

Verteilte Systeme Übung Nr. 6

Alexander Steen , Max Wisniewski

Tutor : Philipp Schmidt

Vorbereitung

Da wir in den letzten 2 Wochen viel zu tun hatten, haben wir unsere letzte Version nicht noch einmal abgeändert, sondern haben diese einfach weiter verwendet. Da wir Schiffe und andocken schon implementiert hatten, haben wir das Protokoll in diesen Punkten nicht noch einmal angepasst.

Dadurch haben wir aber keine allgemeine Kompatibilität zum vorgeschlagenen Protokoll mehr, aber innerhalb unserer Implementierung funktioniert alles wie gefordert.

Aufgabe

In der Aufgabe 6 sollte man das Spiel vom letzten mal um zwei Befehle erweitern. Dies ist zum einen der Befehl **goods**. Wird dieser aufgerufen, sollen alle verfügbaren Waren im bekannten Universum zurückgeliefert werden.

Der zweite Befehl ist **costs**, der einem von einem angegebenen Planeten ausgibt, wie viel einer bestimmten Ware vorrätig ist und zu welchem Preis diese vorhanden ist.

Implementierung

Beim letzten mal haben wir unser Framework für das Programm vorgestellt. Wir haben es so angepasst, dass wir unsere Klassen **Planet** und **Ship** nur noch eine Message- oder Commandinterface implementieren lassen müssen und schon werden die betreffenden Nachricht empfangen und entsprechend geparsed.

Wir müssen uns diesmal also nur noch um die konkrete Umsetzung kümmern. Betrachten wir erst einmal unsere Waren

Market

Durch den Market müssen wir uns später beim Kauf und Verkauf von Waren nicht mehr um die genaue Umsetzung von Preisen kümmern. Wir speichern hier auch alle Waren, die wir persönlich im Markt anbieten.

Algorithm 1 Die Implementierung des Market

```
1 public class Market {
2     private Map<String, Integer> value = new HashMap<String, Integer>();
3     private Map<String, Integer> need = new HashMap<String, Integer>();
4     private Map<String, Integer> capa = new HashMap<String, Integer>();
5     private Map<String, Integer> ttl = new HashMap<String, Integer>();
6
7     public void newGood(String name, int value, int need, int ttl) {
8         this.value.put(name, value);
9         this.need.put(name, need);
10        this.ttl.put(name, ttl);
11        this.capa.put(name, 0);
12    }
13
14    public void delGood(String name) {
15        this.value.remove(name);
16        this.need.remove(name);
17        this.ttl.remove(name);
18        this.capa.remove(name);
19    }
20
21    public int price(String name) {
22        return this.value.get(name)
23            * (this.need.get(name) / this.capa.get(name));
24    }
25
26    public int amount(String name){
27        return this.capa.get(name);
28    }
29
30    public int ttl(String name){
31        return this.ttl.get(name);
32    }
33
34    public Set<String> allGoods() {
35        return this.value.keySet();
36    }
37
38    public int buy(String name, int amount) {
39        if(this.capa.get(name) <= 0) return 0;
40        int price = this.value.get(name)
41            * (this.need.get(name) / this.capa.get(name));
42        int realAmount = this.capa.get(name) < amount ? this.capa.get(name) : amount;
43        this.capa.put(name, this.capa.get(name)-realAmount);
44
45        return price*realAmount + 1;
46    }
47
48    public int sell(String name, int amount) {
49        if(this.need.get(name) <= 0) return 0;
50        int price = this.value.get(name) * (this.capa.get(name) / this.need.get(name));
51        this.capa.put(name, this.capa.get(name) + amount);
52
53        return price*amount + 1;
54    }
55 }
```

Wir können hier im Market einfach neue Waren erschaffen. Dazu brauchen wir einen Bedarf an Waren, einen Standardpreis, eine TimeToLife und einen Namen. In einer weiteren Methode können wir schon einmal eine Startmenge anlegen. Die Berechnung für den eigentlichen Preis ist reichlich willkürlich, aber erfüllte unsere Forderungen an einen Preis für Angebot und Nachfrage.

Goods

Um goods zu Implementieren müssen wir 2 Seiten betrachten. Einmal die Planeten, die diese Nachrichten weiterleiten und immer aktuell halten müssen, und zum anderen die Schiffe, die diese Informationen benötigen um anständig Handl treiben zu können.

Betrachten wir das ganze ersteinmal von der Planetenseite aus:

Algorithm 2 goods from Planet

```
1 public void onTick() {
2     for (String good : this.reachableGoods.keySet()) {
3         int ttl = this.reachableGoods.get(good) - 1;
4         if ((this.market.isGood(good) && ttl < this.market.ttl(good))) {
5             this.reachableGoods.put(good, this.market.ttl(good));
6         } else {
7             this.reachableGoods.put(good, this.reachableGoods.get(good));
8             if (ttl == 0)
9                 this.reachableGoods.remove(good);
10        }
11    }
12    Message m = GameMessage.GOODS.toMessage(this.sendGoods());
13    for (Channel c : this.connectedPeers.values()) {
14        c.send(m);
15    }
16 }
17
18 public void onSdoog(Channel c, String[] goods) {
19     String oMessage = GameMessage.SDOOG.toString();
20     for (int i = 0; i < goods.length; ++i) {
21         oMessage += " " + goods[i];
22     }
23     this.con.println(StdFd.Messages, oMessage);
24     this.updateReachableGoods(goods);
25     this.updatePlanetList();
26 }
27
28 public void onGoods(Channel c, String[] goods) {
29     String oMessage = GameMessage.GOODS.toString();
30     for (int i = 0; i < goods.length; ++i) {
31         oMessage += " " + goods[i];
32     }
33     this.con.println(StdFd.Messages, oMessage);
34     this.updateReachableGoods(goods);
35     c.send(GameMessage.SDOOG.toMessage(this.sendGoods()));
36 }
```

Wir haben uns als aller erstes um die Waren einigermaßen auf dem neusten Stand zu halten einen Timer geschrieben, der den Planeten nach *intervall* Zeiteinheiten (bei uns 10s) benachrichtigt, dass er seine Nachbarn nach **GOODS** fragt. Am Anfang jeder Runde zählen wir die *ttl* einer Ware herunter, außer wenn die *ttl* unter die in unserem Markt fällt oder die *ttl* 0 ist, dann entfernen wir die Ware.

Zu beachten in der Implementierung ist. Dass wir nicht den gesamten Graphen fluten. Dies führt dazu, dass **GOODS** erst nach einiger Zeit die Änderung von Waren an den gesamten Graphen übermittelt hat.

Dies führt bei uns zu 2 Beobachtungen. Erstens braucht es *diam* (Graph) Runden, bis eine neue Ware den gesamten Graphen erreicht hat (unter der Voraussetzung, dass die *ttl* groß genug ist um den gesamten Graphen zu erreichen) [*diam* steht für den Durchmesser des Graphen, also den längsten der Kürzesten Wege eines Graphen von einem Knoten zum anderen]. Zweitens brauchen wir nach dem auslöschen einer Ware *diam* (Graph) + *ttl* Runden, damit der letzte dieses verschwinden bemerkt. Dies gilt natürlich nur, wenn auf dem Weg die Ware nicht mehr vorhanden ist. Wir haben uns dafür entschieden, weil die Benutzung sehr viel leichter ist, als jedes mal den gesamten Graphen zu fluten. Es verursacht pro Runde eine sehr viel kleinere Menge an Nachrichten die ausgetauscht wird.

Dies alles ist nur Sinnvoll, durch die Überlegung, das die Waren, die angeboten werden, sich nicht allzuschnell ändern werden. Was sich schnell Ändern kann sind die Preise. Da die Waren an sich für lange Zeit bestehen werden, wird auch unser gesamter Zustand im Graphen die größte Teil konsitent sein. Die Schwerfälligkeit bei Veränderungen zahlt sich aber spätestens bei der Reaktion in bestehenden Systemen aus. Gerade wenn Schiffe sehr viel Reisen und anfragen.

Costs

Da sich die Preise von Waren recht schnell Ändern können, benötigen wir einen anderen Ansatz als bei **GOODS**. Die Preise im Cache zu halten lohnt sich also nicht, wie bei den bisherigen Möglichkeiten.

Um den Preis zu erfragen, verwenden wir das Softwarerouting, das uns das **PEERS** ermöglicht:

Algorithm 3 goods leitet die Anfrage durch den Graphen

```
1 public void onTsoc(String[] way, String good, int price, int amount) {
2     String oMessage = GameMessage.SD00G.toString();
3     for (int i = 0; i < way.length; ++i) {
4         oMessage += " " + way[i];
5     }
6     oMessage += " # " + good + " # " + price + " " + amount;
7     this.con.println(StdFd.Messages, oMessage);
8     if (way[way.length - 2].equals(this.name)) {
9         //Konstruiere Message way ...
10        Message m = GameMessage.TSOC.toMessage(way);
11        this.dockedShips.get(way[way.length - 1]).send(m);
12    } else {
13        int pos = this.search(way, this.name);
14        if (pos >= 0) {
15            //Konstrukt Message way ...
16            Message m = GameMessage.COST.toMessage(way);
17            this.connectedPeers.get(way[pos + 1]).send(m);
18        }
19    }
20 }
21
22 public void onCost(String[] way, String good) {
23     String oMessage = GameMessage.COST.toString();
24     for (int i = 0; i < way.length; ++i) {
25         oMessage += " " + way[i];
26     }
27     oMessage += " # " + good;
28     this.con.println(StdFd.Messages, oMessage);
29     if (way[way.length - 1].equals(this.name)) {
30         // Wir sind die angefragten
31         if (this.market.isGood(good)) {
32             int price = this.market.price(good);
33             int amount = this.market.amount(good);
34             String[] msg = new String[way.length + 5];
35             //Konstrukt Message way
36             Message m = GameMessage.TSOC.toMessage(msg);
37             if (this.dockedShips.containsKey(msg[1])) {
38                 this.dockedShips.get(msg[1]).send(m);
39             } else {
40                 this.connectedPeers.get(msg[1]).send(m);
41             }
42         }
43     } else {
44         int pos = this.search(way, this.name);
45         if (pos >= 0) {
46             way = this.invertInTo(new String[way.length + 2], way);
47             way[way.length] = "#";
48             way[way.length + 1] = good;
49             Message m = GameMessage.COST.toMessage(way);
50             this.connectedPeers.get(way[pos + 1]).send(m);
51         }
52     }
53 }
```

Algorithm 4 Goods von der Schiffseite aus

```
1 public void goodsCommand() {
2     synchronized (this) {
3         this.con.println("Asked for goods.\n");
4         if (this.pName == null) {
5             return;
6         }
7         String[] empty = { this.name, "#" };
8         this.pChannel.send(GameMessage.GOODS.toMessage(empty));
9     }
10 }
11
12 public void onSdoog(Channel c, String[] goods) {
13     String oMessage = GameMessage.SDOOG.toString();
14     for (int i = 0; i < goods.length; ++i) {
15         oMessage += " " + goods[i];
16     }
17     this.con.println(StdFd.Messages, oMessage);
18     this.con.println("Available Goods: ");
19     for (int i = 2; i < goods.length; ++i) {
20         String[] split = goods[i].split("\\.");
21         this.con.println(" >> " + split[0]);
22     }
23 }
```

Wenn wir eine Nachricht **COST** bekommen, schauen wir nach, ob wir der eientliche Empfänger dieser Nachricht. Sind wir es, packen wir hinten an die Nachricht den Preis und die Menge ran und schicken sie den Weg zurück.

Sind wir in der Mitte des Weges verhalten sich **COST** und **TSOC** gleich. Sie schicken es einfach an den nächsten in der Liste weiter.

Bei **TSOC** müssen wir nur noch darauf achten, dass wir der vorletzte in der Liste sind. Ist dies der Fall müssen wir in unserer Schiffsliste nachschauen, weil der Befehl **COSTS** nur von einem Schiff stammen konnte.

Angekommen geben wir nur den Preis aus, weil uns das speichern aufgrund der schnell schwankenden Preise nicht interessieren würde.

Nachdem wir nun den Planeten betrachtet haben, schauen wir uns das ganze noch einmal von der Schiffseite aus an. Da die Hauptarbeit der Planet erledigt, ist hier aber nicht mehr viel wissenswertes zu holen.

Wie man sieht haben wir mit unserer Goodsimplementierung nicht viel zu tun. Wir schicken einmal die Anfrage ohne neue Waren mitzuschicken und empfangen die Antwort. Der größte Teil der Implementierung verwenden wir auf die Ausgabe.

Cost ist etwa genau so schwer.

Algorithm 5 Cost macht genausoviel arbeit.

```
1 public void onTsoc(String[] way, String good, int price, int amount) {
2     // Nicht machen, einfach weg hauen
3     this.con.println("PriceInfo >> "+ good + " [price: " + price + "; amount: " + amount+ "] on planet "+way[0]);
4 }
5
6
7 public void costCommand(String name, String good) {
8     if(this.peers.containsKey(name)){
9         String[] way = new String[this.peers.get(name).length+2];
10        way = this.copyInto(this.peers.get(name), way);
11        way[this.peers.get(name).length] = "#";
12        way[this.peers.get(name).length+1] = good;
13
14        this.pChannel.send(GameMessage.COST.toMessage(way));
15    }
16 }
```

Bemerkung

Das einzige, dass uns an dieser Implementierung stört, ist dass die GOODS bestimmen sehr lange dauern kann. Allerdings hätte es noch eine TTL gebraucht um eine unkontrollierbare Vermehrung von GOODS Nachrichten zu verhindern. Diese Vermehrung kann ausbrechen, wenn wir nur die gegebene TTL verwenden, weil diese nach spezifikation immer raug gesetzt werden soll, wenn wir über einen Knoten laufen, der die Ware anbietet [und dess ttl größer ist]. Wenn man sich das ganze anguckt, sieht man schnell, dass ein Kreis über einen solchen Knoten die Nachricht endlos kreisen lassen würde. Dies würde aber im Vergleich zu unserer Lösung zu einem viel höheren Kommunikationsaufwand führen. Wir finden für einen stabilen Markt an Waren unsere Lösung an sich praktischer als eine solche Lösung.