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](#_Toc477211619)

[2 Architektur und Konzepte 2](#_Toc477211620)

[2.1 Entwicklungsablauf 2](#_Toc477211621)

[2.2 Architektur 2](#_Toc477211622)

[2.3 Konzepte 2](#_Toc477211623)

[2.3.1 Verhandlungsmechanismus 2](#_Toc477211624)

[2.3.2 Einhaltung der Kapazitätsgrenzen unter Berücksichtigung der Lieferpflicht 3](#_Toc477211625)

[2.3.3 Vorschlagsgenerierung 4](#_Toc477211626)

[2.3.4 Lösungsgenerierung 4](#_Toc477211627)

[3 Realisierungen 4](#_Toc477211628)

[3.1 WebSockets im Kontext des Supply-Chain-Planning 4](#_Toc477211629)

[3.2 Agenten 5](#_Toc477211630)

[3.2.1 register 5](#_Toc477211631)

[3.2.2 Agent Response 5](#_Toc477211632)

[3.3 Mediator 6](#_Toc477211633)

[3.3.1 start Negotiation 6](#_Toc477211634)

[3.3.2 init. Solutions 6](#_Toc477211635)

[3.3.3 Mediator Request 6](#_Toc477211636)

[3.3.4 det. Solutions 6](#_Toc477211637)

[3.3.5 gen. Neighbor 7](#_Toc477211638)

[3.3.6 end Negotiation 7](#_Toc477211639)

[4 Auswertung 8](#_Toc477211640)

[5 Aufgabenverteilung 8](#_Toc477211641)

**Abbildungsverzeichnis**

[Abbildung 1: Baumstrukturen, eigene Darstellung 1](#_Toc477211541)

[Abbildung 2: Sequenzdiagramm: Verhandlung, eigene Darstellung. 2](#_Toc477211542)

[Abbildung 4: WebSockets mit Agenten und Mediator, eigene Darstellung 5](#_Toc477211543)

[Abbildung 3: Bit-Flips-Berechnung, eigene Darstellung 7](file:///C:\Users\pjaehne\Documents\GitHub\docs\Ypsiliform.docx#_Toc477211544)

# 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 : 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.

# Architektur und Konzepte

## 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 Verhandlungsalgorithmus ist das Multi-Negotiation-Text-Protokoll nach Homberger, Fink und Homberger festgelegt worden. Dieses Protokoll soll mit Hilfe von WebSockets und der damit einhergehenden bidirektionalen Verbindung zwischen Client (Agent) und WebSocket-Server (Mediator) realisiert werden.

Die Gesamtkosten der Produktion sollen insgesamt sinken. Dabei soll der Mediator einmal anhand der Gesamtkosten und einmal anhand der Agentenentscheidungen bestimmen, welcher Vorschlag ausgewählt wird.

## Konzepte

### Verhandlungsmechanismus

Nachfolgende Abbildung stellt den Verhandlungsmechanismus zwischen Agenten und Mediator dar:

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

Abbildung : Sequenzdiagramm: Verhandlung, eigene Darstellung.

Eine Verhandlungsrunde startet sobald alle Agenten sich beim Mediator registriert haben. Der Mediator übermittelt die erste Lösung als Mediator Request an Agent 1 und muss nun auf die Antwort dieses Agenten warten, 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 3 zeigen, muss der Mediator nicht auf die Antwort einer Mediator Request warten, sondern kann weitere Mediator Request an die beteiligten der Verhandlungsrunde senden.

### Einhaltung der Kapazitätsgrenzen unter Berücksichtigung der Lieferpflicht

Hier Beschreibung, wie das Problem Kapazitätsgrenze gelöst wurde

Folgender Spezialfall trat bei Vorüberlegungen auf:

Der günstigste Fall nach den Gesamtkosten könnte sich daraus ergeben, dass der erste Agent den gesamten Bedarf aus dessen Lager bezieht. Der Nachfolgende Agent bekommt jedoch nur 12 Periodenbedarfe übergeben. Das initiale Lager wird nicht als Periodenbedarf durch den Mediator übergeben, da dieser nur 5 x 12 Perioden generiert und nicht weiß, dass ein Agent seinen Bedarf über das initiale Lager erfüllt. Wenn einem Agenten kein Bedarf übergeben wird, geht er auch davon aus, dass er nicht Produzieren muss. Deshalb entstehen keine Rüst- und Lagerkosten. Der Mediator würde diesen Fall, aus Sicht der Gesamtkosten, als sehr günstig bewerten.

Um dieses Problem abzufangen wurden zwei mögliche Modellerweiterungen entwickelt:

1. Das initiale Lager, der Bereich vor den betrachteten 12 Perioden, könnte mit erheblichen Kosten versehen werden um zu teuer für eine optimale Lösung zu werden. Dadurch besteht zwar die Möglichkeit, dass der zuvor beschriebene Spezialfall eintritt, dieser würde jedoch vom Mediator nicht als optimale Lösung ausgewählt, wenn der Faktor, um den das initiale Lager teurer wird, hoch genug angesetzt wurde. Es ergibt sich auch ein klarer Wettbewerbsnachteil für den Agenten, der den Bedarf aus seinem initialen Lager bedienen muss, denn die nachfolgenden Agenten haben diese Initialkosten nicht.
2. Die Aufwände des initialen Lagers des Agent 1 können aufsummiert und als Bedarf an den nachfolgenden Agenten 2 weitergegeben werden. Dieser muss den Bedarf direkt in der ersten Produktionsperiode decken. Hat der Agent 2 eine Kapazitätsbeschränkung die kleiner ist als der Bedarf, der ihm mitgegeben wird, muss dieser analog zur Vorgehensweise von Agent 1 seinen Bedarf über das Lager decken und dann wiederum an seinen Nachfolger weitergeben. Hat Agent 2 jedoch eine Kapazitätsbeschränkung, die ihm erlaubt den gesamten Bedarf der ersten Periode zu produzieren kann er seine Kapazität voll ausreizen.

Diese Vorgehensweise eliminiert den Wettbewerbsnachteil der Agenten untereinander und gibt die Möglichkeit Kosten weiterzugeben.

### Vorschlagsgenerierung

Hier beschreiben, wie die Vorschläge generiert werden und warum genau 4

### Lösungsgenerierung

Vote-based vs. Cost-based

# Realisierungen

## 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

## Agenten

### register

Der Agent macht einen Webservice-Call zum Mediator und wird dort in einer Session im Hauptspeicher gehalten. Die Registrierung des Agenten beim Mediator sieht in JSON wie folgt aus:

{

"type" : "agentregistration",

"id" : 1,

"config" : "Bsp1",

"requires" : [2,3],

"demand": [1,2,3,4,5,6,7,8,9,10,11,12]

}

Jeder Agent gibt dem Mediator seine ID, welche ihm bei seinem Aufruf gegeben wird, mit. Darüber hinaus werden seine Nachfolger -requires- und sein Bedarf -demand- an den Mediator übermittelt.

### Agent Response

Als Antwort auf den Mediator Request sendet der Agent seine Agent Response an den Mediator zurück. Diese Agent Response sieht in JSON wie folgt aus:

{

"type" : "agentresponse",

"selection" : 1,

"demands" : [

{"solution":1, "demand":[1,2,3,4,5,6,7,8,9,10,11,12]},

{"solution":2, "demand":[1,2,3,4,5,6,7,8,9,10,11,12]},

{"solution":3, "demand":[1,2,3,4,5,6,7,8,9,10,11,12]},

{"solution":4, "demand":[1,2,3,4,5,6,7,8,9,10,11,12]}

],

"cost":720

}

In der Agent Response wird die gewählte Lösung -selection- des Agenten übermittelt. Der Agent gibt in seiner Response auch noch die Kosten -cost- der gewählten Lösung aus. Diese werden beim Mediator gespeichert und dort nur verwendet, wenn der Mediator anhand der Gesamtkosten entscheidet.

## Mediator

### start Negotiation

Sobald alle Agenten beim Mediator registriert sind, wird durch vom Mediator die Methode init. Solutions aufgerufen.

### init. Solutions

Der Mediator generiert die initiale Lösungsmenge, bestehend aus (Anzahl der Agenten) – 1 Einzellösungen. Eine Einzellösung enthält einen Bit-String der Länge (Anzahl der Agenten) x (Anzahl der Perioden) und besteht, wie der Name schon sagt, aus den booleschen Werten   
true (1) und false (0).

Im gewählten Fall besteht also eine Lösungsmenge aus 4 Einzellösungen, da es 5 Agenten sind. Diese Einzellösungen sind insgesamt 60 Bits lang und geben an, wann ein Agent produzieren kann und wann er es nicht darf.

Die so generierte Lösungsmenge wird dann in die Mediator Request überführt.

### Mediator Request

Der Mediator sendet einem Agenten den Teil der initiierten Lösung, der für ihn bestimmt ist. Dieser Mediator Request sieht in JSON wie folgt aus:

{

"type" : "mediatorrequest",

"solutions" : [

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

"demands":[10,20,30,40,50,60,70,80,90,100,110,120]},

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

"demands":[100,20,30,40,50,60,70,80,90,100,110,120]},

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

"demands":[1,20,30,40,50,60,70,80,90,100,110,120]},

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

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

]

}

Der Mediator Request enthält die 12-Bit Vorgabe wann der Agent Produzieren kann und wann er es nicht darf. Darüber hinaus wird dem Agenten sein Produktionsbedarf -demantd- mitgegeben. Der Produktionsbedarf wird durch den Agenten bei dessen Registrierung an den Mediator übergeben.

### det. Solutions

Der Mediator speichert die Entscheidungen der Agenten. Hierbei entsteht immer die Situation, dass mindestens eine der Lösungen von zwei Agenten gewählt wird. Von dieser Lösung werden dann mit gen. Neighbor Nachbarlösungen gebildet werden. Als Spezialfall gibt es auch die Möglichkeit, dass zwei Lösungen von je zwei Agenten ausgewählt werden und somit eine Patt-Situation entsteht. Die zwei Lösungen werden dann beide in die neue Lösungsmenge überführt und mit Nachbarlösungen der beiden Lösungen versehen.

### gen. Neighbor

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 weitere generiert und den Agenten zur Abstimmung präsentiert werden. Dazu wird ein Parameter ausgelesen, welcher in der Methode gen. Neighbor vorgibt, wie viele Bit-Flips durchgeführt werden. Entsprechend des Parameters ergeben sich dann mehr oder weniger viele verschiedene Nachbarlösungen. Sollte bei det. Solutions zwei Lösungen ausgewählt worden sein, werden diese in der Nächsten Verhandlungsrunde als Lösung eins und zwei verwendet. Von beiden Lösungen werden jeweils eine Nachbarlösung generiert und somit ist die Anzahl der Lösungen wieder vollständig für einen weiteren Mediator Request. Um zu gewährleisten, dass sich die neuen Lösungen zu beginn der Verhandlungen möglichst stark von der gewählten Lösung unterscheiden wurde folgende Formel umgesetzt:

B ist die Anzahl an Bits, die überhaupt per Bit-Flip verändert werden

k ist als Faktor für die Krümmung der Funktion zuständig und gibt an wie schnell die Anzahl der per Bit-Flip veränderten Bits sinkt.

R ist die maximale Anzahl der Verhandlungsrunden.



Abbildung : Bit-Flips-Berechnung, eigene Darstellung

### end Negotiation

Wenn alle Verhandlungen beendet sind und die optimale Lösung für alle Agenten gefunden ist, wird mit end Negotiation 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]

}

# Auswertung

# Aufgabenverteilung

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

Abb 5: Aufgabenverteilung, eigene Darstellung.