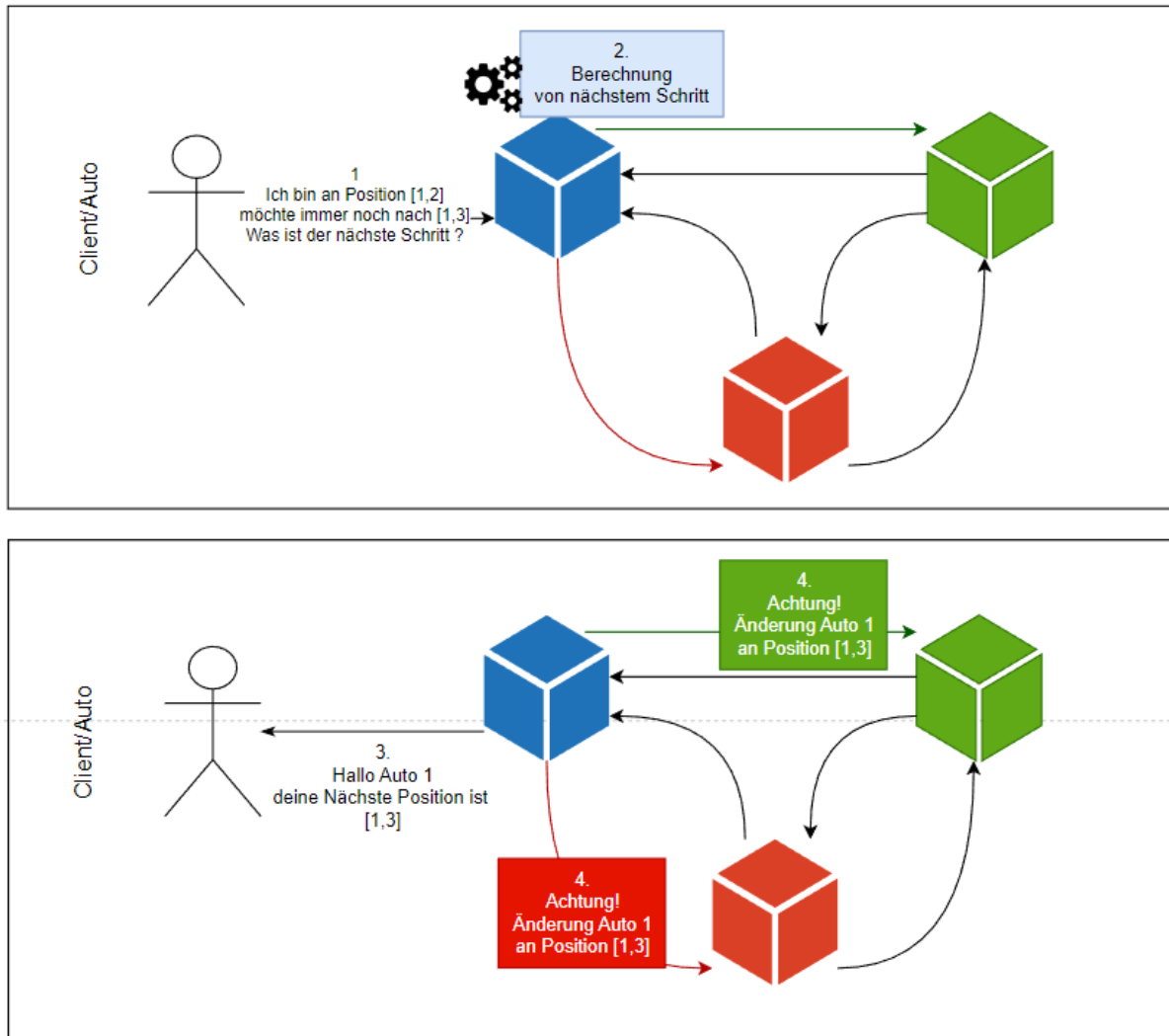
Sobald sich ein Client bei einem Knoten meldet, teilt dieser dem Knoten seine aktuelle Start- und Zielposition mit. Der Knoten berechnet nur den nächsten Schritt für die Anfrage.



Der Knoten setzt nun das Fahrzeug auf die nächste Position in seinem Datenbestand und sendet die neue Position an den Client zurück. Gleichzeitig werden alle anderen Knoten im verteilten System über die Änderung des Autos informiert. Die anderen Knoten erhalten nun die Änderung und speichern die geänderte / neue Position des Autos bei sich intern im Datenbestand (im Arbeitsspeicher).



Dieser Vorgang wird so lange wiederholt, bis der Client sein Ziel erreicht hat, die Verbindung unerwartet getrennt wurde oder die oben genannte maximale Anzahl von Schritten/Feldbewegungen erreicht wurde und der Server die Verbindung trennt, um die **erweiterte Lastverteilung** durchzuführen (siehe Seite 10).



Nachteil dieser Datenhaltung:

Der Nachteil dieser Datenhaltung ist, dass bei einem Ausfall eines Knotens die Daten möglicherweise nicht 100% konsistent sind. Da die Datenhaltung (d.h. die aktuelle Position aller Fahrzeuge) im Arbeitsspeicher, d.h. flüchtig vorhanden ist, gehen diese nach einem Neustart verloren.

Ein neu gestarteter Knoten oder ein neu hinzugefügter Knoten startet mit einem leeren Feld. Dieses Feld wird jedoch nach und nach wieder aufgebaut, nachdem die Änderungen der anderen Knoten empfangen wurden. Es ist daher nicht notwendig, den aktuellen Zustand eines anderen Knotens in irgendeiner Weise zu kopieren, da die Datenkonsistenz langfristig wiederhergestellt wird. Sobald Änderungen vorgenommen werden, d. h. Fahrzeuge sich bewegen, empfängt der neu gestartete

Knoten oder der neue Knoten diese Änderung und baut seinen Datenbestand nach und nach wieder auf.

Selbst wenn alle Knoten des verteilten Systems ausfallen, ist dies kein Problem. Die Clients wissen immer, wo sie sich befinden und sind eigentlich der Single Point of Truth im System. Das System fährt nach einem Totalausfall wieder hoch und weiß nicht, wo sich die Fahrzeuge im Feld gerade befinden. Diese Clients melden sich aber nach und nach im verteilten System bei den einzelnen Knoten und geben ihre aktuelle Position bekannt. Dadurch wird die Datenhaltung wieder identisch aufgebaut.

Ein Problem bleibt jedoch, wie bereits in der Abgrenzung erwähnt. Wenn ein Fahrzeug an seiner Zielposition angekommen ist, bleibt es vorerst dort stehen und kann möglicherweise andere Fahrzeuge blockieren. Diese Information kann nach einem Teil- oder Totalausfall nicht wiedergewonnen werden. Wenn ein Knoten wieder in Betrieb geht, kann er die "geparkten Fahrzeuge" nicht in seinen Datenbestand aufnehmen, da keine aktuellen Positionsdaten gesendet werden.

Dieser "Sonderfall" kann nur gelöst werden, wenn definiert (und programmtechnisch umgesetzt) wird, dass Fahrzeuge, die das Ziel erreicht haben, aus dem Feld entfernt werden.

Die Konsistenz der Datenhaltung hängt auch von der Geschwindigkeit und Zuverlässigkeit der Netzwerkverbindung zwischen den Knoten ab. Eine langsame Netzwerkverbindung zwischen den Knoten kann dazu führen, dass Änderungen an einem Knoten nicht rechtzeitig bei den anderen Knoten ankommen und somit die Konsistenz gefährden.

Zusammenfassung CAP Theorem:

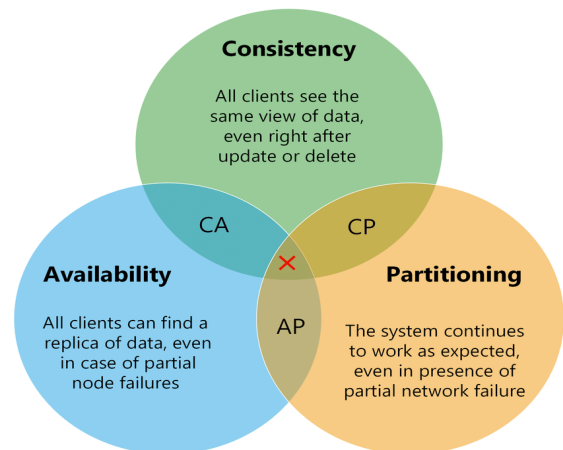
Jedes verteilte System unterliegt dem CAP-Theorem, einem grundlegenden Prinzip, das die Grenzen und Kompromisse von verteilten Systemen beschreibt. Das CAP-Theorem besagt, dass ein verteiltes System nur zwei der folgenden drei Eigenschaften gleichzeitig erfüllen kann:

- **Konsistenz (Consistency):** Jede Leseoperation gibt den zuletzt geschriebenen Wert zurück oder liefert einen Fehler. Konsistenz stellt sicher, dass alle Knoten im verteilten System immer den gleichen Datenstand haben.
- **Verfügbarkeit (Availability):** Jede Anfrage an das System erhält eine Antwort, entweder einen gültigen Wert oder einen Fehler. Verfügbarkeit bedeutet, dass das System immer in der Lage ist, Anfragen zu bearbeiten und zu beantworten, unabhängig vom Ausfall einzelner Knoten.
- **Partitionstoleranz (Partition Tolerance):** Das System kann auch bei Netzwerkpartitionen oder Kommunikationsausfällen zwischen Knoten noch korrekt arbeiten.

Unser verteiltes System kann in AP kategorisiert werden.

Es ist vollständig auf Verfügbarkeit und Ausfallsicherheit ausgerichtet. Client-Anfragen werden immer beantwortet (wenn der Client alle Stellvertreter-IPs kennt). Ausfälle werden vom Client bemerkt, können aber durch einfaches erneutes Senden der Anfrage kompensiert werden. Der Ausfall eines Knotens hat keinen Einfluss auf das verteilte System an sich, es kann normal weiterarbeiten. Es kann immer noch funktionieren, wenn ein oder zwei Knoten ausfallen.

Selbst bei einem Totalausfall ist es nach einem Neustart sofort wieder verfügbar.



Lediglich bei der Datenhaltung muss mit Einschränkungen gerechnet werden. Die Konsistenz (nach einem Ausfall) ist nicht immer gegeben und kann im schlimmsten Fall nicht erreicht werden. Außerdem hängt die Konsistenz der Datenhaltung, wie bereits erwähnt, auch von der Geschwindigkeit und Zuverlässigkeit der Netzwerkverbindung zwischen den Knoten ab.

Schnittstellen

Die Programme Client und Knoten sind über Textdateien konfigurierbar. Diese Textdateien werden beim Programmstart geladen.

Für den Client gibt es zwei Textdateien:

- **config.txt**, in der die Dimensionen des Feldes definiert werden
- **ips.txt**, in der die IPS der Knoten im verteilten System stehen, da diese beim "Login" als Stellvertreter abgefragt werden.

Für den einzelnen Knoten gibt es 3 Textdateien:

- **area.txt**, in der auch die Felddimensionen gespeichert werden müssen (die Felddimensionen beim Client und beim Knoten müssen identisch sein)
- **ips.txt**, in der die IPs der anderen Knoten, aus denen das verteilte System bestehen soll, gespeichert werden.
- **ownip.txt**, für den eigenen Knoten ist es wichtig, seine eigene IP-Adresse zu kennen, unter der er im jeweiligen Netz erreichbar ist.

Beschreibung der Anpassung

Die Funktionen der TrafficControlLogic Class, welche uns bereitgestellt werden, können eins zu eins übernommen werden, an der Logik muss nichts geändert werden. Lediglich die Portierung in die gewählte Programmiersprache ist durchzuführen.

Programm

Voraussetzungen

Die Implementierung des Projekts, sowohl die Knoten für das verteilte System als auch der Client, wurden vollständig in der Programmiersprache C# entwickelt. Um den Client oder die Knoten auf einem Raspberry PI OS (oder einem ähnlichen auf Debian 11 basierenden Betriebssystem) auf den 3 Raspberry PI zu installieren, müssen folgende Befehle mit sudo ausgeführt werden. Für die Installation ist eine Internetverbindung erforderlich.

<https://learn.microsoft.com/en-us/dotnet/core/install/linux-debian?source=recommendations>

```
wget https://packages.microsoft.com/config/debian/11/packages-microsoft-prod.deb -O packages-microsoft-prod.deb
sudo dpkg -i packages-microsoft-prod.deb
rm packages-microsoft-prod.deb

sudo apt-get update && \
sudo apt-get install -y dotnet-sdk-7.0
sudo apt-get update && \
sudo apt-get install -y aspnetcore-runtime-7.0

sudo apt-get install -y dotnet-runtime-7.0
```

Dies installiert die Dotnet Runtime-Umgebung, auf der die Programme ausgeführt werden.

Für den Betrieb und Test müssen die Knoten miteinander netzwerktechnisch kommunizieren können. Dazu muss die Verbindung auf Port 8888 erlaubt sein.

Der Client muss sich ebenfalls mit jedem Knoten auf Port 8888 verbinden können. Theoretisch ist es möglich, dass der Client nicht im selben Netzwerk läuft. Auch wenn er sich hinter einer Firewall / Router befindet und z.B. genannt wird, kann er eine ausgehende Verbindung zu den Knoten aufbauen.

(Voraussetzung ist, dass der Knoten Verbindungen von "außen" von den genannten Geräten zulässt und keine Firewall dies blockiert).

Quellcode

Knoten

Beim Start des Knoten Programms wird zunächst die Konfigurations-Textdateien mit `InitNodeConfig()` eingelesen, welche sich im Ordner `config/` befinden.

```
private static void Main()
{
    InitNodeConfig(); // initialize important node variables

    OpenSocket(_myIp.ToString());
    Logger.HighlightMessage($"Node {_myIp} online");

    foreach (var slave in _slaveIpList)
    {
        TryConnectToSocket(slave);
    }

    // keep Program "minutesToShutdown" alive
    while (true)
    {
        Thread.Sleep(millisecondsTimeout: 1000);
        _minutesToShutdown--;

        if (_minutesToShutdown == 0)
            break;
    }
}

private static void InitNodeConfig()
{
    Logger.IsLoggerEnabled = true;
    Logger.IsDebugEnabled = false;

    _slaveIpList = File.ReadAllLines(path: "config/ips.txt").ToList();
    _myIp = IPAddress.Parse(File.ReadAllLines(path: "config/ownip.txt").ToList()[0]);

    var areaConfig = File.ReadAllLines(path: "config/area.txt").ToList();

    _maxAreaX = short.Parse(areaConfig[0]);
    _maxAreaY = short.Parse(areaConfig[1]);
    _maxCarsPerNode = short.Parse(areaConfig[2]);

    _trafficControlLogic = new TrafficControlLogic(new TrafficArea(_maxCarsPerNode, _maxAreaX, _maxAreaY));
}
```

Die Werte werden Zeilenweise aus den Dateien gelesen und haben folgenden Schema: (es darf keine leere Zeile am ende der Datei sein !)

area.txt:

- 1. Zeil = x-achse Feld
- 2. Zeile y-achse Feld
- 3. Zeile maximale Verkehrslast Pro bestimmtest Feld

area.txt	
1	1000
2	1000
3	2

ips.txt:

- Zeilenweise die IP-Adressen **aller** Knoten angeben

ips.txt	
1	193.196.54.176
2	193.196.55.166
3	193.196.53.217

ownip.txt

- Einzeiler in der die eigene IP-Adresse des jeweiligen Knoten steht

ownip.txt	
1	193.196.54.176

Nach dem Einlesen dieser Dateien wird für den eigene Knoten der Socket auf Port 8888 zum Verbinden geöffnet `OpenSocket(_myIp.ToString());`.

Anschließend wird einmalig versucht, eine Verbindung zu allen in der Liste aufgeführten Knoten per Socket aufzubauen. `TryConnectToSocket(slave);`.

Der Main Thread bleibt dabei dauerhaft aktiv

In der `TryConnectToSocket` Funktion wird versucht, eine Socket-Verbindung zu den übergebenen Knoten aufzubauen. Ist dies erfolgreich, wird der verbundene Knoten in die Liste der `_masterNodes` aufgenommen. Dabei wird der Aspekt der Socketverbindung und der Lock-Mechanismus berücksichtigt.

Hierbei wird mit `Task.Run(() => ListenToNodeSocket(socket));` das Lauschen auf eingehende Verbindungen in einer separaten Task ausgeführt. Dies stellt die Nebenläufigkeit dar, bei der eingehende Verbindungen einen eigenen Thread erhalten.

Im besten Fall sind nun alle Knoten untereinander mit ein und ausgehenden Sockets verbunden.

Anschließend gibt es zwei Möglichkeiten, wie Verbindungen zu dem Knoten kommen können:

Vom Client oder von Knoten

Die `OpenSocket(_myIp.ToString());` Funktion öffnet im einen weiteren Task welcher ankommenden Traffic handelt `Task.Run(() => HandleIncomingConnections(socket));`

Wenn die Verbindung von einem **Client** kommt, wird diese auch wieder durch einen Thread angehört und später durch die Funktion `HandleClientRequest()` weiter verarbeitet.

Kommt die eingehende Verbindung von einem **Knoten**, der eigentlich in der `ips.txt` Liste enthalten ist, aber nicht in der aktuellen `_masterNodes` Liste existiert, wird dieser in das verteilte System aufgenommen und dem Socket wird als separaten Thread gelauscht. (Dies ist der Fall, wenn ein Knoten zuvor noch nicht in System angemeldet war, oder der Knoten offline war und wieder online kommt)

```

/// <summary>
/// Try connect to a given socket from remoteIp
/// </summary>
/// </summary>
private static void TryConnectToSocket(string remoteIp)
{
    if (remoteIp == _myIp.ToString())
        return;

    try
    {
        var socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
        var address = IPAddress.Parse(remoteIp);
        var endPoint = new IPEndPoint(address, OpenPort);

        socket.Connect(endPoint);

        Logger.InfoMessage($"Connected to master: {socket.RemoteEndPoint}");

        Task.Run(() => ListenToNodeSocket(socket));

        lock (_masterNodes)
            _masterNodes.Add(socket);
    }
    catch (Exception)
    {
        Logger.ErrorMessage($"Could not connect to: {remoteIp}");
    }
}

```

```

private static void OpenSocket(string ipAddress)
{
    try
    {
        //open socket on own IpAddress
        var socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);

        var address = IPAddress.Parse(ipAddress);
        var endPoint = new IPEndPoint(address, OpenPort);

        socket.Bind(endPoint);
        socket.Listen(backlog: 1000);

        Logger.InfoMessage($"Socket listening on {ipAddress}:{OpenPort}");

        // Spawn a new thread to handle each incoming connection
        Task.Run(() => HandleIncomingConnections(socket));
    }
}

```

```

private static void HandleIncomingConnections(Socket socket)
{
    while (true)
    {
        // accept new incoming connections
        var clientSocket = socket.Accept();
        var ipString = GetAddressFromRemoteEndPoint(clientSocket);

        // Client connected to socket
        if (!_slaveIpList.Contains(ip))
        {
            Task.Run(() => ListenToClientSocket(clientSocket));
            continue;
        }

        Logger.InfoMessage($"Slave connected: {clientSocket.RemoteEndPoint}");

        //listen to other node on own socket
        Task.Run(() => ListenToNodeSocket(clientSocket));

        // connect to slaves own socket
        lock (_masterNodes)
        {
            if (_masterNodes.Exists(match: s => GetAddressFromRemoteEndPoint(s) == GetAddressFromRemoteEndPoint(clientSocket)))
                continue;

            Task.Run(() => TryConnectToSocket(GetAddressFromRemoteEndPoint(clientSocket)));
        }
    }
}

```

Die vom Client eingehende Verbindung wird durch die Funktion **HandleClientRequest** abgedeckt. Dort wird zunächst geprüft, ob sich der Client mit der Information **"Help"** beim Stellvertreter anmeldet. Dies stellt die „Anmeldung“ einer Anfrage dar.

Nach der Anmeldung muss schließlich der Round-Robin-Lastverteilungsalgorithmus ausgeführt werden, um dem Client mitzuteilen, welcher Knoten für die Berechnung zuständig ist (auch hier wird dies pro Thread gesperrt **lock (_roundRobinLock)**).

```
private static string GetNextRoundRobinIp()
{
    lock (_roundRobinLock)
    {
        while (true)
        {
            //increment counter
            if (_roundRobin >= _slaveIpList.Count - 1)
                _roundRobin = 0;
            else
                _roundRobin++;

            lock (_masterNodes)
            {
                // check if own ip
                if (_slaveIpList[_roundRobin] == _myIp.ToString())
                    return _slaveIpList[_roundRobin];

                // check if next node is online
                if (_masterNodes.Exists((masterNode) => GetAddressFromRemoteEndpoint(s) == _slaveIpList[_roundRobin]))
                    return _slaveIpList[_roundRobin];
            }
        }
    }
}
```

```
/// <summary>
/// Handles incoming Client Requests. Updates TrafficControlLogic with given data.
/// </summary>
/// <usage>
private static bool HandleClientRequest(Socket socket, string data)
{
    if (data.Contains("Help")) // Client asks for navigation
    {
        var roundRobinIp = GetNextRoundRobinIp();

        //check with round robin who should handle request
        if (roundRobinIp == _myIp.ToString())
            SendTcp(socket, message: "Accepted");
        else
        {
            SendTcp(socket, message: $"{roundRobinIp}");
            Logger.InfoMessage($"Connection with {GetAddressFromRemoteEndpoint(socket)} ended.");
            return false;
        }
    }
    else if (data.Contains("Sudo help")) // Client demands navigation
    {
        SendTcp(socket, message: "Accepted");
    }
    else if (data.Contains("Req")) // Client requests next step
    {
        var newPos = CalculateMove(data);

        var id = short.Parse(data.Split(separator: new[] { ' ', '-', '[', ']', ',' }, StringSplitOptions.RemoveEmptyEntries)[0]);
        var messageToNodes = $"Change:{id}-{newPos.X},{newPos.Y}";
        var messageToClient = $"Ass:{newPos.X},{newPos.Y}";

        SendTcp(socket, messageToClient);
        SendChangeToNodes(messageToNodes);

        Logger.InfoMessage(messageToClient);
    }

    return true;
}
```

Der Client, der den Server um die Berechnung der Anfrage bittet, fordert diese beim Knoten mit "sudo help" an. Diese Anfrage wird dann mit **"Accepted"** an den Client zurückgesendet. Dieser kann und wird nun seine Berechnungsanfrage mit dem Parameter "req" senden und der **else if (data.Contains("Req"))** Fall tritt ein. Dabei wird der nächste Schritt mit der Funktion **var newPos = CalculateMove(data);** berechnet.

Die Änderungen der Position werden dann dem Client mit **SendTcp(socket, messageToClient);** und den restlichen Nodes im verteilten System mit **SendChangeToNodes(messageToNodes);** mitgeteilt. (Die **_masterNodes** werde dabei wieder gelockt)

```
/// <summary>
/// Sends message on specific socket.
/// </summary>
/// <usage>
private static async void SendTcp(Socket socket, string message)
{
    lock (_masterNodes)
    {
        foreach (var node in _masterNodes)
        {
            try
            {
                SendTcp(node, message);
            }
            catch
            {
                // ignored
            }
        }
    }

    /// <summary>
    /// </summary>
    /// <usage>
    private static async void SendTcp(Socket socket, string message)
    {
        try
        {
            if (!socket.Connected)
                throw new SocketException();

            var messageBuffer = Encoding.ASCII.GetBytes($"~{message}~");
            await socket.SendAsync(messageBuffer);
        }
        catch
        {
        }
    }
}
```

```
/// <summary>
/// Calculates next move for a navigation request and updates TrafficControlLogic.
/// </summary>
/// <usage>
private static Coordinate CalculateMove(string request)
{
    var parts = request.Split(separator: new[] { ' ', '-', '[', ']', ',' }, StringSplitOptions.RemoveEmptyEntries);

    var id = short.Parse(parts[1]);
    var fromPosParts = parts[2].Split(separator: ',');
    var fromPos = new Coordinate(short.Parse(fromPosParts[0]), short.Parse(fromPosParts[1]));

    var toPosParts = parts[3].Split(separator: ',');
    var toPos = new Coordinate(short.Parse(toPosParts[0]), short.Parse(toPosParts[1]));

    try
    {
        Coordinate nextMove;
        lock (_trafficControlLogic)
        {
            var pos = _trafficControlLogic.TrafficArea.GetPosition(id);

            if (pos == null) // add if new car
                _trafficControlLogic.TrafficArea.Place(id, fromPos);
            else if (pos != fromPos) // update pos if old car
            {
                _trafficControlLogic.TrafficArea.Remove(id, pos);
                _trafficControlLogic.TrafficArea.Place(id, fromPos);
            }

            nextMove = _trafficControlLogic.Move(id, toPos);
        }

        return nextMove;
    }
    catch
    {
        return fromPos;
    }
}
```

Zusätzlich gibt es eine Logger-Funktion, die bei Bedarf konfiguriert wird und über die Parameter in `InitNodeConfig()`.

`Logger.IsLoggerEnabled = true;`

`Logger.IsDebugEnabled = false;`

ein- und ausgeschaltet werden kann.

```
namespace ServerNode;

[16 usages]
public abstract class Logger
{
    [Fields]
    [Properties]

    #region Public Methods
    [5 usages]
    public static void ErrorMessage(string? message)
    {
        LogToConsole( message: "[Error] " + message, ConsoleColor.DarkRed);
    }

    [2 usages]
    public static void HighlightMessage(string? message)
    {
        LogToConsole( message: "[Status] " + message, ConsoleColor.DarkGreen);
    }

    [2 usages]
    public static void TcpInMessage(string? message)
    {
        if(!IsDebugEnabled)
            return;

        LogToConsole( message: "[TCP:in] " + message, ConsoleColor.DarkCyan);
    }

    [5 usages]
    public static void InfoMessage(string? message)
    {
        if(!IsDebugEnabled)
            return;

        LogToConsole( message: "[Info] " + message, ConsoleColor.Gray);
    }
    #endregion

    [Private Method]
}
```

Auch bei dieser selbsterstellten Loggerklasse wurde wieder die Nebenläufigkeit berücksichtigt. Mit `lock(_messageLock)` wird die Log-Ausgabe (Farbe, Text, etc.) thread-sicher gemacht.

```
#region Private Method
[4 usages] Thomas Alpert
private static void LogToConsole(string? message, ConsoleColor color)
{
    if(!IsLoggerEnabled)
        return;

    lock (_messageLock)
    {
        Console.ForegroundColor = color;
        Console.WriteLine(message);
        Console.ResetColor();
    }
}
#endregion
}
```

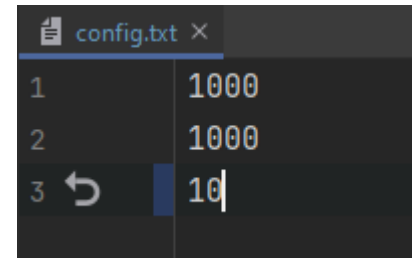
Client

Zu Beginn des Client-Programms werden die Randbedingungen über die **Main**-Funktion mit der Textdatei **config/config.txt** eingelesen. In dieser config.txt werden die Informationen der Feldgröße und der Anzahl der simulierten Clients entnommen:

Die Werte werden Zeilenweise aus den Dateien gelesen und haben folgenden Schema: (es darf keine leere Zeile am ende der Datei sein !)

config.txt:

- 1. Zeil = x-achse Feld
- 2. Zeile y-achse Feld
- 3. Anzahl der zu simulierenden Clients



Die Feldgröße muss hierbei gleichgroß sein (identische Dimensionen) wie sie im Server konfiguriert wurden ist.

```
private static void Main(string[] args)
{
    Logger.IsDebugEnabled = true;

    var areaConfigList<string> = File.ReadAllLines(path:"config/config.txt").ToList();
    _maxAreaX = short.Parse(areaConfig[0]);
    _maxAreaY = short.Parse(areaConfig[1]);
    ClientCnt = short.Parse(areaConfig[2]);

    // start clients
    for (var i = 0; i < ClientCnt; i++)
    {
        var client = new Client(_maxAreaX, _maxAreaY, i);

        Task.Run(() => client.BeginCommunication());

        Thread.Sleep(millisecondsTimeout: ClientStartDelay);
    }

    // keep Program "minutesToShutdown" alive
    while (true)
    {
        Thread.Sleep(millisecondsTimeout: 1000);
        _minutesToShutdown--;

        if (_minutesToShutdown == 0)
            break;
    }
}
```

Für die Anzahl der zu simulierenden Clients wird eine FOR-Schleife ausgeführt. In dieser Schleife wird dann mit **var client = new Client(_maxAreaX, _maxAreaY, i);** ein Client als Objekt einer Klasse initialisiert.

Zusätzlich wird ein separater Task / Thread erstellt, der für jeden Client / ausgehende Verbindung erstellt wird.

Damit der Client nicht „endlos“ läuft, wurde eine Art Kill-Switch eingebaut, der nach einer definierten Anzahl

von Minuten (**_minutesToShutdown**) das Hauptprogramm beendet und damit das Programm schließt.

Der Konstruktor der Klasse Client weist jedem Objekt (jedem simulierten Client) zufällige Start- und Zielwerte zu. Der Seed, der bestimmt, welche "zufälligen" Werte in welcher Reihenfolge erzeugt werden, kann über die Variable **public static readonly Random Random = new(0);** gesteuert werden. Zu Testzwecken und um untereinander vergleichbare Testergebnisse zu erhalten, wurde dieser Seed während des gesamten Test- / Implementierungsprozesses auf 0 gesetzt. Dies stellt sicher, dass die generierten Zufallszahlen gleich sind und somit bei unterschiedlichen Tests die Clients die gleiche Start- und Zieladresse haben.

```
#region Constructors

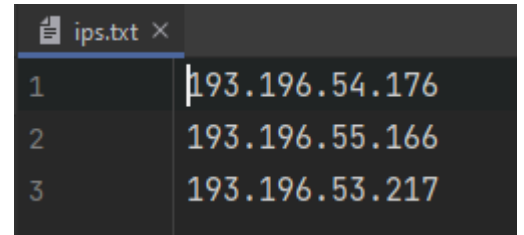
[Usage] Thomas Alpert +1
public Client(short maxX, short maxY, int id)
{
    _start = DateTime.Now;

    //init client with random start and target
    lock (Program.Random)
    {
        var r.Random = Program.Random;

        _serverIps = File.ReadAllLines(path:"config/ips.txt").ToList();
        _currentPos = new Coordinate((short) r.Next(maxX), (short) r.Next(maxY));
        _targetPos = new Coordinate((short) r.Next(maxX), (short) r.Next(maxY));
    }

    _id = id;
}
```

Zusätzlich wird für jeden Client die Liste der IP-Adressen der Knoten im verteilten System eingelesen. Dabei müssen die IP-Adressen zeilenweise (ohne Zeilenumbruch an letzter Stelle) in der Datei configs/ips.txt stehen.)




```

1 193.196.54.176
2 193.196.55.166
3 193.196.53.217

```

Gelingt dies, wird der Server über die Funktion `ConnectToServer()` gefolgt von `ListenToServerSocket()` als Stellvertreter kontaktiert. (Über den Parameter `force` kann mit `false` gesteuert werden, dass Anfragen an den ausgewählten Knoten zunächst als "Stellvertreter"-Anfragen behandelt werden.) Der Client meldet sich also mit „Help“ am Knoten über die `await SendTcp(socket, !force ? "Help" : "Sudo help");` Funktion



```

public async void BeginCommunication()
{
    var ipTarget = 0;
    var res = false;

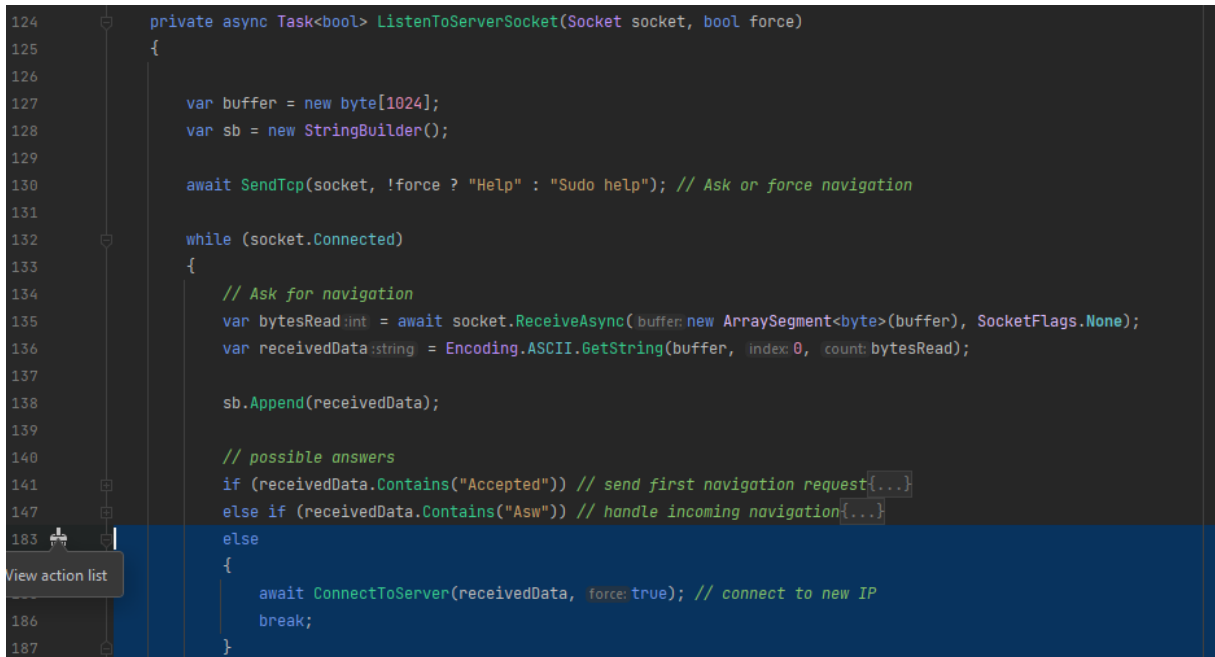
    while (!res)
    {
        try
        {
            Logger.InfoMessage(_serverIps[ipTarget]);

            if (ipTarget >= _serverIps.Count - 1)
                ipTarget = 0;
            else
                ipTarget++;

            //connect to server
            res = await ConnectToServer(_serverIps[ipTarget], force: false);

            Thread.Sleep(millisecondsTimeout: TimeToCheckServer);
        }
        catch // connection denied or crashed
        {
            res = false;
        }
    }
}

```



```

124 private async Task<bool> ListenToServerSocket(Socket socket, bool force)
125 {
126
127     var buffer = new byte[1024];
128     var sb = new StringBuilder();
129
130     await SendTcp(socket, !force ? "Help" : "Sudo help"); // Ask or force navigation
131
132     while (socket.Connected)
133     {
134         // Ask for navigation
135         var bytesRead: int = await socket.ReceiveAsync(buffer: new ArraySegment<byte>(buffer), SocketFlags.None);
136         var receivedData: string = Encoding.ASCII.GetString(buffer, index: 0, count: bytesRead);
137
138         sb.Append(receivedData);
139
140         // possible answers
141         if (receivedData.Contains("Accepted")) // send first navigation request{...}
142         else if (receivedData.Contains("Asw")) // handle incoming navigation{...}
143         else
144         {
145             await ConnectToServer(receivedData, force: true); // connect to new IP
146             break;
147         }
148     }
149 }

```

Anschließend wird in den markierten Else Fall gesprungen, da der Proxy nun eine IP-Adresse zurückgibt, mit der sich der Client verbinden soll. (Mit Round Robin wurde der Knoten ausgewählt, der nun rechnen soll). Der Client verbindet sich also mit der Funktion `ConnectToServer(receivedData, true);` mit der übergebenen IP des Knotens, diesmal aber mit dem Parameter `force auf true`.

Mit diesem Parameter (`true`) wird dann an die Funktion `ListenToServerSocket()` aufgerufen, die nun „Sudo help“ an den neu verbundenen Knoten über den Socket sendet. Damit wird dem Knoten mitgeteilt, dass er kein Stellvertreter mehr ist, sondern die Anfrage bitte berechnen soll. Der Knoten

akzeptiert nun die Anfrage und sendet „Accepted“ an den Client, welcher nun beginnt, seine Start- und Zielposition mit `$"Req:{_id}-[{_currentPos.X},{_currentPos.Y}]-[{_targetPos.X},{_targetPos.Y}];"` an den Knoten sendet.

```
// possible answers
if (receivedData.Contains("Accepted")) // send first navigation request
{
    var message = $"Req:{_id}-[{_currentPos.X},{_currentPos.Y}]-[{_targetPos.X},{_targetPos.Y}];";
    await SendTcp(socket, message);
    Logger.InfoMessage(message);
}
```

Dieser Knoten berechnet nun den nächsten Schritt und gibt diesen dann über den Socket an den Client als **Asw** (Antwort) über denselben Socket zurück.

Der Client erhält nun als Antwort neue Positionsdaten, aktualisiert seine eigene Position (geht also einen Schritt) und sendet seine Anfrage mit den aktualisierten Positionsdaten an denselben Knoten über die immer noch vorhandenen Socket-Verbindung.

```
147 else if (receivedData.Contains("Asw")) // handle incoming navigation
148 {
149     Logger.TcpInMessage(receivedData);
150
151     var parts string[] = receivedData.Split(separator: new[] { ' ', '[', ']', ';' }, StringSplitOptions.RemoveEmptyEntries);
152     var toPos string[] = parts[1].Split(separator: ',');
153
154     var answerPos = new Coordinate((short.Parse(toPos[0]), (short.Parse(toPos[1])));
155
156     if (answerPos.X == _targetPos.X && answerPos.Y == _targetPos.Y)
157     {
158         lock (Program.FinishedCountLock)
159         {
160             socket.Disconnect(reuseSocket: false);
161             Program.FinishedCount++;
162             Logger.HighlightMessage($"({_id} reached destination {(DateTime.Now - _start).TotalSeconds} [{Program.FinishedCount}/{Program.ClientCnt}]");
163         }
164         return true;
165     }
166
167     _currentPos = answerPos;
168
169     Thread.Sleep(millisecondsTimeout: RequestDelay);
170
171     if (_requestCount % MaxRequestPerNode == 0) // reconnect after MaxRequestPerNode in order to rebalance
172     {
173         _requestCount = 1;
174         Logger.InfoMessage("MaxRequestPerNode reached, reconnecting");
175         throw new SocketOverloadException();
176     }
177
178     var message = $"Req:{_id}-[{_currentPos.X},{_currentPos.Y}]-[{_targetPos.X},{_targetPos.Y}];";
179     _requestCount++;
180     await SendTcp(socket, message);
181     Logger.InfoMessage(message);
```

Dieser Vorgang wird solange wiederholt, bis die Position gefunden wurde oder der **erweiterte Lastverteilungsalgorithmus** greift. Anschließend wird geprüft, ob eine Client-Anfrage/Verbindung mehr als die in der Variablen **MaxRequestPerNode** festgelegte Anzahl von Schritten umfasst. Ist dies der Fall, wird der Client getrennt, eine Exception geworfen `throw new SocketOverloadException();` und die Anfrage erneut an das verteilte System gesendet. Als Schwellwert wurde 1/3 der Feldgröße als maximale Schrittzahl definiert, was allerdings jederzeit angepasst werden kann.

```
12 private int _requestCount = 1;
13
14 private const int RequestDelay = 20;
15 private static readonly int MaxRequestPerNode = (Program._maxAreaX + Program._maxAreaY) / (3*2); //make it dependent on field size
16
```


Startanweisung

Sowohl der Client als auch der Server können (nach Installation der Dot-Net Runtime) auf die gleiche Weise gestartet werden.

Vor dem Start der Anwendungen müssen jedoch die Konfigurationsdateien für den Client, config.txt und ips.txt im Verzeichnis /config ausgefüllt werden.

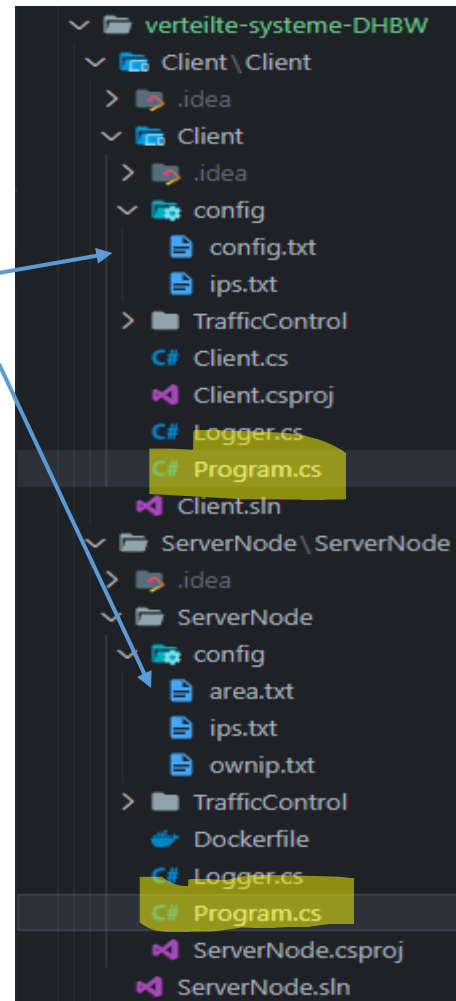
Für den Server müssen die Konfigurationsdateien area.txt, ips.txt, ownip.txt ebenfalls wie oben beschrieben ausgefüllt werden.

Zuerst müssen die **Knoten** gestartet werden, was durch einen Wechsel in das Verzeichnis `verteilte-systeme-DHBW/ServerNode/ServerNode/ServerNode/`, in dem sich die `Program.cs` befindet, möglich ist. Die Knoten/Server können dann nacheinander mit dem einfachen Befehl `dotnet run` gestartet werden.

```
ubuntu@node01:~/verteilte-systeme-DHBW/ServerNode/ServerNode/ServerNode$ dotnet run
[Status] Node 193.196.55.166 online
[Error] Could not connect to: 193.196.54.176
[Error] Could not connect to: 193.196.53.217

ubuntu@node2:~/verteilte-systeme-DHBW/ServerNode/ServerNode/ServerNode$ dotnet run
[Status] Node 193.196.53.217 online
[Error] Could not connect to: 193.196.54.176

ubuntu@node03:~/verteilte-systeme-DHBW/ServerNode/ServerNode/ServerNode$ dotnet run
[Status] Node 193.196.54.176 online
```



Anschließend kann der **Client** gestartet werden, indem in den Pfad `/verteilte-systeme-DHBW/Client/Client/Client/` gewechselt wird, in dem sich die `Program.cs` befindet. Dieser kann dann wie der Server mit `dotnet run` gestartet werden. Die Abfragen/Simulationen starten dann direkt

Beim Testen/Simulieren mehrerer Clients wird empfohlen, die Protokollierung Clientseitig zu deaktivieren, da sonst zu viele Informationen angezeigt werden. `Logger.IsDebugEnabled = false` ;in der `Program.cs`

Logger.IsDebugEnabled = false

```
ubuntu@cloud-computing:~/verteilte-systeme-DHBW/Client/Client/Client$ dotnet run
/home/ubuntu/verteilte-systeme-DHBW/Client/Client/Client/TrafficControl/TrafficArea.sproj]
[Status] 7 reached destination 35.918666 [1/200]
[Status] 0 reached destination 45.5668488 [2/200]
[Status] 6 reached destination 43.2723023 [3/200]
[Status] 16 reached destination 51.1849069 [4/200]
[Status] 19 reached destination 95.0413091 [5/200]
[Status] 14 reached destination 103.0404812 [6/200]
[Status] 3 reached destination 129.2803862 [7/200]
[Status] 17 reached destination 150.9757839 [8/200]
[Status] 25 reached destination 145.8969291 [9/200]
[Status] 10 reached destination 161.746132 [10/200]
[Status] 5 reached destination 179.3073316 [11/200]
```

Logger.IsDebugEnabled = true

```
ubuntu@cloud-computing:~/verteilte-systeme-DHBW/Client/Client/Client$ dotnet run
/home/ubuntu/verteilte-systeme-DHBW/Client/Client/Client/TrafficControl/TrafficArea.sproj]
[Info] 193.196.54.176
[Info] Connected to master: 193.196.55.166:8888
[Info] Req:0-[726,817]-[768,558];
[TCP:in] Accepted
[TCP:in] Asw:[727,816];
[Info] Req:0-[727,816]-[768,558];
[TCP:in] Asw:[727,816];
[TCP:in] Asw:[728,815];
[Info] Req:0-[728,815]-[768,558];
[TCP:in] Asw:[768,558];
[Status] 0 reached destination 14.684159 [1/1]
```

Testplan

Folgende Punkte werden geprüft:

- Grundlegender Funktionstest mit 1 Client
- Extremwerttests (min., max.)
- Last- und Überlasttest
- Erzeugung von HW-Fehlern

Test Nr.	Testbezeichnung	Vorbedingungen	Beschreibung der Testschritte	Erwartetes Ergebnis	Ergebnis	Nachbedingung
1	Grundlegender Funktionstest mit 1 Client	3 Knoten laufen	Es wird getestet, ob der Client über das verteilte System eine Simulation durchführen kann und ob ein Fahrzeug generell an die Zielposition geleitet werden kann.	OK	OK	Keine

```
ubuntu@cloud-computing:~/verteilte-systeme-DHBW/Client/Client/Client$ dotnet run
/home/ubuntu/verteilte-systeme-DHBW/Client/Client/Client/TrafficControl/TrafficArea.sproj]
[Info] 193.196.54.176
[Info] Connected to master: 193.196.55.166:8888
[Info] Req:0-[726,817]-[768,558];
[TCP:in] Accepted
[TCP:in] Asw:[727,816];
[Info] Req:0-[727,816]-[768,558];
[TCP:in] Asw:[727,816];
[TCP:in] Asw:[728,815];
[Info] Req:0-[728,815]-[768,558];
[TCP:in] Asw:[728,815];
[TCP:in] Asw:[729,814];
[Info] Req:0-[729,814]-[768,558];
[TCP:in] Asw:[729,814];
```



```
[Info] Req:0-[768,562]-[768,558];
[TCP:in] Asw:[768,562];
[TCP:in] Asw:[768,561];
[Info] Req:0-[768,561]-[768,558];
[TCP:in] Asw:[768,561];
[TCP:in] Asw:[768,560];
[Info] Req:0-[768,560]-[768,558];
[TCP:in] Asw:[768,560];
[TCP:in] Asw:[768,559];
[Info] Req:0-[768,559]-[768,558];
[TCP:in] Asw:[768,559];
[TCP:in] Asw:[768,558];
[Status] 0 reached destination 14.684159 [1/1]
```

Test Nr.	Testbezeichnung	Vorbedingungen	Beschreibung der Testschritte	Erwartetes Ergebnis	Ergebnis	Nachbedingung
2	Extremwerttests (min., max.)	3 Knoten laufen	Es ist zu prüfen, was passiert, wenn ungültige Start- oder Zielpositionen vom Client an das Verteilsystem gesendet werden. Ungültige Positionen können negative Werte, Fließkommawerte oder Werte außerhalb des Feldes sein.	NOK	NOK	Es findet keine Client und oder Serverseitige Überprüfung der eingegeben Werte statt. Diese kann / ist zu ergänzen.

Test2 schlägt fehl, weil die Validierung der Eingabe auf Client- und Server-Seite nicht erfolgt. Für ein Simulationsszenario, in dem die Eingabewerte (Start- und Zielposition) zufällig generiert werden, ist zu überlegen, ob diese generierten Werte auf Korrektheit überprüft werden sollen. Ist der zu generierende Bereich einmal korrekt gesetzt, was in unserem Fall durch das Einlesen der Geometrie der Felddatei geschieht, werden die Werte auch innerhalb dieser generiert.

Test Nr.	Testbezeichnung	Vorbedingungen	Beschreibung der Testschritte	Erwartetes Ergebnis	Ergebnis	Nachbedingung
3.1	Lasttest mit 10 Clients	3 Knoten laufen wurden allerdings neugestartet	Es wird getestet, ob 10 Clients gleichmäßig auf die 3 Knoten verteilt werden und ob diese ihr Ziel erreichen.	OK	OK	Nodes müssen neugestartet werden, da „parkende“ Autos nicht verschwinden

Node 01 erhält als Stellvertreter 10 Anfragen

```

ubuntu@node01:~/verteilte-systeme-DHBW/ServerNode/ServerNode/ServerNode$ dotnet run
[Status] Node 193.196.55.166 online
[Status] Client 193.196.55.38:35664 connected.
[Status] Client 193.196.55.38:35672 connected.
[Status] Client 193.196.55.38:46194 connected.
[Status] Client 193.196.55.38:46206 connected.
[Status] Client 193.196.55.38:46218 connected.
[Status] Client 193.196.55.38:46222 connected.
[Status] Client 193.196.55.38:46238 connected.
[Status] Client 193.196.55.38:46252 connected.
[Status] Client 193.196.55.38:46254 connected.
[Status] Client 193.196.55.38:46256 connected.

```

→ 3 Anfragen landen bei Knoten 2

```

[Error] connection to 193.196.55.166 lost
ubuntu@node2:~/verteilte-systeme-DHBW/ServerNode/ServerNode/ServerNode$ dotnet run
[Status] Node 193.196.53.217 online
[Error] Could not connect to: 193.196.55.166
[Status] Client 193.196.55.38:38060 connected.
[Status] Client 193.196.55.38:50830 connected.
[Status] Client 193.196.55.38:50834 connected.

```

→ die restlichen 3 landen bei Knoten 3

```

ubuntu@node03:~/verteilte-systeme-DHBW/ServerNode/ServerNode/ServerNode$ dotnet run
[Status] Node 193.196.54.176 online
[Error] Could not connect to: 193.196.55.166
[Error] Could not connect to: 193.196.53.217
[Status] Client 193.196.55.38:43356 connected.
[Status] Client 193.196.55.38:43362 connected.
[Status] Client 193.196.55.38:36012 connected.

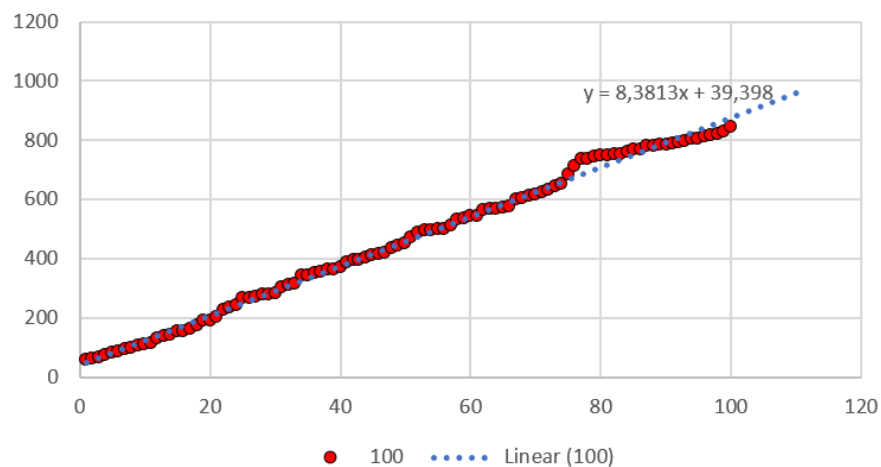
```

Test Nr.	Testbezeichnung	Vorbedingungen	Beschreibung der Testschritte	Erwartetes Ergebnis	Ergebnis	Nachbedingung
3.2	Lasttest mit 100 Clients	3 Knoten laufen wurden allerdings neugestartet	Es wird getestet, ob 100 Clients gleichmäßig auf die 3 Knoten verteilt werden und ob diese ihr Ziel erreichen.	OK	OK	Nodes müssen neugestartet werden, da „parkende“ Autos nicht verschwinden

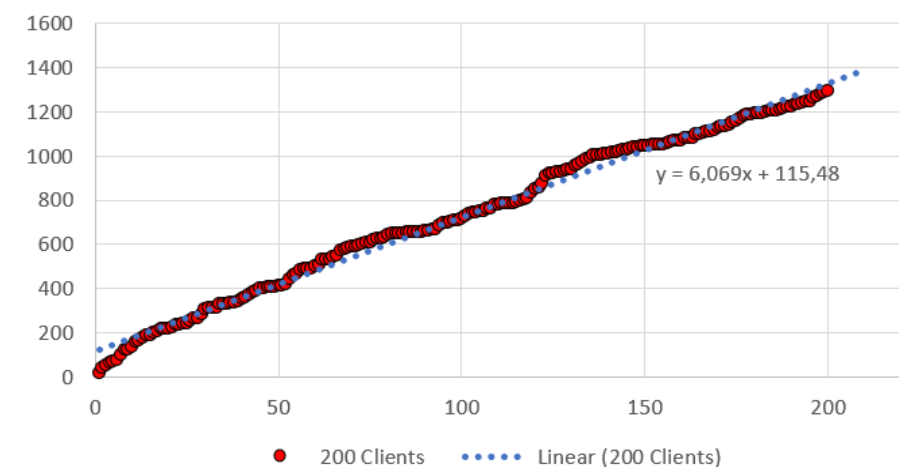
Test Nr.	Testbezeichnung	Vorbedingungen	Beschreibung der Testschritte	Erwartetes Ergebnis	Ergebnis	Nachbedingung
3.3	Lasttest mit 200 Clients	3 Knoten laufen wurden allerdings neugestartet	Es wird getestet, ob 200 Clients gleichmäßig auf die 3 Knoten verteilt werden und ob diese ihr Ziel erreichen.	OK	OK	Nodes müssen neugestartet werden, da „parkende“ Autos nicht verschwinden

Mit dem Lasttest von 100 und 200 Clients kann gezeigt werden, dass der Lasttest mit 100 Clients nach ca. 850 Sekunden (also 14 Minuten) und der Lasttest mit 200 Clients nach ca. 1293 Sekunden (also 22 Minuten) erfolgreich durchgeführt wurde. Die Y-Achse ist die Zeit in Sekunden, die X-Achse ist die Anzahl der Clients.

100

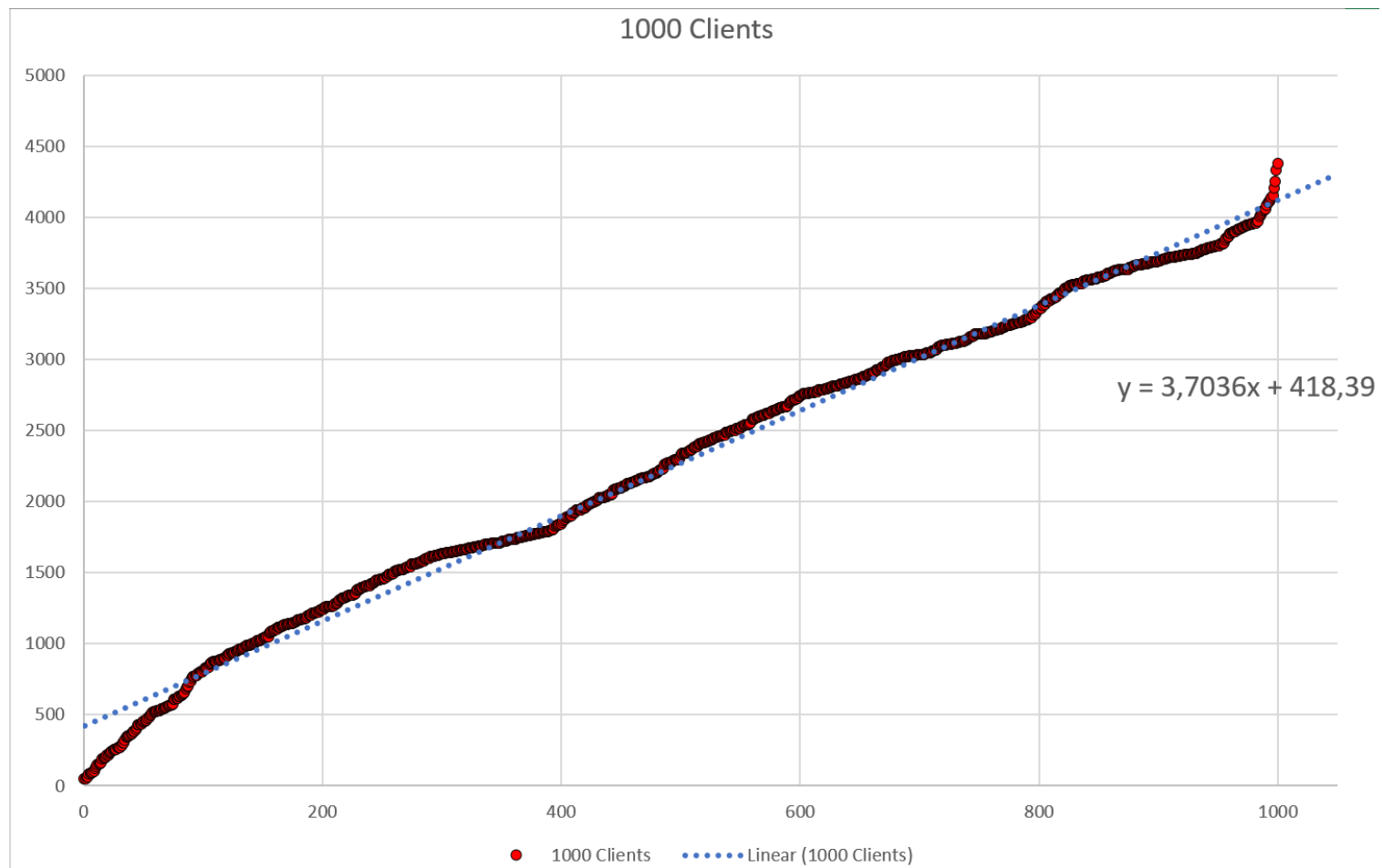


200



Test Nr.	Testbezeichnung	Vorbedingungen	Beschreibung der Testschritte	Erwartetes Ergebnis	Ergebnis	Nachbedingung
3.4	Lasttest mit 1000 Clients	3 Knoten laufen wurden allerdings neugestartet	Es wird getestet, ob 1000 Clients gleichmäßig auf die 3 Knoten verteilt werden und ob diese ihr Ziel erreichen.	OK	OK	Nodes müssen neugestartet werden, da „parkende“ Autos nicht verschwinden

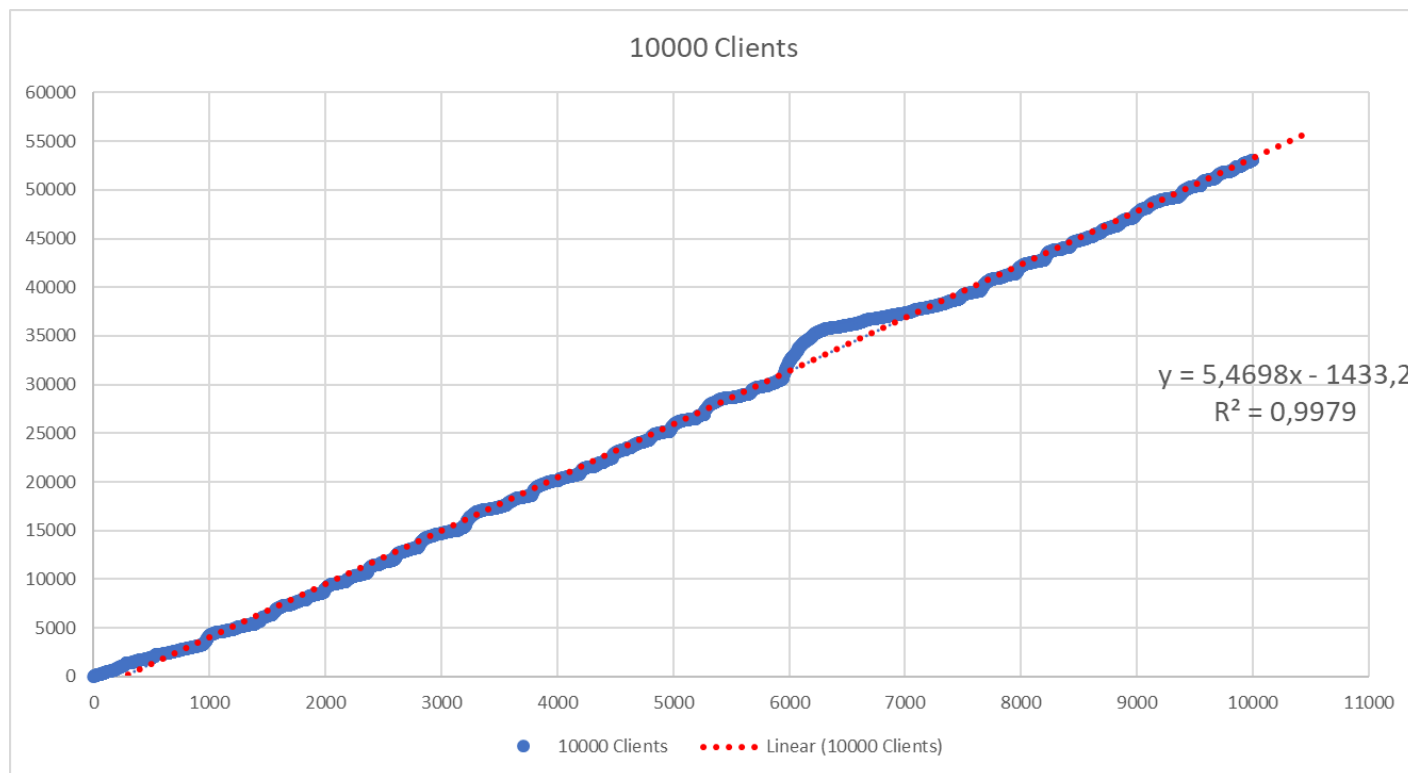
Es lässt sich zeigen, dass 1000 Clients ihr Ziel nach ca. 4377 Sekunden (ca. 72 Minuten) erreichen. Der Verlauf der Zeitlichen dauert ist im folgenden Schaubild zu sehen. Die Y-Achse gibt die Zeit in Sekunden an und die X-Achse sind die einzelnen Clients.



Test Nr.	Testbezeichnung	Vorbedingungen	Beschreibung der Testschritte	Erwartetes Ergebnis	Ergebnis	Nachbedingung
3.5	Überlast-Test mit 10000 Clients	3 Knoten laufen wurden allerdings neugestartet	Es wird getestet, ob 10000 Clients gleichmäßig auf die 3 Knoten verteilt werden und ob diese ihr Ziel erreichen.	OK	OK	Nodes müssen neugestartet werden, da „parkende“ Autos nicht verschwinden

Auch ein Überlast-Test mit 10000 Clients lässt sich mit unserem verteilten System durchführen.

Allerdings dauert dieser Vorgang bereits zirka 53025 Sekunden also rund 14,7 Stunden



Test Nr.	Testbezeichnung	Vorbedingungen	Beschreibung der Testschritte	Erwartetes Ergebnis	Ergebnis	Nachbedingung
4	Erzeugung von HW-Fehlern	3 Knoten laufen wurden allerdings neugestartet	Der Ausfall von 1, 2, ... aller Knoten wird simuliert. Bei jedem Ausfall eines Knotens ist zu überprüfen, ob die bereits angeforderten Verbindungen auf die verbleibenden Knoten umgeleitet werden .	OK	OK	

- Es werden 3 Clients simuliert (oben links). Diese kommen am Stellvertreter Knoten 1 (oben rechts an) und werden gleichmäßig an Knoten 2 und 3 verteilt

```

ubuntu@cloud-computing:~/verteilte-systeme-DHBW/Client/Client$ dotnet run
[Status] Node 193.196.55.166 online
[Error] Could not connect to: 193.196.54.176
[Error] Could not connect to: 193.196.53.217
[Status] Client 193.196.55.38:45396 connected.
[Status] Client 193.196.55.38:45402 connected.
[Status] Client 193.196.55.38:45418 connected.

ubuntu@node01:~/verteilte-systeme-DHBW/ServerNode/ServerNode$ dotnet run
[Status] Node 193.196.55.166 online
[Error] Could not connect to: 193.196.54.176
[Error] Could not connect to: 193.196.53.217
[Status] Client 193.196.55.38:45396 connected.
[Status] Client 193.196.55.38:45402 connected.
[Status] Client 193.196.55.38:45418 connected.

ubuntu@node02:~/verteilte-systeme-DHBW/ServerNode/ServerNode$ dotnet run
[Status] Node 193.196.53.217 online
[Status] Client 193.196.55.38:52926 connected.

ubuntu@node03:~/verteilte-systeme-DHBW/ServerNode/ServerNode$ dotnet run
[Status] Node 193.196.54.176 online
[Status] Client 193.196.55.38:57364 connected.

```

- Der Erste Node (oben rechts fällt aus) und der Client von verbindet sich mit dem nächsten Knoten (gelb Markiert)

```

ubuntu@cloud-computing:~/verteilte-systeme-DHBW/Client/Client$ dotnet run
[Error] Can't connect to lost
[Error] Can't connect to 193.196.55.166 lost

ubuntu@node01:~/verteilte-systeme-DHBW/ServerNode/ServerNode$ dotnet run
[Status] Node 193.196.55.166 online
[Error] Could not connect to: 193.196.54.176
[Error] Could not connect to: 193.196.53.217
[Status] Client 193.196.55.38:40990 connected.
[Status] Client 193.196.55.38:41002 connected.
[Status] Client 193.196.55.38:41010 connected.
^Cubuntu@node01:~/verteilte-systeme-DHBW/ServerNode/ServerNode$

ubuntu@node02:~/verteilte-systeme-DHBW/ServerNode/ServerNode$ dotnet run
[Status] Node 193.196.53.217 online
[Error] Could not connect to: 193.196.54.176
[Status] Client 193.196.55.38:53286 connected.
[Error] Connection to 193.196.55.166 lost
[Status] Client 193.196.55.38:37076 connected.

ubuntu@node03:~/verteilte-systeme-DHBW/ServerNode/ServerNode$ dotnet run
[Status] Node 193.196.54.176 online
[Status] Client 193.196.55.38:47532 connected.
[Error] Connection to 193.196.55.166 lost
[Error] Connection to 193.196.55.166 lost

```


4. Fällt auch dieser Knoten aus kommt es zum Totalausfall
Der Client versucht nun sich weiter zu verbinden.
Die Autos bleiben „stehen“

The image displays four terminal windows from the Cloud Computing IDE, arranged in a 2x2 grid. Each window shows the output of a .NET application running on different nodes.

- Top-Left Window:** Shows the output of `Client/Client$ dotnet run`. It contains multiple "Can't connect to" errors for various IP addresses (e.g., 193.196.55.166, 193.196.53.217) and a final "lost" status.
- Top-Right Window:** Shows the output of `ServerNode/ServerNode$ dotnet run`. It includes a "[Status] Node 193.196.55.166 online" message, followed by "Could not connect to" errors for specific IPs, and then "[Status] Client" connections for 193.196.55.38 at ports 46756, 46770, and 46776.
- Bottom-Left Window:** Shows the output of `ServerNode/ServerNode$ dotnet run`. It features a "[Status] Node 193.196.53.217 online" message, "Could not connect to" errors, and then "[Status] Client" connections for 193.196.55.38 at ports 40728 and 59688. A yellow rectangular box highlights the line "**[Status] Client 193.196.55.38:59688 connected.**", and a yellow arrow points from this box towards the right side of the image.
- Bottom-Right Window:** Shows the output of `ServerNode/ServerNode$ dotnet run`. It includes a "[Status] Node 193.196.54.176 online" message, "Could not connect to" errors, and then "[Status] Client" connections for 193.196.55.38 at ports 40990, 41002, and 41010.

The IDE interface at the top shows tabs for "cloud_computing" and "S. node01-bw". The bottom status bar indicates the current active tab is "S. node01-bw".

5. In der Zwischenzeit ist Knoten 3 (unten rechts) wieder hochgefahren
Dieser bekommt nun alle 3 Clients und rechnet weiter.

```

[Error] Can't connect to 193.196.55.166 lost
[Error] Can't connect to 193.196.54.176 lost
[Error] Can't connect to 193.196.53.217 lost
[Error] Can't connect to 193.196.55.166 lost
[Error] Can't connect to 193.196.54.176 lost
[Error] Can't connect to 193.196.53.217 lost
[Error] Can't connect to 193.196.53.217 lost
[Error] Can't connect to 193.196.54.176 lost
[Error] Can't connect to 193.196.55.166 lost
[Error] Can't connect to 193.196.53.217 lost
[Error] Can't connect to 193.196.54.176 lost
[Error] Can't connect to 193.196.55.166 lost
[Error] Can't connect to 193.196.53.217 lost

ubuntu@node01:~/verteilte-systeme-DHBW/ServerNode/ServerNode$ dotnet run

[Status] Node 193.196.55.166 online
[Error] Could not connect to: 193.196.54.176
[Error] Could not connect to: 193.196.53.217
[Status] Client 193.196.55.38:46756 connected.
[Status] Client 193.196.55.38:46770 connected.
[Status] Client 193.196.55.38:46776 connected.
^Cubuntu@node01:~/verteilte-systeme-DHBW/ServerNode/ServerNode$
  
```

6. Inzwischen ist auch Knoten 2 wieder gestartet und betriebsbereit. Er hat jedoch noch keine Aufgaben/Berechnungen, da keine neuen Clients Anfragen gestellt haben.

```

[Error] Can't connect to 193.196.53.217 lost
[Error] Can't connect to 193.196.54.176 lost
[Error] Can't connect to 193.196.55.166 lost
[Error] Can't connect to 193.196.53.217 lost
[Error] Can't connect to 193.196.53.217 lost
[Error] Can't connect to 193.196.54.176 lost
[Error] Can't connect to 193.196.55.166 lost
[Error] Can't connect to 193.196.53.217 lost

[Status] MaxRequestPerNode reached, reconnecting 1
[Error] Can't connect to 193.196.55.166 lost
[Status] MaxRequestPerNode reached, reconnecting 2
[Error] Can't connect to 193.196.55.166 lost

ubuntu@node01:~/verteilte-systeme-DHBW/ServerNode/ServerNode$ dotnet run

[Status] Node 193.196.55.166 online
[Error] Could not connect to: 193.196.54.176
[Error] Could not connect to: 193.196.53.217
[Status] Client 193.196.55.38:46756 connected.
[Status] Client 193.196.55.38:46770 connected.
[Status] Client 193.196.55.38:46776 connected.
^Cubuntu@node01:~/verteilte-systeme-DHBW/ServerNode/ServerNode$
  
```

7. In der Zwischenzeit haben einige Clients auf Knoten 3 mehr als die erlaubte Anzahl von Berechnungen (Feldbewegungen) durchgeführt.
8. Der **erweiterte Lastverteilungsalgorithmus** tritt in Kraft und trennt diese Clients.
9. Diese getrennten Clients werden nun neu ausbalanciert.

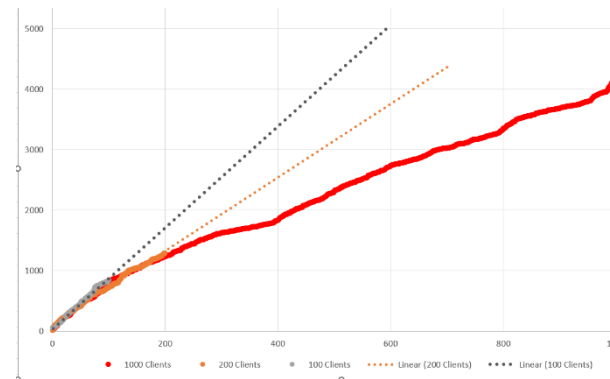
Nachbetrachtung

Das von uns erstellte verteilte System funktioniert und erfüllt alle definierten Anforderungen. Es ist ausfallsicher skalierbar und bewältigt (getestet) bis zu 10.000 Client-Anfragen.

Dass 100 Clients ca. 14 Minuten benötigen und 10 ca. 2 Minuten, halten wir für akzeptabel, sollte dies jedoch zu langsam sein, könnte durch Anpassungen im Quellcode an der Performanzschraube gedreht werden. Derzeit ist der große Performancebremsen die erweiterte Lastverteilung, die die Clients (bei unseren Tests) nach ca. 300 Feldbewegungen trennt. Dieser Wert kann entsprechend angepasst werden, um eine mögliche Performancesteigerung zu erreichen. Es besteht aber auch die Gefahr, dass bei "falscher" Einstellung und Ausfall eines Knotens die Berechnungen sehr viel länger dauern können. (Dies ist auf Seite 10 beschrieben).

Ein weiterer großer Nachteil, der sich ebenfalls auf die Leistung auswirken kann, ist die Tatsache, dass Fahrzeuge, die ihr Ziel erreicht haben, nicht aus dem Feld entfernt werden. Dies kann dazu führen, dass sie für andere Fahrzeuge im Feld im Weg stehen, die Berechnung länger dauert und der Weg, den die Fahrzeuge dann zurücklegen müssen, länger wird. Es wird daher empfohlen, bei mehreren Tests oder generell bei jedem Durchlauf das Verteilsystem neu zu starten. Dieser Nachteil ist nicht mit einem hohen Programmieraufwand verbunden und könnte in der Nachbearbeit implementiert werden, was aber aus Zeitgründen von uns nicht umgesetzt wurde.

Generell ist zu sagen, dass sich das System in dieser Konstellation selbst ausbalanciert. Wenn man sich die Zeiten für 100, 200 und 1000 Clients ansieht, erkennt man, dass bei allen Tests die Berechnungen anfangs recht lange dauern. (Zu sehen an den orange und grau eingezeichneten Trendlinien für die Werte 100 und 200) Je länger das System jedoch läuft und je mehr Anfragen gesendet werden, desto flacher wird die Kurve (zu sehen an der roten Linie für 1000 Clients).



Laufzeitkomplexität:

Obwohl die Kurve mit zunehmender Anzahl von Durchläufen (Clients) flacher wird, könnte man auf den ersten Blick meinen, die Laufzeitkomplexität des Systems sei $O(\log n)$. Es ist jedoch wichtig zu beachten, dass die Verteilung der Anfragen gleichmäßig mit einer einfachen Round Robin Lastverteilung erfolgt, was bedeutet, dass die Ausführungszeit linear mit der Anzahl der Anfragen ansteigt. Daher kann mit großer Sicherheit angenommen werden, dass die Laufzeitkomplexität des Systems $O(n)$ ist.

Zusammenfassend kann man sagen, dass das Projekt, die Umsetzung im Team und die im Unterricht erlernten Methoden uns geholfen haben, das Projekt gut abzuschließen. An dieser Stelle möchten wir uns auch für die ausführliche Dokumentation entschuldigen, die seitentechnisch aus dem Ruder gelaufen ist.