# Verteilte Systeme Übung Nr. 6/7

Alexander Steen , Max Wisniewski

Tutor : Philipp Schmidt

## Vorbereitung

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

Dadurch haben wir aber keine allgemeine Kompatibilität zum vorgeschlagenen Protokoll mehr, aber innerhalb unserer Implementierung funkioniert 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 aufgrerufen, 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.
In Aufgabe 7 sollten wir nun endlich Handeln können. Dazu muss das Schiff von einem Planeten zum anderen reisen können. Dazu brauchen wir das **travel** command, dies kann mit mittels dock, kcod und undock den Planeten wechseln.

Danach brauchen wir noch **sell, buy, (yub, lles)** um den eigentlichen kauf zu realisieren.
Zur besseren Bedienung haben wir noch **status** [gibt Geld, Position und Waren aus], **drop** [schmeißt eine Ware weg] und **path** [gibt den Weg zu einem Planeten auf]. Diese Befehle an sich sind nicht schwer zu implementieren und werden hier deswegen nicht näher erklärt.

## 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 ersteinmal 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       if(this.capa.get(name)==0) return this.value.get(name) *
23             this.need.get(name) * 2;
24       return this.value.get(name)
25          * (this.need.get(name) / this.capa.get(name));
26    }
27
28    public int amount(String name){
29       return this.capa.get(name);
30    }
31
32    public int ttl(String name){
33       return this.ttl.get(name);
34    }
35
36    public Set<String> allGoods() {
37       return this.value.keySet();
38    }
39
40    public int buy(String name, int amount) {
41       if(this.capa.get(name) <= 0) return 0;
42       int price = this.price(name);
43       int realAmount = this.capa.get(name) < amount ? this.capa.get(name) : amount;
44       this.capa.put(name, this.capa.get(name)-realAmount);
45
46       return price*realAmount + 1;
47    }
48
49    public int sell(String name, int amount) {
50       if(this.need.get(name) <= 0) return 0;
51       int price = this.price(name);
52       this.capa.put(name, this.capa.get(name) + amount);
53
54       return price*amount + 1;
55    }
56 }
```

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 Nachrfrage.

### 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.rechableGoods.keySet()) {
3        int ttl = this.rechableGoods.get(good) - 1;
4        if ((this.market.isGood(good) && ttl < this.market.ttl(good))) {
5           this.rechableGoods.put(good, this.market.ttl(good));
6        } else {
7           this.rechableGoods.put(good, this.rechableGoods.get(good));
8           if (ttl == 0)
9              this.rechableGoods.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 gesammten Graphen fluten. Dies führt dazu, dass **GOODS** erst nach einiger Zeit die Änderung von Waren an den gesammten Graphen übermittelt hat.

Dies führt bei uns zu 2 Beobachtungen. Erstens braucht es $diam\,(\text{Graph})$ Runden, bis eine neue Ware den gesammten Graphen erreicht hat (unter der Vorraussetzung, dass die ttl groß genug ist um den gesammten 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 $diam\,(\text{Graph}) + ttl$ 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 gesammten 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 gesammter 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     if (way[way.length - 2].equals(this.name)) {
3        //Konstruiere Message way ...
4        Message m = GameMessage.TSOC.toMessage(way);
5        this.dockedShips.get(way[way.length - 1]).send(m);
6     } else {
7        int pos = this.search(way, this.name);
8        if (pos >= 0) {
9           //Konstrukt Message way ...
10          Message m = GameMessage.COST.toMessage(way);
11          this.connectedPeers.get(way[pos + 1]).send(m);
12       }
13    }
14 }
15
16 public void onCost(String[] way, String good) {
17    if (way[way.length - 1].equals(this.name)) {
18       // Wir sind die angefragten
19       if (this.market.isGood(good)) {
20          int price = this.market.price(good);
21          int amount = this.market.amount(good);
22          String[] msg = new String[way.length + 5];
23          //Konstrukt Message way
24          Message m = GameMessage.TSOC.toMessage(msg);
25          if (this.dockedShips.containsKey(msg[1])) {
26             this.dockedShips.get(msg[1]).send(m);
27          } else {
28             this.connectedPeers.get(msg[1]).send(m);
29          }
30       }
31    } else {
32       int pos = this.search(way, this.name);
33       if (pos >= 0) {
34          way = this.invertInTo(new String[way.length + 2], way);
35          way[way.length] = "#";
36          way[way.length + 1] = good;
37          Message m = GameMessage.COST.toMessage(way);
38          this.connectedPeers.get(way[pos + 1]).send(m);
39       }
40    }
41 }
```

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

**Algorithm 4** Goods von der Schiffseite aus

```
public void goodsCommand() {
   synchronized (this) {
      this.con.println("Asked for goods.\n");
      if (this.pName == null) {
         return;
      }
      String[] empty = { this.name, "#" };
      this.pChannel.send(GameMessage.GOODS.toMessage(empty));
   }
}

public void onSdoog(Channel c, String[] goods) {
   String oMessage = GameMessage.SDOOG.toString();
   for (int i = 0; i < goods.length; ++i) {
      oMessage += " " + goods[i];
   }
   this.con.println(StdFd.Messages, oMessage);
   this.con.println("Available Goods: ");
   for (int i = 2; i < goods.length; ++i) {
      String[] split = goods[i].split("\\.");
      this.con.println(" >> " + split[0]);
   }
}
```

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 nocheinmal 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.

```
public void onTsoc(String[] way, String good, int price, int amount) {
   // Nicht machen, einfach weg hauen
   this.con.println("PriceInfo >> "+ good + " [price: "           +
      price + "; amount: " + amount+ "] on planet "+way[0]);
}

public void costCommand(String name, String good) {
   if(this.peers.containsKey(name)){
      String[] way = new String[this.peers.get(name).length+2];
      way = this.copyInTo(this.peers.get(name), way);
      way[this.peers.get(name).length] = "#";
      way[this.peers.get(name).length+1] = good;

      this.pChannel.send(GameMessage.COST.toMessage(way));
   }
}
```

### Travel

Um reisen zu können, brauchen wir zunächst ein Ziel. Da wir, nach dem ersten andocken, nicht mit IP und Port arbeiten wollen, holen wir uns verdeckt mit der Frage **WHEREIS** vom Planeten beides zum angefragten Planetennamen. Wenn wir die antwort **THEREIS** bekommen, verlassen wir den Planeten und fliegen zu diesem neuen hin.

**Algorithm 6** Der Ablauf beim Wechsel von Planeten, Seite des Schiffes

```
1  SHIP:
2  public synchronized void onThereis(String addr, int port) {
3     InetAddress a;
4     if (addr.equals("127.0.0.1")) {
5        a = ((UdpChannel) pChannel).getRemoteAddress();
6     } else {
7        try {
8           a = InetAddress.getByName(addr);
9        } catch (UnknownHostException e) {
10          this.con.println("Lost connection to the planet.");
11          return;
12       }
13    }
14    this.gold -= Math.abs(port - ((UdpChannel) pChannel).getRemotePort());
15    this.mreg.removePeer(pChannel, pName);
16    pChannel.close();
17    this.pChannel = null;
18    this.pName = null;
19    this.peers.clear();
20    this.onDock(a, port);
21 }
22 public void onTravel(String name) {
23    String[] remotename =
24       { this.name, GameMessage.prepareProtokoll(name) };
25    this.pChannel.send(GameMessage.WHEREIS.toMessage(remotename));
26 }
```

Rufen wir auf der Console **travel** auf, übermitteln wir dem Planeten unseren Namen und den Namen des Planeten, zu dem wir reisen möchten. Bekommen wir die Antwort vom Planeten **THEREIS**, dann schließen wir den Channel zum Planeten, ziehen die Flugkosten ab und bauen eine neue Verbindung mit dem neuen Planeten auf. (Dieser Teil ist schon implementiert, wird hier also verwendet).

**Algorithm 7** Der Ablauf beim Wechsel von Planeten, Seite des Planeten

```
1  public synchronized void onWhereis(Channel c, String ship, String name) {
2     this.con.println(StdFd.Messages, GameMessage.WHEREIS + " " + name);
3
4     Channel that = this.connectedPeers.get(name);
5     if (that == null)
6        return;
7     UdpChannel thatU = (UdpChannel) that;
8     InetAddress inet = thatU.getRemoteAddress();
9     int port = thatU.getRemotePort();
10
11    String[] msg = new String[2];
12    msg[0] = inet.getHostAddress();
13    msg[1] = port + "";
14
15    this.dockedShips.remove(ship);
16    this.mreg.removePeer(c, ship);
17    c.close();
18
19    this.updatePlanetList();
20
21    c.send(GameMessage.THEREISPlaneten.toMessage(msg));
22 }
23 public synchronized void onUndock(Channel c, String name) {
24    this.con.println(StdFd.Messages,GameMessage.UNDOCK +" "+name);
25    this.dockedShips.remove(name);
26    c.close();
27    this.updatePlanetList();
28 }
```

Da wir in unserer Channelabstraktion nicht wissen können, was Addresse und Port der Verbindung sind, müssen wir an dieser Stelle in die Channel rein sehen und unseren UdpChannel zu rate ziehen. Wir suchen

**Algorithm 9** Der Planet bearbeitet den Kaufwunsch

```
public synchronized void onBuy(Channel c, String name, int amount) {
        if (this.market.amount(name) >= amount) {
                int cost = this.market.buy(name, amount);
                String[] bill = new String[3];
                bill[0] = name;
                bill[1] = "" + amount;
                bill[2] = "" + cost;
                Message m = GameMessage.YUB.toMessage(bill);
                c.send(m);
        }
}
```

uns also die Addresse und den Port zu dem gefragten Planeten herraus und schließen die Verbindung. Dies machen wir, da auf der anderen Seite das Schiff nicht warten wird, bis wir fertig sind. Dieses fliegt einfach weg.

### Buy

Da wir mit unserem **Market** schon den logischen Teil des Handelns übernommen haben, müssen wir nun nur noch dem Austausch der Waren annehmen.

**Algorithm 8** Das Schiff fragt an, dass es Waren haben möchte

```
1  public synchronized void onBuy(String name, int amount) {
2     name = GameMessage.prepareProtokoll(name);
3     if (this.gold <= 0) {
4        this.con.println(...);
5        return;
6     }
7     String[] buy = { name, "" + amount };
8     this.pChannel.send(GameMessage.BUY.toMessage(buy));
9  }
10
11 public void onYub(Channel c, String name, int amount, int cost) {
12    this.gold -= cost;
13    if (this.backpack.containsKey(name)) {
14       this.backpack.put(name, amount + this.backpack.get(name));
15    } else {
16       this.backpack.put(name, amount);
17    }
18    this.con.println(...);
19 }
```

Wenn wir den Befehl zum kaufen bekommen, gucken wir zunächst nach ob wir noch Gold besitzen. (Dies ist ein denkbar schlechter Mechanismus zum Missbrauchsschutz, entspricht in seiner Umsetzung aber in etwa unserem Druckservice am Fachbereich)

Ist diese Vorraussetzung erfüllt senden wir dem Planeten diese Bitte.

Bekommen wir die Antwort von Planeten, packen wir die Anzahl in unseren Frachtraum und ziehen den Preis von unserem Gold ab.

Der Planet kann nun den Marktplatz benutzten. Zunächst guckt er nach, ob er überhaupt so viel zu verkaufen hat. Wenn dies der Fall ist, berechnet er die Kosten und schickt diese an das Schiff zurück.

Der eigentliche Warenaustausch findet in dieser Simulation nicht statt. Wenn die Nachricht eintrifft, gelten die Waren als geliefert.

### Sell

Sell sieht äquivalent zu Buy aus. Wir müssen nur prüfen, dass wir so viele Waren besitzen, wie wir verkaufen wollen und am Schluss ziehen wir diese Menge ab und rechnen den Gewinn auf unseren Goldvorrat.

Da es hier nichts neues zu sehen gibt, lassen wir den Teil aus. Im Quellcode ist dieser Teil auch zu bewundern.

## 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 ausftreten, 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 Nacricht 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.

Unsere Lösung für den Markt ist zu der Version, die wir abgeben noch nicht sonderlich gut durchdacht. Wenn man die Waren einfach nur komplett aufkauft und wieder erkauft, dann kann man in null-komma-nichts wahnsinnige Massen an Gold erwirtschaften.

Hier müssen wir noch etwas an der Preisformel herumbasteln.

Das ermitteln der Preise auf den einzelnen Planeten ist etwas schwerfällig, weil man noch nicht weiß, auf welchem Planeten die Waren überhaupt angeboten werden.

Aber wir dachten uns, wer Gewinn machen will, muss schon dafür arbeiten.