Ypsiliform

**Distributed Supply Chain Planning mit Agenten**



Projektarbeit / Programmentwurf

im Studienfach Intelligente Agenten und Multiagentensysteme

am Center for Advanced Studies (CAS) Heilbronn

Wintersemester 16/17

Betreuender Dozent:

Prof. Dr. Jörg Homberger

Verfasser:

Paul Jähne

Christoph Lindemann

Michael Müller

Tim Schöndorfer

**Inhaltsverzeichnis**

[1 Themenstellung 1](#_Toc477424491)

[2 Strukturierung und Vorarbeit 2](#_Toc477424492)

[2.1 Entwicklungsablauf 2](#_Toc477424493)

[2.2 Architektur 2](#_Toc477424494)

[2.3 Beschreibung der Konfigurationsdateien 2](#_Toc477424495)

[3 Konzepte und Realisierungen 4](#_Toc477424496)

[3.1 Verhandlungsmechanismus 4](#_Toc477424497)

[3.2 Mediator 5](#_Toc477424498)

[3.2.1 Vorschlagsformat 5](#_Toc477424499)

[3.2.2 Vorschlagsgenerierung 6](#_Toc477424500)

[3.2.3 Vorschlagsauswertung 7](#_Toc477424501)

[3.3 Agenten 8](#_Toc477424502)

[3.3.1 Einhaltung der Kapazitätsgrenzen unter Berücksichtigung der Lieferpflicht 8](#_Toc477424503)

[3.3.2 Repräsentation des Lagers 8](#_Toc477424504)

[3.3.3 register 9](#_Toc477424505)

[3.3.4 Agent Response 9](#_Toc477424506)

[3.4 WebSockets im Kontext des Supply-Chain-Planning 9](#_Toc477424507)

[4 Auswertung 11](#_Toc477424508)

[5 Aufgabenverteilung 12](#_Toc477424509)

**Abbildungsverzeichnis**

[Abbildung 1: Baumstrukturen, eigene Darstellung 1](#_Toc477424524)

[Abbildung 2: Sequenzdiagramm: Verhandlung, eigene Darstellung. 4](#_Toc477424525)

[Abbildung 3: Bit-Flips-Berechnung, eigene Darstellung 6](file:///C:\Users\pjaehne\Documents\GitHub\docs\Ypsiliform.docx#_Toc477424526)

[Abbildung 4: WebSockets mit Agenten und Mediator, eigene Darstellung 10](#_Toc477424527)

# Themenstellung

Die klassische Supply-Chain-Planning Problemstellung soll im Kontext der Vorlesung „Intelligente Agenten und Multiagentensysteme“ mit mehreren Agenten, welche die einzelnen Glieder der Supply Chain darstellen, gelöst werden. Der grundlegende Aufbau der Supply-Chain ist der, dass es immer genau einen Produzenten gibt, der Teile von 4 anderen Zulieferern bekommt. Die Beziehungen der Zulieferer untereinander können unterschiedlich sein, um den Rahmen dieser Ausarbeitung nicht zu sprengen, hat man sich jedoch auf die Betrachtung der folgenden vier Baumstrukturen festgelegt.

![](data:application/pdf;base64,)

Abbildung 1: Baumstrukturen, eigene Darstellung

Für die Agenten gelten weiterhin die folgenden Restriktionen:

1. Jedes Glied der Kette (Knoten) ist ein **eigenständiger Agent**. Es wird also nicht zwischen 2 Ebenenen (Produzent & Zulieferer) unterschieden, sondern es gibt immer Verhandlungen zwischen den miteinander agierenden Agenten.
2. Jeder Agent hat eine **Kapazitätsgrenze**, kann also nicht bei Bedarf überproduzieren.
3. Jeder **Bedarf muss just-in-time voll gedeckt sein**. Das bedeutet, dass spätestens zum Zeitpunkt X die geforderte Menge geliefert werden muss und nicht erst bei Nachbetrachtung die Stückzahlen übereinstimmen müssen. Es darf also weder zu Nichtlieferung noch zu Verspätungen kommen.

Die Verhandlungen werden mit Hilfe eines Mediators durchgeführt, der Vorschläge zu Produktionszeitpunkten macht – nicht jedoch zu den Mengen, diese sind von den Agenten selbst zu bestimmen.

Ziel der Arbeit war es nicht den bestmöglichen Algorithmus für die Problemstellung zu finden, sondern zu demonstrieren, dass es dieses Konzept überhaupt ermöglicht, mit Kapazitätsgrenzen zu arbeiten. Weiterhin soll die Implementierung verteilt erfolgen und zeigen, dass dies eine gute Alternative zur rein lokalen Entwicklung ist.

# Strukturierung und Vorarbeit

## Entwicklungsablauf

Als Programmiersprache wurde Java 8 ausgewählt, um möglichst Plattformunabhängig zu sein. Das Projekt selbst wird über Github verwaltet (<https://github.com/Ypsiliform>) und kann mit gängigen IDEs wie Ecplise oder IntelliJ einfach verwaltet und entwickelt werden. Als Buildumgebung wurde Maven benutzt, wodurch auch Tests einfach integriert wurden. Wildfly wurde als Anwedungsserver für den Mediator ausgewählt, über den die Agenten und der Mediator Nachrichten im JSON-Format austauschen.

## Architektur

Als Architektur wurde ein verteiltes System gewählt, mit dem Mediator als zentralem Webservice und den Agenten als Clients, die sich mit dem Mediator bidirektional verbinden. Durch diese Struktur soll es zu einem späteren Zeitpunkt möglich sein, den Mediator zentral zu veröffentlichen, sodass er definiert auf Anfragen reagiert. Hierdurch können Agenten entwickelt und getestet werden, ohne dass man eine lokale Webserver-Instanz auf seinem Rechner benötigt. Die Verbindung zwischen Client und Server wurde mit Hilfe von WebSockets realisiert, über die Nachrichten im JSON-Format ausgetauscht werden. Eine genauere Beschreibung und die Begründung dieser Wahl findet sich in Kapitel *3.1 - WebSockets im Kontext des Supply-Chain-Planning*.

Aufgrund des zentralen Webservices wurden zur Vergleichbarkeit alle Log-Dateien zur Auswertung ausschließlich auf den Agenten gesammelt, da man zu einem späteren Zeitpunkt evtl. nicht mehr auf das Dateisystem des Webservers zugreifen kann.

## Beschreibung der Konfigurationsdateien

Um die Kapazitätsgrenzen festzulegen, wurden die bestehenden Konfigurationen angepasst. Dabei wurden pro bestehender Konfiguration 4 neue Konfigurationsdateien erstellt, die unterschiedliche Aspekte der Kapazitätsgrenzen aufzeigen sollen:

1. **Flaschenhals**  
   Beim Flaschenhals geht es darum zu beurteilen, wie sich ein Agent auswirkt, der nur sehr geringe Kapazitäten hat im Vergleich zu den anderen. Die Zahlen wurden so generiert, dass 4 Agenten jeden Bedarf an diesem Tag decken könnten, der mittlere Agent jedoch immer an seine Kapazitätsgrenze stößt. Als Flaschenhals wurde gezielt der mittlere Agent ausgewählt, damit die ersten beiden Agenten u.U. davon nicht betroffen sind, jedoch auch nicht nur der allerletzte Agent die Auswirkungen zu spüren bekommt.  
    *1 400*

*2 400*

*3 30*

*4 400*

*5 400*

1. **Hohe Kapazität**  
   In diesem Versuch sollten alle Agenten den anfallenden Bedarf immer sofort decken können, falls jeden Tag produziert würde.

*1 500*

*2 500*

*3 500*

*4 500*

*5 500*

1. **Niedrige Kapazitäten**  
   Alle Agenten werden ihre Kapazitätsgrenzen ausreizen.

*1 50*

*2 50*

*3 50*

*4 50*

*5 50*

1. **Zufällig verteilt**

Hier ging es darum, eine willkürlich zufällige Verteilung zu betrachten, die nicht aufgrund einer möglichen Problemstellung zu betrachten wäre. Stattdessen soll hiermit überprüft werden, ob andere, im Vorfeld nicht bedachte Probleme zum tragen kommen. Die zufällige Stichprobe unterscheidet sich für jede Konfiguration, damit möglichst viele unterschiedliche zufällig generierte Kapazitätsgrenzen betrachtet werden können.

1 83

2 99

3 117

4 148

5 139

Eine solche Konfiguration besteht nun aus den folgenden Blöcken:

* Anzahl der Agenten, Anzahl der geforderten Mengen, Anzahl Produzenten
  + In diesen Beispielen immer 5 12 1
* Lagerkosten (pro Teil) pro Agent
* Setupkosten (pro Periode) pro Agent
* Geforderte Mengen des Produzenten
* Baumstruktur
* Kapazitätsgrenze pro Agent

Die erstellten Konfigurationen werden mit abgegeben und dürfen auch für zukünftige Gruppen verwendet werden.

|  |
| --- |
|  |

# Konzepte und Realisierungen

## Verhandlungsmechanismus

Nachfolgende Abbildung stellt den Verhandlungsmechanismus zwischen Agenten und Mediator dar:

![](data:application/pdf;base64,)

Abbildung 2: Sequenzdiagramm: Verhandlung, eigene Darstellung.

Wird ein Agent gestartet, so registriert er sich beim Mediator (*register*) und teilt ihm mit, mit welchen Agenten er verhandeln will. Sobald sich alle Agenten beim Mediator registriert haben, startet der Mediator von sich aus die Verhandlung (*start Negotiation*) indem er eine initiale Lösung generiert (*init Solution*) und diese an den ersten Agenten sendet (*Mediator Request*). Anschließend wartet er auf die Antwort (*Agent Response*) dieses Agenten, bevor er Agent 2 die nun berechneten Bedarfe als neuen Mediator Request übermittelt. Hat ein Agent nur einen Nachfolger wird die Verhandlung wie zuvor beschrieben ablaufen. Hat ein Agent zwei oder mehr Nachfolger, wie die Bäume 2-4 der Abbildung 1 auf Seite 1 zeigen, muss der Mediator nicht erst auf die Antwort eines gestellten Mediator Request warten, sondern kann parallel weitere Mediator Request Nachrichten an die beteiligten Agenten der Verhandlungsrunde (gleiche Ebene) senden.

Hat der Mediator alle Antworten empfangen, kalkuliert er die favorisierte Lösung (*det. Solution*) und leitet anhand dieser neue Lösungsvorschläge ab (*gen. Neighbour*). Mit diesen neuen Lösungsvorschlägen beginnt er wieder mit Agent 1 eine neue Verhandlungsrunde.

Insgesamt werden 1000 Verhandlungsrunden durchgeführt, an deren Ende der Mediator den Agenten ihre finale Lösung mitteilt (*end Negotiation*).

## Mediator

### Vorschlagsformat

Zur Generierung der Vorschläge wurden zunächst diese beiden bekannten Verfahren betrachtet:

1. **Verhandlung nach Finke**

Zum Start der Verhandlung müssen beide Agenten ein T berechnen, das zur Sicherstellung einer bestimmten Prozentzahl von angenommenen Vorschlägen dient. Es wird berechnet, indem 100 Lösungen erzeugt und die durchschnittlichen Abweichungen berechnet werden. Das T bleibt auf dem Agenten und wird während der ganzen Verhandlung benutzt – Akzeptanzwahrscheinlichkeit eines Vorschlages:

* + Vorteile:
    - Verhandlung sehr einfach, Antwort ist nur ja / nein
  + Nachteile:
    - Code Duplizierung Mediator / Agent zur Erzeugung der initialen Lösungen

1. **Verhandlung nach Homberger**

Jeder Agent bekommt einen Satz an Vorschlägen und muss die besten auswählen bzw. nach Prioritäten sortieren. Der Mediator entscheidet anschließend anhand der Übereinstimmungen, welcher Satz als nächstes verwendet wird

* + Vorteil:
    - Keine Code-Duplizierung (da keine Init-Phase notwendig)
  + Nachteil:
    - Mögliche Beeinflussung bei lernenden Agenten
    - Pattsituationen möglich?

Das letztendlich herausgekommene Protokoll ist in Anlehnung an das Protokoll Homberger entstanden, unterscheidet sich jedoch in einigen Punkten:

1. Es werden genau **4 Lösungen** (Anzahl Agenten -1) vorgeschlagen

Somit wird garantiert, dass es nur 1 oder 2 favorisierte Lösungen gibt, niemals mehr. Würde man mehr als 4 Lösungen zur Verfügung stellen, so könnte dazu führen, dass es 5 bevorzugte Lösungen gibt und eine faires fortfahren nicht gewährleistet werden kann. 3 Lösungen würden dazu führen, dass man eine Pattsituation nicht sauber auflösen kann (2 vs 2 vs 1 Stimme). 2 Lösungen würden zwar ein eindeutiges Ergebnis liefern, die Anzahl an Vorschlägen ist jedoch zu gering.

1. Jeder **Agent** **wählt dann genau eine** **Lösung** aus

Hierdurch wird die Beeinflussung von lernenden Agenten minimiert. Kann er nur eine Lösung auswählen, so kann er Agenten nur in die Irre führen, indem er sich selbst benachteiligt.

1. **Pattauflösung** durch Verwendung aller Favoriten zur Erstellung neuer Vorschläge

Es werden in jedem Fall (1 oder 2) alle favorisierten Lösungen zur Erstellung eines neuen Lösungssets verwendet. Damit ist der Mediator niemals gezwungen in Pattsituationen zu entscheiden, welche Lösung nun präferiert wird.

1. **Vorherige Lösung bleibt Teil des neuen Lösungssets**

Da ein Agent immer eine Lösung annehmen muss, bleibt der zuvor gewählte Favorit Teil des neuen Lösungssets. Somit wird vermieden, dass sich ein Agent vor allem zu Ende der Verhandlungen verschlechtern muss, obwohl er das eigentlich nicht will.

### Vorschlagsgenerierung

Generell besteht eine Lösung immer aus einem 60-Zeichen langen Bitstrom aus 1(true) und 0(false), wobei 1 für Produktion erlaubt und 0 für Produktion nicht erlaubt steht. Die 60 Zeichen ergeben sich daraus, dass bei 5 Agenten pro Agent 12 Perioden betrachten werden müssen.

Um von einer Lösung zu einer möglicherweise besseren Lösung für einen Agenten zu gelangen werden einzelne Bits der Lösungsmenge per Bit-Flip verändert. Dies bedeutet, dass bei fünf Agenten zur gewählten Lösung drei oder bei Pattsituationen zwei weitere Lösungen generiert und den Agenten zur Abstimmung präsentiert werden. Die zuvor gewählte(n) Lösung(en) wird in das neue Lösungsset übernommen.

Generell berechnet der Mediator als Lösung immer einen 60-Zeichen langen Bitstrom aus 1 oder 0, der dann in 5x12 lange Vorschläge für die Agenten geteilt wird. Damit sich zu Beginn der Verhandlung noch deutliche Änderungen in den Vorschlägen ergeben, berechnet der Mediator abhängig von der Anzahl der bereits erfolgten Verhandlungsrunden wie stark die Veränderungen ausfallen. Eine Veränderung ist das Flippen eines Bits. Ergibt die Funktion z.B. 25, so wird von 60 Bits nicht ganz die Hälfte verändert.

Die Formel lautet wie folgt:



|  |  |
| --- | --- |
| Bez. | Bedeutung |
| B | Anzahl an Bits, die zu Beginn maximal verändert werden sollen |
| x | Anzahl der bereits erfolgten Verhandlungsrunden |
| k | Faktor für die Krümmung der Funktion. Sie gibt an, wie schnell die Zahl der veränderten Bits sinkt. |
| R | Anzahl der Verhandlungsrunden (insgesamt) |

Abbildung 3: Bit-Flips-Berechnung, eigene Darstellung

Das letztendlich erzeugte Lösungsset wird anschließend per Mediator-Request an die Agenten verschickt. Herbei ist auch zu beachten, dass obwohl eine Produktion an einem Tag nicht erlaubt ist, sehr wohl ein Bedarf anliegen kann und umgekehrt.

{

"type" : "mediatorrequest",

"solutions" : [

{"no":1, "solution":[true,false,true,false,true,false,true,false,true,false,true,false],

"demands":[10,20,80,0,50,100,70,80,150,0,40,40]},

{"no":2, "solution":[false,false,true,false,true,false,true,false,true,false,true,false],

"demands":[100,20,30,80,0,0,70,30,90,100,0,120]},

{"no":3, "solution":[true,true,true,false,true,false,true,false,true,false,true,false],

"demands":[0,120,70,0,200,0,0,80,0,100,90,0]},

{"no":4, "solution":[true,false,false,false,true,false,true,false,true,false,true,false],

"demands":[50,50,50,50,50,50,60,60,60,40,60,60]}

]

}

### Vorschlagsauswertung

Obwohl es nicht das primäre Ziel der Arbeit war die Agenten so zu optimieren, dass sie die bestmögliche Gesamtlösung finden, wäre es dennoch interessant ihre Ergebnisse in diesem Kontext einordnen zu können. Deshalb beherrscht der Mediator neben dem bereits oben vorgestellten Mechanismus zur Lösungsfindung noch eine weiter Möglichkeit: Gesamtkostenbasiert.

Wie bisher werden auch Lösungsvorschläge zu den Agenten geschickt, die diese normal auswerten, bei der Lösungsfindung ignoriert der Mediator jedoch die Abgegeben Stimmen der Agenten. Stattdessen addiert er die von jedem Agenten mitgeteilten Kosten und wählt als einzigen Favoriten die Lösung, die die geringste Gesamtkostenzahl hat. Die Anschließende Erzeugung des neuen Lösungssets geht dann wieder wie gehabt vonstatten.

Die Agenten erhalten wie in der normalen (stimmbasierten) Auswertung keine Kenntnis der anderen Agenten oder des Mechanismus, wie der Mediator Lösungen auswählt. Für sie ändert sich also gar nichts.

Wenn alle Verhandlungen beendet sind und die optimale Lösung für alle Agenten gefunden ist, wird mit einer End-Negotiation Nachricht jedem Agenten mitgeteilt, welche Lösung für ihn als Optimum festgelegt wurde. Dies sieht in JSON wie folgt aus:

{

"type" : "endnegotiation",

"solution" : [true,false,true,false,true,false,true,false,true,false,true,false]

"demands" : [10,200,30,40,50,60,70,80,90,100,110,120]

}

## Agenten

### Einhaltung der Kapazitätsgrenzen unter Berücksichtigung der Lieferpflicht

Die Vorgabe der Problemstellung besagt, dass ein Agent in einer Periode nicht mehr produzieren darf, als ihm als Kapazitätsgrenze vorgegeben wurde. Zur Umsetzung dieser Grenze wurden verschiedene Möglichkeiten betrachtet:

1. **Lieferausfall (nicht zulässige Lösung)**

Die plausibelste Lösung für das Problem besteht darin, eine Lösung als nicht zulässig zu deklarieren und das Problem somit weitestgehend zu umgehen. Die Problemlösung wird nun dem Mediator zugeschoben, der nicht nur die Stimmabgabe der Agenten auswerten muss, sondern die Lösungen zusätzlich noch auf Zulässigkeit überprüfen muss. Hier stellen sich jedoch 2 Probleme:

* Die Kapazitätsgrenzen wurden willkürlich festgelegt, es gab also keine Überprüfung ob ein Problem überhaupt lösbar ist. Dadurch erzeugt man auf dem Mediator eine unnötige Komplexität, die seine eigentliche Funktion der Lösungserzeugung in den Hintergrund rückt
* Eventuell zulässige Lösungen könnten sich sehr stark unterscheiden und als nähere Nachbarn nur unzulässige Lösungen haben. Somit würde man das Problem von „finde die möglichst beste Lösung“ zu „finde überhaupt irgendeine Lösung“ verschieben. Dadurch würde auch das Erzeugen der möglichen Lösungen deutlich verkompliziert werden.

Aufgrund dieser beiden Punkte wurde dieser Lösungsansatz verworfen.

1. **Einkauf von zusätzlichem Zulieferer**

Wenn man den persönlichen Bedarf nicht decken kann, so könnte man zusätzlich benötigtes Material von einem externen imaginären Zulieferer erwerben. Das größte Problem bei diesem Ansatz: wie berechnet man die Kosten hierfür?

* Werden die Kosten für den Zulieferer mit fixen Kosten pro Teil berechnet, so müssen diese so hoch sein, dass ein Agent nur als letzte Möglichkeit darauf zurückgreift. Darunter würde jedoch die Vergleichbarkeit leiden, da ein Großteil der entstehenden Kosten unter Umständen von den imaginären Kosten stammt und die eigentlich in den Konfigurationen festgelegten Kosten nur einen Bruchteil der Gesamtkosten ausmachen würden.
* Die Kosten für den Zulieferer können mit einem zusätzlichen Faktor versehen werden, der ihn teurer macht. Neben der Frage der Höhe des Faktors stellt jedoch auch die Frage der Ausgangsbasis. Werden die eigenen Kosten als Ausgangsbasis benutzt, so könnten die entstehenden Kosten dennoch geringer sein als die Kosten beim eigentlichen Zulieferer. Dieser würde somit uninteressant und der Agent würde einfach alle Teile beim imaginären Zulieferer beziehen. Würden die Kosten des eigentlichen Zulieferers benutzt, so hätte der Agent unerlaubte Informationen, die ihm einen großen Vorteil verschaffen würden.

Deshalb wurde auch dieser Lösungsansatz verworfen.

1. **Anlegen eines Lagers**

Kann ein Agent nicht liefern, so muss er im Voraus produzieren. Lagerkosten ganz allgemein sind ja bereits in der Konfiguration für jeden Agenten definiert. Diese Idee wird nun weiter verwendet und ein zusätzliches Lager vor der ersten Periode angelegt. Alle Bedarfe, die der Agent in allen Perioden nicht decken kann, muss er eben zuvor produziert haben und aus dem Lagerbedarf decken.

* Die Kostenberechnung kann in diesem Fall mit den bereits hinterlegten Konstanten erledigt werden, es sind keine imaginären Kosten oder Faktoren notwendig. Unter der Annahme, dass das Lager mit jede Periode im Voraus mit maximaler Kapazität gefüllt werden kann, können sogar Setupkosten mit einbezogen werden.
  + Beispielrechnung:
    - Kapazitätsgrenze 50
    - Lagerbedarf 170
    - Setupkosten 20
    - Lagerkosten 1
  + Der Agent muss 4 Perioden im Voraus produzieren, um sein Lager zu füllen. Am vorteilhaftesten ist es, in der ersten Periode nur 20 Teile zu produzieren und die folgenden 3 Perioden voll auszureizen. Dadurch entstehen Lagerkosten von
* Da beim Füllen des Lagers Setupkosten einberechnet werden und die gelagerten Teile auch in allen anderen Perioden (bis sie verbraucht sind) einberechnet werden, ist eine grundlose Vorproduktion nicht sinnvoll.
* Das Problem beim Anlegen des Lagers ist die Repräsentation und die Mitteilung an den Mediator. Dieser erwartet immer eine Antwort mit der in der Konfiguration festgelegten Anzahl der Perioden. Die Lösung dieses Problems wird im nächsten Abschnitt beschrieben.

Nach Abwägen der Vor- und Nachteile der jeweiligen Lösungen wurde als Lösungsansatz das Anlegen eines Lagers ausgewählt.

### Repräsentation des Lagers

Wie bereits angemerkt wurde, kann das Lager nicht an den Mediator übertragen werden. Wäre das der Fall, so würde das Lager des ersten Agenten als normaler Periodenbedarf für den zweiten Agenten angesehen. Das würde bedeuten, dass der Mediator nun einen 13-Bit langen Vorschlag schicken müsste und evtl. eine 14 Perioden lange Lösung zurückbekommen würde. Dieser Vorgang würde sich mit jedem weiteren Agenten immer weiter fortsetzen.

Selbst wenn der Mediator immer nur die letzten 12-Bit eines Vorschlages neu erzeugt und des Rest mit 1 auffüllt, würde die Datenhaltung deutlich komplizierter. Deshalb wurden 2 Lösungsansätze auf Seiten des Agenten betrachtet, mit denen immer nur die zuvor festgelegte Periodenanzahl an Bedarf übertragen wird.

1. **Das Lager wird nicht mit übertragen**. Das bedeutet, dass die nachfolgenden Agenten auch weniger zu produzieren hätten. Im schlimmsten Fall würde das bedeuten, dass der erste Agent alles aus dem Lager bezieht und die anderen Agenten überhaupt nichts mehr produzieren müssten. Um das zu verhindern, müssten die Lagerkosten nachträglich doch noch mit einem Faktor versehen werden, der es sehr teuer werden lässt, wenn aus dem Lager bezogen wird. Hierbei hätte man aber wieder das Problem, dass man den optimalen Faktor erst wieder ermitteln muss und die Kosten nicht mehr vergleichbar wären. Ein viel größerer Nachteil wäre es jedoch, in der Produktionskette weit oben zu stehen, da die Agenten weiter unten unter Umständen viel geringere Produktzahlen und somit weniger Kosten auflegen müssen. Sie hätten somit einen Wettbewerbsvorteil.
2. **Alle Produkte aus dem Lager werden auf die erste Periode angerechnet**. Dadurch bleibt der Gesamtbedarf über alle Perioden gleich und der Wettbewerbsvorteil der nachfolgenden Agenten wird minimiert. Hat der nächste Agent die gleichen oder sogar geringere Kapazitätsgrenzen, so muss auch er vorher produzieren und die Kosten werden 1:1 übernommen. Hat er jedoch eine höhere Kapazitätsgrenze, so kann er einen größeren Anteil über die normale Produktion abdecken und hat dadurch unter Umständen kaum zusätzliche Kosten.

Lösungsansatz 2 bietet also zumindest die Chance die Produktionskosten teilweise oder ganz weiter zu reichen, während Lösungsansatz 1 immer zu einem Wettbewerbsnachteil führt, falls das Lager benötigt wird.

### Operationen

**Registrierung**

Direkt nach dem Starten meldet sich der Agent beim Mediator an und teilt diesem seine ID, den Namen seiner Konfiguration (wird zur Sessionverwaltung benötigt), die Agenten mit denen er verhandeln möchte und gegebenenfalls seinen Bedarf an (nur Agent 1).

{

"type" : "agentregistration",

"id" : 1,

"config" : "Bsp1",

"requires" : [2,3],

"demand" : [80,50,30,0,25,60,120,90,100,75,60,40]

}

**Agent Response**

Hat der Mediator die Verhandlung gestartet und einem Agenten eine Mediator-Request Nachricht gesendet, so berechnet der Agent für die 4 Vorschläge den benötigten Bedarf sowie die daraus entstehenden Kosten. Diese Agent Response sieht in JSON wie folgt aus:

{

"type" : "agentresponse",

"selection" : 2,

"demands" : [

{"solution":1, "demand":[160,0,0,80,80,80,0,0,0,80,80,80]},

{"solution":2, "demand":[80,80,0,80,80,80,0,0,0,80,80,80]},

{"solution":3, "demand":[320,0,0,0,0,80,0,0,0,80,80,80]},

{"solution":4, "demand":[240,0,0,0,80,80,0,0,0,80,80,80]}

],

"costs": [138.5, 75.0, 249.925, 217.375]

}

In dieser Nachricht ist nun zu sehen, dass der Vorschlag 2 von diesem Agenten bevorzugt wird, da dieser die geringsten Kosten von 75 verursacht. Außerdem ist zu sehen, dass die Kapazitätsgrenze des Agenten bei 80 liegt und er keine Produkte aus dem Lager holen muss, um die Anfrage zu erfüllen. Vorschlag 3 ist stattdessen das genaue Gegenteil, hier sieht man an der ersten Periode, dass der Agent sehr viele Produkte aus dem Lager beziehen muss. Dadurch erklären sich auch die deutlich höheren Kosten.

## WebSockets im Kontext des Supply-Chain-Planning

Anhand der Abbildung 1 in Kapitel 2 ist ersichtlich, dass der Mediator eine zentrale Rolle spielt. Diese Rolle würde unter HTTP implementiert dazu führen, dass nach einer Verhandlungsteilrunde, nämlich bis zur ersten Antwort des Mediators (Server), die HTTP-Verbindung beendet wird und ein erneuter Verbindungsaufbau durch den Agenten (Client) notwendig wird. Dieser kann aber nur abschätzen wann eine erneute Verbindung notwendig ist und somit müsste der Agent nach einer festgelegten Wartezeit einen Verbindungsversuch wagen. Das bedeutet dann wiederum, dass der gesamte HTTP-Handshake abläuft ohne Garantie für eine neue Verhandlungsrunde. Der Wunsch eine dauerhafte Verbindung zwischen Mediator und Agenten zu haben, mit der alle Parteien unabhängig von einander und gleichzeitig Daten Senden können, kann somit per HTTP nicht erfüllt werden.

An dieser Stelle kommt die WebSocket-Spezifikation mit einer eleganten Lösung genau richtig. Es werden „socket“-Verbindungen zwischen Server und den Clients hergestellt. Da diese Verbindung persistent sowohl die Agenten (Clients) als auch den Mediator (Server) dazu befähigen eigenständig mit dem Senden von Daten zu beginnen, fällt der sonst notwendige und ständige Handshake des HTTP zwischen den Beteiligten weg. Die dabei anfallenden Latenzzeiten können eliminiert werden.

Der Mediator ist durch die WebSocket-Spezifikation in der Lage Verhandlungen mit den übrigen Agenten abzuwarten bevor er mit einem wartenden Agenten wieder in Verhandlungen tritt. Dadurch, dass zwischen den Agenten und dem Mediator permanent die Möglichkeit des Datenaustausches besteht muss der Agent keine willkürlichen Verbindungsversuche unternehmen um mit dem Mediator wieder in Verhandlungen zu treten. Ebenfalls muss der Mediator nicht darauf warten, dass sich ein Agent wieder mit ihm in Verbindung setzt. Die nachfolgende Abbildung zeigt die gesamte Verbindung und den vereinfachten Ablauf der Verhandlungen zwischen Agenten und Mediator. Dies stellt auch WebSockets im Kontext der Supply-Chain-Planning Problemstellung dar.

![](data:application/pdf;base64,)

Abbildung 4: WebSockets mit Agenten und Mediator, eigene Darstellung

# Ausführung

## Mediator Webservice

Für den Mediator benötigt man:

* Wildfly-Server (<http://www.wildfly.org/>)
* mediator.jar Archiv (aus Ypsiliform/code/mediator/target/)

Das Download-Archiv des Wildfly Servers muss entpackt werden. Als Unterverzeichnis gibt es den Ordner *WildFly-XX/standalone/deployments/*. In diesen Ordner muss die mediator.jar Datei kopiert werden. Sie wird beim starten des Servers automatisch deployed.

Zum Starten wechselt man in das Verzeichnis *WildFly-XX/bin/* und führt dort, je nach Betriebssystem, die standalone.bat oder standalone.sh Datei aus. Zu Überprüfung kann im Browser localhost:8080 aufgerufen werden und man bekommt einen generischen Willkommenstext, dass der Server gestartet ist.

## Agenten

Für die Agenten benötigt man:

* Java 8 JRE
* agent.jar (aus Ypsiliform/code/agent/target/agent-jar-with-dependencies.jar)
* starter-jar-with-dependencies.jar (aus Ypsiliform/code/agent/target/)

Das agent.jar Archiv muss in denselben Ordner wie die starter-jar-with-dependencies.jar kopiert. Anschließend können die Agenten wie folgt aufgerufen werden:

java –jar starter-jar-with-dependencies.jar <Mediator-URL> <Testdatei> <Anzahl Tests>

Dabei haben die Optionen die folgenden Bedeutungen:

|  |  |
| --- | --- |
| Option | Bedeutung |
| Mediator-URL | URL-Adresse des Mediators. Wird dieser lokal ausgeführt, so muss diese Adresse angegeben werden: ws://localhost:8080/mediator/mediator |
| Testdatei | Name der Datei oder Ordner, der die Testdateien mit Endung .req enthält. Wird ein Ordner angegeben, so werden alle Testdateien darin nacheinander ausgeführt. |
| Anzahl Tests | Anzahl der Testdurchläufe pro Testdatei |

# Auswertung

Auswertung sollte jeweils cost-based vs. Vote-based erfolgen

Kopiere Diagramme aus Auswertung.xls

## Flaschenhals

## Hohe Kapazitäten

## Niedrige Kapazitäten

## Zufall

# Aufgabenverteilung

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Bearbeiter  Aufgabe | Jähne | Lindemann | Müller | Schöndorfer |
| Konzeption |  |  |  |  |
| Agent |  |  |  |  |
| Mediator |  |  |  |  |
| WebSocket  (Middleware) |  |  |  |  |
| Dokumentation |  |  |  |  |
| Auswertung |  |  |  |  |