Bachelor Thesis: A Modern Turing Test
Bot detection in MMORPG's

Adam Cornelissen

Spring 2008

Contents

1 Introduction **5**
- 1.1 Bot consequences 5
- 1.2 Related work 7

2 Experiments **9**
- 2.1 Constraints 9
- 2.2 Strategy 9
- 2.3 Packet Analyzer 10
  - 2.3.1 Packets 11
- 2.4 Session 11
- 2.5 Feature Vector 11
- 2.6 Analyzer 12
- 2.7 Classification 13
  - 2.7.1 Vector Angles 13
  - 2.7.2 Neural Networks 14

3 Results **17**
- 3.1 Vector Angles 17
- 3.2 Neural Network 17

4 Discussion **21**

5 Acknowledgements **23**

6 Appendix **25**
- 6.1 Appendix A: Terms and abbreviations 25
- 6.2 Appendix B: Packet Structure 27
- 6.3 Appendix C: Packet Database Reader Sourcecode 28
- 6.4 Appendix D: Analyzer Sourcecode 30
- 6.5 Appendix E: Session Sourcecode 45
- 6.6 Appendix F: MovePacket Sourcecode 56
- 6.7 Appendix G: Run Sourcecode 58
- 6.8 Appendix H: VectorMath Sourcecode 63

Chapter 1

Introduction

Modern online multiplayer games have become increasingly popular with gamers all around the world. This applies in particular to the kind of games that can be played with hundreds to thousands of players simultaneously, the so called 'massively multiplayer online games', often simply referred to as MMOG or MMO (or, in the case of a Role-Playing Game, a MMORPG).

In these games players play as a virtual character taking on the role of a knight, priest, mage or some other heroic character to defeat enemies, to complete tasks (widely known as 'quests') or to compete in battles with other players.

While doing this, players receive items (such as gold or potions), new equipment (such as swords, shields and armor) or increased experience (how well your character is able to do a certain task) as a reward for their effort.

Not everyone plays according to the rules of the game though. A multitude of ways to cheat in games exist. In this bachelor thesis research I will try to find a method to automatically detect a kind of cheating where players use programs to automate their actions; the use of game bots.

Normally, a player character is operated by a human being who is playing the game. However, tools exist to let your character play automatically, without human interaction. A character not being operated by a real person but by a computer program is called a 'game bot' or simply a 'bot', which is an abbrevation for 'robot'.

Bots and more importantly the detection of bots are the topic of this research paper.

1.1 Bot consequences

As mentioned earlier players advance in the game through completing quests, competing with other players and by defeating enemies. In the beginning every player has the weakest equipment in the game and very little experience at different actions (bad offense, bad defence, no knowledge in how

to do something such as item crafting, ..) but throughout the game players can gain better equipment and higher experience.

To get higher experience players might choose to kill enemies. Certain types of enemies are often slain by using a similar strategy over and over again every time a player encounters such an enemy. While doing this, players gain experience which allows them to access newer and better features in the game.

People that enjoy playing the game but who simply lack the time to engage in the aforementioned repetitive gameplay might consider using bots for this repetitive and non-entertaining task.

Another reason why people use bots is to get better items and equipment. After slaying an enemy there is a change that it will drop a certain item. Thus, if an item has a 0.001% chance to be dropped by a certain enemy, the item will drop approximately once when slaying a thousand of these enemies. By using a bot, getting such an item is merely a matter of letting the bot run a long time instead of actually playing for hours.

Lack of time and/or lust to play the repetitive elements in the game and the will to easily get better in-game items and experience are the primary reasons for the use bots.

At first glance it might seem that there is no reason for disallowing the usage of bots to handle the repetitive gameplay or to get items. However, there are some downfalls in allowing bots.

The most apparent and logical reason for not allowing bots is that they break the game.

The use of bots can break the balance of power in the game. When killing monsters over and over again (much more than one would normally do) the chances of getting a powerful item which rarely drops increases significantly.

Also, because bots can handle things repetitively and often very quickly, they can outplay human-controlled characters, giving them an unfair advantage over legitimate players.

For almost the same reason as breaking the balance of power in the game, the use of bots can also break the economy of the game. In almost any game enemies can drop some kind of currency (cash, gold, zeny, ..). Letting a bot kill a lot of enemies gives the bot owner a lot of this currency, making him (a lot) more wealthier than the average player. The process of letting a bot kill enemies to gain more in-game currency is very common and known by gamers as 'gold farming' or 'farming'.

Another reason to disallow bots is that they have a negative influence on the gaming experience of normal players. The use of bots is viewed as 'unfair' behaviour by the majority of the gaming community.

The most shocking notion about bot might be this: It is known that sometimes in-game items and currency gathered by bots are sold for real money on auction websites. This shows that bots do not only have an

influence on the virtual economy, but also on the real economy.

For these (and other) reasons, companies running online games often disallow the use of bots in their Terms of Service (ToS) or End-User License Agreement (EULA).

1.2 Related work

Due to the fact that bots have several unwanted effects, it is relevant to detect people who are using them.

Most of the time bot detection is done 'by hand'. Players observe another player behaving in a strange way and 'report' this to one of the game maintainers (often called 'game master' or 'gm'). This person then has to go to the spot the potential bot was spotted and try to figure out if it is really a bot or not. This is a very tedious and time consuming process.

There has been more research in the field of bots, their detection and their prevention.

Kuan-Ta Chen has researched the possibility to detect bots with a traffic analysis approach. He proposes strategies to distinguish bots from human players based on their traffic characteristics such as command timing, traffic burstiness and reaction to network conditions. Kuan-Ta Chen has published his research in his paper *Identifying MMORPG Bots: A Traffic Analysis Approach*[1].

Roman V. Yampolskiy and Venu Govindaraju have published a paper titled *Embedded Noninteractive Continuous Bot Detection*[2]. They show how an embedded non-interactive test can be used to prevent bots from participating in online games.

The purpose of my research is to automatically detect whether or not people are using a bot. I will not only use the packets themselves but also the information they contain unlike Chen et al. The methods I use will not interfere with the gaming experience of regular players such as might be the case in the bot detection method presented by Yampolskiy and Govindaraju.

Chapter 2

Experiments

2.1 Constraints

In my research to bot detection I am (ofcourse) unable to handle all aspects of bots and the detection of them. The topic is a broad one. However, in the section 'Related Work' one can find more information on different aspects.

To succesfully finish my research I will have to apply some constraints and set some boundaries within which I will work:

- Game limitation: Ragnarok Online.
- Packet limitation: Most obvious packets will be dealt with.
- Relationship limitation: Between a normal player and a bot.

Research will be limited to only one game, Ragnarok Online. Creating a bot detection method for more games is too much work in this short timespan. More importantly, the detection method might be portable to different games (This will, however, not be investigated further in this paper).

The game has hundreds of different kinds of packets for every action in the game (move, drop item, take item, etc). However, not all packets are usable in my research. I will only pay attention to the packets that have an obvious chance of showing a relation between a bot and a real player, such as the movement packet (it is not unlikely that a bot will move different than a normal player).

With so many packets, many relationships between a player and its packets, and between a bot and its packets, will exist. I will not pay attention to every packet in the game, thus I will automatically limit myself with the number of possible relationships as well.

2.2 Strategy

The strategy for my research is roughly divided in these separate steps:

1. Study background information and related work.
2. Familiarity with the game and its network controls (packet handling, ..).
3. Writing a packet analyzer (in JAVA).
4. Compare packet analyzes of players and bots.
5. Detect relationships, patterns and/or differences between players and bots.
6. Create a detection method to detect bots.

To start, it is necessary to research some background information: What are online games, what are bots, why do people use bots, etcetera. This is essential to get a proper view of the domain in which the research will take place and to get a feeling of 'what is going on' around the topic of bots and bot detection.

Because I will use the game Ragnarok Online for my research, some familiarity with its engine (and in particular its packet handling related workings) is absolutely necessary.

Knowledge of these inner workings is also necessary to be able to write a packet analyzer. I will use this packet analyzer to capture the actions of players and bots.

After filtering out the necessary packets, I will built a 'feature vector' which will contain information about the behaviour of the player/bot. Examples of pieces of information that will be stored in that feature vector are average time between move packets, average move distance, loot time after a kill and more.

Later on I will compare the feature vectors from players with those from bots to find out if there are any oddities, patterns, relationships of differences between the values.

If all goes well, these findings will allow me to create a detection method that will detect a bot as being a bot and a player as being a player.

2.3 Packet Analyzer

To be able to distinguish bots from human players a way has to be found to somehow acquire the information on what is going on. This information is available in the form of packet traffic: The packets travelling between the Client (the player) and the Server (the game host). This packetstream includes packets for authentication, movement, messages, and more.

2.3.1 Packets

A packetstream can best be viewed in a hexadecimal format. An example of a 'received' (as seen by the server, thus sent by the client) packetstream in hexadecimal format:

9B	00	64	00	82	84	1E	00	00	F2	49	02	00	61
39	64	00	74	41	D5	4E	E4	A7	81	0C	01	1D	02
00	00	00	00	7D	00	89	00	35	00	3B	AF	81	0C
4D	01	4F	01	00	00	00	00	4F	01	00	00	00	00
4F	01	01	00	00	00	89	00	62	00	23	DE	81	0C
89	00	34	00	05	0D	82	0C	89	00	65	00	E7	3B
82	0C	89	00	61	00	C8	6A	82	0C	89	00	30	00

The packetstream can be saved to a file on the server. This file will be the input of the packet analyzer program to provide some insight as to what is being sent or received. The above example packetstream on its own says nothing to us, it is just a bunch of numbers. The packet analyzer however can work with this stream: It can figure out where a packet starts, what packet it is, if it is needed for further analysis, what parts of the packet to use and figure out where the packet ends. A more detailed description of the packet structure can be found in Appendix B.

2.4 Session

The featurevector is stored in a Session. A Session (sourcecode included in Appendix E) is used to store information gained directly from the packet stream file and to store information that was derived from those values.

From these values the feature vector (see also the next section) is constructed.

2.5 Feature Vector

The feature vector contains several values which will be used to detect whether we're dealing with a bot or not. These values are:

- Number of packets per second.
- Average time between 'Move' packets.
- Average distance between coordinates of 'Move' packet.
- Average distance between just the X-coordinates of 'Move' packet.
- Average distance between just the Y-coordinates of 'Move' packet.

- Average time between 'Take Item' packets.
- Average time between 'Change Direction' and 'Take Item' packet.
- Average time between a kill and the first following 'Take Item' packet.
- Number of 'continuous' attacks per second.

As the name implies these values are stored in a vector within the Session.

2.6 Analyzer

The Analyzer program (sourcecode included in Appendix D) is used to analyze packetstream files.

'To analyze' in this context means that the program will try to extract relevant data out of the packetstream file. It will store this information in what is commonly refered to as a 'feature vector'. This feature vector (and other possibly relevant information) is stored in a Session.

An analysis roughly goes as follows: The analyzer gets a packetstream file or a directory filled with packetstream files. For all of these files it will create a Session (some files may include multiple play sessions so you can get multiple sessions from one file actually), read the packetstream, extract and derive information for relevant packets, create a feature vector and store this information in variables in the Session.

As is obvious from the above, not all packets are used to extract and derive information. What the program should do with the current packet depends on the kind of packet. For some packets the program does nothing at all (eg, the 'Ticksend' packet) while other packets require a more in-depth analysis or some other action.

Some packets that require action:

Packet 0x0000 - Null End of file has been reached. Stop.

Packet 0x009B - Want To Connection A new connection has been made. A new session should be created. A 'Want To Connection' packet normally appears at the start of a packetstream file, but it can also appear multiple times. Either way, a new session with a new featurevector is created.

Some packets that require more analysis:

Packet 0x0085 - Change Direction It saves the current time to a variable to be used later (See 'Take Item' (0x00F5) packet).

Packet 0x00A7 - Move The information about the coordinates as well as the time the packet was sent is saved in the Session to be used later to calculate several (possibly) interesting features.

Packet 0x00F5 - Take Item Used to calculate the difference with a Change Direction packet, the average time between different Take Item packets and the average time to take an item after a kill.

Packet 0x0190 - ActionRequest The ActionRequest packet is used for multiple actions: Attack Once, Attack Continuous, Sit and Stand. Both of the Attack actions are needed to calculate the number of 'single' attacks per second and the number of 'continuous' attacks per second in the feature vector.

Packet 0x009B - Want To Connection Besides simply creating a new Session object when a 'Want To Connection' packet is found, the analyzer also has to retrieve who exactly is connecting. Although it is not used in the feature vector it is nice to see who's data you are working with.

A more elaborate view on how exactly information is retrieved from the packets can be read in the Analyzer sourcecode (see Appendix).

After reading and cleaning the packet database file, reading the packet-stream files, extractinging information from them and creating feature vectors it is time to put them to good use: A way has to be found to classify bots as bots and players as players.

2.7 Classification

The classification method determines whether or not an unknown feature vector came from a bot or from a human player. I have used two classification methods: Vector Angles and Neural Networks.

2.7.1 Vector Angles

Detection Method

A feature vector looks just like a normal vector. This means that the properties of vectors also work for feature vectors. In the Vector Angles classification method we use these properties. The Vector Angles classification method uses the property that different vectors point to a different direction. The angle between two vectors can be calculated using several formulas:

$u \, . \, v = |u| \, |v| \cos(a)$

where

inproduct: $u \, . \, v = u_1 v_1 + u_2 v_2 + u_3 v_3 + \ldots + u_n v_n$

length: $|u| = \sqrt{u_1^2 + u_2^2 + u_3^2 + ... + u_n^2}$

We can calculate the length of two vectors and their inproduct. From these we can calculate the angle between them by using the first of the above formulas. The ability to calculate angles between different vectors can be used to determine if a feature vector came from a player or a bot. To do this, we need a collection of feature vectors of which we know where they came from (player or bot). We then calculate the angle between the unknown feature vector and the known feature vectors. We then select the vector of which the angle was the smallest; The unknown feature vector looks the most like this feature vector. If this feature vector came from a bot then the unknown feature vector also came from a bot, if the feature vector came from a player then the unknown feature vector also came from a player.

Implementation

The sourcecode of an implementation of the Vector Angle method can be found in Appendix H.

2.7.2 Neural Networks

A neural network is a model based on biological neural networks.

'A biological neural network describes a population of physically interconnected neurons or a group of disparate neurons whose inputs or signalling targets define a recognizable circuit. The interface through which they interact with surrounding neurons usually consists of several dendrites (input connections), which are connected via synapses to other neurons, and one axon (output connection). If the sum of the input signals surpasses a certain threshold, the neuron sends an action potential (AP) at the axon hillock and transmits this electrical signal along the axon.' [1]

The neural network model we use mimics this process; the neurons and the communication between them is simulated by a computer program.

A regular neural network consists of three layers; an input layer, a hidden layer and an output layer. In figure 2.1 such a network is shown. It has four neurons on the input layer, six on the hidden layer and three on the output layer.

[1] Description taken from: http://en.wikipedia.org/wiki/Biological_neural_network

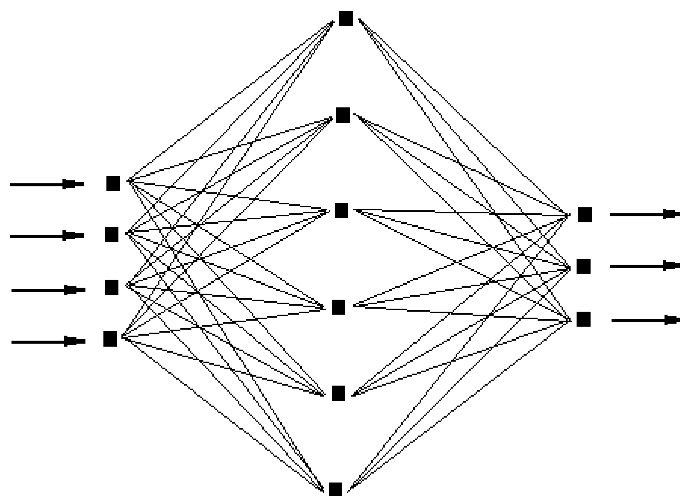

Figure 2.1: A neural network (cortex.snowcron.com)

Implementation

In our implementation the input neurons were set to a value in the feature vector. Since we had nine attributes in our feature vector, we have nine neurons in our input layer. We only need to know if a feature vector retrieved was from a player or a bot, so we only need one neuron in the output layer giving us a value between 0 and 1 indicating how much the behaviour of the analyzed file corresponds with botlike-behaviour. The hidden layer consisted of the same number of neurons as the input layer (nine).

The neurons in the input layer send their values to the neurons in the hidden layer. Each neuron in the hidden layer has a certain threshold. Only when the sum of the input values the neuron receives from the input layer surpasses this threshold the neuron sends its value to the output layer. The height of this threshold has to be learned by the neural network. A neural network is trained using a training set consisting of feature vectors of which we know they came from either bots or players. In other words, we know what the value of the output layer should be (it should be '1' for a bot, '0' for a player). By repeatedly giving the neural network the feature vectors on its input layer and the corresponding 0 or 1 value on the output layer the network learns the best threshold for each of the neurons on the hidden layer. By adjusting the thresholds the neural network error rate for the training set decreases.

Chapter 3

Results

3.1 Vector Angles

The Vector Angles classification method did not give any good results. After inspection it seemed that two different vectors could get an angle of zero. An example illustrates this:

x1 = {10,20,3000}
x2 = {10,10,3000}

length x1 = $\sqrt{(10^2 + 20^2 + 3000^2)} = \sqrt{9000500} = 3000.08$
length x2 = $\sqrt{(10^2 + 10^2 + 3000^2)} = \sqrt{9000200} = 3000.03$

x1 normalized = {10/3000.08, 20/3000.08, 3000/3000.08} = {0.0033, 0.0067, 1.0000}
x2 normalized = {10/3000.03, 10/3000.03, 3000/3000.03} = {0.0033, 0.0033, 1.0000}

inproduct x1 and x2: 0.0033*0.0033 + 0.0067*0.0033 + 1.0000*1.0000 = 0.00001 + 0.00002 + 1.0000 = 1.0000..

As shown in the above example, the inproduct of the vectors x1 and x2 is 1 (round off errors) despite the fact that the second attribute of vector x2 is significantly bigger (it's two times as much as the corresponding value in vector v1!). This classification method did not show to work good enough for our problem.

3.2 Neural Network

We trained our network using 40 feature vectors. 20 of these came regular player sessions and 20 came from bots sessions. The minimal playtime for

each of these sessions was 10 minutes. The network was trained until the error rate was below 0.001. We then took 10 feature vectors which were not in the training set to test our network. Five of these were bots and five were players. The results:

Feature vector 1:	0,0030
Feature vector 2:	0,0026
Feature vector 3:	0,9997
Feature vector 4:	0,9996
Feature vector 5:	0,0067
Feature vector 6:	0,0026
Feature vector 7:	0,9997
Feature vector 8:	0,9997
Feature vector 9:	0,0026
Feature vector 10:	0,9996

Table 3.1: Output activations for players/bots

These numbers indicate how 'botlike' a feature vector was. As you can see vectors 3, 4, 7, 8 and 10 are pretty close to the value '1', indicating they probably came from bots. The vectors 1, 2, 5, 6 and 9 are almost '0', indicating they probably came from players. The results we got are accurate with reality: The vectors the network identified as bots all came from bots, the vectors the network identified as players all came from players.

Besides this test, another test method was also applied. Since the dataset is relatively small, we decided to create a trainingset of 49 sessions, and a test-set of 1 single session. This procedure is repeated 50 times (leaving out every player/bot once). The results are presented in table 3.2. The results show extremely good classification results, except for Player 13, Player 17, Bot 19 and Bot 22. Some further investigation learned that both player 13 and 17 used a special levelling technique (called mobbing) in which large groups of monsters are collected and killed together. It is possible that the character's movement during the collection phase has a mechanical touch and misinterpreted by the network. Both Bot 19 as 22 appeared to have rested long periods during their sessions. Since there was one player who spent her complete session sitting, it is possible that in those cases not enough feature information was present to enable proper determination.

All scores combined yield an average score of 94%. Thresholding at 0.5 would mean 4 errors in 50, so 92%.

test	activation	test	activation
Player 1	0.0004	Bot 1	0.9999
Player 2	0.0006	Bot 2	0.9997
Player 3	0.0005	Bot 3	0.9999
Player 4	0.0010	Bot 4	0.9999
Player 5	0.0003	Bot 5	0.9999
Player 6	0.0053	Bot 6	0.9999
Player 7	0.0003	Bot 7	0.9971
Player 8	0.0003	Bot 8	1.0000
Player 9	0.0002	Bot 9	0.9994
Player 10	0.0013	Bot 10	0.9999
Player 11	0.0004	Bot 11	0.9998
Player 12	0.0007	Bot 12	0.9999
Player 13	0.9975	Bot 13	1.0000
Player 14	0.0004	Bot 14	1.0000
Player 15	0.0007	Bot 15	0.9997
Player 16	0.0010	Bot 16	0.9999
Player 17	0.7023	Bot 17	0.9998
Player 18	0.0002	Bot 18	0.9999
Player 19	0.0007	Bot 19	0.0098
Player 20	0.0008	Bot 20	0.9999
Player 21	0.0008	Bot 21	0.9999
Player 22	0.0009	Bot 22	0.6731
Player 23	0.0001	Bot 23	0.9999
Player 24	0.0006	Bot 24	0.9999
Player 25	0.0007	Bot 25	0.9841

Table 3.2: Output activations for players/bots

Chapter 4

Discussion

Although the method presented in this paper is only validated in a small scale experiment, the results look promising. Knowing that (for now) only simple features were used and the session times where short, there appears to be enough room for improvements. A valuable feature might be the average angle between moves: bots often appear to make strange course corrections, while human players tend to move smoother, according to a plan.

The set of features can be changed to stay a step ahead of the bot creators: If they adjust their bots to prevent a detection feature one could simply pick another feature to use in the feature vector.

Future research might consider the asymmetry in the decision making: having false positives (recognizing players as a bots) is worse then having false negatives (bots are recognized as human players). For example, if the network's outcome is connected to an automatic jailing system[1] people might be rightfully upset if they are jailed without doing anything wrong.

[1]Ragnarok Online has a jail, in which players are placed if they violate the rules.

Chapter 5

Acknowledgements

I would like to thank Franc Grootjen for his dedicated help with my research, paper and presentation. Without him this research would not have been possible. Thank you very much!

I would also like to thank the following people Alchemist, Aligner, Bam-Bam, buf priest, Cartimandua, dark acolyte, Datsylel, Ewan, Frea, Henkie, icarus, ilene, Kugutsu, maat van Ruben, Ophelia, Othello, Pebbles, Raio, Rhytor, Ruin, Rydia, Scarva, sniperboy, sniperboy's priest, and Xena for helping out with the experiments.

Chapter 6

Appendix

6.1 Appendix A: Terms and abbreviations

- Bot - A computer controlled character.
- Character - The virtual representation of a player or bot in an online game.
- Client - Een programma dat gebruikt maakt van de diensten van een server.
- EULA - Abbrevation of 'End-User License Agreement'.
- End-User License Agreement - An End User License Agreement is a legal contract between the author/publisher of a software application and the user of that application.
- Farming - Killing enemies over and over again to gain wealth.
- Game Master - A Game Master is a player who acts as organizer, arbitrator, and officiant in rules situations.
- GM - Abbrevation of 'Game Master'.
- Looting - Picking up items from killed enemies.
- Packet - A packet is the unit of data that is routed between an origin and a destination on the Internet.
- Player - A character controlled by a human being.
- MMOG/MMO - Abbrevation of 'Massively Multiplayer Online Game'.
- Server - Een computer/programma met services voor externe client-computers of toepassingen.

- Terms of Service - A list of the terms that must be agreed on by a user of a particular service.
- TOS - Abbrevation of 'Terms of Service'

6.2 Appendix B: Packet Structure

Per packet the following information is saved: a 4 byte tag (0x4144414D, 'ADAM' in Little Endian format), 4 bytes time in seconds, 4 bytes time in microseconds and the actual packet itself.

To be able to make sense of all the numbers of the packet information the analyzer has to know what they all mean. Some sort of 'Packet Database' to know what packets exist and how the different packets are build up is necessary.

Among the Ragnarok Online sourcecode, there is a file (packet_db.txt) in which all packets are described for every version of the game. A small example of this file:

...
0x0084,2
0x0085,5,walktoxy,2
0x0086,16
0x0087,12
0x0088,10
0x0089,7,actionrequest,2:6
0x008a,29
0x008b,2
0x008c,-1,globalmessage,2:4
...

Every line starts with a hexidecimal number. This number is called the 'packet id' and is used to identify a packet.

The number following the packet id is the 'packet length'. The packet length shows how many bytes (including the packet id) are used. A value of '-1' indicates a dynamic length. If a packet has a dynamic lenght the next two following bytes will give the size.

The next two tokens are optional and represent a function name (such as 'walktoxy') and its arguments (in the 'walktoxy' example this argument is '2'). The functions are looked up in the Ragnarok sourcecode file 'clif.c'.

The file 'clif.c' is what the server uses to handle packets. Not all packets have functions in the database file. The reason that some packets do have a function name followed by their arguments is because Gravity Corp (the creator of Ragnarok) changed the format of these packages a lot, probably partly to make matters difficult for private servers.

The packet information for all versions of the game is contained within a single packet database textfile. A small Java program (PacketDBReader.java) was written which can extract the packets from the latest game version out of the database file. The sourcecode of this program is included in Appendix C.

6.3 Appendix C: Packet Database Reader Sourcecode

```
package nl.ru.ai.turingtest.analyzer;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Hashtable;
import java.util.StringTokenizer;

/**
 * @author Adam Cornelissen
 * @version June 20, 2008
 */

public class PacketDBReader {
  /**
   * Reads in a packet database file and removes 'old'
       entries. Only entries
   * from the most recent game version are saved.
   *
   * @param fileName
   *            Filename of the packet database file
   * @return A hashtable with the packet id as key and
       packetinfo (length,
   *          <function>, <arguments>) as value
   */
  public static Hashtable<Integer, ArrayList<String>>
     read(String fileName) {

    Hashtable<Integer, ArrayList<String>> packetList =
        new Hashtable<Integer, ArrayList<String>>();

    try {
      BufferedReader in = new BufferedReader(new
         FileReader(fileName));
      String line = "";
      while ((line = in.readLine()) != null) {
        if (line.startsWith("0x")) {
          StringTokenizer packetlisttokenizer = new
              StringTokenizer(
```

```
          line);
        ArrayList<String> packetInfo = new ArrayList
          <String>();

        // Get the packet number, eg 0xAAAA
        Integer packetId = Integer.decode(
          packetlisttokenizer
            .nextToken(","));

        /*
         * Search for more tokens: - Length - Name (
            optional) - More
         * info (optional)
         */
        while (packetlisttokenizer.hasMoreTokens())
          packetInfo.add(packetlisttokenizer.
             nextToken());

        packetList.put(packetId, packetInfo);
      }
    }
    in.close();
  } catch (IOException e) {
    System.err.println("File input error");
    System.exit(1);
  }
  return packetList;
 }
}
```

6.4 Appendix D: Analyzer Sourcecode

```
package nl.ru.ai.turingtest.analyzer;

import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.util.ArrayList;
import java.util.Hashtable;
import java.util.List;

/**
 * @author Adam Cornelissen
 * @version June 24, 2008
 */
public class Analyzer {
  private int bytesRead;

  /*
   * The packet database containing all the
       information
   * regarding RRO packets.
   */
  private Hashtable<Integer, ArrayList<String>>
     packets;

  /**
   * Analyser constructor. Reads in the packet
       database file.
   *
   * @param packetDbFileName
   *            Name of the packet database file
   * @param string
   */
  public Analyzer(String packetDbFileName) {
    packets = PacketDBReader.read(packetDbFileName);
  }

  /**
   * Analyzes the named file, if file is a directory
```

```
     it
   * will analyze all files in it (recursively)
   *
   * @param fileName
   *            Name of file (or directory of files)
     to
   *            analyze.
   * @return List of sessions stored in the file(s)
   */
  public List<Session> analyze(String fileName) {
    return analyze(new File(fileName));
  }

  /**
   * Analyzes a file, if the file is a directory it
     will
   * analyze all files in it (recursively)
   *
   * @param file
   *            File (or directory of files) to
     analyze
   * @return List of sessions stored in the file(s)
   */
  public List<Session> analyze(File file) {
    List<Session> sessions = new ArrayList<Session>();
    if (file.isFile()) {
      /*
       * Read the packetstream
       */
      ByteBuffer byteBuffer = readFile(file);

      /*
       * Process the packetstream:
       */
      sessions.addAll(processPacketStream(byteBuffer,
         file));

    } else if (file.isDirectory()) {
      if (!file.getName().equals(".svn")) {
        File[] files = file.listFiles();
        for (File f : files)
          sessions.addAll(analyze(f));
      }
    } else {
```

```
      System.err.println("File '"+file.getAbsolutePath
          ()
          + "' not found");
    }
    return sessions;
  }

  /**
   * Read in the saved packet stream in a bytebuffer
   *
   * @param file
   * @return bytebuffer containing the file's contents
   */
  private ByteBuffer readFile(File file) {
    try {
      ByteBuffer byteBuffer =
        ByteBuffer.wrap(getBytesFromFile(file));
      byteBuffer.order(ByteOrder.LITTLE_ENDIAN);
      return byteBuffer;
    } catch (IOException e) {
      e.printStackTrace();
      return null;
    }
  }

  private List<Session> processPacketStream(
      ByteBuffer byteBuffer,
      File file) {
    double packetTime = 0;
    List<Session> sessions = new ArrayList<Session>();
    Session session = new Session();
    while (byteBuffer.hasRemaining()) {
      /*
       * Detect and handle the 'tag' and 'time'
       * information of a packet.
       */
      packetTime = extractTimeInformation(byteBuffer,
          packetTime);
      if (!byteBuffer.hasRemaining())
        break;
      /*
       * Get the packet id
       */
      int packetId = byteBuffer.getShort();
```

```
if (packetId == 0x00) // End of file
  break;
bytesRead = 2;

/*
 * Check the packet id
 */
if (!packets.containsKey(packetId)) {
  System.out.println("Packet Id "
      +Integer.toHexString(packetId)
      + " in file '" + file.getAbsolutePath()
      + "', offset 0x"
      + Integer.toHexString(byteBuffer.position
          ())
      + " not found in packet database!");
  break;
}
/*
 * Get packet info
 */
ArrayList<String> packetInfo = packets.get(
    packetId);
if (packetInfo.isEmpty()) {
  System.out.println("Packet Info is empty!");
  System.exit(1);
}
/*
 * From the packet info, retrieve the size of a
     packet.
 * If this value is '-1' the packet has a
     variable size.
 * The next two bytes (Short) contain this size.
 */
int packetSize = Integer.parseInt(packetInfo.get
    (0));
if (packetSize == -1) {
  packetSize = byteBuffer.getShort();
  bytesRead += 2;
}
/*
 * If we have a Want To Make Connection packet
     we have
 * to store old session and create a new one.
 */
```

```
      if (packetId == 0x9b) {
        if (session.getPacketsTotal() != 0) {
          session.setTimePlayStop(packetTime);
          sessions.add(session);
        }
        session = new Session();
      }
      /*
       * Handle packet
       */
      session.countPacket();
      if (!session.isTimePlayStartSet())
        session.setTimePlayStart(packetTime);
      handlePackets(packetId, byteBuffer, session,
          packetTime);
      /*
       * Next packet
       */
      byteBuffer.position(byteBuffer.position()+
          packetSize-bytesRead);
    }
    /*
     * Add last collected session
     */
    if (session.getPacketsTotal() != 0) {
      session.setTimePlayStop(packetTime);
      sessions.add(session);
    }
    return sessions;
  }

  private double extractTimeInformation(ByteBuffer
      byteBuffer,
      double oldPacketTime) {
    double packetTime = oldPacketTime;
    /*
     * Extract the support information from the
     * packetstream: A packet normally starts
     * with the time (seconds and microseconds)
     * and the actual packet. However, the server
     * sometimes sends multiple commands in a single
     * packet. The solution to this phenomenon is
     * the use of a tag. The tag 'ADAM' (0x4D414441
     * in hex, reversed due to LITTLE_ENDIAN) is put
```

```
   * in front of the time. Thus a packet is preceded
   * as follows:
   *
   * 4 bytes 4 bytes 4 bytes tag seconds
       microseconds
   * 41 44 41 4D ss ss ss ss mm mm mm mm <packet>
   */

  while (byteBuffer.hasRemaining()) {
    int tag = byteBuffer.getInt();
    if (tag == 0x4D414441) {
      // Tag found, retrieve the time
      double timePacketSeconds = byteBuffer.getInt()
          ;
      double timePacketMicroSeconds=byteBuffer.
          getInt();
      packetTime = timePacketSeconds*1000L
                 + timePacketMicroSeconds/1000;
    } else {
      // No tag found, so it must be another packet
      byteBuffer.position(byteBuffer.position() - 4)
          ;
      return packetTime;
    }
  }
  return packetTime;
}

private void handlePackets(int packetId, ByteBuffer
    byteBuffer,
    Session session, double packetTime) {
  String information;
  switch (packetId) {
  case 0x00:
    information = "End of Session";
    break;
  case 0x007D:
    information = "Load End Ack";
    break;
  case 0x0085:
    session.setLastChangedDirectionTime(packetTime);
    information = "Change dir";
    break;
  case 0x0089:
```

```
    information = "Ticksend";
    break;
  case 0x008C:
    information = "Get Char Name Request";
    break;
  case 0x0096:
    information = "Whisper";
    break;
  case 0x0090:
    information = "NPC Selected";
    break;
  case 0x009B:
    information = handlePacket009B(byteBuffer,
       session);
    break;
  case 0x009F:
    information = "Use Item";
    break;
  case 0x00A7:
    information = handlePacket00A7(byteBuffer,
       session,packetTime);
    break;
  case 0x00A9:
    information = "Equip Item";
    break;
  case 0x00B2:
    information = "Restart";
    break;
  case 0x00B8:
    information = "NPC Select Menu";
    break;
  case 0x00B9:
    information = "NPC Next Clicked";
    break;
  case 0x00BB:
    information = "Status Up";
    break;
  case 0x00C1:
    information = "How Many Connections";
    break;
  case 0x00C5:
    information = "NPC Buy/Sell Selected";
    break;
  case 0x00C8:
```

```
      information = "NPC␣Buy␣List␣Send";
      break;
    case 0x00C9:
      information = "NPC␣Sell␣List␣Send";
      break;
    case 0x00F5:
      information = handlePacket00F5(byteBuffer,
          session,packetTime);
      break;
    case 0x0116:
      information = "Drop␣Item";
      break;
    case 0x0146:
      information = "NPC␣Close␣Clicked";
      break;
    case 0x014D:
      information = "Guild␣Check␣Master";
      break;
    case 0x014F:
      information = "Guild␣Request␣Info";
      break;
    case 0x0190:
      information = handlePacket0190(byteBuffer,
          packetTime, session);
      break;
    case 0x021D:
      information = "got0x21D:␣Anti-Cheat";
      break;
    default:
      information = "--␣Unknown␣Packet";
      break;
    }
    // System.out.println(information);
  }

  private String handlePacket00A7(ByteBuffer
      byteBuffer,
      Session session, double packetTime) {
    /*
     * WalkToXY packet is 8 bytes: bytes 0,1: Variable
     * cmd (packetId), used to find the packet in the
     * database.
     * bytes 2,3,4: Not used (skipped)
     * bytes 5,6: Used to calculate x
```

```
    * bytes 6,7: Used to calculate y
    *
    * cmd = RFIFOW(fd,0);
    * x =RFIFOB(fd,5 )*4+(RFIFOB(fd,5+1) >> 6);
    * y =((RFIFOB(fd,5+1)&0x3f)<<4)+(RFIFOB(fd,5+2)
        >>4);
    */
   int x, y;
   int moveByte5, moveByte6, moveByte7;

   // Bytes 0 and 1 (cmd) were initialized previously
        as packetId

   // Bytes 2, 3 and 4 are skipped
   byteBuffer.get();
   byteBuffer.get();
   byteBuffer.get();

   // Bytes 5, 6 and 7 initialization
   moveByte5 = byteBuffer.get() & 0xff;
   moveByte6 = byteBuffer.get() & 0xff;
   moveByte7 = byteBuffer.get() & 0xff;

   // Calculate coordinates according to 'clif.c'
      code
   x = moveByte5 * 4 + (moveByte6 >> 6);
   y = ((moveByte6 & 0x3f) << 4) + (moveByte7 >> 4);

   // Update the number of bytes read
   bytesRead += 6;

   // Store information regarding position and time
   session.add(new MovePacket(x, y, packetTime));

   return " Walking to (" + x + "," + y + ")";
}

private String handlePacket00F5(ByteBuffer
    byteBuffer,
     Session session, double packetTime) {
   /*
    * map_object_id =
    * RFIFOL(fd,packet_db[sd->packet_ver][RFIFOW(fd
        ,0)].pos[0]);
```

```
     * map_object_id = RFIFOL(fd,4);
     */
    // Bytes 0 and 1 were initialized previously as
       packetId
    // Bytes 2 and 3 are skipped
    byteBuffer.get();
    byteBuffer.get();

    // Get the map_object_id (item)
    int item = byteBuffer.getInt();

    bytesRead += 6;
    session.addTakeItemTime(packetTime);

    if (session.isLastChangedDirectionTimeSet())
      session.addChangeDirectionTakeItemInterval(
         packetTime-session.getLastChangeDirectionTime
            ());

    if (session.isAttacking()) {
      session.addAttackFirstLootInterval(packetTime
          - session.getLastAttackTime());
      session.setAttacking(false);
    }
    return "Take Item " + item + "   Time ChDir-Pickup
       : "
       + (packetTime-session.
          getLastChangeDirectionTime());
}

  private String handlePacket0190(ByteBuffer
     byteBuffer,
      double packetTime, Session session) {
    String result;
    /*
     * 0x0190,19,actionrequest,5:18 void
        clif_parse_ActionRequest :
     *
     * clif_parse_ActionRequest_sub(sd,
     * RFIFOB(fd,packet_db[sd->packet_ver][RFIFOW(fd
        ,0)].pos[1]),
     * RFIFOL(fd,packet_db[sd->packet_ver][RFIFOW(fd
        ,0)].pos[0]),
     * gettick());
```

```
 *
 * clif_parse_ActionRequest_sub(sd,RFIFOB(fd,18),
    RFIFOL(fd,5),
 * gettick())
 */

// Bytes 0 and 1 were initialized previously as
   packetId
// Bytes 2, 3 and 4 are skipped
byteBuffer.get();
byteBuffer.get();
byteBuffer.get();

// Target ID (Byte 5,6,7,8)
int targetId;
targetId = byteBuffer.getInt();

// Bytes 9,10,11,12,13,14,15,16,17 are skipped
byteBuffer.get();
byteBuffer.get();
byteBuffer.get();
byteBuffer.get();
byteBuffer.get();
byteBuffer.get();
byteBuffer.get();
byteBuffer.get();
byteBuffer.get();

// Action ID (Byte 18)
int actionId;
actionId = byteBuffer.get() & 0xff;

switch (actionId) {
case 0x00: // once attack
  session.countOnceAttack();
  result =
    "Attack - Once ("
    + Long.toHexString(targetId)
    + ")";
  session.setAttacking(true);
  session.setLastAttackTime(packetTime);
  break;
case 0x07: // continuous attack
  session.countContinuousAttack();
```

```
      result =
        "Attack␣-␣Continuous␣("
        + Long.toHexString(targetId)
        + ")";
      session.setAttacking(true);
      session.setLastAttackTime(packetTime);
      break;
    case 0x02: // sitdown
      result = "Sit␣down";
      break;
    case 0x03: // standup
      result = "Stand␣up";
      break;
    default:
      result = "?";
      break;
    }
    bytesRead += 17;
    return result;
  }

  private String handlePacket009B(ByteBuffer
     byteBuffer,
      Session session) {
    /*
     * cmd = RFIFOW(fd,0);
     * account_id = RFIFOL(fd,packet_db[packet_ver][
        cmd].pos[0]);
     * char_id = RFIFOL(fd,packet_db[packet_ver][cmd].
        pos[1]);
     * login_id1 = RFIFOL(fd,packet_db[packet_ver][cmd
        ].pos[2]);
     * client_tick = RFIFOL(fd,packet_db[packet_ver][
        cmd].pos[3]);
     * sex = RFIFOB(fd,packet_db[packet_ver][cmd].pos
        [4]);
     *
     * cmd = RFIFOW(fd,0);
     * account_id = RFIFOL(fd, 4);
     * char_id = RFIFOL(fd,9);
     * login_id1 = RFIFOL(fd, 17);
     * client_tick = RFIFOL(fd, 18);
     * sex = RFIFOB(fd, 25);
     *
```

```
 * Strange information: Reading 4 bytes from
    position
 * 17 and reading 4 bytes from position 18?!
    However,
 * we're not interested in these values so leave
    it be.
 *
 * Only interested in 'account_id' (accountId) and
 * 'char_id' (charId). The rest is not saved.
 */

// Bytes 0 and 1 were initialized previously
// as packetId
// Bytes 2 and 3 are skipped
byteBuffer.get();
byteBuffer.get();

// Account ID(Bytes 4, 5, 6, 7)
session.setAccountId(byteBuffer.getInt());

// Byte 8 is skipped
byteBuffer.get();

// Char ID (Bytes 9, 10, 11, 12)
session.setCharId(byteBuffer.getInt());

// Bytes 13, 14, 15 and 16 are skipped
byteBuffer.get();
byteBuffer.get();
byteBuffer.get();
byteBuffer.get();

// Login Id (Bytes 17, 18, 19, 20) (Not needed)
byteBuffer.getInt();

// Not reading Client tick (Not needed)

// Bytes 21, 22, 23 and 24 are skipped
byteBuffer.get();
byteBuffer.get();
byteBuffer.get();
byteBuffer.get();

// Sex (Byte 25) (Not needed)
```

```
    byteBuffer.get();

    bytesRead += 24;
    return "Want To Connect";
  }

  // Source getBytesFromFile: The Java Developers
     Almanac 1.4
  // http://exampledepot.com/egs/java.io/
     File2ByteArray.html
  private byte[] getBytesFromFile(File file) throws
     IOException {
    InputStream is = new FileInputStream(file);

    // Get the size of the file
    long length = file.length();

    // You cannot create an array using a long type,
    // it needs to be an int type.
    // Before converting to an int type, check to
    // ensure that file is not larger than Integer.
       MAX_VALUE.
    if (length > Integer.MAX_VALUE) {
      System.out.println("File is too large");
      System.exit(1);
    }

    // Create the byte array to hold the data
    byte[] bytes = new byte[(int) length];

    // Read in the bytes
    int offset = 0, numRead = 0;
    while (offset < bytes.length
        && (numRead = is.read(bytes, offset,
            bytes.length - offset)) >= 0)
      offset += numRead;

    // Ensure all the bytes have been read in
    if (offset < bytes.length)
      throw new IOException("Could not completely read
           file "
           + file.getName());

    // Close the input stream and return bytes
```

```
    is.close();
    return bytes;
  }
}
```

6.5 Appendix E: Session Sourcecode

```
package nl.ru.ai.turingtest.analyzer;

import java.util.ArrayList;
import java.util.List;
import java.util.Vector;

/**
 * The Session is used to store and calculate certain
 * properties of a packetstream log. These properties
 * will be used to determine whether a log is
 * from a human player or a bot.
 *
 * @author Adam Cornelissen
 * @version June 16, 2008
 */
public class Session {
  private static final int DISTANCE_OUTLIER = 1000;
  private static final double TIME_OUTLIER = 7000;
  private List<MovePacket> movePackets;
  private List<Double>
     changeDirectionTakeItemIntervals;
  private List<Double> takeItemTimes;
  private List<Double> attackFirstLootInterval;
  private boolean timePlayStartSet;
  private double timePlayStart;
  private double timePlayStop;
  private double packetsTotal;
  private int onceAttack;
  private int continuousAttack;
  private int accountId;
  private int charId;
  private double lastChangedDirectionTime;
  private boolean lastChangedDirectionTimeSet;
  private boolean attacking;
  private boolean looting;
  private double lastAttackTime;
  /*
   * Derived values
   */
  private boolean derivedValuesCalculated;
  private double timePlayedTotal;
```

```
private double packetsPerSecond;
private double averageMoveDistanceX;
private double averageMoveDistanceY;
private double averageMoveDistanceXY;
private double averageMoveTime;
private double averagePickupTimeDifference;
private double averageTimeBetweenChDirAndPickup;
private double averageFirstLoot;
private double onceAttackPerSecond;
private double continuousAttackPerSecond;

/**
 * Create a new session
 */
public Session() {
  movePackets = new ArrayList<MovePacket>();
  changeDirectionTakeItemIntervals = new ArrayList<
     Double>();
  takeItemTimes = new ArrayList<Double>();
  attackFirstLootInterval = new ArrayList<Double>();
  timePlayStartSet = false;
  lastChangedDirectionTimeSet = false;
  attacking = false;
  looting = false;
  derivedValuesCalculated = false;
}

/*
 * Calculates the average time between 'move'
 * packets (0x00A7). Times longer than timeOutlier
 * are not used in the calculation.
 */
private void calcAverageMoveTime() {
  double nrMoves = 0;
  double timeDifference = 0, timeTotal = 0;

  for (int position = 1;
       position < movePackets.size();
       position++) {
    MovePacket packetCurrent = movePackets.get(
       position);
    MovePacket packetPrevious = movePackets.get(
       position -1);
```

```
      timeDifference = (packetCurrent.getTime()
          − packetPrevious.getTime() );
      if (timeDifference < TIME_OUTLIER) {
        timeTotal += timeDifference;
        nrMoves++;
      } else {
      //System.out.println("time outlier used");
      }
    }
    if (nrMoves == 0)
      averageMoveTime = TIME_OUTLIER;
    else
      averageMoveTime = timeTotal / nrMoves;
  }

  /**
   * Calculates the average distance between moves.
   * Distances larger than distanceOutlier are not
   * used in the calculation.
   */
  private void calcAverageMoveDistance() {
    double totalX = 0;
    double totalY = 0;
    double totalXY = 0;
    int nrMoves = 0;

    for (int position = 1;
         position < movePackets.size();
         position++) {
      MovePacket packetCurrent = movePackets.get(
          position);
      MovePacket packetPrevious = movePackets.get(
          position −1);

      double differenceX = Math.abs(packetCurrent.getX
          ()
          − packetPrevious.getX());
      double differenceY = Math.abs(packetCurrent.getY
          ()
          − packetPrevious.getY());
      double differenceXY = Math.sqrt(differenceX *
          differenceX
          + differenceY * differenceY);
```

```
      if (differenceX < DISTANCE_OUTLIER
          && differenceY < DISTANCE_OUTLIER
          && differenceXY < DISTANCE_OUTLIER) {
        totalX += differenceX;
        totalY += differenceY;
        totalXY += differenceXY;
        nrMoves++;
      } else {
      //System.out.println("distance outlier used");
      }
    }
    if (nrMoves == 0) {
      averageMoveDistanceX = 0;
      averageMoveDistanceY = 0;
      averageMoveDistanceXY = 0;
    } else {
      averageMoveDistanceX = totalX/nrMoves;
      averageMoveDistanceY = totalY/nrMoves;
      averageMoveDistanceXY = totalXY/nrMoves;
    }
  }

  /**
   * Calculates the average time between 'take item'
   * packets (0x00F5). Times longer than timeOutlier
   * are not used in the calculation.
   */
  private void calcAveragePickupTime() {
    double sumTimes = 0;
    int totalTimes = 0;

    Vector<Double> pickupTimeDifferences = new Vector<
       Double>();

    for (int position = 1;
         position < takeItemTimes.size();
         position++) {
      double pickupTimeCurrent = takeItemTimes.get(
         position);
      double pickupTimePrevious = takeItemTimes.get(
         position - 1);

      double pickupDifference=pickupTimeCurrent-
         pickupTimePrevious;
```

```
      if (pickupDifference < TIME_OUTLIER) {
        pickupTimeDifferences.add(pickupDifference);
        sumTimes += pickupDifference;
        totalTimes++;
      } else {
      //System.out.println("time outlier used");
      }
    }
    if (totalTimes == 0)
      averagePickupTimeDifference = TIME_OUTLIER;
    else
      averagePickupTimeDifference = sumTimes/
          totalTimes;
  }

  /**
   * Calculates the average time between 'chdir'
   * packets (0x0085) and 'take item' packets
   * (0x00F5). Times longer than timeOutlier are
   * not used in the calculation.
   */
  private void calcAverageTimeBetweenChDirAndPickup()
      {
    double sumTimes = 0;
    int totalTimes = 0;

    for (Double time :
        changeDirectionTakeItemIntervals)
      if (time < TIME_OUTLIER) {
        sumTimes += time;
        totalTimes++;
      }
    if (totalTimes == 0)
      averageTimeBetweenChDirAndPickup = TIME_OUTLIER;
    else
      averageTimeBetweenChDirAndPickup = (sumTimes/
          totalTimes);
  }

  private void calcAverageFirstLoot() {
    double sumTimes = 0;
    int totalTimes = 0;
```

```
    for (Double time : attackFirstLootInterval) {
      if (time < TIME_OUTLIER) {
        sumTimes += time;
        totalTimes++;
      }
    }
    if (totalTimes == 0)
      averageFirstLoot = TIME_OUTLIER;
    else
      averageFirstLoot = (sumTimes/totalTimes);
  }

  /**
   * @return total number of packets
   */
  public double getPacketsTotal() {
    return packetsTotal;
  }

  /**
   * Shows the playtime in a 'normal' form.
   * Instead of 'x milliseconds' it shows 'x days,
   * x hours, x minutes, x seconds and x milliseconds'
   *
   * @param timeInMilliSeconds
   * @return human form time
   */
  private String getTimeHumanForm(double
      timeInMilliSeconds) {
    int days, hours, minutes, seconds, milliSeconds;

    days = (int) (timeInMilliSeconds / (1000* 60 * 60
        * 24));
    timeInMilliSeconds -= (days * (1000 * 60 * 60 *
        24));

    hours = (int) (timeInMilliSeconds / (1000 * 60 *
        60));
    timeInMilliSeconds -= (hours * (1000 * 60 * 60));

    minutes = (int) (timeInMilliSeconds / (1000 * 60))
        ;
    timeInMilliSeconds -= (minutes * (1000 * 60));
```

```
    seconds = (int) (timeInMilliSeconds / 1000);
    timeInMilliSeconds -= (seconds * 1000);

    milliSeconds = (int) timeInMilliSeconds;

    return (days + " day(s) " + hours + " hour(s) " +
        minutes
         + " minute(s) " + seconds + " second(s) "
         + milliSeconds + " millisecond(s)");
  }

  /**
   * @param packet
   */
  public void add(MovePacket packet) {
    movePackets.add(packet);
  }

  /**
   * @return True is the start time is
   * already set, false otherwise
   */
  public boolean isTimePlayStartSet() {
    return timePlayStartSet;
  }

  /**
   * @param packetTime
   *        Time to use as the start time of the game
   */
  public void setTimePlayStart(double packetTime) {
    timePlayStart = packetTime;
    timePlayStartSet = true;
  }

  /**
   * @param packetTime
   */
  public void setTimePlayStop(double packetTime) {
    timePlayStop = packetTime;
  }

  public void countPacket() {
    packetsTotal++;
```

```
}

public void countOnceAttack() {
  onceAttack++;
}

public void countContinuousAttack() {
  continuousAttack++;
}

public void addChangeDirectionTakeItemInterval(
    double interval) {
  changeDirectionTakeItemIntervals.add(interval);
}

public void addTakeItemTime(double packetTime) {
  takeItemTimes.add(packetTime);
}

public void addAttackFirstLootInterval(double
    interval) {
  attackFirstLootInterval.add(interval);
}

public void setAccountId(int accountId) {
  this.accountId = accountId;
}

public void setCharId(int charId) {
  this.charId = charId;
}

public void setLastChangedDirectionTime(double
    packetTime) {
  lastChangedDirectionTime = packetTime;
  lastChangedDirectionTimeSet = true;
}

public boolean isLastChangedDirectionTimeSet() {
  return lastChangedDirectionTimeSet;
}

public double getLastChangeDirectionTime() {
  return lastChangedDirectionTime;
```

```
}

public boolean isAttacking() {
  return attacking;
}

public void setAttacking(boolean value) {
  attacking = value;
}

public double getLastAttackTime() {
  return lastAttackTime;
}

public void setLastAttackTime(double packetTime) {
  lastAttackTime = packetTime;
}

public String toString() {
  calculateDerivedVales();
  StringBuffer stringBuffer = new StringBuffer();
  stringBuffer.append("charId: ");
  stringBuffer.append(charId);
  stringBuffer.append("\ntime played: ");
  stringBuffer.append(getTimeHumanForm(
     timePlayedTotal));
  stringBuffer.append("\nnumber of packets: ");
  stringBuffer.append(packetsTotal);
  stringBuffer.append("\npackets/second: ");
  stringBuffer.append(packetsPerSecond);
  stringBuffer.append("\naverage X move distance: ")
     ;
  stringBuffer.append(averageMoveDistanceX);
  stringBuffer.append("\naverage Y move distance: ")
     ;
  stringBuffer.append(averageMoveDistanceY);
  stringBuffer.append("\naverage XY move distance: "
     );
  stringBuffer.append(averageMoveDistanceXY);
  stringBuffer.append("\naverage time between moves:
      ");
  stringBuffer.append(averageMoveTime);
  stringBuffer.append("\naverage pickup time
     difference: ");
```

```
    stringBuffer.append(averagePickupTimeDifference);
    stringBuffer.append("\naverage␣time␣between␣chdir␣
       and␣pickup:␣");
    stringBuffer.append(
       averageTimeBetweenChDirAndPickup);
    stringBuffer.append("\naverage␣first␣loot:␣");
    stringBuffer.append(averageFirstLoot);
    stringBuffer.append("\nonce␣attacks/second:␣");
    stringBuffer.append(onceAttackPerSecond);
    stringBuffer.append("\ncontinuous␣attacks/second:␣
       ");
    stringBuffer.append(continuousAttackPerSecond);
    stringBuffer.append("\n");

    return new String(stringBuffer);
  }

  private void calculateDerivedVales() {
    if (derivedValuesCalculated)
      return;
    timePlayedTotal = timePlayStop-timePlayStart;
    packetsPerSecond = packetsTotal/(timePlayedTotal
       /1000);
    onceAttackPerSecond = onceAttack/(timePlayedTotal
       /1000);
    continuousAttackPerSecond = continuousAttack /
    (timePlayedTotal/1000);
    calcAverageMoveDistance();
    calcAverageMoveTime();
    calcAveragePickupTime();
    calcAverageTimeBetweenChDirAndPickup();
    calcAverageFirstLoot();
    derivedValuesCalculated = true;
  }

  /**
   * @return feature vector for this session
   */
  public Vector<Double> getVector() {
    Vector<Double> vector = new Vector<Double>();
    calculateDerivedVales();
    vector.add(packetsPerSecond);
    vector.add(averageMoveDistanceX);
    vector.add(averageMoveDistanceY);
```

```
    vector.add(averageMoveDistanceXY);
    vector.add(averageMoveTime/100);
    vector.add(averagePickupTimeDifference/1000);
    vector.add(averageTimeBetweenChDirAndPickup/1000);
    vector.add(averageFirstLoot/1000);
    // vector.add(onceAttackPerSecond);
    vector.add(continuousAttackPerSecond);
    return vector;
  }

  /**
   * @return Returns the character id
   */
  public int getCharId() {
    return charId;
  }

  /**
   * Checks if a session is long enough to be valid.
   * The minimal time limit is set as 10 minutes.
   *
   * @return
   */
  public boolean isValid() {
    calculateDerivedVales();
    return (timePlayedTotal > 10*60*1000);
  }

}
```

6.6 Appendix F: MovePacket Sourcecode

```
package nl.ru.ai.turingtest.analyzer;

/**
 * @author Adam Cornelissen
 * @version June 20, 2008
 */
public class MovePacket {
  private int x, y;
  private double time;

  /**
   * Constructor
   *
   * @param x
   *             The x-coordinate of the position
   * @param y
   *             The y-coordinate of the position
   * @param time
   *             Time at which the packet was sent (in
      milliseconds)
   */
  public MovePacket(int x, int y, double time) {
    this.x = x;
    this.y = y;
    this.time = time;
  }

  /**
   * @return x-coordinate of the position
   */
  public int getX() {
    return x;
  }

  /**
   * @return y-coordinate of the position
   */
  public int getY() {
    return y;
  }
```

```
  /**
   * @return Time at which the packet was sent (in
       milliseconds)
   */
  public double getTime() {
    return time;
  }
}
```

6.7 Appendix G: Run Sourcecode

```
package nl.ru.ai.turingtest.analyzer;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.Vector;

import nl.ru.ai.turingtest.network.BackPropagation;
import nl.ru.ai.turingtest.network.Network;

/**
 * Main routine to run analyzer on input
 * files and train the neural network.
 *
 * @author Adam Cornelissen
 * @version June 24, 2008
 */
public class Run {

  /**
   * Main routine which will create feature
   * vectors for all the player files
   * in the playerDirectory variable and
   * for all the bot files in the
   * botDirectory variable.
   *
   * @param args
   *            Unused
   */
  public static void main(String[] args) {
    final String playerDirectory = "players";
    final String botDirectory = "bots";
    final String unknownDirectory = "unknown";
    Analyzer analyzer = new Analyzer("packet_db.txt");
    Map<Integer, String> chars=CharReader.read("
       rrochars.txt");

    /*
     * Create feature vectors for all player files
     */
    List<Vector<Double>> playerVectors =
```

```
      new ArrayList<Vector<Double>>();
    List<String> playerNames =
      new ArrayList<String>();

    List<Session> sessions = analyzer.analyze(
       playerDirectory);
    for (Session session : sessions) {
      if (session.isValid()) {
        // System.out.println(chars.get(session.
            getCharId()));
        Vector<Double> vector = session.getVector();
        playerVectors.add(vector);
        playerNames.add(chars.get(session.getCharId())
            );
      }
    }
    /*
     * Create feature vectors for all bot files
     */
    List<Vector<Double>> botVectors =
      new ArrayList<Vector<Double>>();
    List<String> botNames =
      new ArrayList<String>();
    sessions = analyzer.analyze(botDirectory);
    for (Session session : sessions) {
      if (session.isValid()) {
        Vector<Double> vector = session.getVector();
        botVectors.add(vector);
        botNames.add(chars.get(session.getCharId()));
      }
    }

    /*
     * Create feature vectors for all unknown files
     */
    List<Vector<Double>> unknownVectors =
      new ArrayList<Vector<Double>>();
    List<String> unknownNames =
      new ArrayList<String>();
    sessions = analyzer.analyze(unknownDirectory);
    for (Session session : sessions) {
      if (session.isValid()) {
        Vector<Double> vector = session.getVector();
        unknownVectors.add(vector);
```

```
        unknownNames.add(chars.get(session.getCharId()
            ));
      }
    }
    /*
     * Loop, leaving out a single vector,
     * train with the rest
     */
    for (int leaveOut = 0; leaveOut < playerVectors.
        size()
        + botVectors.size(); leaveOut++) {
      boolean playerSkip;
      int skipNumber;
      if (leaveOut < playerVectors.size()) {
        playerSkip = true;
        skipNumber = leaveOut;
      } else {
        playerSkip = false;
        skipNumber = leaveOut - playerVectors.size();
      }
      /*
       * Create a network
       */
      int inputNodes = playerVectors.get(0).size();
      int hiddenNodes = playerVectors.get(0).size();
      Network network = new BackPropagation(inputNodes
          ,
          hiddenNodes, 1);

      int max = Math.max(playerVectors.size(),
          unknownVectors.size());

      double error = 10;
      /*
       * Train it
       */
      while (error > 0.0001) {
        error = 0;
        for (int i = 0; i < max; i++) {
          /*
           * First a player feature vector will
           * be added to the network, then a
           * bot feature vector, then a player
           * feature vector again, etc.
```

```
     *
     * For each feature vector the input
     * nodes of the neural network will
     * be set to the value of the
     * corresponding field in the feature
     * vector.
     */
    if (i < playerVectors.size()
        && !(playerSkip && i == skipNumber)) {
      Vector<Double> vector = playerVectors.get(
          i);
      for (int j = 0; j < vector.size(); j++)
        network.setInputActivation(j, vector.get
            (j));

      network.propagate();
      // System.out.println(" 0.0 - " +
      // network.getOutputActivation(0));
      double[] desired = new double[1];
      desired[0] = 0; // player
      network.backPropagate(desired);
      error += network.getError();
    }

    if (i < botVectors.size()
        && !(!playerSkip && i == skipNumber)) {
      Vector<Double> vector = botVectors.get(i);
      for (int j = 0; j < vector.size(); j++)
        network.setInputActivation(j, vector.get
            (j));

      network.propagate();
      // System.out.println(" 1.0 - " +
      // network.getOutputActivation(0));
      double[] desired = new double[1];
      desired[0] = 1; // bot
      network.backPropagate(desired);
      error += network.getError();
    }
  }
  // System.out.println(error);
}

/*
```

```
            * Run on skipped one
            */
           Vector<Double> vector;
           if (leaveOut < playerVectors.size()) {
             vector = playerVectors.get(skipNumber);
             for (int i = 0; i < vector.size(); i++) {
               network.setInputActivation(i, vector.get(i))
                   ;
             }
             network.propagate();
             System.out.printf("%-18s %.4f\n",
                 playerNames.get(leaveOut),
                 network.getOutputActivation(0));
           } else {
             vector = botVectors.get(skipNumber);
             for (int i = 0; i < vector.size(); i++) {
               network.setInputActivation(i, vector.get(i))
                   ;
             }
             network.propagate();
             System.out.printf("Bot %2d %.4f\n",
                 leaveOut,
                 network.getOutputActivation(0));
           }
         }
       }
     }
```

6.8 Appendix H: VectorMath Sourcecode

```
package nl.ru.ai.turingtest.analyzer;

import java.util.Vector;

/**
 * @author Adam Cornelissen
 * @version June 16, 2008
 */
public class VectorMath {

  /**
   * Constructor
   */
  public VectorMath() {
  }

  /**
   * Returns the length of a vector
   * according to:
   * (a0^2 + a1^2 + ... + an^2)^0.5
   *
   * @param vector
   *            Vector of which to calculate the
      length
   * @return The length of the vector
   */
  public static double calculateVectorLength(
      Vector<Double> vector) {
    double sumOfSquares = 0;
    for (double attribute : vector)
      sumOfSquares += (Math.pow(attribute, 2));
    return Math.sqrt(sumOfSquares);
  }

  /**
   * Normalizes a vector according to:
   * (1 / |{a0,a1,..,an}| ) * {a0,a1,..,an}
   * (|vector| is the vector length
   * (see 'calculateVectorLength(vector)
   * function).
   *
```

```
 * @param vector
 *            Vector to normalize
 * @param vectorLength
 *            Length of the vector
 * @return A normalized vector
 */
public Vector<Double> normalizeVector(Vector<Double>
    vector,
    double vectorLength) {
  Vector<Double> vectorcopy =
    (Vector<Double>) vector.clone();

  for (int i = 0; i < vectorcopy.size(); i++)
    vectorcopy.set(i, (vectorcopy.elementAt(i) /
       vectorLength));

  return vectorcopy;
}

/**
 * Calculates the inproduct of two
 * vectors according to:
 * {a0,a1,..,an}.{b0,b1,..,bn} =
 * a0*b0 + a1*b1 + .. + an*bn
 * ASSUMPTION: Both vectors are the
 * same size.
 *
 * @param vectorOne
 *            Vector one of two to calculate the
    inproduct
 * @param vectorTwo
 *            Vector one of two to calculate the
    inproduct
 * @return Inproduct of vectorOne and vectorTwo
 */
public double inProduct(Vector<Double> vectorOne,
    Vector<Double> vectorTwo) {
  double inproduct = 0;
  Vector<Double> vector1copy = (Vector<Double>)
     vectorOne.clone();
  Vector<Double> vector2copy = (Vector<Double>)
     vectorTwo.clone();

  for (int i = 0; i < vector1copy.size(); i++) {
```

```
      double element1 = vector1copy.elementAt(i);
      double element2 = vector2copy.elementAt(i);
      double product = element1 * element2;
      inproduct += product;
    }
    return inproduct;
  }

  /**
   * Calculates the angle between two vectors
      according to:
   * {a0,a1,..,an}.{b0,b1,..,bn} =
   * |{a0,a1,..,an}| * |{b0,b1,..,bn}| * cos(angle)
   *
   * @param vectorOne
   *            Vector one of two to calculate the
      angle
   * @param vectorTwo
   *            Vector one of two to calculate the
      angle
   * @return The angle between vectorOne and vectorTwo
   */
  public double calculateAngleBetweenVectors(Vector<
     Double> vectorOne,
      Vector<Double> vectorTwo) {
    Vector<Double> vectorOneCopy =
      (Vector<Double>) vectorOne.clone();
    Vector<Double> vectorTwoCopy =
      (Vector<Double>) vectorTwo.clone();

    Vector<Double> vectorOneNorm = null;
    Vector<Double> vectorTwoNorm = null;

    double lengthOne = calculateVectorLength(
       vectorOneCopy);
    double lengthTwo = calculateVectorLength(
       vectorTwoCopy);

    vectorOneNorm = normalizeVector(vectorOneCopy,
       lengthOne);
    vectorTwoNorm = normalizeVector(vectorTwoCopy,
       lengthTwo);

    double inproduct = inProduct(vectorOneNorm,
```

```
            vectorTwoNorm);
        return inproduct;
    }
}
```

Bibliography

[1] Identifying MMORPG Bots: A Traffic Analysis Approach
Kuan-Ta Chen, National Taiwan University

[2] Embedded Noninteractive Continuous Bot Detection
Roman V. Yampolskiy & Venu Govindaraju, University at Buffalo, Buffalo