Users Manual

RoboCup Soccer Server

for Soccer Server Version 7.07 and later

Mao Chen[†], Ehsan Foroughi,
Fredrik Heintz, ZhanXiang Huang[†],
Spiros Kapetanakis, Kostas Kostiadis,
Johan Kummeneje, Itsuki Noda,
Oliver Obst, Pat Riley,
Timo Steffens, Yi Wang[†] and
Xiang Yin[†]

July 26, 2002

[†] <ustc9811@sina.com>
<foroughi@ce.sharif.edu>
<frehe@ida.liu.se>
<spiros@cs.york.ac.uk>
<kkosti@essex.ac.uk>
<johan.kummeneje@generalwireless.se>
<noda@etl.go.jp>
<fruit@uni-koblenz.de>
<pfr+@cs.cmu.edu>
<timosteffens@gmx.de>

Acknowledgements

We are very grateful for the work of the authors from the previous versions of the manual that could not help out on this version:

- *David Andre* at Berkeley, University of California, USA.
- *Pascal Gugenberger* at Humboldt University, Berlin, Germany.
- *Marius Joldos* at Technical University of Cluj-Napoca, Romania.
- *Paul Arthur Navratil* at University of Texas, USA.
- *Peter Stone* at AT&T, USA.
- *Tomoichi Takahashi* at Chubu University, Japan.
- *Tralvex Yeap* at KRDL, Singapore.
- *Emiel Corten* at University of Amsterdam, Netherlands.
- *Klaus Dorer* at University Freiburg, Germany.
- *Helmut Myritz* at Humboldt University, Germany.
- *Jukka Riekki* at Oulu University, Finland.

Besides the authors, we would also like to thank Stefan Sablatnög from the University of Ulm, Germany, and Mike Hewett from University of Texas, USA, for a thorough proofreading of the soccermanual 4.00. We have also received a lot of good suggestions from Erik Mattsson at the University of Karlskrona/Ronneby, Sweden.

We would not have been able to do this manual without the above mentioned people[1].

[1] The persons listed on the title page are the persons responsible for the different sections of the manual.

Contents

1. Introduction

We are in the early days of RoboCup [7], with half a century to go before we can "...build a team of robot soccer players, which can beat a human world cup champion team" ([6], p. v). The challenge posed by the goal is enormous and inspires hundreds of researchers yearly throughout the world to engage themselves and their students in RoboCup. RoboCup has been used as a research challenge in parallel with a usage for educational purposes, and to stimulate the interest of the public for robotics and artificial intelligence (AI). Each year since 1997, researchers from different countries have gathered to play the world cup. The event has drawn an increasing amount of interest from the public, as robotics is still not commonplace.

The intention of this manual[1] is to guide the developers of simulated league teams in the beginning steps, and also serve as a reference manual for the experienced users.

1.1. Background

Mackworth [11] introduced the idea of using soccer-playing robots in research. Unfortunately, the idea did not get the proper response until the idea was further developed and adapted by Kitano, Asada, and Kuniyoshi, when proposing a Japanese research programme, called Robot J-League[2]. During the autumn of 1993, several American researchers took interest in the Robot J-League, and it thereafter changed name to the Robot World Cup Initiative or *RoboCup* for short. RoboCup is sometimes referred to as the RoboCup challenge or the RoboCup domain.

In 1995, Kitano et al. [7] proposed the first Robot World Cup Soccer Games and Conferences to take place in 1997. The aim of RoboCup was to present a new standard problem for AI and robotics, somewhat jokingly described as the life of AI after Deep Blue[3]. RoboCup differs from previous research in AI by focusing on a distributed solution instead of a centralised solution, and by challenging researchers from not only traditionally AI-related fields, but also researchers in the areas of robotics, sociology, real-time mission critical systems, etc.

To co-ordinate the efforts of all researchers, the RoboCup Federation was formed. The goal of RoboCup Federation is to promote RoboCup, for example by annually arranging the world cup tournament. Members of the RoboCup Federation are all active researchers in the field, and represent a number of universities and major companies. As

[1] Parts of this chapter is taken directly from [16]
[2] The J-League is the professional soccer league in Japan.
[3] In reference to Deep Blue and its games with Kasparov, see `http://www.chess.ibm.com`.

the body of researchers is quite large and widespread, local committees are formed to promote RoboCup-related events in their geographical area.

1.2. The Goals of RoboCup

The RoboCup Federation has set goals and a timetable for the research. Setting goals and a timetable are means of pushing the state-of-the-art further, in conjunction with formalised test-beds. In resemblance with John F. Kennedy's national goal of "landing a man on the moon and returning him safely to earth" ([4], p. 8276), the main accomplishment was not to land a man on the moon and returning him safely, but the overall technological advancement. Therefore, the most important goal of RoboCup is to advance the overall technological level of society, and as a more pragmatic goal to achieve the following:

> By mid-21st century, a team of fully autonomous humanoid robot soccer players shall win the soccer game, comply with the official rule of the FIFA[4], against the winner of the most recent World Cup [13].

There will be several technological advancements, even if the goal of the robotic soccer team is not reached, starting with Team-Partitioned, Opaque-Transition Reinforcement Learning (TPOT-RL) [17] which has found application in the domain of packet routing in computer networks. TPOT-RL is a distributed learning method in domains where "agents have limited information about environmental state transitions" ([17], p. 22).

In most RoboCup leagues, the teams consist of either robots or programs that co-operate in order to defeat the opponent team. RoboCup Rescue and the commentator exhibition diverge from the other RoboCup leagues. The goal of defeating an opponent would raise ethical issues in RoboCup Rescue, since we cannot assign comparable utilities to human lives and buildings. Hence, the focus in RoboCup Rescue is on the co-operative efforts between heterogeneous agents. In the commentator exhibition, the goal is to observe and comment.

Besides the commentator exhibition and RoboCup Rescue, the main body of the RoboCup challenge consists of several leagues for soccer playing. However, as this manual is about the simulated league we will only focus on it.

1.2.1. Simulated League

The RoboCup simulator league is based on the RoboCup simulator called the soccer server [12], a physical soccer simulation system. All games are visualised by displaying the field of the simulator by the soccer monitor on a computer screen. The soccer server is written to support competition among multiple virtual soccer players in an uncertain multi-agent environment, with real-time demands as well as semi-structured conditions. One of the advantages of the soccer server is the abstraction made, which relieves the

[4]Fédération Internationale de Football Association (FIFA) defines the rules of soccer [25].

researchers from having to handle robot problems such as object recognition [9], communications, and hardware issues, e.g., how to make a robot move. The abstraction enables researchers to focus on higher level concepts such as co-operation and learning.

Since the soccer server provides a challenging environment, i.e., the intentions of the players cannot mechanically be deduced, there is a need for a referee when playing a match. The included artificial referee is only partially implemented and can detect trivial situations, e.g., when a team scores. However, there are several hard-to-detect situations in the soccer server, e.g., deadlocks, which brings the need for a human referee. All participating teams are also obliged to play according to a gentlemen's agreement, e.g., not to use loopholes.

Since the first version of the soccer server was completed in 1995, there have been four world cups and one pre-world cup event, not to mention all other RoboCup-related events. The 1996 pre-RoboCup event [5] was held in Osaka, with only seven entrants in the competition which ended with a Japanese victory by the team Ogalets from Tokyo University. In Nagoya the following year, the first formal competition was held in conjunction with the IJCAI'97 conference. The competition had 29 teams participating, and the winner was AT Humboldt [2]. The RoboCup world cup of 1998 was played in conjunction with the human world cup in Paris, and the winning team was CMUnited98 [27]. During the world cup, media was heavily covering the event, as it was public in a museum in the suburbs of Paris. The year after, the world cup was held in conjunction with IJCAI'99 in Stockholm, and the winners (once again) were CMUnited99 [28]. An unchanged version of the champion team must participate, as a benchmark, in the next world cup. The benchmarking teams have always been able to win their group, but only in 2000 did the benchmark team advance further than the first game after group play.

1.2.2. What is the Soccerserver

Soccer Server is a system that enables autonomous agents consisting of programs written in various languages to play a match of soccer (association football) against each other.

A match is carried out in a client/server style: A server, soccerserver, provides a virtual field and simulates all movements of a ball and players. Each client controls movements of one player. Communication between the server and each client is done via UDP/IP sockets. Therefore users can use any kind of programing systems that have UDP/IP facilities.

The soccerserver consists of 2 programs, `soccerserver` and `soccermonitor`. `Soccer Server` is a server program that simulates movements of a ball and players, communicates with clients, and controls a game according to rules. `Soccermonitor` is a program that displays the virtual field from the soccerserver on the monitor using the X window system. A number of `soccermonitor` programs can connect with one `soccerserver`, so we can display field-windows on multiple displays.

A client connects with `soccerserver` by an UDP socket. Using the socket, the client sends commands to control a player of the client and receives information from sensors of the player. In other words, a client program is a brain of the player: The client receives visual and auditory sensor information from the server, and sends control-commands to the server.

Each client can control only one player[5][6]. So a team consists of the same number of clients as players. Communications between the clients must be done via `soccerserver` using `say` and `hear` protocols. (See section 4.2.1.) One of the purposes of soccerserver is evaluation of multi-agent systems, in which efficiency of communication between agents is one of the criteria. Users must realize control of multiple clients by such restricted communication.

1.3. History

In this section we will first describe the history of the soccerserver and thereafter the history of the RoboCup Simulation League. To end the section we will also describe the history of the manual effort.

1.3.1. History of the Soccer Server

The first, preliminary, original system of soccerserver was written in September of 1993 by Itsuki Noda, ETL. This system was built as a library module for demonstration of a programming language called MWP, a kind of Prolog system that has multi-threads and high level program manipulation. The module was a closed system and displayed a field on a character display, that is VT100.

The first version (version 0) of the client-server style server was written in July of 1994 on a LISP system. The server shows the field on an X window system, but each player was shown in an alphabet character. It used the TCP/IP protocol for connections with clients. This LISP version of soccerserver became the original style of the current soccerserver. Therefore, the current soccerserver uses S-expressions for the protocol between clients and the server.

The LISP version of soccerserver was re-written in C++ in August of 1995 (version 1). This version was announced at the IJCAI workshop on Entertainment and AI/Alife held in Montreal, Canada, August 1995.

The development of version 2 started January of 1996 in order to provide the official server of preRoboCup-96 held at Osaka, Japan, November 1996. From this version, the system is divided into two modules, soccerserver and soccerdisplay (currently, soccermonitor). Moreover, the feature of coach mode was introduced into the system. These two features enabled researchers on machine learning to execute games automatically. Peter Stone at Carnegie Mellon University joined the decision-making process for the development of the soccerserver at this stage. For example, he created the configuration files that were used at preRoboCup-96.

After preRoboCup-96, the development of the official server for the first RoboCup, RoboCup-97 held at Nagoya, Japan, August 1997, started immediately, and the version

[5]Technically, it is easy to cheat the server. Therefore this is a gentleman's agreement.

[6]In order to test various kinds of systems, we may permit a client to control multiple players if the different control modules of players are separated logically from each other in the client.

3 was announced in February of 1997. Simon Ch'ng at RMIT joined decisions of regulations of soccerserver from this stage. The following features were added into the new version:

- logplayer
- information about movement of seen objects in visual information
- capacity of hearing messages

The development of version 4 started after RoboCup-97, and announced November 1997. From this version, the regulations are discussed on the mailing list organized by Gal Kaminka. As a result, many contributers joined the development. Version 4 had the following new features:

- more realistic stamina model
- goalie
- handling offside rule
- disabling players for evaluation
- facing direction of players in visual information
- sense_body command

Version 4 was used in Japan Open 98, RoboCup-98 and Pacific Rim Series 98.

Version 5 was used in Japan Open 99, and will also be used in RoboCup-99 in Stockholm during the summer of 1999.

In Melbourne 2000, version 6 was used, and for the world cup in 2001 version 7 will be used.

1.3.2. History of the RoboCup Simulation League

The RoboCup simulation league has had five main official events: preRoboCup-96, RoboCup-97, RoboCup-98, RoboCup-99, and RoboCup 2000. Research results have been reported extensively in the proceedings of the workshops and conferences associated with these competitions. In this section, we focus mainly on the competitions themselves.

preRoboCup-96

preRoboCup-96 was the first robotic soccer competition of any sort. It was held on November 5–7, 1996 in Osaka, Japan [5]. In conjunction with the IROS-96 conference, preRoboCup-96 was meant as an informal, small-scale competition to test the RoboCup soccerserver in preparation for RoboCup-97. 5 of the 7 entrants were from the Tokyo region. The other 2 were from Ch'ng at RMIT and Stone and Veloso from CMU.

The winning teams were entered by:

1. Ogawara (Tokyo University)
2. Sekine (Tokyo Institute of Technology)
3. Inoue (Waseda University)
4. Stone and Veloso (Carnegie Mellon University)

In this tournament, team strategies were generally quite straightforward. Most of the teams kept players in fixed locations, only moving them towards the ball when it was nearby.

RoboCup-97

The RoboCup-97 simulator competition was the first formal simulated robotic soccer competition. It was held on August 23–29, 1997 in Nagoya, Japan in conjunction with the IJCAI-97 conference [6]. With 29 teams entering from all around the world, it was a very successful tournament.

The winning teams were entered by:

1. Burkhard et al. (Humboldt University)
2. Andou (Tokyo Institute of Technology)
3. Tambe et al. (ISI/University of Southern California)
4. Stone and Veloso (Carnegie Mellon University)

In this competition, the champion team exhibited clearly superior low-level skills. One of its main advantages in this regard was its ability to kick the ball harder than any other team. Its players did so by kicking the ball around themselves, continually increasing its velocity so that it ended up moving towards the goal faster than was imagined possible. Since the soccerserver did not (at that time) enforce a maximum ball speed, a property that was changed immediately after the competition, the ball could move arbitrarily fast, making it almost impossible to stop. With this advantage at the low-level behavior level, no team, regardless of how strategically sophisticated, was able to defeat the eventual champion.

At RoboCup-97, the RoboCup scientific challenge award was introduced. Its purpose is to recognize scientific research results regardless of performance in the competitions. The 1997 award went to Sean Luke [10] of the University of Maryland "for demonstrating the utility of evolutionary approach by co-evolving soccer teams in the simulator league."

RoboCup-98

The second international RoboCup championship, RoboCup-98, was held on July 2–9, 1998 in Paris, France [1]. It was held in conjunction with the ICMAS-98 conference.

The winning teams were entered by:

1. Stone et al. (Carnegie Mellon University)
2. Burkhard et al. (Humboldt University)
3. Corten and Rondema (University of Amsterdam)
4. Tambe et al. (ISI/University of Southern California)

Unlike in the previous year's competition, there was no team that exhibited a clear superiority in terms of low-level agent skills. Games among the top three teams were all quite closely contested with the differences being most noticeable at the strategic, team levels.

One interesting result at this competition was that the previous year's champion team competed with minimal modifications and finished roughly in the middle of the final standings. Thus, there was evidence that as a whole, the field of entries was much stronger than during the previous year: roughly half the teams could beat the previous champion.

The 1998 scientific challenge award was shared by Electro Technical Laboratory (ETL), Sony Computer Science Laboratories, Inc., and German Research Center for Artificial Intelligence GmbH (DFKI) for "development of fully automatic commentator systems for RoboCup simulator league."

To encourage the transfer of results from RoboCup to the scientific community at large, RoboCup-98 was the first to host the Multi-Agent Scientific Evaluation Session. 13 different teams participated in the session, in which their adaptability to loss of team-members was evaluated comparatively. Each team was played against the same fixed opponent (the 1997 winner, AT Humboldt'97) four half-games under official RoboCup rules. The first half-game (phase A) served as a base-line. In the other three half-games (phases B-D), 3 players were disabled incrementally: A randomly chosen player, a player chosen by the representative of the fixed opponent to maximize "damage" to the evaluated team, and the goalie. The idea is that a more adaptive team would be able to respond better to these.

Very early on, even during the session itself, it became clear that while in fact most participants agreed intuitively with the evaluation protocol, it wasn't clear how to quantitatively, or even qualitatively, analyse the data. The most obvious measure of the goal-difference at the end of each half may not be sufficient: some teams seem to do better with less players, some do worse. Performance, as measured by the goal-difference, really varied not only from team to team, but also for the same team between phases. The evaluation methodology itself and analysis of the results became open research problems in themselves. To facilitate this line of research, the data from the evaluation was made public at: `http://www.isi.edu/~galk/Eval/`

RoboCup-99

The third international RoboCup championship, RoboCup-99, was held in late July and early August, 1999 in Stockholm, Sweden [3]. It was held in conjunction with the IJCAI-99 conference.

RoboCup 2000

The fourth international RoboCup championship, RoboCup 2000, was held in early September, 2000 in Melbourne, Australia [14]. It was held in conjunction with the PRICAI-2000 conference.

1.3.3. History of the Soccer Manual Effort

The first versions of the manual were written by Itsuki Noda, while developing the soccerserver, and around version 3.00 there were several requests on an updated manual, to better correspond to the server as well as enable newcomers to more easily participate in the RoboCup World Cup Initiative. In the fall of 1998 Peter Stone initiated the Soccer Manual Effort, over which Johan Kummeneje took responsibility to organize and as a result the Soccer Server Manual version 4.0 was released on the 1st of November 1998.

In 1999, the manual for the soccerserver version 5.0 was released. Unfortunately the manual lost part of its pace, and there was no release of the manual for soccerserver version 6.0.

Since 1999, the soccerserver has changed major version to 7 and is continuously developed. Therefore the Soccer Manual Effort has developed a new version, which you are currently reading.

1.4. About This Manual

This manual is the joint effort of the authors from a diverse range of universities and organizations, which build upon the original work of *Itsuki Noda*.
If there are errors, inconsistencies, or oddities, please notify johank@dsv.su.se or fruit@uni-koblenz.de with the location of the error and a suggestion of how it should be corrected.

We are always looking for anyone who has an idea on how to improve the manual, as well as proofread or (re)write a section of the manual. If you have any ideas, or feel that you can contribute with anything to the SoccerServer Manual Effort please mail johank@dsv.su.se or fruit@uni-koblenz.de.

The latest manual can be downloaded at `http://www.dsv.su.se/~johank/RoboCup/manual`.

1.5. Reader's Guide to the Manual

The thesis is written for a wide range of readers, and therefore the chapters are not equally important to all readers. We shortly describe the remaining chapters to give an overview of the thesis.

Chapter 2 introduces the concepts of the simulated league and will help the newcomer to get to terms with the different parts.

Chapter 3 helps the beginners to start compiling and running the software.

Chapter 4 describes the soccerserver.

Chapter 5 describes the soccermonitor.

Chapter 6 describes the soccerclient and how to create one.
Chapter 7 describes the coachclient.
Chapter 8 suggests some further reading.

2. Overview

2.1. Getting Started

This section is designed to give you a quick introduction to the main components of the RoboCup simulator. For each of these components you will find detailed information (i.e. configuration parameters, run-time options, etc.) later on in this manual.

2.1.1. The Server

The server is a system that enables various teams to compete in a game of soccer. Since the match is carried out in a client-server style, there are no restrictions as to how teams are built. The only requirement is that the tools used to develop a team support client-server communication via UDP/IP. This is due to the fact that all communication between the server and each client is done via UDP/IP sockets. Each client is a separate process and connects to the server through a specified port. A team can have up to 11 clients (or players). After a player connects with the server, all messages are transferred through this port. The players send requests to the server regarding the actions they want to perform (e.g. kick the ball, turn, run, etc.). The server receives those messages, handles the requests, and updates the environment accordingly. In addition, the server provides all players with sensory information (e.g. visual data regarding the position of the ball, goals, and other players). It is important to mention that the server is a real-time system working with discrete time intervals (or cycles). Each cycle has a specified duration, and actions that need to be executed in a given cycle, must arrive to the server during the right interval. Therefore, slow performance that results in missing acting opportunities has a major impact on the performance of the team.

2.1.2. The Monitor

The monitor is a visualisation tool that allows people to see what is happening within the server during a game. The information shown on the monitor includes the score, team names, and the positions of all the players and the ball. The monitor also provides a simple interface to the server. For example, when both teams have connected, the "Kick-Off" button on the monitor allows a human referee to start the game. As you will discover later on, to run a game on the server, a monitor is not required. However, if needed, a number of monitors can be connected to the server at the same time (for example if you want to show the same game at different terminals).

2.1.3. The Logplayer

The logplayer can be thought of as a video player. It is a tool that is used to replay matches. When running the server, certain options can be used that will cause the server to store all the data for a given match on the hard drive. (Pretty much like pressing the record button on your video). Then, the logplayer combined with a monitor can be used to replay that game as many times as needed. This is quite usefull for doing team analysis and discovering the strong or weak points of a team. Much like a video player, the logplayer is equipped with play, stop, fast forward and rewind buttons. Also the logplayer allows you to jump to a particular cycle in a game (for example if you only want to see the goals).

2.2. The Rules of the Game

During a game, a number of rules are enforced either by the automated referee within the server, or by a human referee. The aim of this section is to describe how these rules work, and how they affect the game.

2.2.1. Rules Judged by the Automated Referee

Kick-Off

Just before a kick off (either before the game starts, or after every goal), all players must be in their own half. To allow for this to happen, after a goal is scored, the referee suspends the match for an interval of 5 seconds. During this interval, players can use the **move** command to teleport to a position, rather than run to this position, which is much slower and consumes stamina. If a player remains in the opponent half after the 5-second interval has expired, the referee moves the player to a random position within their own half.

Goal

When a team scores, the referee performs a number of tasks. Initially, it announces the goal by broadcasting a message to all players. It also updates the score, moves the ball to the centre mark, and changes the play-mode to kick_off_x (where x is either left or right). Finally, it suspends the match for 5 seconds allowing players to move back to their own half (as described above in the "Kick-Off" section).

Out of Field

When the ball goes out of the field, the referee moves the ball to a proper position (a touchline, corner or goal-area) and changes the play-mode to kick_in, corner_kick, or goal_kick. In the case of a corner kick, the referee places the ball at (1m, 1m) inside the appropriate corner of the field.

Player Clearance

When the play-mode is kick_off, kick_in, or corner_kick, the referee removes all defending players located within a circle centred on the ball. The radius of this circle is a parameter within the server (normally 9.15 meters). The removed players are placed on the perimeter of that circle. When the play-mode is offside, all offending players are moved back to a non-offside position. Offending players in this case are all players in the offside area and all players inside a circle with radius 9.15 meters from the ball. When the play-mode is goal_kick, all offending players are moved outside the penalty area. The offending players cannot re-enter the penalty area while the goal kick takes place. The play-mode changes to play_on immediately after the ball goes outside the penalty area.

Play-Mode Control

When the play-mode is kick_off, kick_in, or corner_kick, the referee changes the play-mode to play_on immediately after the ball starts moving through a **kick** command.

Half-Time and Time-Up

The referee suspends the match when the first or the second half finishes. The default length for each half is 3000 simulation cycles (about 5 minutes). If the match is drawn after the second half, the match is extended. Extra time continues until a goal is scored. The team that scores the first goal in extra time wins the game. This is also known as the “golden goal” rule or “sudden death”.

2.2.2. Rules Judged by the Human Referee

Fouls like “obstruction” are difficult to judge automatically because they concern players’ intentions. To resolve such situations, the server provides an interface for human-intervention. This way, a human-referee can suspend the match and give free kicks to either of the teams. The following are the guidelines that were agreed prior to the RoboCup 2000 competition.

- Surrounding the ball
- Blocking the goal with too many players
- Not putting the ball into play after a given number of cycles
- Intentionally blocking the movement of other players
- Abusing the goalie **catch** command (the goalie may not repeatedly kick and catch the ball, as this provides a safe way to move the ball anywhere within the penalty area).

Flooding the Server with Messages

A player should not send more than 3 or 4 commands per simulation cycle to the soccer server. Abuse may be checked if the server is jammed, or upon request after a game.

Inappropriate Behaviour

If a player is observed to interfere with the match in an inappropriate way, the human-referee can suspend the match and give a free kick to the opposite team.

3. Getting Started

This section contains all the information necessary to get the Soccer Server source files and to install the software. The procedure shown was performed on a computer running GNU/Linux 2.2.17 (check your version with `uname -sr`) with egcs 2.91.66 (check your version with `which g++`) but any reasonably up-to-date installation should do it. In the commands shown below, `->` is supposed to be the command-line prompt.

3.1. Getting and installing the server

① Get source files from one of the Soccer Server sites:

- `http://ci.etl.go.jp/~noda/soccer/server/` (Japan)
- `http://www.robolog.org/server/` (Germany)

This is done by :

a) clicking the Download link and
b) getting the gziped file `sserver-*.tar.gz`

where `*` is the version number[1] of the software. This file contains all the necessary sources for Soccer Server and can be found either by going directly to the Soccer Server FTP site at `http://www.uni-koblenz.de/ag-ki/ROBOCUP/sserver/pub/soccer/server/` or by clicking the Newest version link from the download page.

② Extract the source files by running:

```
-> tar zxvf sserver-*.tar.gz
```

A directory `sserver-*` is created. Now, change the working directory to `sserver-*`. This directory contains the following files:

```
-> ls -a
.               Makefile            logplay         sserver
..              README              logplayer       sserver-csh.tmpl
.cvsignore      coach_lang_grammar  monitor         sserver-tcsh.tmpl
```

[1] Version `7.04` is the current version at the time of writing (March 2001). The example list is output for version `7.04`.

```
Acknowledgement  configure          recfile_change  sserver.org
Changes          configure2         sampleclient    sserver.tmpl
Licence          drawcheck          server          tools
```

Always read the README file first:

```
-> cd sserver-7.04
-> more README
[Directories]
server/
        Source files of soccerserver
monitor/
        Source files of soccermonitor
sampleclient/
        Source files of a very simple client

[How to Make]
        (1) Do configure
        (2) Do make

[How to Start]
        (1) start soccerserver.
        (2) start a couple of soccermonitors you want.

        "sserver" is a sample script to invoke soccerserver and soccermonitor.

[Required Softwares]
        GCC 2.7.0 or later
        X11R5 or R6
                Some old version of R5 may cause problems of display.

[Suppoted OSs]
        SunOS 4.1.x
        Solaris 2.x
        DEC OSF1
        Linux RH 4.xx, 5.xx, 6.xx
```

The rest of the README file contains the license under which you may use and modify the software. Please, make sure you read it in your own time.

③ Do "configure" following the instructions in the README file for your platform:

```
-> ./configure
```

```
Do you use X11R6.x? [y or n]
  [default=y]:y
Enter X11R6 includes directory.
  [default=/usr/X11R6/include]:
Enter X11R6 libraries directory.
  [default=/usr/X11R6/lib]:
Do you use dynamic linking? [y or n]
  [default=y]:y
Enter compiler flag(s).
  [default=-O2 -pipe]:

Configuration Summary:
  OS type = Linux_22
  X11 revision = 6
  X11 include PATH = /usr/X11R6/include
  X11 libraries PATH = /usr/X11R6/lib
  Link style = Dynamic
  Compiler flag(s) = -O2 -pipe

Creating Makefile...[server][monitor][sampleclient][recfile_change][logplayer][drawcheck].
Creating sserver script.
Done.
```

④ Do “make”

```
-> make
g++ -c -pipe -DLinux -DLinux_2_2   main.C
g++ -c -pipe -DLinux -DLinux_2_2   field.C
g++ -c -pipe -DLinux -DLinux_2_2   object.C
g++ -c -pipe -DLinux -DLinux_2_2   netif.C
.
.
.
g++ -c -O2 -pipe -DLinux -DLinux_2_2 -I../server drawcheck.C
g++ -c -O2 -pipe -DLinux -DLinux_2_2 -I../server netif.C
g++ -o drawcheck drawcheck.o netif.o  -lm
```

If there are errors in the `make`, please check the `g++` environment in your system.

3.2. Download sites

There are two download sites for the Soccer Server:

1. `http://ci.etl.go.jp/~noda/soccer/server/` (Japan)

2. http://www.robolog.org/server/ (Germany)

The original home of the Soccer Server is the top site but it is soon to go out of commission. At the time of writing (March 2001) both sites are still up-and-running but the German site is to take over as the official home of the Soccer Server at some point in the summer of 2001. Users are prompted to use the German site to avoid future problems.

3.3. How to start the server

The `sserver-*` directory contains a `sserver` file. This is a shell script that is made by `configure`. Running it with `sserver` does three things :

1. it starts a Soccer Server
2. it starts a Soccer Monitor
3. when no longer needed, it stops the Soccer Server process

The Soccer Server runs in the background and produces some output to the terminal where the script was started from. Also, a window appears on the screen. This window is the Soccer Monitor so the user can actually watch the game. The output of running the `sserver` script should look like this :

```
-> sserver
Soccer Server Ver. 7.04
Copyright (C) 1995, 1996, 1997, 1998, 1999, 2000 Electrotechnical Laboratory.
Itsuki Noda, Yasuo Kuniyoshi and Hitoshi Matsubara.
2001 RoboCup Soccer Server Maintainance Group.
Patrick Riley, Tom Howard, Jan Wendler, Itsuki Noda
wind factor: rand: 0.000000, vector: (0.000000, 0.000000)
```

In order to actually start a match on the Soccer Server, the user must connect some clients to the server (maximum of 11 per side). When these clients are ready, the user can click the `Kick Off` button on the Soccer Monitor to start the game. It is likely that you have not yet programmed your own clients, in which case, you can read section 3.6 for instructions how to set up a whole match with two of the available teams that other RoboCuppers have contributed.

Also, there is a sample client included with every distribution of the Soccer Server which can be found in the `sampleclient` directory. To run one of these clients :

```
-> cd sampleclient
-> ls
Makefile       Makefile.tmpl2  client.c  client.o
Makefile.tmpl  client          client.h

-> client
```

Running `client` attempts to connect to the server using default parameters (host=localhost, port=6000). Of course, these server parameters can be changed using the arguments that Soccer Server accepts when it is started. When the client is started, you need to initialise its connection to the server. The `sampleclient` is made so that the user types Soccer Server commands on the command line. So, to initialise the connection :

```
(init MyTeam)
```

You will notice that one of the two teams is now named "MyTeam" and one of the players that are standing by the side-line is active. This player corresponds to the client you've just initialised. Also, notice the information that the client writes on the terminal. This is what the client receives from the server. In the following text, the first two lines correspond to the initialisation[2] and the other data is the information that the server sends to this client :

```
send 6000 : (init MyTeam)
recv 1067 : (init l 1 before_kick_off)

recv 1067 : (see 0 ((goal r) 66.7 33) ((flag r t) 55.7 3)
((flag p r t) 42.5 23) ((flag p r c) 53.5 43))

recv 1067 : (see 0 ((goal r) 66.7 33) ((flag r t) 55.7 3)
((flag p r t) 42.5 23) ((flag p r c) 53.5 43))
```

You can still type commands (such as `(move 0 0)` or `(turn 45)`) that the player will then send to the server. You should be able to see the result of these commands on the Soccer Monitor window.

3.4. How to stop the server

The correct procedure for stopping the server is :

1. Stop all clients (players)
2. Stop all Soccer Monitors by clicking on the `Quit` button
3. Hit CTRL-C at the terminal window where you started the Soccer Server in order to terminate it

If you follow this procedure, you will not only stop all visible running processes but also make sure that all those processes that may be running in the background (such as the Soccer Server) are also stopped. The problem that arises when you don't properly shut down the Soccer Server is that you may not be able to start another process unless you start it with different parameters.

[2] The response from the server means that the client plays for the left side, has the number one and the play mode is before_kick_off

NOTE : It is sometimes useful and convenient to terminate processes using their name. Using the `kill` operating system command involves finding the process number of the process you want to stop using the `ps` command. A simpler way to eradicate all processes that have a specific name is by means of the `killall` command, for example : "`killall soccermonitor`" is sufficient to kill all processes with the name Soccer Monitor.

3.5. Supported platforms

The Soccer Server supports the following platforms[3] :

- SunOS 4.1.x
- Solaris 2.x
- DEC OSF1
- Linux RedHat 4.xx, 5.xx, 6.xx

3.6. The process of a match

[3]Supported platforms may change. Please, check the `README` file that came with your version of the software.

4. Soccer Server

4.1. Objects

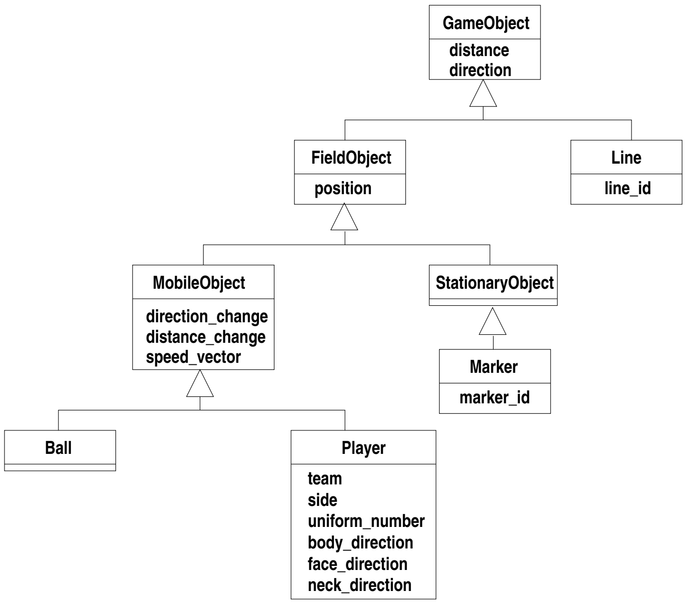

Figure 4.1.: UML diagram of the objects in the simulation

4.2. Protocols

4.2.1. Client Command Protocol

Connecting, reconnecting, and disconnecting

From client to server	From server to client
(init *TeamName* [(version *VerNum*)] [(goalie)]) *TeamName* ::= $(-\|\_\|a-z\|A-Z\|0-9)^+$ *VerNum* ::= the protocol version (e.g. 7.0)	**(init *Side Unum PlayMode*)** *Side* ::= l | r *Unum* ::= **1** ~ 11 *PlayMode* ::= one of play modes **(error no_more_team_or_player_or_goalie)**
(reconnect *TeamName Unum*) *TeamName* ::= $(-\|\_\|a-z\|A-Z\|0-9)^+$	**(reconnect *Side PlayMode*)** *Side* ::= l | r *Unum* ::= **1** ~ **11** *PlayMode* ::= one of play modes **(error no_more_team_or_player)** **(error reconnect)**
(bye)	

If your client connects or reconnects sucessfully with a protocol version ≥ 7.0, the server will additionally send following messages: a message containing the server parameters, a message containing the player parameters and a message containing the player types. The format is given below. Finally, the player will receive a message on changed players (see Sec. 4.6).

- **(server_param** ***gwidth inertia_moment psize pdecay prand pweight pspeed_max paccel_max stamina_max stamina_inc recover_init recover_dthr recover_min recover_dec effort_init effort_dthr effort_min effort_dec effort_ithr effort_inc kick_rand team_actuator_noise prand_factor_l prand_factor_r kick_rand_factor_l kick_rand_factor_r bsize bdecay brand bweight bspeed_max baccel_max dprate kprate kmargin ctlradius ctlradius_width maxp minp maxm minm maxnm minnm maxn minn visangle visdist windir winforce winang winrand kickable_area catch_area_l catch_area_w catch_prob goalie_max_moves ckmargin offside_area win_no win_random say_cnt_max SayCoachMsgSize clang_win_size clang_define_win clang_meta_win clang_advice_win clang_info_win clang_mess_delay clang_mess_per_cycle half_time sim_st send_st recv_st sb_step lcm_st SayMsgSize hear_max hear_inc hear_decay cban_cycle slow_down_factor useoffside kickoffoffside offside_kick_margin audio_dist dist_qstep land_qstep dir_qstep dist_qstep_l dist_qstep_r land_qstep_l land_qstep_r dir_qstep_l dir_qstep_r CoachMode CwRMode old_hear sv_st start_goal_l start_goal_r fullstate_l fullstate_r drop_time*****)**
- **(player_param** ***player_types subs_max pt_max player_speed_max_delta_min player_speed_max_delta_max stamina_inc_max_delta_factor player_decay_delta_min player_decay_delta_max inertia_moment_delta_factor dash_power_rate_delta_min dash_power_rate_delta_max player_size_delta_factor kickable_margin_delta_min kickable_margin_delta_max kick_rand_delta_factor extra_stamina_delta_min extra_stamina_delta_max effort_max_delta_factor effort_min_delta_factor*****)**

- for each available player type a message of the form
  (player_type *id player_speed_max stamina_inc_max player_decay inertia_moment dash_power_rate player_size kickable_margin kick_rand extra_stamina effort_max effort_min*)

Client Control

From client to server	Only once per cycle
(catch *Direction*) *Direction* ::= ***minmoment*** ~ ***maxmoment*** degrees	Yes
(change_view *Width Quality*) *Width* ::= narrow | normal | wide *Quality* ::= high | low	No
(dash *Power*) *Power* ::= ***minpower*** ~ ***maxpower*** Note: backward dash consumes double stamina.	Yes
(kick *Power Direction*) *Power* ::= ***minpower*** ~ ***maxpower*** *Direction* ::= ***minmoment*** ~ ***maxmoment*** degrees	Yes
(move *X Y*) *X* ::= **-52.5** ~ **52.5** *Y* ::= **-34** ~ **34**	Yes
(say *Message*) *Message* ::= a message	No
(sense_body) The server returns (**sense_body** *Time* **(view_mode {high | low} {narrow | normal | wide})** **(stamina *Stamina Effort*)** **(speed *AmountOfSpeed DirectionOfSpeed*)** **(head_angle *HeadAngle*)** **(kick *KickCount*)** **(dash *DashCount*)** **(turn *TurnCount*)** **(say *SayCount*)** **(turn_neck *TurnNeckCount*)** **(catch *CatchCount*)** **(move *MoveCount*)** **(change_view *ChangeViewCount*)**)	No
(score) The server returns (**score** *Time OurScore TheirScore*)	No
(turn *Moment*) *Moment* ::= ***minmoment*** ~ ***maxmoment*** degrees	Yes
(turn_neck *Angle*) *Angle* ::= ***minneckmoment*** ~ ***maxneckmoment*** degrees turn_neck is relative to the direction of the body. Can be invoked in the same cycle as a turn, dash or kick.	Yes

The server may respond to the above commands with the errors:

(error unknown_command)

(error illegal_command_form)

4.2.2. Client Sensor Protocol

From server to client
(**hear** *Time Sender* "*Message*") (**hear** *Time Online_Coach Coach_Language_Message*) *Time* ::= simulation cycle of the **soccerserver** *Sender* ::= **online_coach_left** | **online_coach_right** | **coach** | **referee** | **self** | *Direction* *Direction* ::= **-180** ∼**180** degrees *Message* ::= string *Online_Coach* ::= **online_coach_left** | **online_coach_right** *Coach_Language_Message* ::= see the standard coach language section
(**see** *Time ObjInfo*$^{+}$) *Time* ::= simulation cycle of the **soccerserver** *ObjInfo* ::= (***ObjName Distance Direction DistChange DirChange BodyFacingDir HeadFacingDir***) | (***ObjName Distance Direction DistChange DirChange*** | (***ObjName Distance Direction***) | (***ObjName Direction***) *ObjName* ::= (**p** ["*Teamname*" [*UniformNumber* [**goalie**]]]) | **(b)** | **(g** [**l**|**r**]) | **(f c)** | **(f** [**l**|**c**|**r**] [**t**|**b**]) | **(f p** [**l**|**r**] [**t**|**c**|**b**]) | **(f g** [**l**|**r**] [**t**|**b**]) | **(f** [**l**|**r**|**t**|**b**] **0)** | **(f** [**t**|**b**] [**l**|**r**] [**10**|**20**|**30**|**40**|**50**]) | **(f** [**l**|**r**] [**t**|**b**] [**10**|**20**|**30**]) | **(l** [**l**|**r**|**t**|**b**]) | **(B)** | **(F)** | **(G)** | **(P)** *Distance* ::= positive real number *Direction* ::= **-180** ∼**180** degrees *DistChange* ::= real number *DirChange* ::= real number *HeadFaceDir* ::= **-180** ∼**180** degrees *BodyFaceDir* ::= **-180** ∼**180** degrees *Teamname* ::= string *UniformNumber* ::= 1 ∼11
(**sense_body** *Time* **(view_mode** {**high** | **low**} {**narrow** | **normal** | **wide**} **)** **(stamina *Stamina Effort*)** **(speed *AmountOfSpeed DirectionOfSpeed*)** **(head_angle *HeadAngle*)** **(kick *KickCount*)** **(dash *DashCount*)** **(turn *TurnCount*)** **(say *SayCount*)** **(turn_neck *TurnNeckCount*)** **(catch *CatchCount*)** **(move *MoveCount*)** **(change_view *ChangeViewCount*)**) *Time* ::= simulation cycle of the **soccerserver** *Stamina* ::= positive real number *Effort* ::= positive real number *AmountOfSpeed* ::= positive real number *DirectionOfSpeed* ::= **-180** ∼**180** degrees *HeadAngle* ::= **-180** ∼**180** degrees **Count* ::= positive integer

4.3. Sensor Models

A RoboCup agent has three different sensors. The aural sensor detects messages sent by the referee, the coaches and the other players. The visual sensor detects visual information about the field, like the distance and direction to objects in the player's current field of view. The visual sensor also works as a proximity sensor by “seeing” objects that are close, but behind the player. The body sensor detects the current “physical” status of the player, like its stamina, speed and neck angle. Together the sensors give the agent quite a good picture of the environment.

4.3.1. Aural Sensor Model

Aural sensor messages are sent when a client or a coach sends a **say** command. The calls from the referee is also received as aural messages. All messages are received immediately.

The format of the aural sensor message from the soccer server is:

> **(hear *Time Sender* "*Message*")**
>
> *Time* indicates the current time.
>
> *Sender* is the relative direction to the sender if it is another player, otherwise it is one of the following:
>
> > **self**: when the sender is yourself.
> >
> > **referee**: when the sender is the referee.
> >
> > **online_coach_left** or **online_coach_right**: when the sender is one of the online coaches.
>
> *Message* is the message. The maximum length is ***say_msg_size*** bytes. The possible messages from the referee are described in Section 4.7.1.

The server parameters that affects the aural sensor are described in Tab. 4.1.

Parameter in `server.conf`	**Value**
audio_cut_dist	50.0
hear_max	2
hear_inc	1
hear_decay	2
say_msg_size	512

Table 4.1.: Parameters for the aural sensor

Capacity of the Aural Sensor

A player can only hear a message if the player's hear capacity is at least ***hear_decay***, since the hear capacity of the player is decreased by that number when a message is heard. Every cycle the hear capacity is increased with ***hear_inc***. The maximum hear capacity is ***hear_max***. To avoid a team from making the other team's communication useless by overloading the channel the players have separate hear capacities for each team. With the current server.conf file this means that a player can hear at most one message from each team every second simulation cycle.

If more messages arrive at the same time than the player can hear the messages actually heard are undefined. (The current implementation choose the messages according to the order of arrival.) This rule does not include messages from the referee, or messages from oneself. In other words, a player can say a message and hear a message from another player in the same timestep.

Range of Communication

A message said by a player is transmitted only to players within ***audio_cut_dist*** meters from that player. For example, a defender, who may be near his own goal, can hear a message from his goal-keeper but a striker who is near the opponent goal can not hear the message. Messages from the referee can be heard by all players.

Aural Sensor Example

This example should show which messages get through and how to calculated the hear capacity.

Example: Each coach sends a message every cycle. The referee send a message every cycle. The four players in the example all send a message every cycle. Show which messages gets through during 10 cycles (6 might be enough).

4.3.2. Vision Sensor Model

The visual sensor reports the objects currently seen by the player. The information is automatically sent to the player every ***sense_step***, currently 150, milli-seconds.

Visual information arrives from the server in the following basic format:

(**see** *ObjName Distance Direction DistChng DirChng BodyDir HeadDir*)

where

$$
\begin{aligned}
\textit{ObjName} \;::=\; & (\mathbf{p}\ \text{"}\textit{Teamname}\text{"}\ \textit{UniformNumber}\ \mathbf{goalie}) \\
& \mid (\mathbf{g}\ [\mathbf{l}|\mathbf{r}]) \\
& \mid (\mathbf{b}) \\
& \mid (\mathbf{f}\ \mathbf{c}) \\
& \mid (\mathbf{f}\ [\mathbf{l}|\mathbf{c}|\mathbf{r}]\ [\mathbf{t}|\mathbf{b}]) \\
& \mid (\mathbf{f}\ \mathbf{p}\ [\mathbf{l}|\mathbf{r}]\ [\mathbf{t}|\mathbf{c}|\mathbf{b}]) \\
& \mid (\mathbf{f}\ \mathbf{g}\ [\mathbf{l}|\mathbf{r}]\ [\mathbf{t}|\mathbf{b}]) \\
& \mid (\mathbf{f}\ [\mathbf{l}|\mathbf{r}|\mathbf{t}|\mathbf{b}]\ \mathbf{0}) \\
& \mid (\mathbf{f}\ [\mathbf{t}|\mathbf{b}]\ [\mathbf{l}|\mathbf{r}]\ [\mathbf{10}|\mathbf{20}|\mathbf{30}|\mathbf{40}|\mathbf{50}]) \\
& \mid (\mathbf{f}\ [\mathbf{l}|\mathbf{r}]\ [\mathbf{t}|\mathbf{b}]\ [\mathbf{10}|\mathbf{20}|\mathbf{30}]) \\
& \mid (\mathbf{l}\ [\mathbf{l}|\mathbf{r}|\mathbf{t}|\mathbf{b}])
\end{aligned}
$$

Distance, *Direction*, *DistChng* and *DirChng* are calculated in the following way:

$$
\begin{aligned}
p_{rx} &= p_{xt} - p_{xo} && (4.1)\\
p_{ry} &= p_{yt} - p_{yo} && (4.2)\\
v_{rx} &= v_{xt} - v_{xo} && (4.3)\\
v_{ry} &= v_{yt} - v_{yo} && (4.4)\\
\textit{Distance} &= \sqrt{p_{rx}^2 + p_{ry}^2} && (4.5)\\
\textit{Direction} &= \arctan(p_{ry}/p_{rx}) - a_o && (4.6)\\
e_{rx} &= p_{rx}/\textit{Distance} && (4.7)\\
e_{ry} &= p_{ry}/\textit{Distance} && (4.8)\\
\textit{DistChng} &= (v_{rx} * e_{rx}) + (v_{ry} * e_{ry}) && (4.9)\\
\textit{DirChng} &= [(-(v_{rx} * e_{ry}) + (v_{ry} * e_{rx}))/\textit{Distance}] * (180/\pi) && (4.10)\\
\textit{BodyDir} &= \textit{PlayerBodyDir} - \textit{AgentBodyDir} - \textit{AgentHeadDir} && (4.11)\\
\textit{HeadDir} &= \textit{PlayerHeadDir} - \textit{AgentBodyDir} - \textit{AgentHeadDir} && (4.12)
\end{aligned}
$$

where (p_{xt}, p_{yt}) is the absolute position of the target object, (p_{xo}, p_{yo}) is the absolute position of the sensing player, (v_{xt}, v_{yt}) is the absolute velocity of the target object, (v_{xo}, v_{yo}) is the absolute velocity of the sensing player, and a_o is the absolute direction the sensing player is facing. The absolute facing direction of a player is the sum of the *BodyDir* and the *HeadDir* of that player. In addition to it, (p_{rx}, p_{ry}) and (v_{rx}, v_{ry}) are respectively the relative position and the relative velocity of the target, and (e_{rx}, e_{ry}) is the unit vector that is parallel to the vector of the relative position. *BodyDir* and *HeadDir* are only included if the observed object is a player, and is the body and head directions of the observed player relative to the body and head directions of the observing player. Thus, if both players have their bodies turned in the same direction, then *BodyDir* would be 0. The same goes for *HeadDir*.

The **(goal r)** object is interpreted as the center of the right hand side goalline. **(f c)** is a virtual flag at the center of the field. **(f l b)** is the flag at the lower left of the field. **(f p l b)** is a virtual flag at the lower right corner of the penalty box on the left side

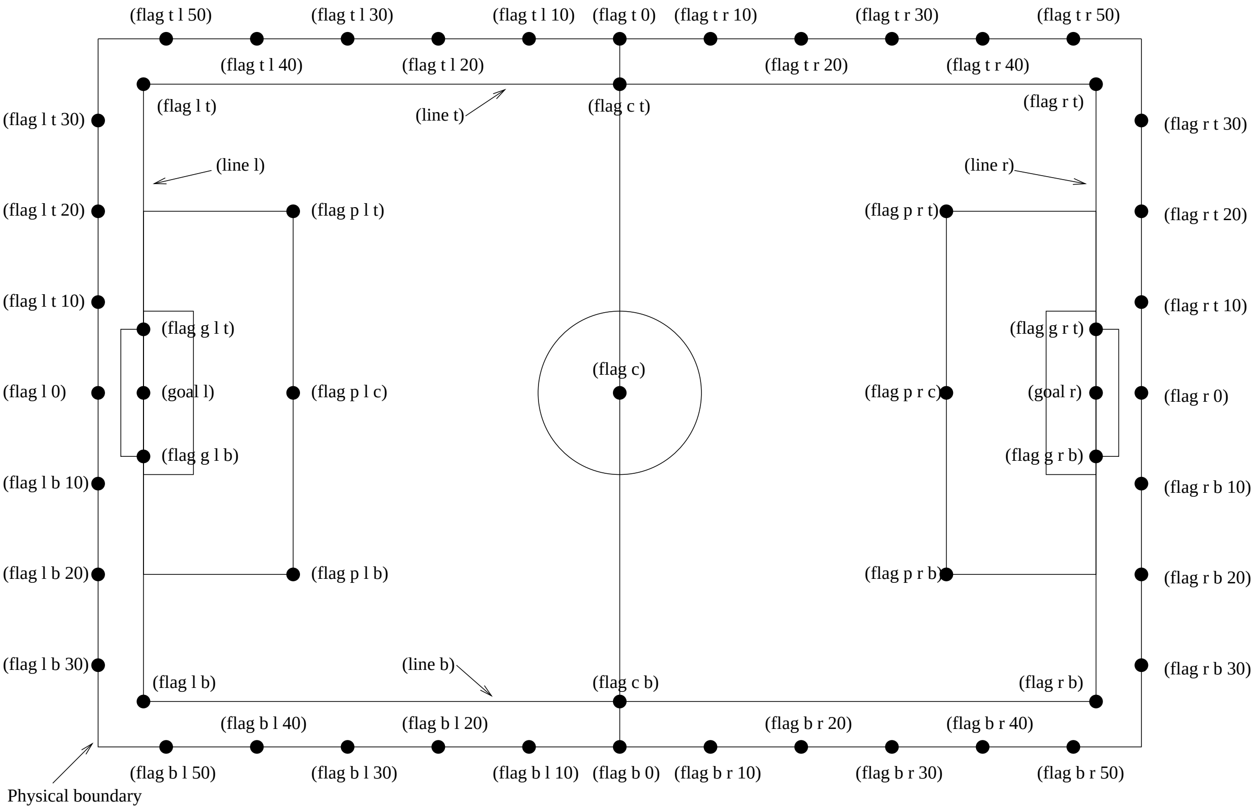

Figure 4.2.: The flags and lines in the simulation.

of the field. **(f g l b)** is a virtual flag marking the right goalpost on the left goal. The remaining types of flags are all located 5 meters outside the playing field. For example, **(f t l 20)** is 5 meters from the top sideline and 20 meters left from the center line. In the same way, **(f r b 10)** is 5 meters right of the right sideline and 10 meters below the center of the right goal. Also, **(f b 0)** is 5 meters below the midpoint of the bottom sideline.

In the case of **(l ...)**, *Distance* is the distance to the point where the center line of the player's view crosses the line, and *Direction* is the direction of the line.

Range of View

The visible sector of a player is dependant on several factors. First of all we have the server parameters ***sense_step*** and ***visible_angle*** which determines the basic time step between visual information and how many degrees the player's normal view cone is. The current default values are 150 milli-seconds and 90 degrees.

The player can also influence the frequency and quality of the information by changing *ViewWidth* and *ViewQuality*.

To calculate the current ***view_frequency*** and ***view_angle*** of the agent use equations 4.13 and 4.14.

$$\boldsymbol{view\_frequency} = \boldsymbol{sense\_step} * \boldsymbol{view\_quality\_factor} * \boldsymbol{view\_width\_factor} \quad (4.13)$$

where ***view_quality_factor*** is 1 iff *ViewQuality* is ***high*** and 0.5 iff *ViewQuality* is ***low***; ***view_width_factor*** is 2 iff *ViewWidth* is ***narrow***, 1 iff *ViewWidth* is ***normal***, and 0.5 iff *ViewWidth* is ***wide***.

$$\boldsymbol{view\_angle} = \boldsymbol{visible\_angle} * \boldsymbol{view\_width\_factor} \quad (4.14)$$

where ***view_width_factor*** is 0.5 iff *ViewWidth* is ***narrow***, 1 iff *ViewWidth* is ***normal***, and 2 iff *ViewWidth* is ***wide***.

The player can also "see" an object if it's within ***visible_distance*** meters of the player. If the objects is within this distance but not in the view cone then the player can know only the type of the object (ball, player, goal or flag), but not the exact name of the object. Moreover, in this case, the capitalized name, that is "**B**", "**P**", "**G**" and "**F**", is used as the name of the object rather than "**b**", "**p**", "**g**" and "**f**".

The following example and Fig. 4.3 are taken from [17].

The meaning of the ***view_angle*** parameter is illustrated in Fig. 4.3. In this figure, the viewing agent is the one shown as two semi-circles. The light semi-circle is its front. The black circles represent objects in the world. Only objects within ***view_angle***°/2, and those within ***visible_distance*** of the viewing agent can be seen. Thus, objects *b* and *g* are not visible; all of the rest are.

As object *f* is directly in front of the viewing agent, its angle would be reported as 0 degrees. Object *e* would be reported as being roughly -40°, while object *d* is at roughly 20°.

Also illustrated in Fig. 4.3, the amount of information describing a player varies with how far away the player is. For nearby players, both the team and the uniform number of the player are reported. However, as distance increases, first the likelihood that the uniform number is visible decreases, and then even the team name may not be visible. It is assumed in the server that ***unum_far_length*** $\leq$ ***unum_too_far_length*** $\leq$ ***team_far_length*** $\leq$ ***team_too_far_length***. Let the player's distance be *dist*. Then

- If *dist* $\leq$ ***unum_far_length***, then both uniform number and team name are visible.
- If ***unum_far_length*** $<$ *dist* $<$ ***unum_too_far_length***, then the team name is always visible, but the probability that the uniform number is visible decreases linearly from 1 to 0 as *dist* increases.
- If *dist* $\geq$ ***unum_too_far_length***, then the uniform number is not visible.
- If *dist* $\leq$ ***team_far_length***, then the team name is visible.
- If ***team_far_length*** $<$ *dist* $<$ ***team_too_far_length***, then the probability that the team name is visible decreases linearly from 1 to 0 as *dist* increases.

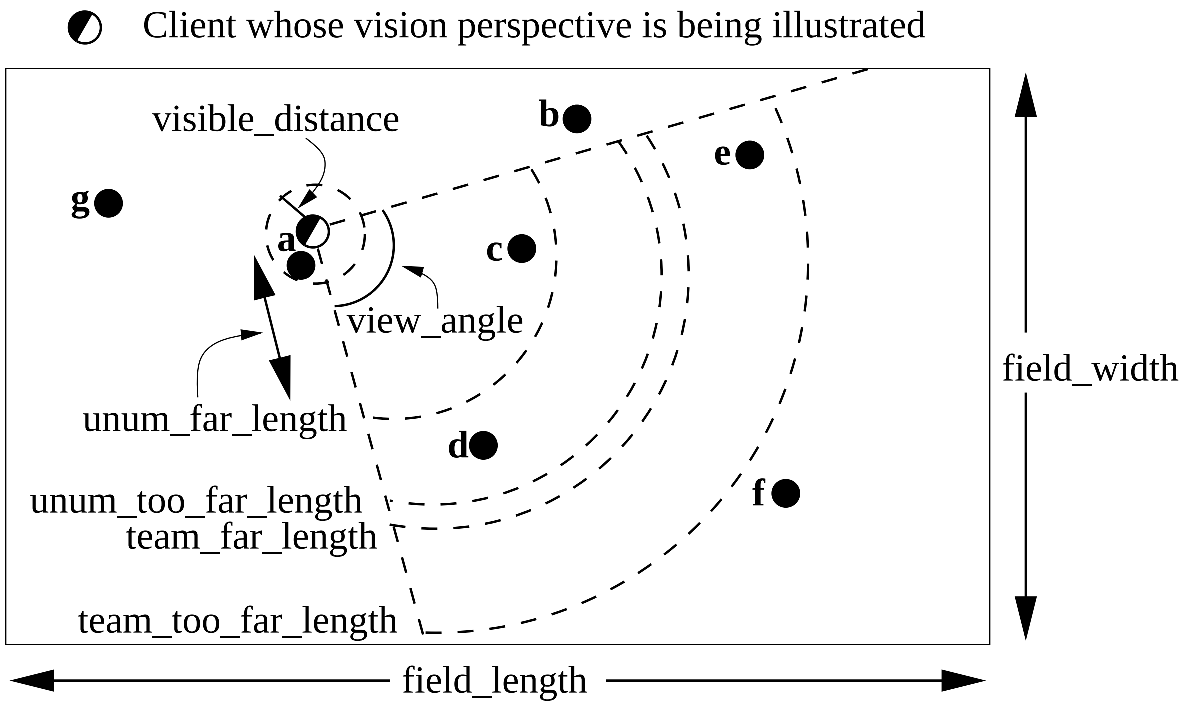

Figure 4.3.: The visible range of an individual agent in the soccer server. The viewing agent is the one shown as two semi-circles. The light semi-circle is its front. The black circles represent objects in the world. Only objects within ***view_angle***°/2, and those within ***visible_distance*** of the viewing agent can be seen. ***unum_far_length***, ***unum_too_far_length***, ***team_far_length***, and ***team_too_far_length*** affect the amount of precision with which a players' identity is given. Taken from [17].

- If $dist \geq$ ***team_too_far_length***, then the team name is not visible.

For example, in Fig. 4.3, assume that all of the labeled circles are players. Then player c would be identified by both team name and uniform number; player d by team name, and with about a 50% chance, uniform number; player e with about a 25% chance, just by team name, otherwise with neither; and player f would be identified simply as an anonymous player.

Parameter in `server.conf`	**Value**
sense_step	150
visible_angle	90.0
visible_distance	3.0
unum_far_length[a]	20.0
unum_too_far_length[a]	40.0
team_far_length[a]	40.0
team_too_far_length[a]	60.0
quantize_step	0.1
quantize_step_l	0.01

[a]Not in `server.conf`, but compiled into the server

Table 4.2.: Parameters for the visual sensors

Visual Sensor Noise Model

In order to introduce noise in the visual sensor data the values sent from the server is quantized. For example, the distance value of the object, in the case where the object in sight is a ball or a player, is quantized in the following manner:

$$d' = \mathrm{Quantize}(\exp(\mathrm{Quantize}(\log(d), \boldsymbol{quantize\_step})), 0.1) \tag{4.15}$$

where d and d' are the exact distance and quantized distance respectively, and

$$\mathrm{Quantize}(V, Q) = \mathrm{ceiling}(V/Q) \cdot Q \tag{4.16}$$

This means that players can not know the exact positions of very far objects. For example, when distance is about 100.0 the maximum noise is about 10.0, while when distance is less than 10.0 the noise is less than 1.0.

In the case of flags and lines, the distance value is quantized in the following manner.

$$d' = \mathrm{Quantize}(\exp(\mathrm{Quantize}(\log(d), \boldsymbol{quantize\_step\_l})), 0.1) \tag{4.17}$$

4.3.3. Body Sensor Model

The body sensor reports the current "physical" status of the player. The information is automatically sent to the player every ***sense_body_step***, currently 100, milli-seconds.

The format of the body sensor message is:

(**sense_body** *Time*
(view_mode ViewQuality ViewWidth)
(stamina Stamina Effort)
(speed AmountOfSpeed DirectionOfSpeed)
(head_angle HeadDirection)
(kick KickCount)
(dash DashCount)
(turn TurnCount)
(say SayCount)
(turn_neck TurnNeckCount)
(catch CatchCount)
(move MoveCount)
(change_view ChangeViewCount))

ViewQuality is one of ***high*** and ***low***.

ViewWidth is one of ***narrow***, ***normal***, and ***wide***.

AmountOfSpeed is an approximation of the amount of the player's speed.

DirectionOfSpeed is an approximation of the direction of the player's speed.

HeadDirection is the relative direction of the player's head.

The *Count* variables are the total number of commands of that type executed by the server. For example *DashCount* = 134 means that the player has executed 134 **dash** commands so far.

The semantics of the parameters are described where they are actually used. The *ViewQuality* and *ViewWidth* parameters are for example described in the Section 4.3.2.

The server parameters that affects the body sensor are described in Tab. 4.3.

Parameter in `server.conf`	**Value**
sense_body_step	100

Table 4.3.: Parameters for the body sensor

4.4. Movement Model

In each simulation step, movement of each object is calculated as following manner:

$$\begin{aligned}
(u_x^{t+1}, u_y^{t+1}) &= (v_x^t, v_y^t) + (a_x^t, a_y^t)\text{: accelerate} \\
(p_x^{t+1}, p_y^{t+1}) &= (p_x^t, p_y^t) + (u_x^{t+1}, u_y^{t+1})\text{: move} \\
(v_x^{t+1}, v_y^{t+1}) &= \text{decay} \times (u_x^{t+1}, u_y^{t+1})\text{: decay speed} \\
(a_x^{t+1}, a_y^{t+1}) &= (0, 0)\text{: reset acceleration}
\end{aligned} \tag{4.18}$$

where, (p_x^t, p_y^t), and (v_x^t, v_y^t) are respectively position and velocity of the object in timestep t. decay is a decay parameter specified by **ball_decay** or **player_decay**. (a_x^t, a_y^t) is acceleration of object, which is derived from *Power* parameter in **dash** (in the case the object is a player) or **kick** (in the case of a ball) commands in the following manner:

$$(a_x^t, a_y^t) = \mathit{Power} \times \mathit{power\_rate} \times (\cos(\theta^t), \sin(\theta^t))$$

where θ^t is the direction of the object in timestep t and *power_rate* is **dash_power_rate** or is calculated from **kick_power_rate** as described in Sec. 4.5.3. In the case of a player, this is just the direction the player is facing. In the case of a ball, its direction is given as the following manner:

$$\theta^t_{\text{ball}} = \theta^t_{\text{kicker}} + \mathit{Direction}$$

where θ^t_{ball} and θ^t_{kicker} are directions of ball and kicking player respectively, and *Direction* is the second parameter of a **kick** command.

4.4.1. Movement Noise Model

In order to reflect unexpected movements of objects in real world, ß adds noise to the movement of objects and parameters of commands.

Concerned with movements, noise is added into Eqn. 4.18 as follows:

$$(u_x^{t+1}, u_y^{t+1}) = (v_x^t, v_y^t) + (a_x^t, a_y^t) + (\tilde{r}_{\text{rmax}}, \tilde{r}_{\text{rmax}})$$

where $\tilde{r}_{\text{max}}$ is a random number whose distribution is uniform over the range $[-\text{rmax}, \text{rmax}]$. rmax is a parameter that depends on amount of velocity of the object as follows:

$$\text{rmax} = \text{rand} \cdot |(v_x^t, v_y^t)|$$

where rand is a parameter specified by **player_rand** or **ball_rand**.

Noise is added also into the *Power* and *Moment* arguments of a command as follows:

$$\mathit{argument} = (1 + \tilde{r}_{\text{rand}}) \cdot \mathit{argument}$$

4.4.2. Collision Model

If at the end of the simulation cycle, two objects overlap, then the objects are moved back until they do not overlap. Then the velocities are multiplied by -0.1. Note that it is possible for the ball to go through a player as long as the ball and the player never overlap at the end of the cycle.

4.5. Action Models

4.5.1. Catch Model

The goalie is the only player with the ability to catch a ball. The goalie can catch the ball in play mode '`play_on`' in any direction, if the ball is within the catchable area and the goalie is inside the penalty area. If the goalie catches into direction φ, the catchable area is a rectangular area of length ***catchable_area_l*** and width ***catchable_area_w*** in direction φ (see Fig. 4.4). The ball will be caught with probability ***catch_probability***, if it is inside this area (and it will not be caught if it is outside this area). For the current values of catch command parameters see Tab. 4.4.

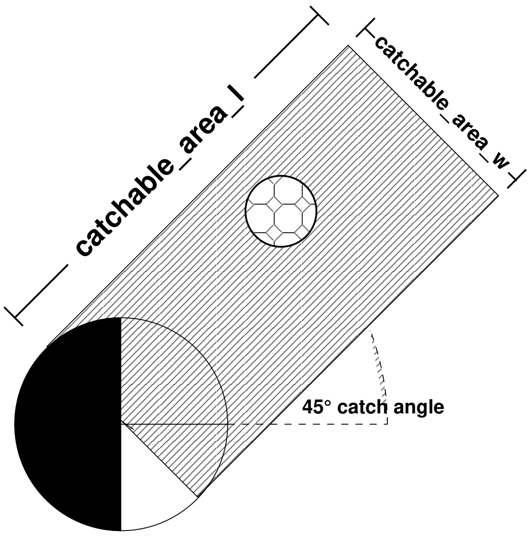

Figure 4.4.: Catchable area of the goalie when doing a catch 45

If a catch command was unsuccessful, it takes ***catch_ban_cycle*** cycles until another catch command can be used (catch commands during this time have simply no effect). If the goalie succeeded in catching the ball, the play mode will change to '`goalie_catch_ball_[l|r]`' first and '`free_kick_[l|r]`', after that during the same cycle. Once the goalie caught the ball, it can use the **move** command to move with the ball inside the penalty area. The goalie can use the **move** command ***goalie_max_moves*** times before it kicks the ball. Additional **move** commands do not have any effect and the server will respond with `(error too_many_moves)`. Please note that catching the ball, moving around, kicking the ball a short distance and immediately catching it again to move more than ***goalie_max_moves*** times is considered as ungentlemanly play.

Parameter in `server.conf`	Value
catchable_area_l	2.0
catchable_area_w	1.0
catch_probability	1.0
catch_ban_cycle	5
goalie_max_moves	2

Table 4.4.: Parameters for the goalie catch command

4.5.2. Dash Model (incl. stamina model)

Dash Model

The **dash** command is used to accelerate the player in direction of its body. **dash** takes the acceleration *power* as a parameter. The valid range for the acceleration *power* can be configured in `server.conf`, the respective parameters are ***minpower*** and ***maxpower***. For the current values of parameters for the dash model, see Tab. 4.5.

Each player has a certain amount of stamina that will be consumed by **dash** commands. At the beginning of each half, the stamina of a player is set to ***stamina_max***. If a player accelerates forward ($power > 0$), stamina is reduced by *power*. Accelerating backwards ($power < 0$) is more expensive for the player: stamina is reduced by $-2 \cdot power$. If the player's stamina is lower than the power needed for the **dash**, *power* is reduced so that the **dash** command does not need more stamina than available. Heterogeneous players will use some extra stamina every time the available power is lower than the needed stamina. The amount of extra stamina depends on the player type and the parameters ***extra_stamina_delta_min*** and ***extra_stamina_delta_max***.

After reducing the stamina, the server calculates the *effective dash power* for the **dash** command. The effective dash power *edp* depends on the ***dash_power_rate*** and the current effort of the player. The effort of a player is a value between ***effort_min*** and ***effort_max***; it is dependent on the stamina management of the player (see below).

$$\text{edp} = \text{effort} \cdot \text{dash\_power\_rate} \cdot \text{power} \tag{4.19}$$

edp and the players current body direction are tranformed into vector and added to the players current acceleration vector $\vec{a}_n$ (usually, that should be 0 before, since a player cannot dash more than once a cycle and a player does not get accelerated by other means than dashing[1]).

At the transition from simulation step n to simulation step $n+1$, acceleration $\vec{a}_n$ is applied:

[1] is that so?

1. $\vec{a}_n$ is normalized to a maximum length of ***player_accel_max***.

2. $\vec{a}_n$ is added to current players speed $\vec{v}_n$. $\vec{v}_n$ will be normalized to a maximum length of ***player_speed_max***. For heterogeneous players, the maximum speed is a value between ***player_speed_max*** + ***player_speed_max_delta_min*** and ***player_speed_max*** + ***player_speed_max_delta_max*** in `player.conf`.

3. Noise $\vec{n}$ and wind $\vec{w}$ will be added to $\vec{v}_n$. Both noise and wind are configurable in `server.conf`. Parameters responsible for the wind are ***wind_force***, ***wind_dir*** and ***wind_rand***. With the current settings, there is no wind on the simulated soccer field. The responsible parameter for the noise is ***player_rand***. Both direction and length of the noise vector are within the interval $[-|\vec{v}_n| \cdot \text{player\_rand} \ldots |\vec{v}_n| \cdot \text{player\_rand}]$.

4. The new position of the player $\vec{p}_{n+1}$ is the old position $\vec{p}_n$ plus the velocity vector $\vec{v}_n$ (i.e. the maximum distance difference for the player between two simulation steps is ***player_speed_max***).

5. ***player_decay*** is applied for the velocity of the player: $\vec{v}_{n+1} = \vec{v}_n \cdot \text{player\_decay}$. Acceleration $\vec{a}_{n+1}$ is set to zero.

Stamina Model

For the stamina of a player, there are three important variables: the *stamina* value, *recovery* and *effort*. *stamina* is decreased when dashing and gets replenished slightly each cycle. *recovery* is responsible for how much the *stamina* recovers each cycle, and the *effort* says how effective dashing is (see section above). Important parameters for the stamina model are changeable in the files `server.conf` and `player.conf`, see also Tab. 4.5. Basically, the algorithm shown in Fig. 4.5 says that every simulation step the stamina is below some threshold, effort or recovery are reduced until a minimum is reached. Every step the stamina of the player is above some threshold, *effort* is increased up to a maximum. The *recovery* value is only reset to 1.0 each half, but it will not be increased during a game.

4.5.3. Kick Model

There are no principal changes to the kick model from soccer server version 6 to soccer server version 7, so your old implementation should still work. However, due to changes in the server parameter file, in some cases multiple kicks are not necessary anymore.

The **kick** command takes two parameters, the *kick power* the player client wants to use (between ***minpower*** and ***maxpower***) and the *angle* the player kicks the ball to. The angle is given in degrees and has to be between ***minmoment*** and ***maxmoment*** (see Tab. 4.6 for current parameter values).

Once the **kick** command arrived at the server, the kick will be executed if the ball is kick-able for the player and the player is not marked offside. The ball is kick-able

Basic Parameters `server.conf`		**Parameters for heterogeneous Players** `player.conf`		
Name	**Value**	**Name**	**Value**	**Range**
minpower	-100			
maxpower	100			
stamina_max	4000			
stamina_inc_max	45	***stamina_inc_max_delta_factor***	-100.0	25 — 45
		player_speed_max_delta_min	0.0	
		player_speed_max_delta_max	0.2	
extra_stamina[a]	0.0	***extra_stamina_delta_min***	0.0	0.0
		extra_stamina_delta_max	100.0	— 100.0
dash_power_rate	0.006	***dash_power_rate_delta_min***	0.0	0.006
		dash_power_rate_delta_max	0.002	— 0.008
effort_min	0.6	***effort_min_delta_factor***	-0.002	0.4 — 0.6
		extra_stamina_delta_min	0.0	
		extra_stamina_delta_max	100.0	
effort_max[a]	1.0	***effort_max_delta_factor***	-0.002	0.8 — 1.0
		extra_stamina_delta_min	0.0	
		extra_stamina_delta_max	100.0	
effort_dec_thr	0.3			
effort_dec	0.005			
effort_inc_thr	0.6			
effort_inc	0.01			
recover_dec_thr	0.3			
recover_dec	0.002			
recover_min	0.5			
player_accel_max	1.0			
player_speed_max	1.0	***player_speed_max_delta_min***	0.0	1.0
		player_speed_max_delta_max	0.2	— 1.2
player_rand	0.1			
wind_force	0.0			
wind_dir	0.0			
wind_rand	0.0			
player_decay	0.4	***player_decay_delta_min***	0.0	0.4
		player_decay_delta_max	0.2	— 0.6

[a]Not in `server.conf`, but compiled into the server

Table 4.5.: Dash and Stamina Model Parameters for Soccer Server 7

```
{if stamina is below recovery decrement threshold, recovery is reduced}
if stamina ≤ recover_dec_thr · stamina_max then
  if recovery > recover_min then
    recovery ← recovery − recover_dec
  end if
end if

{if stamina is below effort decrement threshold, effort is reduced}
if stamina ≤ effort_dec_thr · stamina_max then
  if effort > effort_min then
    effort ← effort − effort_dec
  end if
  effort ← max(effort, effort_min)
end if

{if stamina is above effort increment threshold, effort is increased}
if stamina ≥ effort_inc_thr · stamina_max then
  if effort < effort_max then
    effort ← effort + effort_inc
    effort ← min(effort, effort_max)
  end if
end if

{recover the stamina a bit}
stamina ← stamina + recovery · stamina_inc_max
stamina ← min(stamina, stamina_max)
```

Figure 4.5.: The stamina model algorithm

for the player, if the distance between the player and the ball is between 0 and ***kickable_margin***. Heterogeneous players can have different kickable margins. For the calculation of the distance during this section, it is important to know that if we talk of distance between player and ball, we talk about the minimal distance between the outer shape of both player and ball. So the distance in this section is the distance between the center of both objects **minus** the radius of the ball **minus** the radius of the player.

The first thing to be calculated for the kick is the effective kick power ep:

$$\text{ep} = \text{kick power} \cdot \text{kick\_power\_rate} \tag{4.20}$$

If the ball is not directly in front of the player, the effective kick power will be reduced by a certain amount dependent on the position of the ball with respect to the player. Both angle and distance are important.

If the relative angle of the ball is 0° wrt. the body direction of the player client — i.e. the ball is in front of the player — the effective power will stay as it is. The larger the angle gets, the more the effective power will be reduced. The worst case is if the ball is lying behind the player (angle 180°): the effective power is reduced by 25%.

The second important variable for the effective kick power is the distance from the ball to the player: it is quite obvious that – should the kick be executed – the distance between ball and player is between 0 and ***kickable_margin***. If the distance is 0, the effective kick power will not be reduced again. The further the ball is away from the player client, the more the effective kick power will be reduced. If the ball distance is ***kickable_margin***, the effective kick power will be reduced by 25% of the original kick power.

The overall worst case for kicking the ball is if a player kicks a distant ball behind itself: 50% of *kick power* will be used. For the effective kick power, we get the formula 4.21. (dir_diff means the absolute direction difference between ball and the player's body direction, dist_diff means the absolute distance between ball and player.)
$0 \leq \text{dir\_diff} \leq 180° \quad \wedge \quad 0 \leq \text{dist\_diff} \leq \text{kickable\_margin}$:

$$\text{ep} = \text{ep} \cdot \left(1 - 0.25 \cdot \frac{\text{dir\_diff}}{180°} - 0.25 \cdot \frac{\text{dist\_ball}}{\text{kickable\_margin}}\right) \tag{4.21}$$

The effective kick power is used to calculate $\vec{a}_{n_i}$, an acceleration vector that will be added to the global ball acceleration $\vec{a}_n$ during cycle n (remember that we have a multi agent system and *each* player close to the ball can kick it during the same cycle).

There is a server parameter, ***kick_rand***, that can be used to generate some noise to the ball acceleration. For the default players, ***kick_rand*** is 0 and no noise will be generated. For heterogeneous players, ***kick_rand*** depends on ***kick_rand_delta_factor*** in `player.conf` and on the actual kickable margin. In RoboCup 2000, ***kick_rand*** was used to generate some noise during evaluation round for the normal players.

During the transition from simulation step n to simulation step $n+1$ acceleration $\vec{a}_n$ is applied:

1. $\vec{a}_n$ is normalized to a maximum length of ***baccel_max***. Currently (Server 7), the maximum acceleration is equal to the maximum effective kick power.

2. $\vec{a}_n$ is added to the current ball speed $\vec{v}_n$. $\vec{v}_n$ will be normalized to a maximum length of ***ball_speed_max***.

3. Noise $\vec{n}$ and wind $\vec{w}$ will be added to $\vec{v}_n$. Both noise and wind are configurable in `server.conf`. Parameters responsible for the wind are ***wind_force***, ***wind_dir*** and ***wind_rand***. The responsible parameter for the noise is ***ball_rand***. Both direction and length of the noise vector are within the interval $[-|\vec{v}_n| \cdot \text{ball\_rand} \ldots |\vec{v}_n| \cdot \text{ball\_rand}]$.

4. The new position of the ball $\vec{p}_{n+1}$ is the old position $\vec{p}_n$ plus the velocity vector $\vec{v}_n$ (i.e. the maximum distance difference for the ball between two simulation steps is ***ball_speed_max***).

5. ***ball_decay*** is applied for the velocity of the ball: $\vec{v}_{n+1} = \vec{v}_n \cdot \text{ball\_decay}$. Acceleration $\vec{a}_{n+1}$ is set to zero.

With the current settings the ball covers a distance up to 45, assuming an optimal kick. 53 cycles after an optimal kick, the distance from the kick off position to the ball is about 43, the remaining velocity is smaller than 0.1. 15 cycles after an optimal kick, the ball covers a distance of 27 – 28 and the remaining veloctity is slightly larger than 1.

Implications from the kick model and the current parameter settings are that it still might be helpful to use several small kicks for a compound kick – for example stopping the ball, kick it to a more advantageous position within the kickable area and kick it to the desired direction. It would be another possibility to accelerate the ball to maximum speed without putting it to relative position (0,0°) using a compound kick.

4.5.4. Move Model

The **move** command can be used to place a player directly onto a desired position on the field. **move** exists to set up the team and does not work during normal play. It is available at the beginning of each half (play mode '`before_kick_off`') and after a goal has been scored (play modes '`goal_r_`*n*' or '`goal_l_`*n*'). In these situations, players can be placed on any position in their own half (i.e. $X < 0$) and can be moved any number of times, as long as the play mode does not change. Players moved to a position on the opponent half will be set to a random position on their own side by the server.

A second purpose of the **move** command is to move the goalie within the penalty area after the goalie caught the ball (see also Sec. 4.5.1). If the goalie caught the ball, it can move together with the ball within the penalty area. The goalie is allowed to move ***goalie_max_moves*** times before it kicks the ball. Additional **move** commands do not have any effect and the server will respond with `(error too_many_moves)`.

4.5.5. Say Model

Using the **say** command, players can broadcast messages to other players. Messages can be ***say_msg_size*** characters long, where valid characters for **say** messages are from

Basic Parameters `server.conf`		**Parameters for heterogeneous Players** `player.conf`		
Name	**Value**	**Name**	**Value**	**Range**
minpower	-100			
maxpower	100			
minmoment	-180			
maxmoment	180			
kickable_margin	0.7	***kickable_margin_delta_min*** ***kickable_margin_delta_max***	0.0 0.2	0.7 — 0.9
kick_power_rate	0.027			
kick_rand	0.0	***kick_rand_delta_factor*** ***kickable_margin_delta_min*** ***kickable_margin_delta_max***	0.5 0.0 0.2	0.0 — 0.1
ball_size	0.085			
ball_decay	0.94			
ball_rand	0.05			
ball_speed_max	2.7			
ball_accel_max	2.7			
wind_force	0.0			
wind_dir	0.0			
wind_rand	0.0			

Table 4.6.: Ball and Kick Model Parameters

Parameter in `server.conf`	**Value**
goalie_max_moves	2

Table 4.7.: Parameter for the move command

the set [-0-9a-zA-Z ().+*/?<>_] (without the square brackets). Messages players say can be heard within a distance of ***audio_cut_dist*** by members of both teams (see also Sec. 4.3.1). Say messages sent to the server will be sent back to players within that distance immediately. The use of the **say** command is only restricted by the limited capacity of the players of hearing messages.

Parameter in `server.conf`	Value
say_msg_size	512
audio_cut_dist	50
hear_max	2
hear_inc	1
hear_decay	2

Table 4.8.: Parameters for the say command

4.5.6. Turn Model

While **dash** is used to accelerate the player in direction of its body, the **turn** command is used to change the players body direction. The argument for the **turn** command is the moment; valid values for the moment are between ***minmoment*** and ***maxmoment***. If the player does not move, the moment is equal to the angle the player will turn. However, there is a concept of inertia that makes it more difficult to turn when you are moving. Specifically, the actual angle the player is turned is as follows:

$$\text{actual\_angle} = \text{moment}/(1.0 + \text{inertia\_moment} \cdot \text{player\_speed}) \tag{4.22}$$

inertia_moment is a `server.conf` parameter with default value 5.0. Therefore (with default values), when the player is at speed 1.0, the maximum effective turn he can do is ±30. However, notice that because you can not dash and turn during the same cycle, the fastest that a player can be going when executing a **turn** is ***player_speed_max*** · ***player_decay***, which means the effective turn for a default player (with default values) is ±60.

For heterogeneous players, the inertia moment is the default ***inertia_value*** plus a value between min. ***player_decay_delta_min*** · ***inertia_moment_delta_factor*** and max. ***player_decay_delta_max*** · ***inertia_moment_delta_factor***.

4.5.7. TurnNeck Model

With **turn_neck**, a player can turn its neck somewhat independently of its body. The angle of the head of the player is the viewing angle of the player. The **turn** command changes the angle of the body of the player while **turn_neck** changes the neck angle of the player relative to its body. The minimum and maximum relative angle for the player's neck are given by ***minmoment*** and ***maxmoment*** in `server.conf`. Remember

Basic Parameters `server.conf`		**Parameters for heterogeneous Players** `player.conf`		
Name	**Value**	**Name**	**Value**	**Range**
minmoment	-180			
maxmoment	180			
inertia_moment	5.0	***player_decay_delta_min***	0.0	5.0 10.0
		player_decay_delta_max	0.2	
		inertia_moment_delta_factor	25.0	

Table 4.9.: Turn Model Parameters

that the neck angle is relative to the body of the player so if the client issues a **turn** command, the viewing angle changes even if no **turn_neck** command was issued.

Also, **turn_neck** commands can be executed during the same cycle as **turn**, **dash**, and **kick** commands. **turn_neck** is not affected by momentum like **turn** is. The argument for a **turn_neck** command must be in the range between ***minneckmoment*** and ***maxneckmoment***.

Parameter in `server.conf`	**Value**
minneckang	-90
maxneckang	90
minneckmoment	-180
maxneckmoment	180

Table 4.10.: Parameter for the turn_neck command

4.6. Heterogeneous Players

With Soccer Server 7, heterogeneous players were introduced. For heterogeneous players, Soccer Server generates ***player_types*** random player types at startup. The player types have different abilities based on the tradeoffs defined in the `player.conf` file. Both teams of a match use the same player types. Type 0 is the default type and always the same.

When the players connect to the server, the receive information on the available player types (see Sec. 4.2.1). The online coach can change player types unlimited times in '`before_kick_off`' mode and change player types ***subs_max*** times during other non-'`play_on`' play modes using the **change_player_type ...** command (see Sec. 7.4).

Each time a player is substituted by some other player type, its stamina, recovery and effort is reset to the initial (maximum) value of the respective player type.

Parameter in `player.conf`	**Value**
player_types	7
subs_max	3

Table 4.11.: Parameter for substitutions and heterogeneous player types

4.7. Referee Model

The Automated Referee sends messages to the players, so that players know the actual play mode of the game. The rules and the behavior for the automated referee are described in Sec. 2.2.1. Players receive the referee messages as *hear* messages. A player can hear referee messages in every situation independent of the number of messages the player heard from other players.

4.7.1. Play Modes and referee messages

The change of the play mode is announced by the referee. Additionally, there are some referee messages announcing events like a goal or a foul. If you have a look into the server source code, you will notice some additional play modes that are currently not used. Both play modes and referee messages are announced using `(referee` ***`String`***`)`, where *String* is the respective play mode or message string. The play modes are listed in Tab. 4.12, for the messages see Tab. 4.13.

Play Mode	t_c	subsequent play mode	comment
'`before_kick_off`'	0	'`kick_off_`*`Side`*'	at the beginning of a half
'`play_on`'			during normal play
'`time_over`'			
'`kick_off_`*`Side`*'			announce start of play (after pressing the `Kick Off` button)
'`kick_in_`*`Side`*'			
'`free_kick_`*`Side`*'			
'`corner_kick_`*`Side`*'			
'`goal_kick_`*`Side`*'		'`play_on`'	play mode changes once the ball leaves the penalty area
'`goal_`*`Side`*'			currently unused (but see Tab. 4.13).
'`drop_ball`'	0	'`play_on`'	
'`offside_`*`Side`*'	30	'`free_kick_`*`Side`*'	for the opposite side

where *Side* is either the character '`l`' or '`r`', *OSide* means opponent's side.
t_c is the time (in number of cycles) until the subsequent play mode will be announced

Table 4.12.: Play Modes

Message	t_c	subsequent play mode	comment
goal_*Side*_*n*	50	'kick_off_*OSide*'	announce the nth goal for a team
foul_*Side*	0	'free_kick_*OSide*'	announce a foul
goalie_catch_ball_*Side*	0	'free_kick_*OSide*'	
time_up_without_a_team	0	'time_over'	sent if there was no opponent until the end of the second half
time_up	0	'time_over'	sent once the game is over (if the time is $\geq$ second half **and** the scores for each team are different)
half_time	0	'before_kick_off'	
time_extended	0	'before_kick_off'	

where *Side* is either the character 'l' or 'r', *OSide* means opponent's side.
t_c is the time (in number of cycles) until the subsequent play mode will be announced

Table 4.13.: Referee Messages

4.8. The Soccer Simulation

In Sec. 4.4, we gave a description on how the objects are moved with respect to their accelerations and velocities. In this section, we describe at what point in time acceleration and velocities are applied to the objects during the simulation.

4.8.1. Description of the simulation algorithm

In Soccer Server, time is updated in discrete steps. A simulation step is 100ms. During each simulation step, objects (i.e. players and the ball) stay on their positions. If players decide to act within a step, actions are applied to the players and the ball at the transition from one simulation cycle to the next. Depending on the play mode, not all actions are allowed for the players (for instance in 'before_kick_off' mode, players can **turn** and **move**, but they cannot **dash**), so only allowed actions will be applied and take effect.

If during a step, several players kick the ball, all the kicks are applied to the ball and a resulting acceleration is calculated. If the resulting acceleration is larger than the maximum acceleration for the ball, acceleration is normalized to its maximum value. After moving the objects, the server checks for collisions and updates velocities if a collision occurred (see also Sec. 4.4.2).

When applying accelerations and velocities to the objects, the order of application is randomized. After changing objects positions, and updating velocities and accelerations, the automated referee checks the situation and changes the play mode or the object positions, if necessary. Changes to the play mode are announced immediately. Finally, stamina for each player is updated.

4.9. Using Soccerserver

4.9.1. The Soccerserver Parameters

Table 4.14.: Parameters adjustable in `server.conf`

Name	Default Value	Current Value in `server.conf`	Description
goal_width	7.32	14.02	goal width
player_size		0.3	player size
player_decay		0.4	player decay
player_rand		0.1	
player_weight		60.0	player weight
player_speed_max		1.0	max. player velocity
player_accel_max		1.0	max. player acceleration
stamina_max		4000.0	max. player stamina
stamina_inc_max		45.0	max. player stamina increment
recover_dec_thr		0.3	player recovery decrement threshold
recover_min		0.5	min. player recovery
recover_dec		0.002	player recovery decrement
effort_dec_thr		0.3	player dash effort decrement threshold
effort_min		0.6	min. player dash effort
effort_dec		0.005	dash effort decrement
effort_inc_thr		0.6	dash effort increment treshold
effort_inc		0.01	dash effort increment
kick_rand		0.0	noise added directly to kicks
team_actuator_noise			flag whether to use team specific actuator noise
prand_factor_l			factor to multiply prand for left team
prand_factor_r			factor to multiply prand for right team
kick_rand_factor_l			factor to multiply kick_rand for left team
kick_rand_factor_r			factor to multiply kick_rand for right team
ball_size		0.085	ball size
ball_decay		0.94	ball decay
ball_rand		0.05	
ball_weight		0.2	weight of the ball
ball_speed_max		2.7	max. ball velocity
ball_accel_max		2.7	max. ball acceleration
dash_power_rate		0.006	dash power rate
kick_power_rate		0.027	kick power rate
kickable_margin		0.7	kickable margin
control_radius			control radius
catch_probability		1.0	goalie catch probability
catchable_area_l		2.0	goalie catchable area length
catchable_area_w		1.0	goalie catchable area width
goalie_max_moves		2	goalie max. moves after a catch
maxpower		100	max power

Table 4.14.: (continued)

Name	Default Value	Current Value in `server.conf`	Description
minpower		-100	min power
maxmoment		180	max. moment
minmoment		-180	min. moment
maxneckmoment		180	max. neck moment
minneckmoment		-180	min. neck moment
maxneckang		90	max. neck angle
minneckang		-90	min. neck angle
visible_angle		90.0	visible angle
visible_distance			visible distance
audio_cut_dist		50.0	audio cut off distance
quantize_step		0.1	quantize step of distance for movable objects
quantize_step_l		0.01	quantize step of distance for landmarks
quantize_step_dir			
quantize_step_dist_team_l			
quantize_step_dist_team_r			
quantize_step_dist_l_team_l			
quantize_step_dist_l_team_r			
quantize_step_dir_team_l			
quantize_step_dir_team_r			
ckick_margin		1.0	corner kick margin
wind_dir	0.0	0.0	wind direction
wind_force	10.0	0.0	
wind_rand	0.3	0.0	
wind_none			wind factor is none
wind_random	false		wind factor is random
inertia_moment		5.0	intertia moment for turn
half_time		300	length of a half time in seconds
drop_ball_time		200	number of cycles to wait until dropping the ball automatically
port		6000	player port number
coach_port		6001	(offline) coach port
olcoach_port			online coach port
say_coach_cnt_max		128	upper limit of the number of online coach's message
say_coach_msg_size		128	upper limit of length of online coach's message
simulator_step		100	time step of simulation [unit:msec]
send_step		150	time step of visual information [unit:msec]
recv_step		10	time step of acception of commands [unit: msec]
sense_body_step		100	
say_msg_size		512	string size of say message [unit:byte]
clang_win_size		300	time window which controls how many messages can be sent (coach language)
clang_define_win		1	number of messages per window
clang_meta_win		1	

Table 4.14.: (continued)

Name	Default Value	Current Value in `server.conf`	Description
clang_advice_win		1	
clang_info_win		1	
clang_mess_delay		50	delay between receipt of message and send to players
clang_mess_per_cycle		1	maximum number of coach messages sent per cycle
hear_max		2	
hear_inc		1	
hear_decay		2	
catch_ban_cycle		5	
coach			
coach_w_referee			
old_coach_hear			
send_vi_step		100	interval of online coach's look
use_offside		on	flag for using off side rule
offside_active_area_size		5	offside active area size
forbid_kick_off_offside		on	forbid kick off offside
log_file			
record			
record_version		3	flag for record log
record_log		on	flag for record client command log
record_messages			
send_log		on	flag for send client command log
log_times		off	flag for writing cycle lenth to log file
verbose		off	flag for verbose mode
replay			
offside_kick_margin		9.15	offside kick margin
slow_down_factor			
start_goal_l			
start_goal_r			
fullstate_l			
fullstate_r			

5. The Soccer Monitor

5.1. Introduction

Soccermonitor provides a visual interface. Using the monitor we can watch a game vividly and control the proceeding of the game. By cooperating with logplayer, soccermonitor can replay games, so that it becomes very convenient to analyze and debug clients.

5.2. Getting started

To connect the soccermonitor with the soccerserver, you can use the command following:

```
-> cd monitor
-> soccermonitor -f ConfFileName [-ParameterName Value]*
```

By specifying the arguments, you can modify the parameters of soccermonitor (See 5.5 Settings and Parameters) instead of modifying monitor configuration file.

If you use script "sserver" to start soccerserver, a monitor will be automatically started and connected with the server:

```
-> sserver
```

5.3. Soccermonitor Communication

Soccermonitor and soccerserver are connected via UDP/IP on port 6000 (default).

5.3.1. Information From Server to Monitor

When the server is connected with the monitor , it will send information to the monitor every cycle. Soccerserver 7.xx provides two different formats information (version 1 and version 2).The server will decide which format be used according to the initial command sent by the monitor. (See 5.3.2 Commands From Monitor to Server)

Version 1

Soccerserver and logplayer send dispinfo_t structs to the soccermonitor. Dispinfo_t contains a union with three different types of information:

- showinfo_t: information needed to draw the scene
- msginfo_t : contains the messages from the players and the referee shown in the bottom windows
- drawinfo_t: information for monitor to draw circles, lines or points (not used by the server)

The size of dispinfo_t is determined by its largest subpart (msg) and is 2052 bytes (the union causes some extra network load and may be changed in future versions). In order to keep compatibility between different platforms, values in dispinfo_t are represented by network byte order.

Following is a description of these structs and the ones contained:

```
showinfo_t:
typedef struct {
   short enable ;
   short side ;
   short unum ;
   short angle ;
   short x ;
   short y ;
} pos_t ;
```

Values of the elements can be

- enable:

  state of the object. Players not on the field (and the ball) have state DISABLE. The other bits of enable allow monitors to draw the state and action of a player more detailed (server/types.h).

  ```
  DISABLE     (0x00)
  STAND       (0x01)
  KICK        (0x02)
  KICK_FAULT  (0x04)
  GOALIE      (0x08)
  CATCH       (0x10)
  CATCH_FAULT (0x20)
  ```

- side:

  side the player is playing on. LEFT means from left to right, NEUTRAL is the ball (server/types.h).

  ```
  LEFT     1
  NEUTRAL  0
  RIGHT   -1
  ```

- unum:

  uniform number of a player ranging from 1 to 11

- angle:

  angle the agent is facing ranging from -180 to 180 degrees, where -180 is view to the left side of the screen, -90 to the top, 0 to the right and 90 to the bottom.

- x, y:

  position of the player on the screen. (0, 0) is the midpoint of the field, x increases to the right, y to the bottom of the screen. Values are multiplied by SHOWINFO_SCALE (16) to reduce aliasing, so field size is PITCH_LENGTH * SHOWINFO_SCALE in x direction and PITCH_WIDTH * SHOWINFO_SCALE in y direction.

```
typedef struct {
   char  name[16];  /* name of the team */
   short score;     /* current score of the team */
} team_t ;

typedef struct {
   char   pmode ;
   team_t team[2] ;
   pos_t  pos[MAX_PLAYER * 2 + 1] ;
   short  time ;
} showinfo_t ;
```

A showinfo_t struct is passed every cycle (100 ms) to the monitor and contains the state and positions of players and the ball.

values of the elements can be

- pmode:

  currently active playmode of the game (server/types.h)

```
PM_Null,
PM_BeforeKickOff,
PM_TimeOver,
PM_PlayOn,
PM_KickOff_Left,
PM_KickOff_Right,
PM_KickIn_Left,
PM_KickIn_Right,
PM_FreeKick_Left,
PM_FreeKick_Right,
PM_CornerKick_Left,
```

```
        PM_CornerKick_Right,
        PM_GoalKick_Left,
        PM_GoalKick_Right,
        PM_AfterGoal_Left,
        PM_AfterGoal_Right,
        PM_Drop_Ball,
        PM_OffSide_Left,
        PM_OffSide_Right,
        PM_MAX
```

- team:

  structs containing the teams (see above). Index 0 is for team playing from left to right.

- pos:

  position information of player (see above). Index 0 is the ball, indices 1 to 11 is for team[0] (left to right) and 12 to 22 for team[1].

- time:

  current time ranging from 1 to 12000 (in extra time)

- msginfo_t:

```
typedef struct {
   short board ;
   char  message[2048] ;
} msginfo_t ;
```

- board:

  indicates the type of message. A message with type MSG_BOARD is a message of the referee for the left text window, LOG_BOARD are messages from and to the players. (server/param.h)

```
        MSG_BOARD 1
        LOG_BOARD 2
```

- message:

  zero terminated string containing the message.

- drawinfo_t:

  allows the server to tell the monitor to draw simple graphics elements.

```
typedef struct {
   short x ;
   short y ;
   char  color[COLOR_NAME_MAX] ;
} pointinfo_t ;

typedef struct {
   short x ;
   short y ;
   short r ;
   char  color[COLOR_NAME_MAX] ;
} circleinfo_t ;

typedef struct {
   short x1 ;
   short y1 ;
   short x2 ;
   short y2 ;
   char  color[COLOR_NAME_MAX] ;
} lineinfo_t ;

typedef struct {
   short mode ;
   union {
      pointinfo_t  pinfo ;
      circleinfo_t cinfo ;
      lineinfo_t   linfo ;
   } object ;
} drawinfo_t ;
```

- mode:

  determines the kind of message the union object contains (server/param.h)

  ```
          DrawClear  0
          DrawPoint  1
          DrawCircle 2
          DrawLine   3
  ```

- dispinfo_t:

  container for the different messages from server to monitor.

  ```
  typedef struct {
     short   mode ;
  ```

```
    union {
       showinfo_t show ;
       msginfo_t  msg ;
       drawinfo_t draw ;
    } body ;
 } dispinfo_t ;
```

- mode:

  determines the kind of message the union body contains. NO_INFO indicates no valid info contained (never sent by the server), BLANK_MODE tells the monitor to show a blank screen (used by logplayer) (server/param.h).

  ```
  NO_INFO    0
  SHOW_MODE  1
  MSG_MODE   2
  DRAW_MODE  3
  BLANK_MODE 4
  ```

Version 2

Soccerserver and logplayer send dispinfo_t2 structs to the soccermonitor instead of dispinfo_t structs which is used in version 1. Dispinfo_t2 contains a union with five different types of information:

- showinfo_t2: information needed to draw the scene
- msginfo_t : contains the messages from the players and the referee shown in the bottom windows
- player_type_t: information describes different player's ability
- server_params_t: parameters and configurations of soccerserver
- player_params_t: parameters of player

Following is a description of these structs and the ones contained:
showinfo_t2:

```
typedef struct {
  short mode;
  short type;
  long  x;
  long  y;
  long  deltax;
  long  deltay;
  long  body_angle;
```

```
  long  head_angle;
  long  view_width;
  short view_quality;
  long  stamina;
  long  effort;
  long  recovery;
  short kick_count;
  short dash_count;
  short turn_count;
  short say_count;
  short tneck_count;
  short catch_count;
  short move_count;
  short chg_view_count;
} player_t;

values of the elements can be
/*****************************************/
/********** NEEDS TO BE EXPANDED *********/
/*****************************************/

typedef struct {
  long x;
  long y;
  long deltax;
  long deltay;
} ball_t;

values of the elements can be
/*****************************************/
/********** NEEDS TO BE EXPANDED ********/
/*****************************************/

typedef struct {
   char  name[16];  /* name of the team */
   short score;     /* current score of the team */
} team_t ;

typedef struct {
  char     pmode ;
  team_t   team[2] ;
  ball_t   ball;
  player_t pos[MAX_PLAYER * 2] ;
  short    time ;
```

```
} showinfo_t2 ;
```

A showinfo_t2 struct is passed every cycle (100 ms) to the monitor and contains the state and positions of players and the ball.

values of the elements can be

- pmode:

  several new playmodes are added based on version 1 (server/types.h)

```
PM_Null,
PM_BeforeKickOff,
PM_TimeOver,
PM_PlayOn,
PM_KickOff_Left,
PM_KickOff_Right,
PM_KickIn_Left,
PM_KickIn_Right,
PM_FreeKick_Left,
PM_FreeKick_Right,
PM_CornerKick_Left,
PM_CornerKick_Right,
PM_GoalKick_Left,
PM_GoalKick_Right,
PM_AfterGoal_Left,
PM_AfterGoal_Right,
PM_Drop_Ball,
PM_OffSide_Left,
PM_OffSide_Right,

// added for 3D viewer/commentator/small league
PM_PK_Left,
PM_PK_Right,
PM_FirstHalfOver,
PM_Pause,
PM_Human,
PM_Foul_Charge_Left,
PM_Foul_Charge_Right,
PM_Foul_Push_Left,
PM_Foul_Push_Right,
PM_Foul_MultipleAttacker_Left,
PM_Foul_MultipleAttacker_Right,
PM_Foul_BallOut_Left,
PM_Foul_BallOut_Right,
PM_MAX
```

- team:

  structs containing the teams (see above). Index 0 is for team playing from left to right.

- ball:

  position information of ball (see above).

- pos:

  position information of player (see above). Indices 0 to 10 is for team[0] (left to right) and 11 to 21 for team[1].

- time:

  current time ranging from 1 to 12000 (in extra time)

- msginfo_t:

```
typedef struct {
   short board ;
   char  message[2048] ; /* max_message_length_for_display */
} msginfo_t ;
```

- board:

  indicates the type of message. A message with type MSG_BOARD is a message of the referee for the left text window, LOG_BOARD are messages from and to the players. (server/param.h)

```
        MSG_BOARD 1
        LOG_BOARD 2
```

- message:

  zero terminated string containing the message.

```
typedef struct {
    short id;
    long  player_speed_max;
    long  stamina_inc_max;
    long  player_decay;
    long  inertia_moment;
    long  dash_power_rate;
    long  player_size;
    long  kickable_margin;
    long  kick_rand;
    long  extra_stamina;
    long  effort_max;
```

```
    long  effort_min;

  // spare variables which are to be used for paramenter added in the future
    long sparelong1;
    long sparelong2;
    long sparelong3;
    long sparelong4;
    long sparelong5;
    long sparelong6;
    long sparelong7;
    long sparelong8;
    long sparelong9;
    long sparelong10;
} player_type_t;

typedef struct
{
    long  gwidth ;            /* goal width */
    long  inertia_moment ;    /* intertia moment for turn */
    long  psize ;             /* player size */
    long  pdecay ;            /* player decay */
    long  prand ;             /* player rand */
    long  pweight ;           /* player weight */
    long  pspeed_max ;        /* player speed max */
    long  paccel_max ;        /* player acceleration max */
    long  stamina_max ;       /* player stamina max */
    long  stamina_inc ;       /* player stamina inc */
    long  recover_init ;      /* player recovery init */
    long  recover_dthr ;      /* player recovery decriment threshold */
    long  recover_min ;       /* player recovery min */
    long  recover_dec ;       /* player recovery decriment */
    long  effort_init ;       /* player dash effort init */
    long  effort_dthr ;       /* player dash effort decriment threshold */
    long  effort_min ;        /* player dash effrot min */
    long  effort_dec ;        /* player dash effort decriment */
    long  effort_ithr ;       /* player dash effort incriment threshold */
    long  effort_inc ;        /* player dash effort incriment */
    long  kick_rand;          /* noise added directly to kicks */
    short team_actuator_noise; /* flag whether to use team specific actuator noise */
    long  prand_factor_l;     /* factor to multiple prand for left team */
    long  prand_factor_r;     /* factor to multiple prand for right team */
    long  kick_rand_factor_l; /* factor to multiple kick_rand for left team */
    long  kick_rand_factor_r; /* factor to multiple kick_rand for right team */
    long  bsize ;             /* ball size */
```

```
long  bdecay ;            /* ball decay */
long  brand ;             /* ball rand */
long  bweight ;           /* ball weight */
long  bspeed_max ;        /* ball speed max */
long  baccel_max;         /* ball acceleration max */
long  dprate ;            /* dash power rate */
long  kprate ;            /* kick power rate */
long  kmargin ;           /* kickable margin */
long  ctlradius ;         /* control radius */
long  ctlradius_width ;   /* (control radius) - (plyaer size) */
long  maxp ;              /* max power */
long  minp ;              /* min power */
long  maxm ;              /* max moment */
long  minm ;              /* min moment */
long  maxnm ;             /* max neck moment */
long  minnm ;             /* min neck moment */
long  maxn ;              /* max neck angle */
long  minn ;              /* min neck angle */
long  visangle ;          /* visible angle */
long  visdist ;           /* visible distance */
long  windir ;            /* wind direction */
long  winforce ;          /* wind force */
long  winang ;            /* wind angle for rand */
long  winrand ;           /* wind force for force */
long  kickable_area ;     /* kickable_area */
long  catch_area_l ;      /* goalie catchable area length */
long  catch_area_w ;      /* goalie catchable area width */
long  catch_prob ;        /* goalie catchable possibility */
short goalie_max_moves;   /* goalie max moves after a catch */
long  ckmargin ;          /* corner kick margin */
long  offside_area ;      /* offside active area size */
short win_no ;            /* wind factor is none */
short win_random ;        /* wind factor is random */
short say_cnt_max ;       /* max count of coach SAY */
short SayCoachMsgSize ;   /* max length of coach SAY */
short clang_win_size;
short clang_define_win;
short clang_meta_win;
short clang_advice_win;
short clang_info_win;
short clang_mess_delay;
short clang_mess_per_cycle;
short half_time ;         /* half time */
short sim_st ;            /* simulator step interval msec */
```

```
short send_st ;           /* udp send step interval msec */
short recv_st ;           /* udp recv step interval msec */
short sb_step ;           /* sense_body interval step msec */
short lcm_st ;            /* lcm of all the above steps msec */
short SayMsgSize ;        /* string size of say message */
short hear_max ;          /* player hear_capacity_max */
short hear_inc ;          /* player hear_capacity_inc */
short hear_decay ;        /* player hear_capacity_decay */
short cban_cycle ;        /* goalie catch ban cycle */
short slow_down_factor ;  /* factor to slow down simulator and send intervals */
short useoffside ;        /* flag for using off side rule */
short kickoffoffside ;    /* flag for permit kick off offside */
long  offside_kick_margin ;/* offside kick margin */
long  audio_dist ;        /* audio cut off distance */
long  dist_qstep ;        /* quantize step of distance */
long  land_qstep ;        /* quantize step of distance for landmark */
long  dir_qstep ;         /* quantize step of direction */
long  dist_qstep_l ;      /* team right quantize step of distance */
long  dist_qstep_r ;      /* team left quantize step of distance */
long  land_qstep_l ;      /* team right quantize step of distance for landmark */
long  land_qstep_r ;      /* team left quantize step of distance for landmark */
long  dir_qstep_l ;       /* team left quantize step of direction */
long  dir_qstep_r ;       /* team right quantize step of direction */
short CoachMode ;         /* coach mode */
short CwRMode ;           /* coach with referee mode */
short old_hear ;          /* old format for hear command (coach) */
short sv_st ;             /* online coach's look interval step */

// spare variables which are to be used for paramenter added in the future
long  sparelong1;
long  sparelong2;
long  sparelong3;
long  sparelong4;
long  sparelong5;
long  sparelong6;
long  sparelong7;
long  sparelong8;
long  sparelong9;
long  sparelong10;

short start_goal_l;
short start_goal_r;
short fullstate_l;
short fullstate_r;
```

```
    short drop_time;
    short spareshort6;
    short spareshort7;
    short spareshort8;
    short spareshort9;
    short spareshort10;
} server_params_t;

typedef struct {
    short player_types;
    short subs_max;
    short pt_max;

    long  player_speed_max_delta_min;
    long  player_speed_max_delta_max;
    long  stamina_inc_max_delta_factor;

    long  player_decay_delta_min;
    long  player_decay_delta_max;
    long  inertia_moment_delta_factor;

    long  dash_power_rate_delta_min;
    long  dash_power_rate_delta_max;
    long  player_size_delta_factor;

    long  kickable_margin_delta_min;
    long  kickable_margin_delta_max;
    long  kick_rand_delta_factor;

    long  extra_stamina_delta_min;
    long  extra_stamina_delta_max;
    long  effort_max_delta_factor;
    long  effort_min_delta_factor;

    long  sparelong1;
    long  sparelong2;
    long  sparelong3;
    long  sparelong4;
    long  sparelong5;
    long  sparelong6;
    long  sparelong7;
    long  sparelong8;
    long  sparelong9;
    long  sparelong10;
```

```
    short spareshort1;
    short spareshort2;
    short spareshort3;
    short spareshort4;
    short spareshort5;
    short spareshort6;
    short spareshort7;
    short spareshort8;
    short spareshort9;
    short spareshort10;

} player_params_t;
```

See Server Parameters, Player Parameters and Player Types

- dispinfo_t2:

  container for the different messages from server to monitor.

```
typedef struct {
  short mode;
  union {
      showinfo_t2     show;
      msginfo_t       msg;
      player_type_t   ptinfo;
      server_params_t sparams;
      player_params_t pparams;
  } body;
} dispinfo_t2 ;
```

- mode:

  determines the kind of message the union body contains. NO_INFO indicates no valid info contained (never sent by the server). BLANK_MODE tells the monitor to show a blank screen (used by logplayer) (server/param.h).

```
NO_INFO     0
SHOW_MODE   1
MSG_MODE    2
BLANK_MODE  4
PT_MODE     7
PARAM_MODE  8
PPARAM_MODE 9
```

5.3.2. Commands From Monitor to Server

The monitor can send to the server the following commands:

`(dispinit) | (dispinit version 2)`

sent to the server as first message to register as monitor (opposed to a player, that connects on port 6000 as well) . "(dispinit)" is for information version 1, while "(dispinit version 2)" is for version 2. You can change the version by setting the according monitor parameter. (See 5.5 Parameters and Configurations)

`(dispstart)`

sent to start (kick off) a game, start the second half or extended time. Ignored, when the game is already running.

`(dispfoul x y side)`

sent to indicate a foul situation. x and y are the coordinates of the foul, side is LEFT (1) for a free kick for the left team, NEUTRAL (0) for a drop ball and RIGHT (-1) for a free kick for the right team.

`(dispdiscard side unum)`

sent to show a player the red card (kick him out). side can be LEFT or RIGHT, unum is the number of the player (1 - 11).

`(dispplayer side unum posx posy ang)`

sent to place player at certain position with certain body angle, side can be LEFT (1) or RIGHT (-1), unum is the number of the player(1 - 11). Posx and posy indicate the new position of the player, which will be divided by SHOWINFO_SCALE. And ang indicate the new angle of a player in degrees. This command is added in soccerserver 7.02.

5.4. How to record and playback a game

To record games, you can call server with the argument:

`-record LOGFILE`

(LOGFILE is the logfile name) or set the parameter in server.conf file:

`record.log : on.`

To specify the logfile version, you can call server with the argument:

`-record_version [1/2/3]`

or set the parameter in server.conf file:

`record_version : 2`

The logplayer allows you to replay recorded games. Logfiles can be read in by the logplayer and sent to the connected soccermonitors. To replay logfiles just call logplayer with the logfile name as argument, start a soccermonitor and then use the buttons on the logplayer window to start, stop, play backward, play stepwise.

5.4.1. Version 1 Protocol

Logfiles of version 1 (server versions up to 4.16) are a stream of consecutive dispinfo_t chunks. Due to the structure of dispinfo_t as a union, a lot of bytes have been wasted leading to impractical logfile sizes. This lead to the introduction of a new logfile format 2.

5.4.2. Version 2 Protocol

Version 2 logfile protocol tries to avoid redundant or unused data for the price of not having uniform data structs. The format is as follows:

- head of the file:
  the head of the file is used to autodetect the version of the logfile. If there is no head, Unix-version 1 is assumed. 3 chars 'ULG' : indicating that this is a Unix logfile (to distinguish from Windows format)

- char version :
  version of the logfile format

- body:
  the rest of the file contains the data in chunks of the following format:

- short mode:
  this is the mode part of the dispinfo_t struct (see 5.4.1 Version 1) SHOW_MODE for showinfo_t information MSG_MODE for msginfo_t information
  - If mode is SHOW_MODE, a showinfo_t struct is following.
  - If mode is MSG_MODE, next bytes are
    - ▷ short board: containing the board info
    - ▷ short length: containing the length of the message (including zero terminator)
    - ▷ string msg: length chars containing the message

Other info such as DRAW_MODE and BLANK_MODE are not saved to logfiles. There is still room for optimization of space. The team names could be part of the head of the file and only stored once. The unum part of a player could be implicitly taken from array indices.

Be aware of, that information chunks in version 2 do not have the same size, so you can not just seek SIZE bytes back in the stream when playing logfiles backward. You have to read in the whole file at once or (as is done) have at least to save stream positions of the showinfo_t chunks to be able to play logfiles backward.

In order to keep compatibility between different platforms, values are represented by network byte order.

5.4.3. Version 3 Protocol

The version 3 protocol contains player parameter information for heterogenous players and optimizes space. The format is as follows:

- head of the file:
  Just like version 2, the file starts with the magig characters 'ULG'.
- char version :
  version of the logfile format, i.e. 3
- body:
  The rest of the file contains shorts that specify which data structures will follow.
  - If the short is PM_MODE,
    - ▷ a char specifying the play mode follows.

    This is only written when the playmode changes.
  - If the short is TEAM_MODE,
    - ▷ a team_t struct for the left side and
    - ▷ a team_t struct for the right side follow.

    Team data is only written if a new team connects or the score changes.
  - If the short is SHOW_MODE,
    - ▷ a short_showinfo_t2 struct specifying ball and player positions and states follows.
  - If the short is MSG_MODE,
    - ▷ a short specifying the message board,
    - ▷ a short specifying the length of the message,
    - ▷ a string containing the message will follow.
  - If the short is PARAM_MODE,
    - ▷ a server_params_t struct specifying the current server parameters follows.

    This is only written once at the beginning of the logfile.
  - If the short is PPARAM_MODE,
    - ▷ a player_params_t struct specifying the current hetro player parameters.

    This is only written once at the beginning of the logfile.
  - If the short is PT_MODE,
    - ▷ a player_type_t struct specifying the parameters of a specific player type follows.

    This is only written once for each player type at the beginning of the logfile.

Data Conversion:

- Values such as x, y positions are meters multiplied by SHOWINFO_SCALE2.
- Values such as deltax, deltay are meters/cycle multiplied by SHOWINFO_SCALE2.
- Values such as body_angle, head_angle and view_width are in radians multiplied by SHOWINFO_SCALE2.
- Other values such as stamina, effort and recovery have also been multiplied by SHOWINFO_SCALE2.

5.5. Settings and Parameters

Soccermonitor has the following modifiable parameters:

"Used Value" is the current value of the parameter which is encoded in the monitor.conf file. "Default Value" is the value encoded in the source files and will be used if the user doesn't give one.

You can specify parameters described in the table above in command line as following: You can also modify the parameters by specifying them in configuration file monitor.conf. In the configuration file, each line consists a pair of name and value of a parameter as follows: ParameterName : Value Lines that start with '#' are comment lines.

5.6. What's New

5.6.1. [7.07]

- The logplayer did not send server param, player param, and player type messages. This has been fixed.
- The monitor would crash on some logfiles because stamina_max seemed to be set to 0. The monitor will no longer crash this way.

5.6.2. [7.05]

- For quite some time, the logplayer has occasionally "skipped" so that certain cycles were never displayed by the logplayer. This seems to be caused by the logplayer sending too many UDP packets for the monitor to receive. Therefore, a new parameter has been added to the logplayer 'message_delay_interval'. After sending that many messages, the logplayer sleeps for 1 microsecond, giving the monitor a chance to catch up. This is not a guaranteed to work, but it seems to help significantly. If you still have a problem with the logplayer/monitor "skipping", try reducing message_delay_interval from it's default value of 10. Setting message_delay_interval to a negative number causes there to be no delay.
- The server used to truncate messages received from the players and coach to 128 characters before recording them in the logfile. This has been fixed.

Parameter Name	*Used Value*	*Default Value*	*Explanation*
host	localhost	Localhost	hostname of soccerserver
port	6000	6000	port number of soccerserver
version	2	1	monitor protocol version
length_magnify	6.0	6.0	magnification of size of field
goal_width	14.02	7.32	goal width
print_log	off	On	flag for display log of communication [on/off]
Log_line	6	6	size of log window
Print_mark	on	On	flag for display mark on field [on/off]
mark_file_name	mark.RoboCup.grey.xbm	Mark.xbm	mark on field use file name
ball_file_name	ball-s.xbm	Ball.xbm	ball use file name
player_widget_size	9.0	1.0	size of player widget
player_widget_font	5x8	Fixed	font(uniform number) of player widget
Uniform_num_pos_x	2	2	position (X) of player uniform number
Uniform_num_pos_y	8	8	position (Y) of player uniform number
Team_l_color	Gold	Gold	Team_L color
team_r_color	Red	Red	Team_R color
goalie_l_color	Green	Green	Team_L Goalie color
goalie_r_color	Purple	Purple	Team_R Goalie color
neck_l_color	Black	Black	Team_L Neck color
neck_r_color	Black	Black	Team_R Neck color
Goalie_neck_l_color	Black	Black	Team_L Goalie Neck color
Goalie_neck_r_color	Black	Black	Team_R Goalie Neck color
status_font	7x14bold	Fixed	status line font [team name and score,time,play_mode]
popup_msg	off	Off	flag for pop up and down "GOAL!!" and "Offside!" [on/off]
Goal_label_width	120	120	pop up and down "GOAL!!" label width
Goal_label_font	-adobe-times-bold-r-*-*-34-*-*-*-*-*-*-*	Fixed	pop up and down "GOAL!!" label font
Goal_score_width	40	40	pop up and down "GOAL!!" score width
Goal_score_font	-adobe-times-bold-r-*-*-25-*-*-*-*-*-*-*	Fixed	pop up and down "GOAL!!" score font
Offside_label_width	120	120	pop up and down "Offside!" label width
Offside_label_font	-adobe-times-bold-r-*-*-34-*-*-*-*-*-*-*	Fixed	pop up and down "Offside!" label font
eval	off	Off	flag for evaluation mode
redraw_player	on	Off	always redraw player (needed for RH 5.2)

5.6.3. [7.04]

- If a client connects with version > 7.0, all angles sent out by the server are rounded instead of truncated (as they were previously) This makes the error from quantization of angles (i.e. conversion of floats to ints) both uniform throughout the domain and two sided. This change was also made to all values put into the dispinfo_t structure for the monitors and logfiles.

5.6.4. [7.02]

- A new command has been added to the monitor protocol:
  (dispplayer side unum posx posy ang)
  (contributed by Artur Merke)
  See 5.3.2 Commands From Monitor to Server

5.6.5. [7.00]

- Included the head angle into the display of the soccermonitor. (source contributed by Ken Nguyen)
- Included visualization effect when the player collided with the ball or the player collided with another player. The monitor displays both cases with a black circle around the player.
- Introduced new monitor protocol version 2. (See 5.4.2 Version 2 and 5.3.2 Commands From Monitor to Server)
- Introduced new logging protocol version 3. (See 5.4.3 Version 3 Protocol)
- Fixed logging so that the last cycle of a game is logged.

6. Soccer Client

6.1. Protocols

This section provides a brief overview of the protocol between the Soccer Client and the Soccer Server. More details on these protocols can be found in the Soccer Server section.

Note that the init and reconnect commands should be send to the player's UDP port (default: 6000) of the Soccer Server machine, and after the response they sould be sent to the port assigned to your player by the server, in a valid format. The server sends the init response from this port (refer to section 1.2.1) . All the commands sent to or received from the server are strings of common character and are in a pair of parenthesis.

6.1.1. Initialization and Reconnection

Every player wanting to connect to the server should introduce himself. This is like a handshake and is done only at the beginning and optionally in the half time when you want to reconnect.

Initialization

Your client should send an init command to the server in the following format :

(init *TeamName* [(version *VerNum*)] [(goalie)])

The goalie should include the "(goalie)" in the init command to be allowed by the server to catch the ball or do another special goalie action. Note there can only be one or no goalie in each team. (You are not obliged to use a goalie)

The Server welcomes you with a response to your init message in the following format:

(init *Side UniformNumber PlayMode*)

Or by an error message (if there is an error, i.e. you have initiated more than two team, more than 11 players in a team or more than one goalie in a team):

(error no_more_team_or_player_or_goalie)

Side is your team's side of play, a character, **l**(left) or **r**(right). *UniformNumber* is the player's uniform number (the players of each team are known by their uniform number). *PlayMode* is a string representing one of the valid play modes.

If you connect to server with versions 7.00 or higher you will receive additional server parameters, player parameters and player types information (the last two are related to the hetero players feature). For the exact format refer to the appendix.

(server_param *Parameters . . .*)

(player_param *Parameters . . .*)

(player_type *id Parameters . . .*)

Here the hand shaking is finished and your client is known as a valid player.

Reconnection

Reconnection is useful for changing the client program of a player without restarting the game. It can only be done in a non-PlayOn playing mode (e.g. in the half time).

For reconnection you should send a reconnect command in the following format:

(reconnect *TeamName UniformNumber*)

And you will receive a response in the following format:

(reconnect *Side PlayMode*)

Or one of the following errors:

(can't reconnect)

if the game is in the PlayOn mode.

(error reconnect)

when no client reconnected due to an error. You may also receive the following error if the team name is invalid **(error no_more_team_or_player_or_goalie)**

Here again if you are connecting to the server with version 7.00 or higher you will receive additional server parameters, player parameters and player types information.

Disconnection

Before you disconnect, you can send a bye command to the server. This command will remove the player from the field.

(bye)

There will be no answers from the server.

Version Control

Due to the progressive development of the Soccer Server, new features have been added every year and this resulted in changes and improvements in the protocols to support these features. In order to keep compatibility with the older clients and making it easier to work with (specially for researchers), a system has been implemented for the Protocols Version Control. Every client should tell the server the version of its communication protocol in the **init** command so that the server would be able to send the messages in the proper format.

But note that although the communication protocol remains unchanged, the judgment and the simulation rules may change and this will affect the whole game.

6.1.2. Control Commands

During the game each player can send action commands. The server executes the commands at the end of the cycle and simulates the next cycle regarding the received commands and the previous cycles data.

Body Commands

All the playing and movement behaviors of the player are consisted from a few commands known as body commands that are presented briefly below.

The results of these commands are a little complicated and depend on many simulation factors. For the details of the execution of each command refer to the Soccer Server Section.

(turn *Moment*)

The *Moment* is in degrees from -180 to 180. This command will turn the player's body direction *Moment* degrees relative to the current direction.

(dash *Power*)

This command accelerates the player in the direction of its body (not direction of the current speed). The *Power* is between ***minpower*** (used value: -100) and ***maxpower*** (used value: 100).

(kick *Power Direction*)

Accelerates the ball with the given *Power* in the given *Direction*. The direction is relative to the the *Direction* of the body of the player and the power is again between ***minpower*** and ***maxparam***.

(catch *Direction*)

Goalie special command: Tries to catch the ball in the given *Direction* relative to its body direction. If the catch is successful the ball will be in the goalie's hand until kicked away.

(move *X Y*)

This command can be executed only before kick off and after a goal. It moves the player to the exact position of X (between -54 and 54) and Y (between -32 and 32) in one simulation cycle. This is useful for before kick off arrangements.

Note that in each simulation cycle, only one of the above five commands can be executed (i.e. if the client sends more than one command in a single cycle, one of them will be executed randomly, usually the one received first)

(turn_neck *Angle*)

This command can be sent (and will be executed) each cycle independently, along with other action commands. The neck will rotate with the given *Angle* relative to previous Angle. Note that the resulting neck angle will be between ***minneckang*** (default: -90) and ***maxneckang*** (default: 90) relative to the player's body direction.

Communication Commands

The only way of communication between two players is broadcasting of messages through the **say** command and hearing through the **hear** sensor.

(say *Message*)

This command broadcasts the *Message* through the field, and any player near enough (specified with ***audio_cut_dist***, default: 50.0 meters), with enough hearing capacity will hear the *Message*. The message is a string of valid characters.

(ok say)

Command succeeded.
In case of error there will be the following response from the Server:

(error illegal_command_form)

Misc. Commands

Other commands are usually of two forms:

- Data Request Commands

  (sense_body)

Requests the server to send sense body information. Note the server sends sense body information every cycle if you connect with version 6.00 or higher.

(score)

Request the server to send score information. The server's reply will be in this format:

(score *Time OurScore OpponentScore*)

- Mode Change Commands

(change_view *Width Quality*)

Changes the view parameters of the player. *Width* is one of **narrow**, **normal** or **wide** and *Quality* is one of **high** or **low**. The amount and detail of the information returned by the visual sensor depends on the width of the view and the quality. But note that the frequency of sending information also depends on these parameters (e.g. if you change the quality from high to low, the frequency doubles, and the time between two see sensors will be cut to half).

6.1.3. Sensor Information

Sensor information are the messages that are sent to all players regularly (e.g. each cycle or each one and half a cycle). There is no need to send any message to the server to get these information.

All the returned information of the sensors have a time label, indication the cycle number of the game when the data have been sent (indicated by *Time*). This time is very useful.

Visual Sensor

Visual Sensor is the most important sensor, and is a little bit complicated. This sensor returns information about the objects that can be seen from the player's view (i.e. objects that are in the view angle and not very far).

The main format of the information is:

(see *Time ObjInfo ObjInfo* . . .)

The *ObjInfo*s are of the format below:

(*ObjName Distance Direction* [*DistChange DirChange* [*BodyFacingDir HeadFacingDir*]])

or

(***ObjName Direction***)

Note that the amount of information returned for each object depends on its distance. The more distant the object is the less information you get. For more detailed information regarding *ObjInfo* refer to Appendix.

ObjName is in one of the following formats:
(p [*TeamName* [*Unum*]])
(b)
(f *FlagInfo*)
(g *Side*)
p stands for player, **b** stands for ball, **f** stands for flag and **g** stands for goal.
Side is one of **l** for left or **r** for right. For more information on *FlagInfo* refer to Appendix.

Audio Sensor

Audio sensor returns the messages that can be heard through the field. They may come from the online coach, referee, or other players.

The format is as follows:

(hear *Time Sender Message*)

Sender is one of the followings:
self: when the sender is yourself.
referee: when the sender is the referee of the game.
online_coach_l or **online_coach_r**
Direction: when the sender is a player other than yourself the relative direction of the sender is returned instead.

Body Sensor

Body sensor returns all the states of the player such as remaining stamina, view mode and the speed of the player at the beginning of each cycle:

(sense_body *Time* (view_mode { high | low } { narrow | normal | wide }) (stamina *Stamina Effort*) (speed *Speed Angle*) (head_angle *Angle*) (kick *Count*) (dash *Count*) (turn *Count*) (say *Count*) (turn_neck *Count*) (catch *Count*) (move *Count*) (change_view *Count*))

The last eight parameters are counters of the received commands. Use the counters to keep track of lost or delayed messages.

6.2. How to Create Clients

This section provides a brief description to write a first-step program of soccer client.

6.2.1. Sample Client

The Soccer Server distribution includes a very simple program for soccer clients, called `sampleclient`. It is under the "`sampleclient`" directory of the distribution, and is automatically compiled when you make the Soccer Server.

The `sampleclient` is not a stand-alone client: It is a simple 'pipe' that redirects commands from its standard input to the server, and information from the server to its standard output. Therefore, nothing happens when users invoke the `sampleclient`. The users must type-in commands from keyboards, and read the sensor information displayed on the terminal. (Actually it is impossible to read sensor information, because the server sends about 17 sensor informations (`see` information and `sense_body` information) per second.)

The `sampleclient` is useful to understand what clients should do, and what the clients will receive from the server.

How to Use `sampleclient`

Here is a typical usage of the `sampleclient`.

1. Invoke `client` under `sampleclient` directory of the Soccer Server.

   ```
   % ./client SERVERHOST
   ```

   Here, `SERVERHOST` is a hostname on which Soccer Server is running.

   Then the program awaits user input.

   If the Soccer Server uses an unusual port, for example 6005, instead of the standard port (6000), the users should use the following form.

   ```
   % ./client SERVERHOST 6005
   ```

2. Type in `init` command from the keyboard.

   ```
   (init MYTEAMNAME (version 7))
   ```

   Here `MYTEAMNAME` is a team name the users want to use.

   Then a player appears on the field. In the same time, the program starts to output the sensor information sent from the server to the terminal. Here is a typical output:

   ```
   send 6000 : (init foo (version 7))
   recv 1567 : (init r 1 before_kick_off)
   recv 1567 : (server_param 14.02 5 0.3 0.4 0.1 60 1 1 4000 45 0 0.3 0.5 ...
   recv 1567 : (player_param 7 3 3 0 0.2 -100 0 0.2 25 0 0.002 -100 0 0.2 ...
   recv 1567 : (player_type 0 1 45 0.4 5 0.006 0.3 0.7 0 0 1 0.6)
   recv 1567 : (player_type 1 1.16432 28.5679 0.533438 8.33595 0.00733326 ...
   recv 1567 : (player_type 2 1.19861 25.1387 0.437196 5.92991 0.00717675 ...
   ```

```
recv 1567 : (player_type 3 1.04904 40.0956 0.436023 5.90057 0.00631769 ...
recv 1567 : (player_type 4 1.1723 27.7704 0.568306 9.20764 0.00746072 ...
recv 1567 : (player_type 5 1.12561 32.4392 0.402203 5.05509 0.00621539 ...
recv 1567 : (player_type 6 1.02919 42.0812 0.581564 9.53909 0.00688457 ...
recv 1567 : (sense_body 0 (view_mode high normal) (stamina 4000 1) ...
recv 1567 : (see 0 ((g r) 61.6 37) ((f r t) 49.4 3) ((f p r t) 37 27) ...
recv 1567 : (sense_body 0 (view_mode high normal) (stamina 4000 1) ...
...
```

The first line, "`send 6000 : (init foo (version 7))`", is a report what the `client` sends to the server. The second line,"`recv 1567 : (init r 1 before_kick_off)` is a report of the first response from the server. Here, the server tells the client that the assigned player is the right side team (`r`), its uniform number is 1, and the current playmode is `before_kick_off`. The next 9 lines are `server_param` and `player_param`, which tells various parameters used in the simulation. Finally, the server starts to send the normal sensor informations, `sense_body` and `see`. Because the server sends these sensor information every 100ms or 150ms, the `client` continues to output the information endlessly.

3. Type in `move` command to place the player to the initial position. The player appears on a bench outside of the field. Users need to move it to its initial position by `move` command like:

   ```
   (move -10 10)
   ```

   Then the player moves to the point (-10,10).

   Because, as mentioned before, the `client` program outputs sensor information endlessly, users can not see strings they type in. So, they must type-in commands blindly. [1]

4. Click 'Kick-Off' button on the Soccer Server. Then the game starts. The users can see that the time data in each sensor information (the first number of `see` and `sense_body` information) are increasing.

5. After then, users can use any normal command, `turn`, `dash`, `kick` and so on. For example, users can turn the player to the right by typing:

   ```
   (turn 90)
   ```

   The player can dash forward with full power by typing:

   ```
   (dash 100)
   ```

[1] Users can redirect the output to any file or program. For example, you can redirect it to `/dev/null` to discard the information by invoking "`% client SERVERHOST > /dev/null`". Then, the users can see the string they type-in.

When the player is near enough to the ball, it can kick the ball to the left with power 50 by:

```
(kick 50 -90)
```

Note again that because of endless sensor output, users must type-in these commands blindly.

Overall Structure of Sample Client

The structure of the `sampleclient` is simple. The brief process the `client` does is as follows:

1. Open a UDP socket and connect to the server port. (`init_connection()`)
2. Enter the read-write loop (`message_loop`), in which the following two processes are executed in parallel.
   - read user's input from the standard input (usually a keyboard) and send it to the server (`send_message()`).
   - receive the sensor information from the server (`receive_message()`) and output it to the standard output (usually a console).

In order to realize the parallel execution, `sampleclient` uses the `select()` function. The function enables to wait for multiple input from sockets and streams in a single process. When `select()` is called, it waits until one of the sockets and streams gets input data, and tells which sockets or streams got the data. For more details of the usage of `select()`, please refer to the man page or manual documents.

An important tip in the `sampleclient` is that the client must change the server's port number when it receives sensor informations from the server. This is because the server assign a new port to a client when it receives an `init` command. This is done by the following statement in "`client.c`" (around line 147):

```
        printf( "recv %d : ", ntohs(serv_addr.sin_port));
+       sock->serv_addr.sin_port = serv_addr.sin_port ;
        buf[n] = '\0' ;
```

6.2.2. Simple Clients

In order to develop complete soccer clients, what users must do is to write code of a 'brain' part, which performs the same thing as users do with the `sampleclient` described in the previous section. In other words, users must write a code to generate command strings to send to the server based on received sensor information.

Of course it is not a simple task (so that many researchers tackle RoboCup as a research issue), and there are various ways to implement it. Simply saying, in order to develop player clients, users need to realize the following functions:

[Sensing] To analyze sensor information: As shown in the previous section, the server sends various sensor information in S-expressions. Therefore, a client needs to parse the S-expressions. Then, the client must analyze the information to get a certain internal representation. For example, the client needs to analyze a visual information to estimate player's location and field status, because the visual information only include relative locations of landmarks and moving objects on the field.

[Action Interval] To control interval of sending commands: Because the server accepts a body command (turn, dash and kick) per 100ms, the client needs to wait appropriate interval before sending a command.

[Parallelism] To execute sensor and action processes in parallel: Because the Soccer Server processes sensor information and command asynchronously, clients need to execute a sensor process, which deals with sensor information, and an action process, which controls to send commands, in parallel.

[Planning] To make a plan of play: Using sensor information, the client needs to generate appropriate command sequences of play. Of course, this is the final goal of developing soccer clients!!

Here are two simple examples of stand-alone players, `sclient1` and `sclient2`, which just chase the ball and kick it to the opponent goal. The sources are available from:

`ftp://ci.etl.go.jp/pub/soccer/client/noda-client-2.0.tar.gz`

In the examples, the functions listed above are realized as follows:

- For *Sensing* function, both examples use common facilities of `class BasePlayer`, `class FieldState`, and `estimatePos` functions. By these facilities, the example programs do:
  - receive data from a socket connected with the server,
  - parse the data as S-expression,
  - interpret the expression into internal data format (`class SensorInfo`),
  - and in the case the received data is visual sensor information, estimate player's and other object's positions.

  For more detail, please read the source code.
- For *Action Interval* and *Parallelism* functions, the two examples use different methods. The first example, `sclient1` uses timeout of `select()` function. The second one, `sclient2` uses the multi-thread (pthread) facility. These are described below.
- For *Planning* function, both examples have very simple planners as follows:
  - If the player does not see the ball in recent 10 steps, or if the player can not estimate its position in recent 10 steps, it looks around.

- If the ball is in kickable area, it kicks the ball to the opponent goal.
- Otherwise, the player rushes to the ball (turns to the ball and dashes).

`sclient1`

The `sclient1` uses the timeout facility of `select()` function to realize *Action Interval* and *Parallelism*.

The key part of the program is in `MyPlayer::run()`. Here is the part of the source code:

```
//-----------------------------------------
// enter main loop

SocketReadSelector selector ;

TimeVal nexttic ;  // indicate the timestamp for next command send
nexttic.update() ; // set nexttic to the current time.

while(True) {

   //-----------------------------------------------
   // setup selector

   selector.clear() ;
   selector.set(socket) ;

   //-----------------------------------------------
   // wait socket input or timeout (100ms) ;

   Int r = selector.selectUntil(nexttic) ;

   if(r == 0) { // in the of timeout. (no sensor input)
      doAction() ;  // enter action part
      nexttic += TimeVal(0,100,0) ; // increase nexttimetic 100ms
   } else {  // got some input
      doSensing() ; // enter sensor part
   }
}
```

Here, `class SocketReadSelector` is a class to abstract facilities of `select()` and is defined in "`itk/Socket.h`". In the line "`Int r = selector.selectUntil(nexttic) ;`", the program awaits the socket input or timeout indicated by `nexttic`, which holds the timestamp of the next tic (simulation step). The function returns 0 if timeout, or the number of receiving sockets. In the case of timeout, the program calls `doAction()` in which a command is generated and sent to the server, or otherwise, it calls `doSensing()` in which a sensor information is processed.

`sclient2`

The `sclient2` uses the POSIX thread (pthread) facilities to realize *Action Interval* and *Parallelism*.

The key part of the program is also in `MyPlayer::run()`. Here is the part of the source code:

```
//----------------------------------------
// fork sensor thread

forkSensor() ;

//----------------------------------------
// main loop

while(True) {
   if        (!isBallSeenRecently(10)) {
      //------------------------------
      // if ball is not seen recently
      // look around by (turn 60)

      for(UInt i = 0 ; i < 6 ; i++) {
         turn(60) ;
      }
   } else if (kickable()) {
      ...
   }
}
```

The statement "`forkSensor() ;`" invokes a new thread for receiving and analyzing the sensor information. (The behavior of the sensor thread are defined in "`SimpleClient.*`" and "`ThreadedClient.*`".) Then the main thread enters the main loop in which action sequences of "*chasing the ball and kick to the goal*" are generated. Because *Sensing* function is handled in the sensor thread in parallel, the main thread needs not take care of the sensor input.

In order to keep action interval to be 100ms, the `sclient2` waits for the next simulation step by the function `ThreadedPlayer::sendCommandPre()` defined in "`ThreadedPlayer.cc`" as follows:

```
Bool ThreadedPlayer::sendCommandPre(Bool bodyp) {
   cvSend.lock() ;
```

```
    if(bodyp) {
       while(nextSendBodyTime.isFuture())
          cvSend.waitUntil(nextSendBodyTime) ;
    }

    while(nextSendTime.isFuture()) {
       cvSend.waitUntil(nextSendTime) ;
    }

    return True ;
 } ;
```

In this function, `MutexCondVar cvSend` provide a similar timeout facility of `select()` function used in `sclient1` described above. (`MutexCondVar` is a combination of condition variable (`pthread_cond_t`) and mutex (`pthread_mutex_`), and is defined in "`itk/MutexCondVar.h`".) Because the function is called just before the player sends a command to the server, and `nextSendBodyTime` is controlled to indicate the timestamp of the next simulation step, the thread waits to send a command in the next tic.

6.2.3. Tips

Here we collect tips to develop soccer client programs.

- Debugging is the main problem in developing your own team. So try to find easy debuging methods.
- A nice and simple way to see your program's variables in a condition is to use an **abort()** command or some **assert**s to force the program to core-dump; And debug the core using `gbd`.
- Log every message received from the server and sent to the server. It is very useful for debugging.
- Using ready to use libraries for socket and parsing problems is useful if you are a beginner.
- Remember to pass the version number to the server in the init command. Although it is optional, the default is 3.00 which usually is not desired.
- Even if the catch probability is 1.00 your catch command may be unsuccessful because of errors in returned sensors about the positions.
- The first serious problem you may encounter is the timing problem. There are many methods to synchronize your client's time with server. One simple methods is to use received sense body information.

- Beware of slow networks. If your timing is not very powerful your client's will behave abnormaly in a crowded or slow network or if they are out of process resources (e.g. you run many clients on one slow machine). In this case they may see older positions and will try to act in these positions and this will result in confusion (e.g. they will turn around themselves)

- The main usage of flags are for the player to find the position of himself in the field. Your very first clients may ignore flags and play with relative system of positions. But you may need a positioning module in the near future. There are many of the in the ready to use libraries.

- The program should check the end of buffer in analyzing sensor information. The sensor information uses S-expressions. But the expression may not be completed when the sensor data is longer than the buffer, so that some closing parentheses are lost. In this case, the program may core-dump if it parses the expression naively.

7. The coach

7.1. Introduction

Coaches are privileged clients used to provide assistance to the players. There are two kinds of coaches, the online coach and the trainer. The latter is often called 'off-line coach' as well, but for clarity sake we will use the term 'trainer'.

7.2. Distinction between trainer and online coach

In general, the trainer can exercise more control over the game and may be used only in the development stage, whereas the online coach may connect to official games. The trainer is useful during development for such tasks as running automated learning or managing games. The on-line coach is used during games to provide additional advice and information to the players.

While developing player clients, for example when applying machine learning methods to learn skills like dribbling or kicking, it might be useful to create training sessions in an automated way. Therefore, the trainer has the following capabilities:

- It can control the play-mode.
- It can broadcast audio messages. Such a message can consist of a command or some information intended for one or more of the player-clients. Its syntax and interpretation are user-defined.
- It can move the players and the ball to any location on the field and set their directions and velocities.
- It can get noise-free information about the movable objects.

For details on these capabilities see Section 7.3.

The online coach is intended to observe the game and provide advice and information to the players. Therefore, it's capabilities are somewhat limited:

- It can communicate with the players.
- It can get noise-free information about the movable objects.

To prevent the coach from controlling each client in a centralized way, communication is restricted in several ways as described in Section 7.7. The online coach is a good tool for opponent modelling, game analysis, and giving strategic tips to its teammates. Since the coach gets a noise-free, global view over the field and has less real-time demands, it is expected that it can spend more time deliberating over strategies. See Section 7.6 for more details about the online coach.

7.3. Trainer

7.3.1. Connecting with and without the soccerserver referee

By default, an internal referee module is active within the soccerserver that controls the match (see Section 4.7). If the trainer should have complete control over the match, the soccerserver must be instructed to deactivate the referee module. This means for example, that the play-mode will not change and players will not be moved back to their sides after a goal. The trainer has to react to these events by its own rules.

The soccerserver must be informed at startup-time that a trainer-client will be used. Add the option -*coach*[1] to the command arguments of the soccerserver application when a coach-client is used and the internal referee module of the server must be deactivated. You can also add the line *coach* to the server.conf.

If you want to connect a trainer but let the server referee remain activated, add the option -*coach_w_referee* to the command arguments of the server or add *coach_w_referee* to the server configuration file.

If the server is invoked with one of the trainer modes, it prepares a UDP socket to which the trainer-client can connect. The default port number is 6001[2]. If a different port number is needed the new port can be set by assigning its value to the *coach_port* parameter (see Section 4.9.1).

7.4. Commands

The trainer and the online coach can use the following set of commands. The items are listed in three categories. The first category includes commands that can be used only by the trainer, the second includes commands that can be used also by the online coach with certain restrictions, and the third lists commands that can be used by both trainer and online coach.

7.4.1. Commands that can be used only by the trainer

- **(change_mode PLAY_MODE)**

[1]Note: The name of this parameter refers to the notion of 'offline-coach', not to be mixed up with the online-coach.

[2]The default port number for online coaches is 6002.

Change the play-mode to PLAY_MODE. PLAY_MODE must match one of the modes defined in Section 4.7.1. Note that for most play-mode requests the soccerserver will only change the play-mode. The position of the ball usually remains unchanged, but in some cases players will be moved. E.g. in free-kick and kick-in playmodes they will be moved away from the ball if they stand within a certain radius. When changing to '`before_kick_off`' they will be even moved to their own side.

Possible replies by the soccerserver:

- **(ok change_mode)**
  The command succeeded.
- **(error illegal_mode)**
  The specified mode was not valid.
- **(error illegal_command_form)**
  The PLAY_MODE argument was omitted.

- **(move OBJECT X Y [VDIR [VEL$_X$ VEL$_Y$]])**

  This command will move OBJECT, which may be a player or the ball (see Section sec:sensormodels for format information), to absolute position (X, Y). If VDIR is specified, it will also change its absolute facing direction to VDIR (this only matters for players). Additionally, if VEL$_X$ and VEL$_Y$ are specified, the object's velocity will be set accordingly.

  The trainer always uses left-hand coordinates.

  Possible replies by the soccerserver:

  - **(ok move)**
    The command succeeded.
  - **(error illegal_object_form)**
    The OBJECT specification was not valid.
  - **(error illegal_command_form)**
    The position, direction, and/or velocity specification was not valid.

- **(check_ball)**

  Ask the soccerserver to check the position of the ball. Four positions are defined:

  - **in_field**
    The ball is within the boundaries of the field.
  - **goal_l**
    The ball is within the area assigned to the goal at the left side of the field.
  - **goal_r**
    The ball is within the area assigned to the goal at the right side of the field.

  - **out_of_field**
    The ball is somewhere else.

  Note that the states '`goal_l`' and '`goal_r`' do not necessary imply that the ball actually crossed the goal line.

  Possible replies by the soccerserver:

  - **(ok check_ball TIME BALLPOSITION)**
    BALLPOSITION will be one of the states specified above.

- **(start)**

  This commands starts the server, e.g. sets the play-mode to '`kick_off_l`'. This essentially simulates pressing the kick off button on the monitor.

  If the trainer does *not* send an init command, then the first commands of any type received from the trainer will cause the server to start, e.g. set the play-mode to '`kick_off_l`'.

  Possible replies by the soccerserver:

  - **(ok start)**
    The command succeeded.

- **(recover)**

  This command resets players' stamina, recovery, effort and hear capacity to the values at the beginning of the game.

  Possible replies by the soccerserver:

  - **(ok recover)**
    The command succeeded.

- **(ear MODE)**

  It turns on or off the sending of auditory information to the trainer. MODE must be one of **on** and **off**. If **(ear on)** is sent, the server sends *all* auditory information to the trainer. See Table 7.3 for the format. If **(ear off)** is sent, the server stops sending auditory information to the trainer.

  Possible replies by the soccerserver:

  - **(ok ear on)**
    (ok ear off)
    Both replies indicate that the command succeeded.
  - **(error illegal_mode)**
    MODE did not match **on** or **off**.
  - **(error illegal_command_form)**
    The MODE argument was omitted.

7.4.2. Commands that can also be used by the online coach with certain restrictions

- **(init (version VERSION))** for the trainer and
- **(init TEAMNAME (version VERSION))** for the online coach.

  These commands tell the server which protocol version should be used to communicate with the trainer or coach. In the case of the online coach TEAMNAME has to be specified to indicate which team the coach belongs to. Note that the coach must connect after at least one player from its team.

  The trainer is *not* required to issue an init command. However, it is recommended that the trainer does so. Otherwise, the server will communicate with an older protocol.

  It should be mentioned that the default port is 6001 for the trainer and 6002 for the online coach.

  Possible replies by the soccerserver:

  - **(init ok)**
    The command succeeded in case of the trainer.
  - **(init SIDE ok)**
    The command succeeded in case of the online coach. SIDE is either 'l' or 'r'.

- **(say MESSAGE)**

  Note that the online coach can use this command with the same syntax, but there are more restrictions. See Section 7.6.2 for details.

  This command broadcasts the message MESSAGE to all clients in the case of the trainer and only to teammates in the case of the online coach. For the trainer the format of MESSAGE is the same as for a player-client. It must be a string whose length is less than *say_coach_msg_size*(see Section 4.9.1) and it must consist of alphanumeric characters and/or the symbols ().+*/?<>_

  The format which the players hear these messages can be found in Section 4.3.1.

  Possible replies by the soccerserver:

  - **(ok say)**
    The command succeeded.
  - **(error illegal_command_form)**
    MESSAGE did not match the required format.

- **(change_player_type TEAM_NAME UNUM PLAYER_TYPE)** for the trainer and

- **(change_player_type UNUM PLAYER_TYPE)** for the online coach.

  These commands can be used to change the heterogeneous player type (see Section 4.6) of the player with the number UNUM of team TEAM_NAME to the type PLAYER_TYPE. PLAYER_TYPE is a digit between 0 and 6, where 0 denotes the default player type. Note that in the case of the online coach the argument TEAM_NAME is missing, because it can only change player types in its own team.

  The trainer does not have to comply to the rule that a maximum of three (specified by *subs_max*) players of each type can be on the field.

  See Section 7.6.3 for details about the restrictions as to when and how the online coach may substitute players.

  Possible replies by the soccerserver to both trainer and online coach:

  - **(warning no_team_found)**
    The team does not exist.
  - **(error illegal_command_form)**
    If **change_player_type** is not followed by a string, two integers and a close bracket.
  - **(warning no_such_player)**
    If there is no player with that uniform number on that team.
  - **(ok change_player_type TEAM UNUM TYPE)**
    The command succeeded.

  Additionally, the soccerserver can send the following replies to the online coach:

  - **(warning cannot_sub_while_playon)**
    If the play-mode is '`play_on`'.
  - **(warning no_subs_left)**
    If the coach has already made its three (specified by *subs_max*) subs for the game.
  - **(warning max_of_that_type_on_field)**
    If the player-type is not the default and there are three (specified by *subs_max*) of that type already on the field.
  - **(warning cannot_change_goalie)**
    If the coach tries to change the player type of the goalie.

  The server responds to the teammates with:

  - **(change_player_type UNUM TYPE)**

  and opponents (including opponent coach) with:

  - **(change_player_type UNUM)**

7.4.3. Commands that can be used by both trainer and online-coach

- **(look)**

  This command provides information about the positions of the following objects on the field:

  - The left and right goals.
  - The ball.
  - All active players.

  Note that the trainer and online coach for *both* sides receive left hand coordinates. That is, the coaches receive information in the global coordinates that the left hand team uses. In general, the players receive no global information (the one exception being the **move** command), but it is common for teams to localize themselves so that the negative x direction is towards the goal they defend.

  Possible replies by the soccerserver:

  - **(ok look TIME (OBJ$_1$ OBJDESC$_1$) (OBJ$_2$ OBJDESC$_2$) ...)**
    OBJ$_j$ can be any of the objects mentioned above. See Section 4.3 for information about the way the names for those objects are composed. OBJDESC$_j$ have the following form:
    - ▷ For goals : X Y
    - ▷ For the ball: X Y DELTA$_X$ DELTA$_Y$
    - ▷ For players : X Y DELTA$_X$ DELTA$_Y$ BODYANGLE NECKANGLE

  The coordinates are always in left-hand orientation, no matter whether a trainer or online coach is used.

  If the trainer/coach should receive visual information periodically, use the **(eye on)** command.

- **(eye MODE)**

  MODE must be one of **on** and **off**. If **(eye on)** is sent, the server starts sending **(see_global ...)** information (see Section 7.5) every 100 ms (the interval is specified by the *send_vi_step* parameter automatically to the client. If **(eye off)** is sent, the server stops to send visual information automatically. In this case the trainer/coach has to ask actively with **(look)**, if it needs visual information.

  Possible replies by the soccerserver:

  - **(ok eye on)**
    (ok eye off)
    Both replies indicate that the command succeeded.
  - **(error illegal_mode)**
    MODE id not match **on** or **off**.

  – **(error illegal_command_form)**
    The MODE argument was omitted.

- **(team_names)**

  This command makes the trainer/coach receive information about the names of both teams and which side they are playing on.

  Possible replies by the soccerserver:

  – **(ok team_names [(team l TEAMNAME$_1$) [(team r TEAMNAME$_2$)]])**
    Depending on whether the teams already connected no, one, or both team name(s) will be supplied. Recall that the first team that connects will be on the left side.

7.5. Messages from the server

Apart from the replies to the commands mentioned above the server also sends some messages to the trainer and online coach. If the clients connect to the server with a version >= 7.0 (using the **init**-command), they will receive the following parameter messages just like player clients:

- **(server_param ...)** once
- **(player_param ...)** once
- **(player_type ...)** once for each player type

See Section 4.2.2 for details on the parameter messages.

If the client chooses to receive visual information in each cycle by sending **(eye on)** it will receive messages in the following format every 100 ms (*send_vi_step*):
(see_global (OBJ$_1$ OBJDESC$_1$) (OBJ$_2$ OBJDESC$_2$) ...)
OBJ$_j$ denotes the name of the object. See Table 4.3 for information about the way the names for those objects are composed. OBJDESC$_j$ have the following form:

- For goals : X Y
- For the ball: X Y DELTA$_X$ DELTA$_Y$
- For players : X Y DELTA$_X$ DELTA$_Y$ BODYANGLE NECKANGLE

The syntax is the same as in the reply to the **(look)** command, so coordinates are always in left-hand orientation.

If the client wants to receive auditory information and sent **(ear on)** to the server, it will receive all auditory information, from both the referees and all of the players. There are two kinds of hear messages:

- **(hear TIME referee MESSAGE)** for all referee messages, such as "play_on" and "free_kick_left". See Section 4.7 for a list of the valid messages from the referee.
- **(hear TIME (p "TEAMNAME" NUM) "MESSAGE")** for all player messages. Note the quotes around the message.

See Section 4.3.1 for more details about the players speaking and listening abilities.

7.6. Online coach

7.6.1. Introduction

The online coach is a privileged client that can connect to the server in official games. It has the capability of receiving global and noise-free information about the objects on the field. In order to encourage research in this area starting in 2001 there will be special coach contests. This way, research groups that do not want to develop a team of player clients can participate in the RoboCup challenge by focussing on the online coach. Additionally, in order to make it possible to use a single coach with a variety of teams, a standard coach language has been developed that can be used to communicate with the players.

See Section 7.4 and 7.5 for details about the commands that can be used by the online coach and messages that will be sent by the server.

7.6.2. Communication with the players

Prior to version 7.00, the online coach could say say short (128 characters, *say_coach_msg_size*) alphanumeric (plus the symbols ().+*/?<>_) messages when the play-mode is not '`play_on`'. This type of message still exists as a "freeform" message, but there are now other standard message types.

To prevent coaches from micro-controlling every single action of the players communication is restricted in the following ways. In the standard coach language there are four other types of messages: advice, define, info, and meta messages. Per 300 cycles (specified by *clang_win_size*) the coach can send one of each. Note that the number of allowed messages can be changed by setting the *clang_advice_win*, *clang_define_win*, *clang_info_win*, and *clang_meta_win* parameters (see Section 4.9.1). The messages are heard by the players 50 (specified by *clang_mess_delay*) cycles later. If the play-mode is not '`play_on`', one (specified by *clang_mess_per_cycle*) message is sent to the players in each cycle, even if the delay time has not elapsed. Messages that are sent while the play mode is not '`play_on`' do not count towards the message number restriction. For example, if the default values are used the coach can send one message per cycle during breaks that will be heard by the players without delay. The server guarantees that messages will be sent to the players in the same order in which they were received from the coach.

The language grammar developed below does not place restrictions on the length of the messages which can be sent to the server. However, for very practical reasons, any message in the standard language can not be longer than 2013 characters (this is so the maximum message which should be sent to the player is 2048 characters).

For freeform messages, the coach can only speak in non-'`play_on`' modes. The coach can send *say_coach_cnt_max* freeform messages per game. The length of these messages has to be less than *say_coach_msg_size*. If the game continues into extended time, the online coaches are given an additional *say_coach_cnt_max* messages to say every additional 6000 cycles (or whatever the normal length of a game is). Allowed messages are cumulative, so if the coach does not use all its allowed messages, it can use them in the extended time. The server will send **(error said_too_many_messages)** if the coach tries to send messages after it reached the maximum number.

The standard coach language will be described in detail in Section 7.7.

7.6.3. Changing player types in a real game

Using the **change_player_type**-command (described in in Section 7.4) the online coach can change player types unlimited times in '`before_kick_off`' play-mode. Of course these changes have to comply with the general rules about heterogeneous players (see Section 4.6). After kick-off player types can be changed three (*subs_max*) times during play-modes that are not '`play_on`'.

See the description of the **change_player_type**-command in Section 7.4 for details about the possible replies from the server.

Note: A player client will be informed about substitutions that occurred before the client connected by the message **(change_player_type UNUM TYPE)** for substitutions in it own team and **(change_player_type UNUM)** for substitutions in the opponent team.

7.7. The standard coach language

7.7.1. General properties

The standard coach language was developed to enable coaches to work together with teams from different research groups. One of the design goals was to have clear semantics that should prevent misinterpretation from both the players and the coach. The language is based on low-level concepts that can be combined to construct new high level concepts.

Additionally, coaches can communicate a certain number of freeform messages that may be arbitrary strings to the players during non-'`play_on`'-modes. See Section 7.6.2 for details. Be aware though, that freeform messages probably will not be understood by other teams if you plan to use your coach with other teams.

The language description below is the just the first version of the language developed by the community. It is hoped that all interested researchers will continue to develop this language and it will be improved over time.

Note that the server itself parses all the coach messages using flex and bison (the GNU replacements for lex and yacc) and constructs a simple representation based on a C++ class hierarchy. Please feel free to use and modify this code from the server to handle the parsing of the coach messages. In particular, look at the `coach_lang.[Chly]` and `coach_lang_comp.[Ch]` files.

7.7.2. Overview of the five message types

There are five types of coach messages in the standard coach language: Info, Advice, Define, Meta, and Freeform.

Info Info messages carry information that the coach believes the players should know, e.g. frequent positions of the opponent or the player type of opponent players. Info messages can carry information about the opponent (single players, sets of players, or the whole team), or about the coach's own team and players.

Format: **(info TOKEN$_1$ TOKEN$_2$ . . . TOKEN$_n$)**

The format of TOKEN$_i$ will be described in Section 7.7.3.

Advice Advice messages tell the players what the coach believes they should do, either at an individual, group or team level. Advice messages can thus never instruct opponent players what they should do.

Format: **(advice TOKEN$_1$ TOKEN$_2$ . . . TOKEN$_n$)**

The format of TOKEN$_i$ will be described in Section 7.7.3.

Define Define messages introduce names i.e. shortcuts for regions, directives, conditions, and actions. (these will be described later). While these messages do not add expressivity, it could allow the coach to deal with the absolute message length restriction, reduce network load, and make the messages more understandable for humans.

Format: **(define DEFINE_TOKEN$_1$ DEFINE_TOKEN$_2$ . . . DEFINE_TOKEN$_n$)**

There are four types of DEFINE_TOKEN:

- **(definer name REGION)**
- **(definec name CONDITION)**
- **(defined name DIRECTIVE)**
- **(definea name ACTION)**

Note that the name must be delimінted by quotes and it limited to 40 characters.

Meta Meta messages are used to carry meta-level information about the coach-players interactions, such as the number of messages sent, the standard language version supported, etc. These messages are intended purely to support debugging and upward-compatibility.

Format: **(meta META_TOKEN$_1$ META_TOKEN$_2$ . . .)**

Only one form of META_TOKEN exists at this stage:

- **(version X)**
  The coach uses protocol version X.

Freeform Freeform messages allow an arbitrary short string (less that *say_coach_msg_size* characters), but only when the play-mode is not '`play_on`'. The message can consist on alphanumeric characters plus the symbols ().+*/?<>_ Only *say_coach_cnt_max* messages can be sent per 6000 cycles.

Format: **(freeform "STRING")**

Note that STRING must be included in quotes.

7.7.3. Semantics and syntax details of info and advice messages

In the following the syntax and semantics of the aforementioned TOKENs in info and advice messages will be described.

There is one special token, **(clear)**, which tells the player to ignore all previous info/advice (this is like forcing the time to live of all messages to 0).

All other token involve a condition in which it applies, and a list of directives for the team and/or individual players. Basically, a token specifies a set play, or a reactive plan.

Format: **(TTL CONDITION DIRECTIVE$_1$ DIRECTIVE$_2$ . . . DIRECTIVE$_n$)**

TTL means time-to-live and specifies the coach's assumption as to when it thinks the information will become obsolete. After TTL cycles since the coach sent the message it becomes irrelevant. Note that the time to live depends on when the coach *sent* the message, not when the player *received* it.

As INFO messages, tokens report on the coach's beliefs about the opponent's or its own team's plans and behavior. As ADVICE messages, tokens direct the team to act in particular ways.

This form makes using ADVICE messages very easy. Each such token is basically a simple rule with a condition on the left-hand side, and a set of actions on the right hand side. Thus each rule can be thought of as essentially specifying an if-then statement:

```
if CONDITION
then { DIRECTIVE_1 DIRECTIVE_2 ... }
```

In the player's programs, it is easy to represent all the advice given by the coach as a small rule-base. Following the advice would be easy by matching the current world state against the condition, and trying to act on the directives. Note: If more than one condition applies to the current situation and the corresponding directives differ, it is up to the player to choose the directive. Note that the player should also exercise some discretion in following directives. For example, if the only directive which matches is to pass to player 5, but player 5 is well-covered by opponents, the player with the ball may choose to ignore the directive for now.

- Conditions:

  A condition is made from the logical connectives over atomic state description propositions:

- **(true)**
  Always true.
- **(false)**
  Always false.
- **(ppos TEAM UNUM_SET INT INT REGION)**
  The first INT is the MINIMUM and the second is the MAXIMUM At least MINUMUM but no more than MAXIMUM players in UNUM_SET from team TEAM are in region REGION. Regions and unum sets are more precisely defined below. TEAM is either "our" or "opp". There is no ambiguity since the coach can only be heard by its own players.
- **(bpos REGION)**
  The ball is in region REGION.
- **(bowner TEAM UNUM_SET)**
  The ball is controlled by *some* player in UNUM_SET of team TEAM. The ball-owner is the last player that had ball contact (i.e. the ball was in his kickable area), even if the ball left his control after that.
- **(playm PLAY_MODE)**
  The play-mode is PLAY_MODE. See Section 7.7.4 for the valid values of PLAY_MODE.

The logical connectives are:

- **(and CONDITION$_1$ CONDITION$_2$ ... CONDITION$_n$)**
- **(or CONDITION$_1$ CONDITION$_2$ ... CONDITION$_n$)**
- **(not CONDITION)**

An example condition: "When opponent player 3 is in region X and controls the ball" would be

(and (player_pos opponent {3} X) (ball_owner opponent {3}))

- Directives:

  Directives are basically lists of actions for individual sets of players and come in two forms:

  - **(do TEAM UNUM_SET ACTION)** (players should take this action)
  - **(dont TEAM UNUM_SET ACTION)** (players should avoid taking this action)

  In INFO messages, directives convey knowledge about the plans/behaviors of the players or their opponents.

- Actions:

- **(pos REGION)**
  The player should position itself in REGION.
- **(home REGION)**
  The player's default position should be in REGION. This directive is intended largely to specify formations for the team.
- **(bto REGION BALLMOVE_SET)**
  The ball should be moved towards the region REGION. BALLMOVE_SET can include the characters 'p','d','c', and/or 's' for pass, dribble, clear (kick to no particular player), and score. Note that no spaces are needed and order does not matter. Some valid BALLMOVE_SET values are "pd," "s," and "pdcs." Of course it is up to the players to decide, especially which of the actions in BALLMOVE_SET it will pursue. Also if the specified REGION in the case of a **score** directive is not in the goal area, the player should decide where to shoot.
- **(bto UNUM_SET)**
  The ball should be passed to some player in UNUM_SET.
- **(mark UNUM_SET)**
  The player should mark some opponent player in UNUM_SET.
- **(markl REGION)**
  The passing lane from the current ball position to REGION should be marked.
- **(markl UNUM_SET)**
  The passing lane from the current ball position to some opponent player in UNUM_SET should be marked.
- **(oline REGION)**
  The offside-trap line for the player/team should be set at REGION.
- **(htype TYPE)**
  The player is of heterogeneous type TYPE. The TYPE number is as described in Section 4.6. A value of -1 should clear the player's idea of the heterogeneous type.

• Regions:

Any REGION token can be any of the following:

- a POINT
  This is defined more precisely below
- **(quad POINT$_1$ POINT$_2$ POINT$_3$ POINT$_4$)**
  Defines a four-sided shape given by the four coordinates, using the global coordinate system and point primitive above. POINT$_1$ is connected to POINT$_2$, which is connected to POINT$_3$, which is connected to POINT$_4$, which is connected back to PAINT$_1$.

- **(arc POINT RADIUS_SMALL RADIUS_LARGE ANGLE_BEGIN ANGLE_SPAN)**
  Defines a donut-arc: the area between two circles co-centered at point POINT, having the given radii, with the arc defined starting at the beginning angle and covering the spannign angle. For example a, a circle with radius r could be defined as "(arc (pt 0 0) 0 r 0 360)", and a U-shaped region could be defined as "(arc (pt 0 0) 5 10 0 180)"
- **(null)**
  The null (empty) region.
- **(reg REG$_1$ REG$_2$ ... REG$_n$)**
  Defines a region made up from the union of the given regions.

A POINT is any of the following:

- **(pt X Y)**
  X and Y are reals and in global coordinates. This is the absolute position (X,Y);
- **(pt X Y POINT)** This is the point (X,Y)+POINT.
- **(pt ball)** The current global position of the ball.
- **(pt TEAM UNUM)** The current position of player number UNUM on team TEAM (either 'our' or 'opp')

The use of these relative points makes it easy to express ideas such as "Move to the ball", "If there are 2 teammates within 10m of the ball", etc.

Remember that the online coach receives visual information alway in left-hand orientation, no matter which side its team plays on. Yet, when sending messages to a team that plays on the right side, the coach must use right-hand orientation in the messages. Transforming coordinates from left- to right-hand orientation is done by negating them.

- UNUM SETS:

  Unum sets are sets of player numbers. These are sets in the sense that order does not matter and may be changed by the server. If 0 is included anywhere in the set, then the set contains all players 1 - 11

  Format: { NUM$_1$ NUM$_2$ ... NUM$_n$ }

7.7.4. Syntax

The whole grammar of the standard coach language:

```
<MESSAGE> -> <INFO_MESS>
           | <ADVICE_MESS>
```

```
           | <META_MESS>
           | <DEFINE_MESS>
           | <FREEFORM_MESS>

#Advice and Info messages
<INFO_MESS> -> (info <TOKEN_LIST>)
<ADVICE_MESS> -> (advice <TOKEN_LIST>)
<TOKEN_LIST> -> <TOKEN_LIST> <TOKEN> | <TOKEN>
<TOKEN> -> (<TIME> <CONDITION> <DIRECTIVE_LIST>)
         | (clear)
<CONDITION> -> (true)
             | (false)
#the int's are the min/max number of players which must match
             | (ppos <TEAM> <UNUM_SET> <INT> <INT> <REGION>)
             | (bpos <REGION>)
             | (bowner <TEAM> <UNUM_SET>)
             | (playm <PLAY_MODE>) |
             | (and <CONDITION_LIST>) |
             | (or <CONDITION_LIST>) |
             | (not <CONDITION>)
             | "STRING"
<CONDITION_LIST> -> <CONDITION_LIST> <CONDITION>
<DIRECTIVE_LIST> -> <DIRECTIVE_LIST> <DIRECTIVE> | <DIRECTIVE>
<DIRECTIVE> -> (do <TEAM> <UNUM_SET> <ACTION>) |
             | (dont <TEAM> <UNUM_SET> <ACTION>)
             | "STRING"
<ACTION> -> (pos <REGION>) |
          | (home <REGION>) |
          | (bto <REGION> <BMOVE_SET>) |
          | (bto <UNUM_SET>) |
          | (mark <UNUM_SET>) |
          | (markl <UNUM_SET>) |
          | (markl <REGION>) |
          | (oline <REGION>) |
          | (htype <HET_TYPE>)
          | "STRING"

#Misc. bits of data
<PLAY_MODE> -> bko | time_over | play_on
             | ko_our | ko_opp | ki_out | ki_opp | fk_our | fk_opp
             | ck_our | ck_opp | gk_our | gk_opp | gc_our | gc_opp
             | ag_our | ag_opp
<TIME> -> [int]
<HET_TYPE> -> [int] #0-based, -1 to clear knowledge
```

```
<TEAM> -> our | opp
<UNUM> -> [int(0-11)]
<UNUM_SET> -> { <UNUM_LIST> }
<UNUM_LIST> -> <UNUM_LIST> <UNUM> | e #e here means epsilon,
                                      # i.e. the empty string
<BMOVE_SET> -> { <BMOVE_LIST> }
<BMOVE_LIST> -> <BMOVE_LIST> <BMOVE_TOKEN> | <BMOVE_TOKEN>
<BMOVE_TOKEN> -> p | d | c | s #pass dribble clear score; note
                               # these do not need spaces between them


#Regions
<REGION> -> <POINT> |
          | (null)
          | (quad <POINT> <POINT> <POINT> <POINT>) |
          | (arc <POINT> [real] [real] [real] [real]) |
             #small radius, large radius, start angle, angle span
          | (reg <REGION_LIST>)
          | "STRING"
<REGION_LIST> -> <REGION_LIST> <REGION> | <REGION>
<POINT> -> (pt [real] [real]) #xcoord, ycoord
         | (pt [real] [real] <POINT>)
         | (pt ball)
         | (pt <TEAM> <UNUM>)

#Meta messages
<META_MESS> -> (meta <META_TOKEN_LIST>)
<META_TOKEN_LIST> -> <META_TOKEN_LIST> <META_TOKEN> | <META_TOKEN>
<META_TOKEN> -> (ver [int])

#Define messages
<DEFINE_MESS> -> (define <DEFINE_TOKEN_LIST>)
<DEFINE_TOKEN_LIST> -> <DEFINE_TOKEN_LIST> <DEFINE_TOKEN>
                     | <DEFINE_TOKEN>
<DEFINE_TOKEN> -> <CONDITION_DEFINE>
                | <DIRECTIVE_DEFINE>
                | <REGION_DEFINE>
                | <ACTION_DEFINE>
<CONDITION_DEFINE> -> (definec "[string]" <CONDITION>)
<DIRECTIVE_DEFINE> -> (defined "[string]" <DIRECTIVE>)
<REGION_DEFINE> ->    (definer "[string]" <REGION>)
<ACTION_DEFINE> ->    (definea "[string]" <ACTION>)

#Freeform messages
```

```
<FREEFORM_MESS> -> (freeform "[string]")
```

Parameter name	Used value	Default value	Explanation
coach_port	6001	6001	The port number the trainer connects to.
say_msg_size	512	256	Maximum length of a freeform message a player, trainer, or coach can say.
say_coach_cnt_max	128	128	Upper limit of freeform messages an online coach can say
send_vi_step	100	100	Interval of online coach's look.
clang_win_size	100	100	Number of cycles that lie between online coach messages
clang_advice_win	1	1	Number of advice messages that can be sent in the aforementioned interval.
clang_define_win	1	1	Number of define messages that can be sent in the aforementioned interval.
clang_info_win	1	1	Number of info messages that can be sent in the aforementioned interval.
clang_meta_win	1	1	Number of meta messages that can be sent in the aforementioned interval.
clang_mess_delay	50	50	Number of cycles messages from the online coach will be delayed.
clang_mess_per_cycle	1	1	Number of messages that will be sent to the players during non-play_on modes.

From trainer to server	From server to trainer
(init (version VERSION)) VERSION ::= a real number	trainer: (init ok)
(change_mode PLAY_MODE) PLAY_MODE ::= one of the play-modes	(ok change_mode) (error illegal_mode) (error illegal_command_form)
(move OBJECT X Y [VDIR [DELTA_X DELTA_Y]]) OBJECT ::= One of object names X ::= -52–52 Y ::= -32–32 VDIR ::= -180–180 DELTA_X, DELTA_Y ::= [float]	(ok move) (error illegal_object_form) (error illegal_command_form)
(check_ball)	(ok check_ball TIME BPOS) TIME ::= sim. time of server BPOS ::= in_field | goal_SIDE | out_of_field SIDE ::= l | r
(start) (recover)	(ok start) (ok recover)
(change_player_type TEAM_NAME UNUM PLAYER_TYPE) TEAM_NAME ::= string UNUM ::= 1–11 PLAYER_TYPE ::= 0–6	(warning no_team_found) (error illegal_command_form) (warning no_such_player) (ok change_player_type TEAM UNUM TYPE)
(ear MODE) MODE ::= on | off	(ok ear on) (ok ear off) (error illegal_mode) (error illegal_command_form)

Table 7.1.: Trainer Interactions with the Server

From online coach to server	From server to online coach
(init TEAMNAME (version VERSION)) VERSION ::= a real number TEAMNAME ::= string	(init SIDE ok) SIDE ::= l | r
(change_player_type UNUM PLAYER_TYPE) UNUM ::= 1–11 PLAYER_TYPE ::= 0–6	(warning no_team_found) (error illegal_command_form) (warning no_such_player) (ok change_player_type TEAM UNUM TYPE) (warning cannot_sub_while_playon) (warning no_subs_left) (warning max_of_that_type_on_field) (warning cannot_change_goalie)

Table 7.2.: Online Coach Interactions with the Server

From client to server	From server to client
(say MESSAGE) (see Section 7.4.2)	(ok say) (error illegal_command_form)
(look)	(ok look TIME (OBJ_1 $OBJDESC_1$) (OBJ_2 $OBJDESC_2$) . . .) OBJ_j ::= object name (see Section 4.3 $OBJDESC_j$ ::= X Y | X Y $DELTA_x$ $DELTA_y$ | X Y $DELTA_x$ $DELTA_y$ BODYANG NECKANG
(eye MODE) MODE ::= on | off	(ok eye on) (ok eye off) (error illegal_mode) (error illegal_command_form)
This message is sent automatically every *send_vi_step* milliseconds when the coach/trainer eye is on (see the "eye" commands below).	(see_global TIME (OBJ_1 $OBJDESC_1$) (OBJ_2 $OBJDESC_2$) . . .)
The trainer must use the 'ear' command to get these messages. The online coach always gets these messages.	(hear TIME referee MESSAGE) (hear TIME (p "TEAMNAME" NUM) "MESSAGE") TIME ::= time message was sent TEAMNAME ::= string NUM ::= 1–11 MESSAGE ::= string
(team_names)	(ok team_names [(team l TEAMNAME1) [(team r TEAMNAME2)]])

Table 7.3.: Server Interactions with Trainer/Coach

8. References and Further Reading

8.1. General papers

[1] Minoru Asada and Hiroaki Kitano, editors. *RoboCup-98: Robot Soccer World Cup II.* LNAI 1604. Springer, Berlin, Heidelberg, New York, 1999.

[2] Hans-Dieter Burkhard, Markus Hannebauer, and Jan Wendler. AT Humboldt — Development, Practice and Theory. In Hiroaki Kitano, editor, *RoboCup-97: Robot Soccer World Cup I*, volume 1395 of *Lecture Notes in Computer Science*, pages 357–372. RoboCup Federation, Springer–Verlag, 1997.

[3] Silvia Coradeschi, Tucker Balch, Gerhard Kraetzschmar, and Peter Stone, editors. *Team Descriptions Simulation League RoboCup'99*, Stockholm, Sweden, July 1999.

[4] John F. Kennedy. Urgent National Needs. Congressional Record – House (25 may 1961), 1961.

[5] Hiroaki Kitano, editor. *Proceedings of the IROS-96 Workshop on RoboCup*, Osaka, Japan, November 1996.

[6] Hiroaki Kitano, editor. *RoboCup-97: Robot Soccer World Cup I.* Springer Verlag, Berlin, 1998.

[7] Hiroaki Kitano, Minoru Asada, Yasou Kuniyoshi, Itsuki Noda, and Eiichi Osawa. RoboCup: The Robot World Cup Initiative. In *Proc. of IJCAI-95 Workshop on Entertainment and AI/Alife*, pages 19–24, 1995.

[8] Hiroaki Kitano, Minoru Asada, Yasuo Kuniyoshi, Itsuki Noda, and Eiichi Osawa. RoboCup: The robot world cup initiative. In W. Lewis Johnson and Barbara Hayes-Roth, editors, *Proceedings of the First International Conference on Autonomous Agents (Agents '97)*, pages 340–347, New York, 5–8 1997. ACM Press.

[9] Stefan Lanser, Christoph Zierl, Olaf Munkelt, and Bernd Radig. MORAL - A Vision-based Object Recognition System for Autonomous Mobile Systems. In 7^{th} *International Conference on Computer Analysis of Images and Patterns, Kiel*, pages 33–41. Springer–Verlag, September 1997.

[10] Sean Luke, Charles Hohn, Jonathan Farris, Gary Jackson, and James Hendler. Coevolving Soccer Softbot Team Coordination with Genetic Programming. In Hiroaki

Kitano, editor, *Proceedings of the RoboCup−97 Workshop at the 15th International Joint Conference on Artificial Intelligence (IJCAI97)*, pages 115–118, 1997.

[11] Alan Mackworth. *On Seeing Robots*, chapter 1, pages 1–13. World Scientific Press, 1993.

[12] Itsuki Noda, Shoji Suzuki, Hitoshi Matsubara, Minoru Asada, and Hiroaki Kitano. Overview of RoboCup-97. In Hiroaki Kitano, editor, *RoboCup-97: Robot Soccer World Cup I*, pages 20–41. Springer–Verlag, 1997.

[13] The goals of RoboCup. by RoboCup Federation on `http://www.robocup.org/overview/22.html`, 2000. Verified on 12th February 2001.

[14] Peter Stone, Tucker Balch, and Gerhard Kraetszchmar, editors. *RoboCup-2000: Robot Soccer World Cup IV*, Berlin, 2001. Springer Verlag. To appear.

8.2. Doctoral Theses

[15] Klaus Dorer. *Motivation, Handlungskontrolle und Zielmanagement in autonomen Agenten*. PhD thesis, Albert-Ludwigs-Universität Freiburg, Freiburg, December 1999. (German only).

[16] Johan Kummeneje. RoboCup as a Means to Research, Education, and Dissemination. Ph. Lic. Thesis, March 2001. Department of Computer and Systems Sciences, Stockholm University and the Royal Institute of Technology.

[17] Peter Stone. *Layered Learning in Multi-Agent Systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, December 1998.

8.3. Undergraduate and Master's Theses

[18] Fredrik Heintz. RoboSoc a System for Developing RoboCup Agents for Educational Use. Master's thesis, IDA 00/26, Linköping university, Sweden, March 2000.

[19] Jan Murray. My goal is my castle – Die höheren Fähigkeiten eines RoboCup-Agenten am Beispiel des Torwarts. Studienarbeit, Universität Koblenz-Landau, Germany, March 1999. (German only).

[20] Jan Murray. Soccer Agents Think in UML. Diploma thesis, Universität Koblenz-Landau, 2001.

[21] Oliver Obst. RoboLog: Eine deduktive Schnittstelle zum RoboCup Soccer Server. Diploma thesis, Universität Koblenz-Landau, February 1999. (German only).

8.4. Platforms to start building team upon

8.5. Education-related articles

8.6. Machine Learning

[22] Sebastian Buck and Martin A. Riedmiller. Learning situation dependent success rates of actions in a robocup scenario. In *Pacific Rim International Conference on Artificial Intelligence*, page 809, 2000.

[23] Peter Stone. *Layered Learning in Multiagent Systems: A Winning Approach to Robotic Soccer*. MIT Press, 2000.

8.7. Decision Making

[24] V.S. Subrahmanian, Piero Bonatti, Jürgen Dix, Thomas Eiter, Sarit Kraus, Fatma Ozcan, and Robert Ross. *Heterogeneous Agent Systems.* MIT Press, Cambridge, Massachusetts, 2000.

8.8. Other supporting documents

[25] Laws of the games. by FIFA on `http://www.fifa.com`, 2000. Verified on 12^{th} February 2001.

[26] W.R. Stevens. *UNIX Network Programming.* Prentice Hall, 1990.

8.9. Team Descriptions

8.9.1. 1996

8.9.2. 1997

8.9.3. 1998

[27] Peter Stone, Manuela Veloso, and Patrick Riley. The CMUnited-98 Champion Simulator Team. In Minoru Asada and Hiroaki Kitano, editors, *RoboCup-98: Robot Soccer World Cup II.* RoboCup Federation, Springer–Verlag, 1998.

8.9.4. 1999

[28] Peter Stone, Manuela Veloso, and Patrick Riley. The CMUnited-99 Simulator Team. In Silvia Coradeschi, Tucker Balch, Gerhard Kraetzschmar, and Peter Stone, editors, *Team Descriptions Simulation League RoboCup'99*, pages 7–11. RoboCup Federation, Linköping University Electronic Press, 1999.

8.9.5. 2000

8.9.6. 2001

A. GNU Free Documentation License

Version 1.1, March 2000

Preamble

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

A.1. Applicability and Definitions

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the

Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A.2. Verbatim Copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section

3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

A.3. Copying in Quantity

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

A.4. Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

- List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
- State on the Title page the name of the publisher of the Modified Version, as the publisher.
- Preserve all the copyright notices of the Document.
- Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- Include an unaltered copy of this License.
- Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.
- Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties – for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

A.5. Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

A.6. Collections of Documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

A.7. Aggregation With Independent Works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

A.8. Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

A.9. Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this

License will not have their licenses terminated so long as such parties remain in full compliance.

A.10. Future Revisions of This License

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Index