Verteilte Systeme Übung 5

Alexander Steen , Max Wisniewski

Tutor : Philipp Schmidt

Aufgabe

Die Aufgabe bestand darin ein kleines Framework für eine verteilte Wirtschaftssimulation zu schreiben. Das Programm soll über eine Console laufen und in dieser ersten Aufgabe dazu ersteinmal die Befehle

connect : Verbindet einen Knoten (Planet) mit einem anderen

peers : Gibt eine Liste von allen Knoten (Planeten) an, die unter einander Verbunden sind implementieren.

Damit die Programme unter einander Funktionieren, wurden die ersten Nachrichten die wir benötigen durch ein Protokoll festgelegt. Wir unterstützen hierbei Stringnachrichten, wobei die Bestandteile durch Whitespaces getrennt werden. Implementiert werden, sollten die Befehle:

HELLO : Teilt seinen Namen einem anderen Knoten mit (baut eine Verbindung auf)

OLLEH : Antwort auf ein Hello, gibt den Namen des anderen Planeten zurück

PEERS : Gibt einen kompletten Weg an, an den die Nachricht geschickt werden soll.

SREEP : Antwort auf PEERS. Dreht den Weg von PEERS um und gibt nach einem # eine Liste mit von dem Endknoten erreichbaren Knoten zurück.

Implementierung

Console

Für eine bessere Darstellung haben wir uns eine eigene Consolenklasse geschrieben (schönere Aufarbeitung in der GUI). Für möglicherweise verschiedene Darstellungen von Klassen (Planeten und Raumschiffe waren ersteinmal angedacht) haben wir ein Interface bereitgestellt:

```
1  public interface Console {
2      public enum StdFd{
3          ...
4      }
5
6      public void setInputHandler(InputHandler handler);
7
8      public void println(String text);
9
10     public void println(int fd, String text) throws IllegalArgumentException;
11
12     public void println(StdFd fd, String text) throws IllegalArgumentException;
13
14     public void clear();
15
16     public void clear(int fd) throws IllegalArgumentException;
17
18     public void clear(StdFd fd) throws IllegalArgumentException;
19
20     public boolean testFd(int fd);
21
22     public boolean testFd(StdFd fd);
23
24     public void setVisible(boolean visible);
25
26     public String waitForName();
27 }
```

Wir haben ersteinmal einen **enum** erschaffen um die verschiedenen Textbereiche leichter dereferenzieren zu können. (Wir habe zum einen StdOut, Planets[Liste von Planet], Messages[anzeige welche Nachrichten uns erreich haben], Market[nicht implementiert, aber für die Wirtschaft angedacht]).

Die Console bekommt einen InputHander, der auf alle Eingaben der Console reagieren kann. InputHandler ist ein Interface, mit der einzigen Methode *onInput*, die wir später bei der Implementierung der Knoten noch einmal betrachten werden.

Es ist zu beachten, dass wir die meisten Methoden dreifach überladen haben um den Umgang mit den Filedescriptoren zu erleichtern (Kein fd bedeutet immer die Standardconsole, sonst kann man den enum angeben oder direkt den int des fd).

Die eigentliche Implementierung der Console lassen wir an dieser Stelle einmal weg. Es werden nur verschiedene Textareas erzeugt und ein Textfield. Dann setzen wir einen Listener auf das Textfield, so dass wir bei einem Druck auf Enter den InputHandler aufrufen können.

Zur besseren Benutzung haben wir noch ein Speicher für die letzen 20 Befehle eingerichtet, so dass man mit *UP* und *DOWN* durchblättern kann.

Command Interfaces

Um möglichst generisch zu bleiben für die Erweiterungen in den folgenden Erweiterungen haben wir uns eines einfachen Commandpatterns für die einzugebenden Befehle bedient. Dabei ist es wiederum in ein Handlerinterface und ein Commandinterface getrennt:

```
1  public interface Handler {
2  }
3
4  public abstract class Command<V extends Handler> {
5
6      protected List<V> reg = new ArrayList<V>();
7
8      public abstract void execute(String[] paras) throws IllegalArgumentException;
9
10     public void register(V handler){
11         reg.add(handler);
12     }
13
14     public abstract String usage();
15     public abstract String description();
16
17     public abstract String command();
18 }
```

Das Handlerinterface ist in dieser Abstraktionsebene ersteinmal ein Markerinterface ohne Methoden (wir schauen uns gleich Beispielhaft eine Implementierung an). Die abstrakte Klasse *Command* kümmert sich nun schon einmal um die Registrierung der Handler. Dabei können nur Untertypen des gerade erwähnten Handlers registeriert werden. Die Generizität erlaubt es uns später in Unterklassen nur die Handler zu registrieren, die wirklich zu diesem Befehl gehören.

Wir haben uns dafür entschieden für einen speziellen Befehl mehrere Handler zuzulassen, obwohl wir dies bisher noch nicht gebracht haben. Es könnte später notwendig sein an mehreren Stellen auf ein solches Ereigniss zu reagieren.

Die restlichen Methoden beziehen sich auf die Ausführung und Benutzung. Der Inputhandler der Console wird seine Eingabe parsen und dann das *execute* des betreffenden Befehls aufrufen. Wirft diese Methode eine Exception werden wir *usage* benutzen um nochmal die genaue Benutzung auf der Console auszugeben.

Der Befehl *description* gibt nocheinmal eine kurze Beschreibung an, was der Befehl macht. Diese Funktion brauchen wir vor allem in der Hilfefunktion.
Zuletzt haben wir noch die Methode *command*, die uns den String zurückgibt, der das Commando auslöst. So haben wir die Befehlssequence an einer zentralen Stelle und müssen bei Änderungen nicht alle Klassen durchsuchen.

Schauen wir uns nun examplarisch das *ConnectCommand* an.

```
1  public interface ConnectHandler extends Handler{
2      void onConnect(InetAddress host, int port);
3  }
4
5  public class ConnectCommand extends Command<ConnectHandler> {
6
7      @Override   public void execute(String[] paras) throws IllegalArgumentException {
8          InetAddress host;
9          int port;
10         try {
11             host = InetAddress.getByName(paras[0]);
12             port = Integer.parseInt(paras[1]);
13
14             for(ConnectHandler h : reg){
15                 h.onConnect(host, port);
16             }
17
18         } catch (UnknownHostException e) {
19             System.out.println("Unkown Host");
20             throw new IllegalArgumentException("Ungültiger Host");
21         } catch (Exception e) {
22             System.out.println(e.getMessage());
23             throw new IllegalArgumentException();
24         }
25     }
26     @Override   public String usage() {
27         return "connect <host: IP or DNS> <port: Int>";
28     }
29     @Override   public String description() {
30         return "Connect to another planet. Loses the old connection.";
31     }
32     @Override   public String command() {
33         return "connect";
34     }
35 }
```

Wir der Befehl ausgeführt, versuchen wir als erstes die ersten beiden Eingaben in eine Addresse und einen Port zu übersetzen, schlägt dieser Versuch fehl, werden wir eine Exception werden, was zu einer Erklärung des Commandos auf der Console führen wird.

Da wir die Klase von *Command<ConnectHandler>* erben lassen, wissen wir, dass in unserer Liste nur *ConnectHandler* sein können. Wir können also ohne zu casten auf die Methode *onConnect* zugreifen.

Unten sieht man nocheinmal die Beschreibung und Benutzung des Befehls.

So sehen die Commandos alle aus, so dass sich der Handler nun wirklich nur noch darum kümmern muss, auf welche er überhauptreagieren will und wie er sei bearbeitet.

Bisher von uns Implementierte Commandos sind *close,cls, connect, dock, global, help, local, peer.* Die Befehle sind überwiegen selbsterklärend oder vorgegeben. Die Befehle *doc, global* und *local* sind Befehle einer schon animplementierten *Ship* Klasse. *Dock* verbindet ein Shiff mit einem Planeten (anderer Befehl und andere Nachricht, damit Planeten den Unterschied zwischen Schiffen und Planeten feststellen können. Würde sich auf durch einen weiteren Zusatz in *CONNECT* lösen lassen. Das ist aber leicht zu ändern, sobald wir uns auf ein Protokoll geeinigt habe).

Local und *Global* sind schon gut funktionierende Chatfunktionen zwischen den Schiffen. *Local* verbreitet dabei eine Nachricht nur auf dem aktuellen Planeten und *Global* schickt es an alle bekannten Planeten weiter.

Message Interfaces

Prinzipiell ist der Aufbau der *MessageCommands* der *Commands* sehr ähnlich. Wir betrachten hier deshalb nur die Interfaces.

```
1  public interface Handler {
2  }
3
4  public abstract class CommandMessage<V extends Handler> {
5
6  protected List<V> reg = new ArrayList<V>();
7
8      public abstract void execute(Channel c, String[] paras) throws IllegalArgumentException;
9
10     public void register(V handler){
11         reg.add(handler);
12     }
13
14     public abstract GameMessage message(); }
```

Insgesammt gesehen, macht die Klasse etwas weniger, weil es keine Benutzung oder Beschreibung für die Nachrichten gibt. Wenn etwas falsch war, dann auf der anderen Seite eines Kanels. Man könnte sich höchstens Fehlernachrichten ausdenken.

Im Gegensatz zu einem einfachen String für die Schlüsselwörter einer Nachricht, geben wir hier eine *GameMessage* zurück. Dies stellt einen *enum* dar, der alle Nachrichten des Protokolls kapselt.

In dieser Klasse gibt es über die reine Message implementierung noch 3 praktische Methoden. Eine kann aus einer *Message* ermitteln um welche GameMessage es sich handelt. Die anderen beiden können zur GameMessage einen Liste von Parametern hinzufügen. Dies ist pratkisch um später die endgültigen Nachrichten zusammen zu setzen.

Wir unterstützen die Nachrichten: *DOCK, KCOD, GLOBAL, LOCAL, HELLO, OLLEH, PEERS, SREEP*. Die ersten 4 Nachrichten wurden im vorigen Abschnitt schon erklärt. Die letzten 4 sind vom Protokoll her vorgegeben gewesen.

Inputhandler

Da die Klassen an sich schon riesig genug geworden sind, haben wir uns dafür entschieden die 3 Klassen *Planet, PlanetCommandRegistration, PlanetMessageRegistration*. Alle 3 sind sehr eng miteinander Verbunden und kennen sich gegenseitig. Vom Prinzip her würden sie alle in eine Klasse passen, aber so ist es etwas übersichtlicher.

Betrachten wir nun zuerst *PlanetCommandRegistration*, die Klasse die den InputHandler implementiert.

```
1  public class PlanetCommandRegistration implements InputHandler {
2      final private Planet planet;
3
4      final Map<String, Command<?>> commands = new HashMap<String, Command<?>>();
5
6      public PlanetCommandRegistration(Planet planet) {
7          this.planet = planet;       addCommandHandler();
8      }
9
10     private void addCommandHandler() {
11         {
12             PeersCommand h = new PeersCommand();
13             h.register(planet);
14             commands.put(h.command(), h);
15         }
16         ... //And many more
17     }
18
19     @Override   public void onInput(String input) {
20         String[] commandParts = input.split(" ");
21         Command<?> command = commands.get(commandParts[0]);
22         if (command != null) {
23             try {
24             command.execute(Arrays.copyOfRange(commandParts, 1, commandParts.length));
25             } catch (IllegalArgumentException e) {
```

```
                if(e.getMessage() != null){
                    if (!e.getMessage().equals("")) {
                        this.planet.getConsole().println(e.getMessage());
                    }
                }
                this.planet.getConsole().println(command.usage());
            }

        } else {
            this.planet.getConsole().println("Unknown Command");
            this.planet.getConsole().println(" >>\"help\"");
        }

    }

    String helpText(){
        StringBuilder sb = new StringBuilder("\n\n-------- Commands ---------\n");
        for (String command : this.commands.keySet()) {
            sb.append(command);
            sb.append(" - ");
            sb.append(this.commands.get(command).description());
            sb.append("\n");
            sb.append("    ->");
            sb.append(this.commands.get(command).usage());
            sb.append("\n");
        }
        sb.append("----------------------------");
        return sb.toString();
    }
}
```

Diese Klasse kümmert sich am Anfang darum, dass alle Befehle angelegt werden (in *addCommandHandler*) und trägt auch gleich die Klasse *Planet* als Handler für die Befehle ein.

Am Schluss sehen wir noch, wie der Hilfetext zusammengestellt wird. Dies muss hier geschehen, da diese Klasse die Befehle kennt.

Die Hauptarbeit befindet sich natürlich in *onInput*. Wir teilen hier als erstes die Eingabe anhand der Leerzeichen auf. Nun können wir schauen, ob der erste Teil als Befehl eingetragen ist. Ist die der Fall, kopieren wir den Rest als Parameter und rufen die *execute* Funktion auf. Sollte der Befehl nicht existieren, geben wir den Hinweis aus, es einmal mit **help** zu probieren. Sollte bloß die Ausführung scheitern geben wir die Benutzung des Befehls aus.

MessageHandler

Die Klasse *PlanetMessageRegistration* ähnelt im großen und ganzen dem InputHander. Bloß anstatt auf einen expliziten Aufruf zu warten werden wir hier auf alle unsere Channel lauschen, ob wir eine Nachricht bekommen.

Zusätzlich zu den Eingetragenen, werden wir immer eine handvoll nichtverbundene offen halten um neue Verbindungen zu akzeptieren.

```
public class PlanetMessageRegistration {

    final private Planet planet;

    final Map<String, CommandMessage<?>> messages = new HashMap<String, CommandMessage<?>>();

    private List<Channel> connectedPeers = new ArrayList<Channel>();
    private List<Channel> removedPeers = new ArrayList<Channel>();
    private List<Channel> addedPeers = new ArrayList<Channel>();

    final int LOCALPORT;
    final int LISTEN_NUM = 5;

    private List<Channel> listenChannel = new ArrayList<Channel>();

    public PlanetMessageRegistration(Planet planet, int port) {
        this.planet = planet;
        this.LOCALPORT = port;
```

```
19          this.fillListenChannel();
20          addMessageHandler();
21          new Communicaton();
22      }
23
24      ...
25
26      // ----------------- Handle Connections -------------------------------
27      protected class Communicaton extends Thread {
28          Message actMessage;
29
30          public Communicaton() {
31              this.start();
32          }
33
34          public void run() {
35              while (true) {
36                  try {
37                      UdpChannelFactory.waitOnPort(LOCALPORT);
38                  } catch (InterruptedException e) {
39                      continue;
40                  }
41                  for (Channel c : listenChannel) {
42
43                      try {
44                          if ((actMessage = c.nrecv()) != null) {
45                              String[] input = new String(Arrays.copyOfRange(
46                                      actMessage.getData(), 0,
47                                      actMessage.getLength())).split(" ");
48                              if (messages.containsKey(input[0]))
49                                  messages.get(input[0]).execute(
50                                                  input.length));
51                          }
52                      } catch (IllegalArgumentException e) {
53                          if (e.getMessage() != null) {
54                              if (!e.getMessage().equals("")) {
55                                  System.err.println(e.getMessage());
56                              }
57                          }
58                          System.err.println("Received Unknown Message");
59                      }
60                  }
61
62                  for (Channel c : connectedPeers) {
63                      ... // Selbe wir oben
64                  }
65
66                  synchronized (addedPeers) {
67                      connectedPeers.addAll(addedPeers);
68                      listenChannel.removeAll(addedPeers);
69                      fillListenChannel();
70                      addedPeers.clear();
71                  }
72                  synchronized (removedPeers) {
73                      connectedPeers.removeAll(removedPeers);
74                      removedPeers.clear();
75                  }
76              }
77          }
78      }
79  }
```

Wenn am Anfang jedes Durchganges legen wir uns schlafen und warten darauf, dass wir eine Nachricht auf unserem Port bekommen. Diese Methode ist leider noch nicht sehr ausgereift, so dass ich einen Standard-timeout dadrin habe. Wir werden war aufgeweckt, wenn wir schlafen und eine Nachricht ankommt. Leider kann es passieren, dass wir eine Nachrich bekommen uns dann schlafen legen und nicht geweckt werden. Dieses Problem werden wir hoffenltich zum nächsten mal noch lösen können.

Bekommen wir nun eine Nachricht, schauen wir zuerst nach, ob der erste Teil dieser Nachricht in userer Liste von Nachricht vorkommt. Wenn er das nicht tat, geben wir eine kurze Rückmeldung aus. Kommt die

Nachricht, hingegen vor, rufen wir deren *execute* Funktion mit den restlichen Parametern auf. Dies machen wir 2mal. Einmal für die nichtverbundenen Kanäle und einmal für die verbundenen.

Am Ende jedes Durchgangs schauen wir uns an, ob der Planet uns neue Kanäle hinzugefügt hat. (addChannel und removeChannel wurde in dieser Darstellung erst einmal rausgenommen).

Planet

Im Planeten ist nun die Hauptarbeit zu erledigen. Der Planet implementiert die verschiedenen Handler um Commands und Nachrichten zu verarbeiten. Wir werden hier nun eine Auswahl dieser Methoden vorstellen. Zunächst haben wir eine aufstellung von internen Variablen:

```
1  protected Console con;
2  final protected String name;
3
4  private Map<String, Channel> connectedPeers = new HashMap<String, Channel>();
5  private List<Channel> pendingPeers = new LinkedList<Channel>();
6  private Map<String, Channel> dockedShips = new HashMap<String, Channel>();
7
8  final private PlanetCommandRegistration reg;
9  final private PlanetMessageRegistration mreg;
10
11 private Map<String, String[]> peersToWork = new HashMap<String, String[]>();
12 private Map<String, String[]> peers = new HashMap<String, String[]>();
```

Ersteinmal speichern wir die Console, die beiden Registerklassen von ebend und unseren eigenen Name. Dann brauchen wir noch, für die unmittelbare Umgebung die *connectedPeers* von direkt verbundenen Planeten, *pendingPeers* sind alle angefragten Planeten von denen wir noch auf ein *OLLEH* warten und *dockedShips* schon für die Erweiterung auf Schiffe.

Explizit für das Routing im Netz brauchen wir noch *peersToWork* in der alle Knoten stehen, die wir angefragt haben, aber von denen wir noch keine Nachricht bekommen haben und *peers*. In peers stehen alle Knoten die wir kennen mit dem Weg zu ihnen, wie er mit Protokoll steht.

Betrachten wir zuerst ein *HELLO* als einfache Nachricht:

```
1  public void onHello(Channel c, String name) {
2      synchronized (this) {
3          this.connectedPeers.put(name, c);
4          mreg.addPeer(c);
5          String[] myName = { this.name };
6          c.send(GameMessage.OLLEH.toMessage(myName));
7          this.con.println("A new planet was discoverd right next to us.");
8          String[] way = { name };
9          this.peers.put(name, way);
10         this.con.println(StdFd.Messages, GameMessage.HELLO.toString()+ " " + name);
11         this.updatePlanetList();
12     }
13 }
```

Wir synchronisieren ersteinmal auf uns, weil prinzipiell ein Befehl und eine Nachricht gleichzeit auf die Listen zugreifen könnte.

Dann packen wir denjenigen, der uns die Nachricht geschickt hat in unsere private Liste und geben den Kanel noch unserer *MessageRegistration* mit, damit die auf den Channel lauscht.

Danach setzen wir eine neue *OLLEH* Nachricht zusammen, die wir zurücksenden mit unserem Name drin. Nun machen wir noch ein paar Ausgaben auf den Consolen und legen den neuen Knoten in den *peers* und *connectedPeers* an.

Die Reaktion auf *OLLEH* sieht ähnlich simpel aus und ist im Code nachzulesen.

Betrachten wir nun einmal die den etwas schwierigeren Umgang mit der Nachricht *SREEP*:

```
1  public void onSreep(Channel c, String[] inc) {
2      synchronized (this) {
3          String oMessage = "SREEP";
4          for (int i = 0; i < inc.length; ++i) {
5              oMessage += " " + inc[i];
6          }
7          this.con.println(StdFd.Messages, oMessage);
8          if (inc.length < 4)
9              throw new IllegalArgumentException("To small way");
10         int pos = this.search(inc, "#");
11         if (pos == -1)
12             throw new IllegalArgumentException("Error in Message");
13         if (inc[pos - 1].equals(this.name)) {
14             // Wir haben das Ziel erreicht!
15             String[] newEdges = Arrays
16                     .copyOfRange(inc, pos + 1, inc.length);
17             this.peersToWork.remove(inc[0]);
18                 // Weg wiederum umdrehen
19             String[] way = new String[pos + 1];
20             for (int i = 0; i < pos; ++i) {
21                 way[(pos - 1) - i] = inc[i];
22             }
23             // Neue Kanten hinzufügen und ansprechen
24             for (int i = 0; i < newEdges.length; ++i) {
25                 if (this.peersToWork.containsKey(newEdges[i])
26                         || this.peers.containsKey(newEdges[i])
27                         || newEdges[i].equals(this.name))
28                     continue;
29                 way[pos] = newEdges[i];
30                 this.peersToWork.put(newEdges[i], way);
31                 this.peers.put(newEdges[i], way);
32                 this.connectedPeers.get(way[1]).send(
33                         GameMessage.PEERS.toMessage(way));
34             }
35             this.updatePlanetList();
36         } else {
37             String next = "";
38             for (int i = 1; i < pos; ++i) {
39                 if (inc[i].equals(this.name)) {
40                     // wir haben uns gefunden
41                     next = inc[i + 1];
42                     break;
43                 }
44             }
45             this.connectedPeers.get(next).send(
46                     GameMessage.SREEP.toMessage(inc));
47         }
48     }
49 }
```

SREEP ist die Antwort auf die Nachricht *PEERS*. Letzte muss nur nachschauen, ob man selber der letzte Knoten in der Reihe ist (siehe Protokoll). Ist man es nicht schickt man die Nachricht weiter an den Knoten, den in der Reihe direkt nacheinem kommt. Ist man der letzte, dreht man die Reihe um, schreibt alle seine *connectedPeers* in die Nachricht (nach einem #) und schickt es zurück an den Absender.

Die *SREEP* Nachricht ist insofern schwieriger. als dass hier das exlipzite Routing statt findet. Als erstes müssen wir die Nachricht wieder nur weitersenden, wenn wir nicht der letzte Knoten in der Reihe sind.

Sind wir es allerdings doch, so betrachten wir jetzt alle Knoten hinter dem #, die uns von der Quelle ja als verbundene Knoten mitgegeben wurden. Nun betrachten wir, ob diese Knoten in einer unserer Listen steht (schon verbunden, wird schon untersucht oder wir sind es zufällig selber). Ist uns der neue Knoten wirklich noch nicht bekannt, setzen wir eine neue Nachricht zusammmen. Wir drehen wieder den Weg um und stecken den neuen Knoten als Endpunkt rein.

Diesen Weg können wir jetzt zusammen mit diesem Knoten in *peers* speichern (und in *peersToWork*) und an diesen Knoten eine *PEERS* Nachricht senden um deren Nachbarn herraus zu finden.

Nun können wir den Knoten, von dem wir die Nachbern bekommen hatten aus unserer *peersToWork* Liste entfernen. Alles in allem geschieht nicht viel aufregendes.

Weitergehende Implementierungen

Damit wir die Planeten schoneinmal sinnvoll testen konnten, haben wir uns schoneinmal eine Klasse *Ship* geschrieben. Diese können an einen Planeten andocken und Chatnachrichten über diesen schicken.

Der lokale Chat wird nur auf dem eigenen Planeten verbreitet. Dazu sendet der Planet, der die Nachricht bekommt die Nachricht an alle Schiffe, die an den Planeten angedockt sind.

Beim globalen Chat wird die Nachricht nicht nur an die Schiffe, sondern auch noch an alle verbundenen Planeten weiter gesendet.

Dies ist nur eine kleine Anregung gewesen und baut uns prinzipiell ein PTP-Chat auf, mit Serverknoten und Clients.

Verbesserungen

Was noch eine Verbesserung wäre, damit man das *peers* nicht ständig selber aufrufen muss, wäre eine automatische Ausführung, wenn ein neuer Planet eine Verbindung eingeht.

Um den Kommunikationsaufwand zu verringern, könnte man auch die vorbeigehenden *PEERS* und *SREEP* Nachrichten scannen und neue Planeten hinzufügen, ohne explizit alle zu fragen müssen.
Um bei erneuten *peers* Aufruf schneller alle zu finden, könnte man die schon bekannte Knoten direktanfragen. Dadurch verringert sich der Kommunikationsaufwand zwar nicht, aber man schickt gleich zu beginn die meisten Nachricht schon raus und muss nicht immer erst auf die Antworten warten.

Ein weiteres unser Anliegen ist es noch, den Ansatz des Selects (der schon angedacht ist auf den UDPChannels) zu verbessern.

Wie schon in *MessageRegistration* erwähnt, tut die Funktion noch nicht ganz was sie soll, weil es dazu kommen kann, dass das eintreffen von Nachrichten verpasst wurde.

Wünschenswert wäre auch noch ein anderer Mechanismus für *MessageRegistration*, da bis auf das hinzufügen der Handler und der Ansprechpartner ja doch gleich aussieht. (Ebenso *CommandRegistration*) Man könnte dies über eine abstrakte Klasse ersteinmal lösen, aber das Problem bleibt immer noch, dass man *Planet* und *Ship* über die implementierten Klassen parsen müsste.

Dies würde mit Reflektions leicht gehen, aber der Klarheit wegen, möchte ich Reflektions ersteinmal vermeiden.

Probleme

Ein Problem, das uns lange Zeit beschäftigt hat, war ein Problem unserer UPDChannel.

Wir haben in unseren Channelobjekte die eingegebenen Bytearrays nicht kopiert. Dies viel bei einfacher Benutzung nie auf und auch die meisten unserer Test hier im Programm funktionierte. Bei großen Datenmengen und hohem Nachrichtenaufwand, kam es allerdings doch häufiger dazu, dass manche unserer Nachrichten sich verschluckt haben.

Es vielen einfach ein paar Byte am Ende einer Nachricht weg. Dies führte dazu, dass Routing nicht mehr möglich war, da die betreffenden Namen nicht mehr zu lesen waren.

Diesen Fehler zu finden hat uns einiges an Mühe gekostet, da beim testen der Fehler, durch verzögerungen beim betrachten, in fast allen Fällen verschwand.

Nach einigen Stunden wurde dann aber langsam offensichtlich woran es lag.

Diesen Fehler haben wir beseitigen können. Beim Chatten zeigte sich dann der nächste Fehler, nachdem man Stunden suchen konnte. Wir haben scheinbar vergessen in unserer UDPDispatcherimplementierung die offenen Channel beim belegen aus der Liste zu nehmen. Es kam also öfters zu dem Fall, dass ein Channel mehrfach vergeben wurde. Dies ist natürlich dumm, da dann ein Knoten keine Nachrichten mehr bekommt oder wie in anderen Fällen zu beobachten war, eine Endlosschleife an Nachrichten entstand, weil ein Channel ohne es zusehen an sich selber Nachrichten schicken konnte.