RoboCup-Tutorial

Gerald Kühne, Markus Beier
Version 0.2
18.04.2002

Praktikum Programmiermethodik SS 2002
Lehrstuhl für Praktische Informatik IV, Universität Mannheim

Inhaltsverzeichnis

0 Modifikationen

Im Verlauf des Praktikums werden möglicherweise Änderungen oder Ergänzungen am vorliegenden Dokument erforderlich. Die entsprechenden Modifikationen werden zur besseren Übersicht in diesem Abschnitt zusammengefaßt.

Version 0.1	Initialversion
Version 0.2	Änderungen in Kapitel 5 zur Anpassung an die SimpleClient-Version 0.2

1 Einführung

Das vorliegende RoboCup-Tutorial soll über die ersten Hürden bei der Entwicklung eines Clients für das RoboCup-Simulationssystem hinweghelfen. In Verbindung mit dem offiziellen Handbuch zum RoboCup-Soccer-Server, welches von den Praktikumswebseiten (Rubrik [Dokumente]) heruntergeladen werden kann, sollte ein schneller Einstieg in das System möglich sein.

Das RoboCup-Simulationssystem besteht aus einem Server (*rcssserver*), einem Viewer (*rcssmonitor*) sowie einem einfachen Kommandozeilen-Client (*rcssclient*). Auf der Grundlage des im Server implementierten Kommunikationsprotokolls lassen sich eigene virtuelle Fußballmannschaften erstellen. Ein Server verwaltet dabei genau ein Spiel und bis zu 22 Spieler. Jeder Spieler agiert eigenständig – prinzipiell ist es sogar möglich, daß jeder Spieler auf einem anderen Rechner läuft.

Das Tutorial ist in die folgenden Kapitel unterteilt. In Kapitel 2 nehmen wir ersten Kontakt mit dem RoboCup-Simulationssystem auf. Dazu starten wir die verschiedenen Systemkomponenten, plazieren einen Spieler auf dem Spielfeld und machen unseren ersten Schuß aufs Tor. Im darauffolgenden Kapitel 3 konzentrieren wir uns auf die Netzwerkkommunikation und erläutern den Austausch von Nachrichten zwischen Server und Client auf Basis von UDP/IP. Das nachfolgende Kapitel 4 verwendet die besprochenen Netzwerkroutinen zur Implementierung eines kommandozeilenorientierten Client. Die eingegebenen Kommandos werden über das Netz an den Server übermittelt und dessen Rückmeldungen werden entsprechend ausgegeben. Kapitel 5 beschreibt einen einfachen autonom spielenden Client. Auf seiner Basis könnte man eine gesamte Mannschaft zusammenstellen. Der Client arbeitet nach einer sehr simplen Strategie, aber er verdeutlicht grundlegende Prinzipien. Im Kapitel 6 schließlich stellen wir verschiedene Methoden vor, die man benötigt, um ausgefeiltere Mannschaften zu erstellen. Dazu gehört z.B. eine Positionsroutine, die aus den erhaltenen visuellen Informationen die eigene absolute Position auf dem Spielfeld ermittelt.

In verschiedenen Kapiteln werden Themen wie Ausnahmebehandlung (*exceptions*), Netzwerkprogrammierung und nebenläufige Prozesse (*threads*) angesprochen. Wir konzentrieren uns hierbei auf Aspekte, die das RoboCup-Simulationssystem betreffen. Eine gute java-orientierte Einführung zu diesen Themen findet sich im Java-Lehrbuch von Schader und Schmidt-Thieme [1].

Die im Text eingestreuten Verweise auf Internetseiten (gekennzeichnet durch []) beziehen sich auf die Homepage des Programmierpraktikums SS 2002, Lehrstuhl Praktische Informatik IV, Universität Mannheim [http://www.informatik.uni-mannheim.de/informatik/pi4/stud/veranstaltungen/ss2002/pm/].

2 Erstkontakt

In unserer ersten Kontaktaufnahme mit dem RoboCup-Simulationssystem werden wir mit einem Kommandozeilen-Client einen Spieler auf dem Spielfeld positionieren und den Ball kicken lassen. Dazu benötigen wir zwei Softwarekomponenten, die von der Praktikumsseite [Software] heruntergeladen werden können:

- *rcssserver* - Enthält den RoboCup-Simulationserver `rcssserver` sowie den einfachen Kommandozeilenclient `rcssclient`.

- *rcssmonitor* - Dieses Programm dient der Anzeige eines Spiels, das auf dem RoboCup-Server läuft.

Im Folgenden gehen wir davon aus, daß die Programme `rcssserver`, `rcssclient` und `rcssmonitor` auf dem System installiert sind. Mit `rcssmonitor` bezeichnen wir sowohl das Linux- als auch das Windowsprogramm (welches eigentlich `soccermonitor_newest` heißt).

Zunächst starten wir auf der Kommandozeile den Server durch die Eingabe von `rcssserver` was zu folgender Ausgabe führt:

```
Copyright (C) 1995, 1996, 1997, 1998, 1999 Electrotechnical Laboratory.
Itsuki Noda, Yasuo Kuniyoshi and Hitoshi Matsubara.
2000, 2001 RoboCup Soccer Server Maintainance Group.
Patrick Riley, Tom Howard, Jan Wendler, Itsuki Noda
Hetero Player Seed: 119056
wind factor: rand: 0.000000, vector: (0.000000, 0.000000)
Hit CTRL-C to exit
```

Nun, das sieht nicht besonders spannend aus. Auf jeden Fall wissen wir aber, daß der Server läuft und für entsprechende Clientanfragen bereit ist. Wir können also nun den Kommandozeilenclient `rcssclient` in einem anderen Konsolenfenster starten. Dieses ergibt keine Ausgabe, der Client wartet lediglich auf unsere Kommandos. Als erstes müssen wir Kontakt zum Server herstellen und ihm mitteilen über welche Protokollversion wir kommunizieren wollen. Im Lauf der Zeit haben die Protokolle verschiedene Stadien durchlaufen, in späteren Versionen sind zusätzliche Eigenschaften hinzugekommen – wir verwenden standardmäßig die Protokollversion 7.0. Nach einem kurzen Blick in das Server-Manual, das in der Rubrik [Dokumente] heruntergeladen werden kann, finden wir auf Seite 22 das `init`-Kommando. Dieses erwartet einen Teamnamen sowie optional die Protokollversion. Also geben wir folgende Zeichenkette ein:

```
(init OurTeamName (version 7.0))
```

D.h. wir haben als Teamnamen `OurTeamName` gesetzt und die Protokollversion 7.0 gewählt. Den optionalen Parameter `goalie` des `init`-Befehls ignorieren wir an dieser Stelle erstmal. Mit der Eingabe dieser Zeile haben wir einen Spieler beim `rcssserver` angemeldet und der Server liefert uns einen Haufen Textzeilen folgender Art:

```
send 6000 : (init OurTeamName (version 7.0))
recv 43791 : (init l 1 before_kick_off)

recv 43791 : (server_param 14.02 5 0.3 0.4 0.1 60 1 1 4000 45 0 0.3
0.5 0.002 0 0.3 0.6 0.005 0.6 0.01 0 0 1 1 1 1 0.085 0.94 0.2 0.2
2.7 2.7 0.006 0.027 0.7 2 1.7 100 -100 180 -180 180 -180 90 -90 90
3 0 0 0 0 1.085 2 1 1 2 1 5 0 0 128 128 300 1 1 1 1 50 1 3000 100
150 10 100 300 512 2 1 2 5 1 1 0 9.15 50 0.1 0.01 -1 -1 -1 -1 -1 -1
-1 0 0 0 100 0 0 0 0 200)

recv 43791 : (player_param 7 3 3 0 0.2 -100 0 0.2 25 0 0.002 -100 0
0.2 0.5 0 100 -0.002 -0.002)

recv 43791 : (see 0 ((f r t) 55.7 3) ((f g r b) 70.8 38)
((g r) 66.7 34) ((f g r t) 62.8 28) ((f p r c) 53.5 43)
((f p r t) 42.5 23) ((f t 0) 3.6 -34 0 0) ((f t r 10) 13.2 -9 0 0)
((f t r 20) 23.1 -5 0 0) ((f t r 30) 33.1 -3) ((f t r 40) 42.9 -3)
((f t r 50) 53 -2) ((f r 0) 70.8 31) ((f r t 10) 66 24)
((f r t 20) 62.8 16) ((f r t 30) 60.9 7) ((f r b 10) 76.7 38)
((f r b 20) 83.1 43))
```

Abbildung 1: Anzeige des rcssmonitor nach dem `init`-Kommando des rcssclient.

```
recv 43791 : (sense_body 0 (view_mode high normal) (stamina 4000 1)
(speed 0 0) (head_angle 0) (kick 0) (dash 0) (turn 0) (say 0)
(turn_neck 0) (catch 0) (move 0) (change_view 0))
```

Wir gehen diese Nachrichten Stück für Stück durch. Zunächst teilt uns der Client mit, daß er den `init`-Befehl versendet hat. Die 6000 nach dem `send`-String zeigt uns an, an welchen Port[1] des lokalen Rechners das Kommando gesendet wurde. Danach erhält der Client die erste Nachricht vom Server. Dieses wird über das Kürzel `recv` vor der eigentlichen Nachricht gekennzeichnet. Im Anschluß an das Kürzel steht der Port an dem diese Nachricht empfangen wurde.

Die Serverantwort `(init l 1 before_kick_off)` besagt (siehe Server-Manual, S. 22), daß wir von links nach rechts spielen, unser angemeldeter Spieler die Nummer 1 hat, und das wir uns in der Phase vor dem Anstoß befinden. An diesem kurzen Ausschnitt aus der Client-Server-Kommunikation erkennen wir bereits das Grundprinzip des `rcssserver`-Protokolls. Zuerst steht immer der Name des Kommandos (`init`) bzw. der Information (`see`, `sense_body`) und danach folgen eine Reihe von Parametern, wobei es möglich ist, daß Parameter mittels Klammerung zu Gruppen zusammengefaßt werden. Die Mitteilungen `server_param` und `player_param` betreffen Einstellungen des Servers – wir ignorieren sie an dieser Stelle. Interessanter sind nun die nachfolgenden Nachrichten `see` und `sense_body`. Hiermit teilt der Server mit, was unser Spieler (Nr. 1) sieht und wie er sich fühlt. Wir erhalten alle 150ms eine `see`-Nachricht und alle 100ms erfahren wir etwas über unseren körperlichen Zustand. An dieser Stelle wollen wir nur exemplarisch Bestandteile aus diesen Nachrichten herausgreifen. Genaue Informationen finden sich im Server-Manual. Bei den `see`-Nachrichten steht nach dem Schlüsselwort `see` und dem Zeitstempel (hier: 0) eine Aufzählung von Objekten die unser Spieler in seiner aktuellen Position und Blickrichtung sehen kann. Nehmen wir als Beispiel den Teilstring `((g r) 66.7 34)`. Dieser besagt, daß der Spieler das rechte Tor `(g r)` in einer Entfernung von 66,7 Metern im

[1] Mittels Rechneradresse (IP-Adresse) und Port wird ein Dienst eines Rechners angegeben. In unserem Fall läuft der `rcssserver` auf dem lokalen Rechner `localhost` bzw. `127.0.0.1` am Port 6000. Das soll uns in diesem Zusammenhang nicht weiter interessieren. Interessant wird das erst, wenn Server und Client auf verschiedenen Rechnern laufen.

Abbildung 2: Anzeige des rcssmonitor nach dem `kick`-Kommando des rcssclient.

Winkel von 34 Grad sieht. Die anderen Objekte (`f`) beziehen sich auf Flaggen, die um das Spielfeld positioniert sind.

Aus der `sense_body`-Nachricht greifen wir das Teilstück `(view_mode high normal)` heraus. Hier erfahren wir, daß der Spieler mit hoher Genaugkeit sein normales Gesichtsfeld überblickt.

Soviel zu den Nachrichten – nun wollen wir unseren Spieler sehen und bewegen. Dazu starten wir den `rcssmonitor`, und verbinden ihn mit dem Server. Dieses geschieht entweder automatisch oder über einen Connect-Button. Die Abbildungen zeigen die Unix-Variante des Viewers, das Prinzip ist bei der Windows-Implementierung das gleiche.

Abbildung 1 auf der vorherigen Seite zeigt den Zustand nach dem `init`-Kommando des rcssclient an. Der Spieler Nr. 1 der linken Mannschaft ist farblich hervorgehoben. Der Ball (das ist der weiße Punkt) liegt übrigens auf dem Anstoßpunkt im Mittelkreis.

Nun wollen wir ein wenig Leben auf das Spielfeld bringen. Dazu setzen wir den Spieler Nr. 1 über die Kommandozeile des rcssclient an eine bestimmte Position – in diesem Beispiel an den Anstoßpunkt. Dieses geschieht über das `move`-Kommando, welches nur in bestimmten Spielsituationen (z.B. vor dem Anstoß) möglich ist. D.h. wir geben über den rcssclient folgende Zeile ein:

```
(move 0 0)
```

Daraufhin erscheint unser Spieler im Mittelkreis in der Nähe des Balls. Über den Menüeintrag *kick off* starten wir nun das Spiel. In der Statuszeile sieht man daraufhin das Verstreichen der Spielzyklen. Jetzt wollen wir unseren Spieler agieren sehen: Durch das Kommando `(kick 100 0)` schießt der Spieler den Ball (mehr oder minder) in Richtung des gegnerischen Tors (siehe Abbildung 2).

Anhand obiger Beispiele haben wir die Grundprinzipien des RoboCup-Simulationssystems durchgespielt. Weitere Aktionsmöglichkeiten des Spielers sowie zusätzliche sensorische Informationen sind im Server-Manual aufgeführt.

Wir sehen, daß jeder Spieler separat gesteuert wird und nur das mitbekommt, was der Server ihm übermittelt.[2] An unseren ersten Schußversuchen können wir erkennen, daß es gar nicht so einfach ist, den Ball in das gegnerische Tor zu schießen. Auch ist die Orientierung auf dem Spielfeld nicht ganz einfach – vielmehr müssen wir aus den relativen visuellen Information die eigenen Position errechnen.

3 Netzwerkkommunikation

Zur Kommunikation tauschen `rcssserver` und `rcssclient` Textnachrichten gemäß des im Server-Manual beschriebenen Protokolls über das Internet-Transportprotokoll UDP (User Datagram Protocol) aus. Hierbei findet die Übertragung der Textnachrichten verbindungslos mit Datagrammen statt. Der `rcssserver` sendet Datenpakete an die Clients ohne auf Bestätigung zu warten, ob sie korrekt empfangen wurden oder nicht. Der `rcssclient` sendet seine Befehle auf die gleiche Weise.

Mit JAVA kann man diese Art der Kommunikation recht einfach realisieren. Einen guten Überblick gibt das Netzwerk-Kapitel im Java-Lehrbuch von Schader und Schmidt-Thieme [1]. Im Folgenden beschränken wir uns auf die Komponenten, die im Zusammenhang mit dem RoboCup-Simulationssystem notwendig sind.

Das Package `java.net` bietet alle nötigen Klassen zur Netzwerkkommunikation. Die UDP-Pakete heißen hier `DatagramPacket`. Sie können allerdings nur Daten vom Typ `byte[]` transportieren. Zudem muß man jedem Paket auch noch die Zieladresse mitgeben, da sich jedes Paket seinen eigenen Weg durch die Weiten des Internets sucht.

Ein Paket, an den laufenden `rcssserver` auf `localhost:6000` mit der Nachricht `(init OurTeamName (version 7.0))` wird auf folgende Weise erstellt:

```
InetAddress targetAddr = InetAddress.getByName("localhost");
int serverPort = 6000;
String message = "(init OurTeamName (version 7.0))";
byte[] data = message.getBytes();

DatagramPacket packet = new DatagramPacket(data, data.length,
                                           targetAddr, serverPort);
```

Jetzt haben wir ein Paket, welches nur noch verschickt werden muß. Als Verbindungspunkt zur Außenwelt dient der `DatagramSocket`. Über ihn können Pakete empfangen und versendet werden.

```
DatagramSocket socket = new DatagramSocket();
socket.send(packet);
```

erzeugt einen Socket und sendet das Paket an die im Paket angegebene Adresse.

Die Nachricht ist jetzt auf dem Weg zum Server. Um an die Antwort zu kommen, muß man ein Paket erzeugen, daß die Antwort aufnehmen soll und dann das Antwortpaket vom Socket lesen, was z.B. so aussehen kann:

```
byte[] buff = new byte[1024];
DatagramPacket answer = new DatagramPacket(buff, buff.length);

socket.receive(answer);

String output = new String(answer.getData());
System.out.println(output.trim());
```

Beim Empfang braucht man dem Paket keine Zieladresse zu geben, da es lediglich als Container für das ankommende Paket dient. `answer.getData()` liefert das `byte`-Feld der Antwort zurück, mit dem man

[2] Kommunikation zwischen Spielern kann über das Kommando `hear` sowie auditive Informationen (`say`-Nachrichten) erfolgen. Diese werden ebenfalls über den Server abgewickelt.

dann einen `String` erzeugen kann. Der Aufruf `output.trim()` gibt einen `String` zurück, bei dem alle Leerzeichen an Anfang und Ende des Strings entfernt wurden.

Wenn man nun den ganzen Code in eine eigene Klasse eingebaut hat und das Programm compilieren will, meldet der Compiler, daß *exceptions* (Ausnahmen) abgefangen oder deklariert werden müssen und bricht damit die Übersetzung ab. Bei der Netzwerkkommunikation kann es zu einer Vielzahl von Problemen kommen. Diese werden in JAVA mit Exceptions behandelt, die abgefangen werden müssen. Wenn man einen `DatagramSocket` erzeugt, kann es zu einer Ausnahme (`SocketException`) kommen, wenn z.B. kein Port mehr frei ist, oder der Port an den man den Socket explizit binden will, schon belegt ist. Dieses müssen wir in unserem Code berücksichtigen.

Die Fehlerarten, die bei unserem Beispiel vorkommen können sind:

- **`SocketException`** wenn der Socket nicht erzeugt werden kann,
- **`UnknownHostException`** wenn die Zieladresse nicht, gefunden werden kann
- **`IOException`** wenn beim Senden oder Empfangen was schief geht.

Man kann alle Fehler separat behandeln, in dem man die folgenden `try-catch`-Blöcke einsetzt:

```
try {
    // code from above
} catch (SocketException se) {
    // handle SocketExceptions
} catch (UnknownHostException ioe) {
    // handle IOExceptions
} catch (IOException ioe) {
    // handle IOExceptions
}
```

Die Variable der `Exception` kann dazu benutzt werden, nähere Informationen zum genauen Fehler zu bekommen. Wenn man wissen will, wo genau die IOException aufgetreten ist, kann man sich mit

```
ioe.printStackTrace();
```

diese Information anzeigen lassen.

Als Antwort auf das obige `init`-Kommando sollte man folgende Nachricht vom Server bekommen:

```
(init l 1 before_kick_off)
```

Der Kontakt zum Server ist also hergestellt.

4 Ein Kommandozeilen-Client

Nach Abschnitt 3 sollte die Kommunikation mit dem Server kein Problem mehr sein und man kann endlich anfangen, einen Client zu schreiben, mit dem man beliebige Kommandos senden und alle Nachrichten des `rcssserver` empfangen kann. Als erstes bringt man die Kommunikation ein wenig in Form. Dazu nehmen wir eine Klasse `CommunicationChannel` und spendieren ihr die Methoden `send` und `receive`. Das kann folgendermaßssen aussehen:

```
public void send(String message) {
    try {
        // convert the message to a byte array
        byte[] buffer = message.getBytes();
        // create the packet from the buffer and set
```

```
        // the server's IP address and port
        DatagramPacket packet =
            new DatagramPacket(buffer, buffer.length,
                               serverAddress, serverPort);
        // send the packet
        socket.send(packet);
    } catch (IOException ioe) {
        System.err.println("Could not send packet.");
    }
}

// receive a message string from the server
public String receive() {
    // allocate buffer for server message
    byte[] buffer = new byte[1024];
    // create datagram packet for receiving
    DatagramPacket packet =
        new DatagramPacket(buffer, buffer.length);
    try {
        // receive packet, the method blocks until
        // a datagram arrives
        socket.receive(packet);
    } catch (IOException ioe) {
        System.err.println("Could not receive packet.");
    }
    // return the message as string
    return new String(packet.getData());
}
```

Im Konstruktor der Klasse muß noch der Socket erzeugt sowie die Adresse und Portnummer des Ziels festgelegt werden. `socket`, `serverAddress` und `serverPort` sind hierbei Instanzvariablen der Klasse.

```
CommunicationChannel(String serverName, int serverPort) {
    try {
        // construct datagram socket and
        // bind it to any available port on the local machine
        this.socket        = new DatagramSocket();
        // set target address
        this.serverAddress = InetAddress.getByName(serverName);
        // set target port
        this.serverPort    = serverPort;
    } catch (SocketException se) {
        System.err.println("Could not create datagram socket.");
        System.exit(1);
    } catch (UnknownHostException uhe) {
        System.err.println("Could not determine the server's IP.");
        System.exit(1);
    }
}
```

Jetzt kann man über die beiden Methoden Pakete empfangen und senden. Fehlt nur noch Code, der diese benutzt. Wir schreiben eine Klasse `Player`, die dies tun wird:

```
import java.io.*;
```

```
public class Player {
    public static void main(String[] args) {
        String host = args[0];
        int port    = Integer.parseInt(args[1]);
        CommunicationChannel com = new CommunicationChannel(host, port);

        BufferedReader in = new BufferedReader(
            new InputStreamReader(System.in));
        String command = "";
        while (!command.equals("exit")) {
            try {
                command = in.readLine();
            } catch (IOException ioe) {
                System.out.println("Keyboard read failed");
            }
            com.send(command);
            System.out.println(com.receive().trim());
        }
    }
}
```

Mit dieser Einstiegsversion, kann man ein Kommando eingeben und genau eine folgende Antwort lesen. Der Befehle `in.readLine()` und `com.receive()` sind blockierend. Das heißt, daß die Programmausführung unterbrochen wird, bis die Befehle abgearbeitet sind, in unserem Fall also bis eine Eingabe von der Tastatur mit der Eingabetaste abgeschlossen wurde, bzw. ein Paket eingetroffen ist. Die eigentliche Blockade findet im letzten Fall in der Klasse `CommunicationChannel` in der Zeile `socket.receive()` statt. Hier wird gewartet bis ein Paket eingetroffen oder ein Timer abgelaufen ist.

Mit dieser Version stoßen wir zuerst auf das Problem, daß der `rcssserver` ständig Informationen über die Spielsituation sendet und nicht immer nur genau eine Antwort auf genau ein Kommando. Wir benötigen also etwas, das die ganze Zeit auf hereinkommende Pakete wartet und diese ausgibt, unabhängig davon, ob ein Kommando gesendet wurde oder nicht. Diese Nebenläufigkeit kann man in JAVA durch sogenannte `Threads` erreichen. Diese arbeiten unabhängig voneinander. Wir schreiben daher eine Klasse `Receiver`, die von Thread erbt und damit unabhängig vom Senden der Befehle arbeiten kann. Sie kann z.B. so aussehen:

```
public class Receiver extends Thread {
    private CommunicationChannel com;

    public Receiver(CommunicationChannel com) {
        this.com = com;
    }

    // this method needs to be overwritten
    public void run() {
        while (true) {
            System.out.println(com.receive().trim());
        }
    }
}
```

Die `run`-Methode muß umgeschrieben werden. Sie enthält den Code, der beim Start des Threads ausgeführt wird und ist vergleichbar mit der `main`-Methode. Die `main`-Methode des `Player` muß dementsprechend erweitert werden:

```
public static void main(String[] args) {
    String host = args[0];
```

```
    int port    = Integer.parseInt(args[1]);
    CommunicationChannel com = new CommunicationChannel(host, port);

    // start a new Receiver Thread
    (new Receiver(com)).start();

    BufferedReader in = new BufferedReader(
        new InputStreamReader(System.in));
    String command = "";
    while (!command.equals("exit")) {
        try {
            command = in.readLine();
        } catch (IOException ioe) {
            System.out.println("Keyboard-read failed");
        }
        com.send(command);
    }
}
```

Man kann sich nun nach Starten des Clients mit

```
java Player localhost 6000
```

durch Eingeben von `(init OurTeamName (version 7.0))` beim Server anmelden und sieht sofort, daß nun fortlaufend alle Nachrichten des Servers angezeigt werden. Die Abarbeitung des `Receiver`-Threads wird immer unterbrochen, bis ein Paket eingetroffen ist.

Wenn man nun versucht, seinen Spieler mit dem Befehl `(move -10 10)` auf die Position -10 10 zu stellen, merkt man, daß nichts passiert. Dies liegt an der Art wie der `rcssserver` die Clients identifiziert. Da es zu umständlich ist, immer zu prüfen, von welcher Adresse die Nachricht an Port 6000 des Servers gesendet wurde, macht der Server für jeden Client einen neuen Socket auf, über den er die Nachrichten an den entsprechenden Client schickt. Der Server sendet die Pakete weiterhin an die gleiche Adresse, weswegen die Nachrichten weiter erscheinen. Sendet man nun selbst aber immer noch an Port 6000, so kann der Server das Kommando keinem Client mehr zuordnen und ignoriert das Paket. Der Anfangsport des Servers dient also nur der Kontaktaufnahme. Um an die neue Portnummer des Servers zu kommen, muß man die `receive` Methode der Klasse `CommunicationChannel` ein wenig modifizieren:

```
serverPort = packet.getPort();
```

Diese Zeile muß nach Lesen des Pakets und vor der `return`-Anweisung stehen. So wird bei jedem Lesen die aktuelle Portnummer eingetragen, die dann in der `send`-Methode zur Adressierung des `DatagramPacket` benutzt wird.

Jetzt sollte der Client wie gewünscht funktionieren. Ein wenig störend ist, daß die Kommandos durch die eintreffenden Servernachrichten “zerstückelt” werden. Sie werden trotzdem richtig eingelesen. Man muß sich nur merken, was man schon getippt hat.

5 Ein F-Jugend-Client

Die Kommandos von Hand einzugeben ist auf die Dauer ziemlich lästig. Deswegen wird hier beschrieben wie man einen einfachen Client implementieren kann, der autonom agiert. Er soll sich alleine auf dem Feld zurechtfinden und zielorientiert handeln. Im einfachsten Fall heißt dies: Er soll den Ball finden, hinlaufen und auf das oder besser ein Tor schießen. Ein einfacher Client (*PI4-SimpleClient*), der dies kann steht in der Rubrik [Software] zum Download bereit. Das dahinter stehende Konzept wird in diesem Abschnitt erläutert. Ungeduldige können den Client einfach mit

```
javac Player.java
```

übersetzen und mit

```
java Player localhost 6000 Player1
```

starten. Der `rcssserver` muß vorher gestartet sein. Um den Client mit der Arbeit beginnen zu lassen, startet man einen Monitor, baut mit diesem eine Verbindung zum Server auf und startet das Spiel mit dem *kick off*-Button.

Das Kernmodell des Clients besteht aus vier Hauptkomponenten:

1. Kommunikation mit dem Server
2. Verarbeitung der Informationen des Servers
3. Senden vom Kommandos
4. Treffen von Entscheidungen aufgrund der Informationen und Umwandlung in Kommandos

Die Kommunikation wird genau wie beim Kommandozeilenclient realisiert, weswegen im Folgenden nur noch die restlichen Komponenten beschrieben werden.

5.1 Informationsbeschaffung

Um sinnvolle Aktionen durchführen zu können, braucht man Informationen über die Umwelt und sich selbst. Diese Informationen bekommt man vom Server in Form verschiedener Nachrichten. In gewissen Zeitabständen werden diese an alle Clients gesendet. Es gibt `see`, `hear` und `sense_body` Nachrichten, die angeben was der jeweilige Client sieht, hört bzw. fühlt. Diese Informationen müssen in für den Client verwertbare Form gebracht werden.

Wir benutzen für unseren SimpleClient nur die visuellen Informationen also die `see`-Nachrichten. Die ankommenden Nachrichten werden in der `Actor`-Klasse vom Server empfangen und weiterverarbeitet. Dies geschieht auf Basis des Nachrichtentyps. Aus dem empfangenen String wird mit der Methode `event()` ein Ereignis (*event*) generiert. In unserem Fall entstehen aus `see`-Nachrichten `VisualEvents`. Die Klasse `VisualEvent` zerlegt die Nachricht und speichert die Informationen in einer Liste von `ObjectPerception`-Objekten. Hier werden alle wahrgenommenen Informationen über ein gesehenes Objekt eingetragen. Wichtig für unsere *kick-and-rush*-Strategie ist die Position des Balls, sowie die des gegnerischen Tors. In der Klasse `VisualInfo` werden daher auch nur die Teile der Nachricht in Objektinformationen umgewandelt, die sich auf diese beiden Objekte beziehen, also mit einem “b” oder “g” beginnen. Wir tragen den Namen sowie Information über Sichtwinkel und Entfernung in die `ObjectPerception` ein. Eventuell vorhandene Zusatzinformationen sind für unsere Strategie irrelevant und werden daher einfach weggelassen.

Zusätzlich beachten wir noch die `sense_body`-Nachrichten. Die enthaltene Information wird ignoriert, aber die Nachricht an sich wird benötigt, um die Aktionen des Clients mit dem Server zu koordinieren. Der Server teilt das Spiel in Zyklen der Länge 100ms ein. Pro Zyklus kann in der Regel ein Kommando gesendet werden. Die `see` Nachrichten werden aber alle 150ms gesendet. Eine Steuerung der Form: “Sehe, entscheide, handle” würde also pro Zyklus 50ms “verschenken”. Die `sense_body`-Nachrichten werden hingegen alle 100ms gesendet. Man kann also einen Steuerungsablauf der Form: “Fühle, entscheide, handle” wählen. So nutzt man die Möglichkeiten Kommandos zu senden optimal aus. Der große Nachteil dieser Variante ist jedoch, dass man sich nicht mehr nur nach dem aktuellen `VisualEvent` richten kann. Sieht man den Ball bei 40° und dreht sich 40° rechts, so muß man dies intern vermerken, da sonst evtl. bei einem vor der nächsten `see` Nachricht eintreffenden `sense_body` in der nicht mehr aktuellen `VisualEvent` noch die 40 steht. Obwohl man genau auf den Ball schaut, würde der Client sich nochmals um 40° drehen. Um dies zu vermeiden, muß man die eigene “Weltsicht” des Clients von den `see`-Nachrichten abkoppeln und diese nur noch zur Aktualisierung des internen Modells benutzen.

5.2 Kommandos

Auf Basis der vorhandenen Informationen kann der Client nun seine Entscheidungen treffen und muß diese in Kommandos an den Server umsetzen. Um im Quelltext die Kommandos nicht immer als Strings stehen zu haben, wurde von uns die Klasse `Actor` um Methoden erweitert, die eine Abstraktion von den Klartextkommandos bieten. bietet. Die Methoden dienen außerdem dazu, die Kommandos gleich zu verschicken. Es gibt Methoden um den Client anzumelden, zu positionieren, zu drehen sowie ihn laufen und kicken zu lassen. Dies ist selbstverständlich nur eine kleine Auswahl der möglichen Kommandos, so dass die Klasse noch zu erweitern ist.

5.3 Die Zentrale

Es muß eine zentrale Stelle geben, an der die Informationen hereinkommen und in Aktionen umgewandelt werden. Wir haben dafür den `Player`. Zuerst meldet er sich beim Server an, setzt sich auf eine zufällig gewählte Position in der linken Spielhälfte und wartet dann auf Nachrichten vom Server, die dann wie oben beschrieben in `VisualEvents` transformiert werden (der Client spielt immer nur von links nach rechts). Wenn in der Liste der Objektinformationen ein Ball und oder Tor vorhanden ist, so werden sich diese Informationen gemerkt. Bei Eintreffen eines `BodyEvents` wird ein Spielzug ausgeführt (siehe 5.1).

Ein Spielzug wird nach folgenden Logik ausgeführt:

1. Wenn der Ball noch nicht gesehen wurde, drehe 40° nach rechts, bis der Ball gefunden wird.
2. Ist der relative Winkel zum Ball groß, so drehe zu diesem hin.
3. Ist der relative Winkel klein und die Entfernung zum Ball noch zu groß, so laufe in Richtung Ball. Hierbei bestimme die Stärke des Laufs in Abhängigkeit von der Entfernung zum Ball.
4. Stimmt Richtung und Entfernung zum Ball und wurde das Tor noch nicht gesehen, so drehe 40° nach rechts, bis das Tor gesehen wird.
5. Wenn alles paßt, kicke in Richtung Tor.

Bei manchen Situationen wird man merken, dass sich der Client irrational zu verhalten scheint. Zum Beispiel kann es vorkommen, dass er zielstrebig auf den Ball zuläuft und diesen dann in die komplett falsche Richtung kickt. Diese Fehler haben meist etwas mit der Aktualität der Informationen zu tun, da ja nicht alle Objekte ständig gesehen werden. Die Fragestellung lautet hier: Soll man sich die visuellen Informationen merken, oder nicht? Dass es nicht ganz ohne “Erinnerung” geht, wurde schon in Abschnitt 5.1 deutlich (Differenz von 50ms). Wie man das innere Weltmodell des Clients gestaltet, ist einer der zentralen Punkte, wenn es an die Erweiterung des Clients geht.

6 Tips & Tricks

6.1 Speichern und Aktualisieren von Objektinformationen

Man bekommt pro `see`-Nachricht nur eine kleine Auswahl von Objekten im Sichtbereich des Spielers mitgeteilt. Meist orientiert man sich in Richtung des Balls. Dann gehen einem jedoch wichtige andere Informationen verloren wie zum Beispiel die Positionen der anderen Spieler. Es empfiehlt sich daher sich einmal gesehene Dinge für eine gewisse Zeit zu merken. Hierbei muß man zwischen Aktualität und Genauigkeit der Information abwägen. Merkt man sich die Positionen der Spieler länger, so können sich diese mittlerweile ganz woanders befinden und man plant den nächsten Zug aufgrund völlig falscher Daten.

Eine mögliche Strategie ist, daß man sich alle beweglichen Objekte merkt. Man speichert diese zusammen mit der Zeit der Wahrnehmung ab. Wenn die Wahrnehmungszeit von der aktuellen Zeit um mehr als eine vorgegeben Spanne abweicht, wird die Objektinformation gelöscht. Selbstverständlich werden ältere Daten mit neueren überschrieben und man wartet nicht erst, bis man sie "vergisst".

Ein kleineres Problem beim Merken der Spielerpositionen ist, daß man unterschiedliche Informationen in Abhängigkeit von der Entfernung vom Spieler sowie der Größe des aktuellen Gesichtsfeldes des Spielers erhält. Zur Speicherung wäre es am Besten wenn man immer Teamzugehörigkeit und Nummer des wahrgenommenen Spielers wüsste. Sind die Spieler jedoch weiter entfernt, so gehen zuerst die Informationen über die Nummer des Spielers, dann die über die Teamzugehörigkeit verloren, so daß es schwieriger ist, den Spieler zu identifizieren. Hierdurch kann es dazu kommen, daß der Spieler denkt, es seien mehr als 22 Spieler auf dem Platz. Wenn eben ein Gegner noch mit Nummer wahrgenommen wurde, im nächsten Zyklus die Nummer nicht mehr gesehen wird, so merkt man sich die Position des Spielers mit der Nummer und fügt einen neuen Spieler ohne Nummer in die Weltsicht ein. Bei zwei aufeinanderfolgenden Zyklen kann man diesen Defekt noch recht einfach beheben, da man den Spieler noch über die vergangene Information identifizieren kann, bei Drehungen des Clients kann dies jedoch nicht so einfach erfolgen.

Man sollte sich sehr genau überlegen, welche Informationen man wie lange merkt. Sonst kann es z.B. vorkommen, daß Spieler einem "Geisterball" hinterherjagen oder versuchen einen Paß an einen Phantomspieler zu spielen.

6.2 Positionsbestimmung

Fängt man an, sich die Objekte zu merken, wird sehr schnell klar, daß die vom Server übermittelten relativen Informationen (bezogen auf Spielerposition und dessen Blickrichtung) hierfür ungeeignet sind. Dreht sich der Spieler, so werden alle Winkelangaben falsch. Man sollte meinen, dies sei recht einfach zu korrigieren, da einem der vorige Winkel und die eigene Drehung bekannt ist. Hierbei macht einem jedoch der Server einen Strich durch die Rechnung. Das eigene Kommando zur Drehung wird nicht immer exakt ausgeführt; je größer die Drehung desto ungenauer. Noch schlimmer wird es, wenn der Spieler anfängt sich zu bewegen. Zum einen müßte man dann die Entfernungen und Winkel aktualisieren, zum anderen kennt man seine eigene Entfernungsänderung nicht. Das `dash` Kommando gibt nur die Stärke des Laufimpulses an und nicht die in der nächsten Runde zurückzulegende Distanz.

Die Lösung dieser Probleme stellt die Umwandlung der relativen Daten in absolute Positionen dar. Hierbei bietet es sich an, das interne Koordinatensystem des Servers zu benutzen, wie wir es schon beim `move` Kommando kennengelernt haben. Den Nullpunkt stellt der Mittelpunkt des Feldes dar. Schaut man von diesem aus in Richtung des gegnerischen Tors, so befindet sich der negative Teil der X-Achse hinter dem Client, vor ihm entsprechend der positive Teil. Rechts des Clients ist der positive, links der negative Bereich der Y-Achse. Die Wertebereiche sind $[-52,5;52,5]$ sowie $[-34,0;34,0]$. Da auf dem bzw. am Spielfeld viele unbewegliche Objekte (Tore, Flaggen) vorhanden sind, deren absolute Positionen bekannt sind, sollte es nun nicht schwer fallen, zwei Objekte anzupeilen und hieraus die eigene Position zu errechnen. Eine Kleinigkeit, die diesem Vorhaben im Weg steht, ist die Unschärfe mit der die visuelle Wahrnehmung belegt wird in Abhängigkeit der Entfernung der Objekte zum Spieler. Die Fehler bei der Wahrnehmung zweier Objekt summieren sich dann und führen zu einer sehr ungenauen Positionsfeststellung. Dies läßt sich verbessern, indem man statt zweier Objekte, eine Linie und ein Objekt nimmt. Zur Bestimmung der aktuellen Blickrichtung wird der mittgeteilte Schnittwinkel mit einer Außenlinie genommen. Dieser wird nicht mit Unschärfe belegt. Die Position kann dann in Abhängigkeit von einem unbeweglichen Objekt festgestellt werden. Um noch höhere Genauigkeit zu erreichen, kann man die Position in Abhängigkeit von mehreren Objekten bestimmen und dann ein entsprechend der Entfernung der Objekte gewichtetes Mittel bilden.

6.2.1 Berechnung des Winkels

Die vier Linien sind mit **t**op, **b**ottom, **r**ight und **l**eft bezeichnet. Der Schnittwinkel wird nicht nach einer einheitlichen Ausrichtung angegeben, sondern es wird immer ein Winkel betragsmäßig kleiner 90° angegeben. Das Vorzeichen gibt an aus welcher Richtung man auf die Linie schaut.

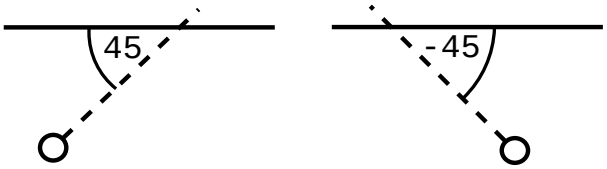

Abbildung 3: Beispiel für die obere Linie. Links: (a) Blickrichtung -45°; Rechts: (b) Blickrichtung -135°

Die Abbildung 3 zeigt ein Beispiel, bei dem ein nach rechts spielender Client die obere Linie sieht. Die gestrichelte Linie zeigt die Blickrichtung. Die angegebenen Winkel sind die Werte, die vom Server als Sichtwinkel ($w_{S1} = 45°$, $w_{S1} = -45°$) mitgeteilt werden. Als Null-Richtung wählen wir die X-Achse in Richtung des gegnerischen Tors. Die absolute Blickrichtung des Spielers bezogen auf diese Null-Richtung in Abbildung 3a beträgt also $w_{A1} = -45°$ bzw. $w_{A2} = -135°$ in Abbildung 3b. Zur Ermittlung der absoluten Blickrichtung aus Serverdaten muß im ersten Fall (Sichtwinkel $w_{S1} > 0$) der angegebene Wert von 0 abzogen werden, im zweiten Fall (Sichtwinkel $w_{S1} < 0$) von 180. Im zweiten Fall beträgt das Ergebnis also 225. Es ist sinnvoll die Winkel auf das Intervall [-180°,180°] zu normalisieren, d.h. man zieht im zweiten Fall vom Ergebnis 360 ab und erhält so die gewünschten -135°.

Die vom Server übermittelten Schnittwinkel sind unabhängig von der Spielrichtung. Spielt man in die umgekehrte Richtung (von rechts nach links), so muß man die Null-Richtung umkehren (0 ist nun in Richtung des anderen Tors). D.h. die korrekte Bestimmung des absoluten Blickwinkels erfolgt durch Vertauschung der Werte von denen man abzieht:

$$\begin{array}{lcccl} \omega'_{A1} & = & 180° - 45° & = & 135° \quad \text{(da rel. Winkel positiv)} \\ \text{bzw.} & & & & \\ \omega'_{A2} & = & 0° - (-45)° & = & 45° \quad \text{(da rel. Winkel negativ)} \end{array}$$

Gleiches gilt für alle anderen Linien. Hierbei ändern sich nur die Werte von denen man die wahrgenommenen Schnittwinkel abziehen muß.

6.2.2 Berechnung der Position

Hat man die absolute Blickrichtung bestimmt, kann man sich an die Bestimmung der absoluten Position anhand eines Objekts machen. Zweckmäßigerweise sollte man das am nächsten liegende, unbewegliche Objekt wählen.

Nehmen wir für ein Beispiel an, daß unser absoluter Blickwinkel von uns mit -19° bestimmt wurde und wir die Fahne `((f g r b ) 9.2 -41)` sehen. Der linke Pfosten des rechten Tors befindet sich also in einem Winkel ω_{rel} von -41° in einer Enfernung d von 9,2 Metern relativ zu unserer eigenen Position.

Der absolute Winkel ω_{abs} in dem die Flagge steht beträgt also $\omega_{abs} = -19 + (-41) = -60$. Nun kennen wir die Position der Flagge in Polarkoordinaten relativ zu unserer aktuellen Position und bestimmen einen Vektor im zweidimensionalen euklidischen Koordinatensystem (beachte: Die Vorzeichen der Y-Achse sind in unserem Fall vertauscht). Daher ist die Formel zur Umrechnung leicht modifiziert:

$$\begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix} = d * \begin{pmatrix} \cos \omega_{abs} \\ \sin \omega_{abs} \end{pmatrix} = 9,2 * \begin{pmatrix} 0,5 \\ -0,8660254 \end{pmatrix} = \begin{pmatrix} 4,6 \\ -7,9674337 \end{pmatrix}$$

Die Position des Torpfostens ist bekannt. Man braucht also nur den auf diesen Pfosten zeigenden Vektor von

dieser abzuziehen und erhält die eigene Position in absoluten Koordinaten:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 52.5 \\ 7.01 \end{pmatrix} - \begin{pmatrix} 4,6 \\ -7,9674337 \end{pmatrix} = \begin{pmatrix} 47,9 \\ 14.977434 \end{pmatrix}$$

Zum Vergleich: Für dieses Beispiel wurden die Befehle `(move 48 15)` und `(turn -20)` benutzt.

Literatur

[1] Martin Schader, Lars Schmidt-Thieme. Java: Eine Einführung. Springer-Verlag, 1999.