Soccerserver Manual

Ver. 5 Rev. 00 beta

(for Soccerserver Ver.5.00 and later)

Emiel Corten,Klaus Dorer
Fredrik Heintz, Kostas Kostiadis,
Johan Kummeneje, Helmut Myritz,
Itsuki Noda, Jukka Riekki,
Patrick Riley, Peter Stone,
Tralvex Yeap

mielko@wins.uva.nl klaus@cognition.iig.uni-freiburg.de
frehe@ida.liu.se kkosti@essex.ac.uk
johank@dsv.su.se myritz@informatik.hu-berlin.de
noda@etl.go.jp jpr@ees2.oulu.fi
priley+@andrew.cmu.edu Peter_Stone@zico.prodigy.cs.cmu.edu
tralvex@krdl.org.sg

May 28, 1999

Contents

Chapter 1

Introduction

Soccerserver is a system that enables autonomous agents consisting of programs written in various 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`. `Soccerserver` 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[1][2]. 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 A.2.) One of purposes of soccerserver is evaluation of multi-agent systems, in which efficiency of communication between agents is one of the criteria. Users must realise control of multiple clients by such restricted communication.

1.1 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.

1.1.1 History of the Soccerserver

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 have 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 LISP system. The server shows the field on X window system, but each player was shown in an alphabet character. It used 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-expression 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.

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

[2]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.

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 3 was announced in February of 1997. Simon Ch'ng at RMIT joined to 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 regulation are discussed on the mailing list organized by Gal Kaminka. As a result, many contributer joined to the development. The 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

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

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

1.1.2 History of the RoboCup Simulation League

The RoboCup simulation league has had three main official events: preRoboCup-96, RoboCup-97, and RoboCup-98. 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 [2]. In conjunction with the IROS-96 conference, Pre-RoboCup-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 [3]. 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 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:

1.1.3 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 organise and as a result the Soccerserver Manual version 4.0 was released on the 1st of November 1998.

Since November, the soccerserver has changed version to 5.0 (currently 5.14) and is continously developed. Therefore the Soccer Manual Effort has developed a new version, which you are currently reading.

1.2 About This Manual

This manual is the joint effort of the authors from a diverse range of universities, which build upon the original work of *Itsuki Noda*.
If there are errors, inconsistencies, or oddities, please notify johank@dsv.su.se 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.

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

Acknowledgements

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, Fredrik Heintz at Linköping University. We are alsovery 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.
- *Tomoichi Takahashi* at Chubu University, Japan.

We would not have been able to do this manual without the above mentioned people[3].

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

Chapter 2

User's Manual

2.1 Getting Started

2.1.1 Installation

1. Get source files from the soccerserver site:

   `http://ci.etl.go.jp/~noda/soccer/server/index.html`,

   (a) click Download,

   (b) get gziped file `sserver-*.tar.gz` by clicking

   - Newest version, or
   - FTP service of the Soccerserver is here

   where `*` is the version number [1].

2. Extract source files:

   ```
   % gzip -dc sserver-*.tar.gz | tar xvf -
   ```

   the following files and directories are created under the directory `sserver-*`.

   ```
   % ls sserver-5.12
   Acknowledgement     Changes             Changes~
   configure*          configure2*         drawcheck/
   logplayer/          Makefile            monitor/
   prints.txt          README              recfile_change/
   sampleclient/       server/             sserver*
   sserver-csh.tmpl    sserver-tcsh.tmpl   sserver.tmpl
   %
   ```

3. Change working directory to `sserver-*` and read the file `README`.

   ```
   % cd sserver-5.12
   % 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
   ```

[1] Version `5.12` is the newest one at April '99. The example list is output for version `5.12`.

```
        (2) Do make

[How to Start]
        (1) start \ss{}.
        (2) start a couple of soccermonitors you want.

        "sserver" is a sample script to invoke \ss{} 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
%
```

4. Do "`configure`" according to `README` and answer questions about directories.

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

Configuration Summary:
  OS type = SunOS_5
  X11 revision = 6
  X11 include PATH = /usr/openwin/include
  X11 libraries PATH = /usr/openwin/lib
  RUNPATH = -R/local/lib:/usr/openwin/lib
  Link style = Dynamic
  Compiler flag(s) = -O2 -pipe

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

5. Do "`make`".

```
% make
g++ -c -pipe -DSolaris -DSYSV  main.C
g++ -c -pipe -DSolaris -DSYSV  field.C
  .
  .
  .
g++ -o drawcheck drawcheck.o netif.o  -lm -lsocket -lnsl -R/local/lib
%
```

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

2.1.2 How to Start Soccerserver

1. Start `sserver`. `sserver` is shell script made by `configure`.

   ```
   % sserver
   Soccer Server Ver. 5.12
   Copyright (C) 1995, 1996, 1997, 1998  Electrotechnical Laboratory.
   Itsuki Noda, Yasuo Kuniyoshi and Hitoshi Matsubara.
   wind factor: rand: 0.000000, vector: (0.000000, 0.000000)
   ```

   A window of `soccermonitor` appears on CRT.

2. When all clients are ready, click the `kick-off` button on a window of `soccermonitor`. If you have not yet programmed your own client, you can use the sample clients.

   - using a sample included in the soccerserver software, you can get used to protocols between clients and the soccerserver:

     ```
     % cd sampleclient/
     % ls
     client*          client.c         client.h         client.o
     Makefile         Makefile.tmpl    Makefile.tmpl2
     tk32 5% client     <---  connect Server with default parameters -
                              hocalhost 6000. You can change them as argument.
     (init "Myteam")    <---  This program sends 'stdin' data to Server.
     send 6000 : (init "Myteam")
     recv 33097 : (init l 1 before_kick_off)
                        <---  display data received from Server.
                              It says Left side, Uniform Number = 1 & PlayMode.
     recv 33097 : (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))
             :          <---  Continuosly displaying the recieving data,
                              you can key in (move 0 0), for example.
     recv 33097 : (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 notice messages appears in left
                             lower window of soccermonitor.
     ```

   - The soccerserversite contains several sample clients. These programs have been contributed from RoboCuppers and have not been updated to the newest soccerserverversion. So, you need to check whether you are running the corresponding soccerserverversion. `Changes` file in distribution shows the history of soccerserver.

2.1.3 How to Stop Soccerserver

1. Stop all clients.
2. Stop all `soccermonitor`s by clicking the `quit` buttons.
3. Hit CNTL-C to terminate `soccerserver`, or `kill soccerserver`.

2.1.4 sserver Script

The script file "`sserver`" included in the distribution makes it simple to start and stop soccerserver. This script does the following things:

1. Setup environment variables.
2. Fork "`soccerserver`" with standard configuration file.
3. Start"`soccermonitor`" with standard configuration file, and wait until it ends.

4. Kill "`soccerserver`".

This script uses "`sh`". You may modify the path of `sh` in the 1st line of the script and the parameters for `soccerserver` and `soccermonitor`.

2.1.5 Supported Platforms, Required Facilities and More Information

Soccerserversupports the following platforms [2]

- DEC OSF/1
- SunOS 4.1.x and Solaris 2.x

Soccerserver requires the following facilities.

- gcc versions 2.7.x or later
- libg++ version 2.7.x or later
- X11R5 or R6

Soccerserver is freely distributed from the following access point.

- `http://ci.etl.go.jp/~noda/soccer/server/index.html`

For further information, please refer to the official RoboCup pages at

- `http://www.robocup.org`

Or, you can join the mailing list "robocup-sim-l@usc.edu" by sending a message to "listproc@usc.edu" with the BODY containing the following listproc command: "sub robocup-sim-l [your name]". All messages sent to "robocup-sim-l@usc.edu" will automatically be distributed to all subscribers. Any comments and suggestions are welcome to be sent to this list. You must be a subscriber to post a message to the mailing list.

2.2 Whole Process of a Match

A match organized by Soccerserver is carried out in the following steps:

1. Each client of each team connects with the server by an `init` command.
2. When all clients are ready to play, the match referee (a person who invokes the server) starts the match by pressing the *kick-off* button of the server window. Then the first half starts.
3. The first half is 5 minutes[3]. When the first half finishes, the server suspends the match.
4. The half-time is 5 minutes. During the half-time, competitors can change client programs.
5. Before the second half, each client re-connects with the server by a `reconnect` command.
6. When all clients are ready, the referee starts the second half by pressing the *kick-off* button. Then the second half starts.
7. The second half is also 5 minutes. After the second half, the server stops the match.
8. If the match is a draw, an extra half starts. The extra half ends immediately when a team gets a new goal (Victory-goal style or sudden-death).

2.3 Parameters

In the Soccerserver System there are a number of parameters for the Soccerserver (soccerserver) and Soccermonitor (soccermonitor).

[2]Supported Platforms may change, please check `README`. Soccerserver should also run on Pentium with Linux.
[3]The length of a half may be changed.

2.3.1 Parameters of `Soccerserver`

`Soccerserver` has the following modifiable parameters:

Parameter Name	Used Value	Explanation
goal_width	14.02	Width of the goal. Default value: 7.32
player_size	0.3	Radius of a player. Default value: 1.0
player_decay	0.4	Decay rate of speed of a player. If this is 1.0 a player keeps its speed, and if this is 0.0, a player loses its speed in one simulation cycle. Default value: 0.5
player_rand	0.1	Amount of noise added in player's movements and turns. Default value: 0.1
player_weight	60.0	Weight of a player. This parameter concerns the wind factor. Default value: 60.0
player_speed_max	1.0	Maximum speed of a player during one simulation cycle (i.e. the player can achieve a maximum speed of 10 m/sec) Default value: 32.0
stamina_max	3500	Maximum stamina of a player. Default value: 2500
stamina_inc_max	20.0	Amount of stamina that a player gains in a simulation cycle. Default value: 20.0
recover_dec_thr	0.3	Decrement treshold for player's recovery. Default value: 0.3
recover_dec	0.002	Decrement step for player's recovery. Default value: 0.05
recover_min	0.5	Minimum player recovery. Default value: 0.1
effort_dec_thr	0.3	Decrement treshold for player's effort capacity. Default value: 0.4
effort_dec	0.05	Decrement step for player's effort capacity. Default value: 0.05
effort_inc_thr	0.6	Increment treshold for player's effort capacity. Default value: 0.9
effort_inc	0.1	Increment step for player's effort capacity. Default value: 0.05
effort_min	0.6	Minimum value for player's effort capacity. Default value: 0.1
hear_max	2	Maximum hearing capacity of a player. A player can listen N(=**hear_inc**) messages and M(=**hear_decay**) simulation cycles. Default value: 2
hear_inc	1	Minimum hearing capacity of a player. Default value: 1
hear_decay	2	Decay of hearing capacity of a player. Default value: 1
inertia_moment	5.0	Inertia moments of a player. It affects it's moves. Default value: 5.0
sense_body_step	100	Interval of sense_body informations. Default value: 100
catchable_area_l	2.0	Goalie catchable area length. Default value: 2.0
catchable_area_w	1.0	Goalie catchable area width. Default value: 1.0
catch_probability	1.0	The probabilty for a goalie to catch the ball (if it is not during the catch ban interval). Default value: 1.0

Parameter Name	Used Value	Explanation
catch_ban_cycle	5	The number of cycles for a goalie is banned from catching the ball after a successful catch. Default value: 5
ball_size	0.085	Radius of the ball. Default value: 0.15
ball_decay	0.94	Decay rate of the ball. Default value: 0.8
ball_rand	0.05	Amount of noise added in the movements of the ball. Default value: 0.2
ball_weight	0.2	Weight of the ball. This parameter concerns the wind factor. Default value: 0.2
ball_speed_max	2.7	Maximum speed of the ball during one simulation cycle (i.e. the ball can achieve a maximum speed of 27 m/sec) Default value: 32.0
wind_force	0.0	The amount of force of wind. Default value: 10.0
wind_dir	0.0	The direction of wind. Default value: 0.0
wind_rand	0.0	The amount of change of direction of wind. Default value: 0.3
kickable_margin	0.7	The area within which the ball is kickable is: kickable_area = kickable_margin + ball_size + player_size Default value: 1.0
ckick_margin	1.0	Corner kick margin. Default value: 1.0
dash_power_rate	0.006	Rate of which *Power* argument in **dash** command is multiplied. Default value: 0.1
kick_power_rate	0.016	Rate of which *Power* argument in **kick** command is multiplied. Default value: 0.1
visible_angle	90.0	Angle of view cone of a player in the standard view mode Default value: 90.0
audio_cut_dist	50.0	Maximum distance a message said by a player can reach. Default value: 50.0
quantize_step	0.1	The step of quantization of distance for moving objects (players and ball). Default value: 0.1
quantize_step_l	0.01	The step of quantization of distance for landmarks (flags, lines and goals). Default value: 0.01
maxpower	100	Maximum value of *Power* in **dash** and **kick** commands. Default value: 100
minpower	-100	Minimum value of *Power* in **dash** and **kick** commands. Default value: -30
maxmoment	180	Maximum value of *Moment* and *Direction* in **turn** and **kick** commands. Default value: 180
minmoment	-180	Minimum value of *Moment* and *Direction* in **turn** and **kick** commands. Default value: -180
port	6000	The port number of UDP/IP that connects with playerclients and `soccermonitor`. Default value: 6000
coach_port	6001	The port number of UDP/IP that connects with coachclients. Default value: 6001
olcoach_port	6002	The port number of UDP/IP that connects with onlinecoachclients. Default value: 6002
say_coach_cnt_max	128	Upper limit of the number of online coach's messages. Default value: 128
say_coach_msg_size	128	Upper limit of length of a online coach's message. Default value: 128
send_vi_step	100	Interval of online coach's look. Default value: 100

Parameter Name	Used Value	Explanation
simulator_step	10	Length of period of simulation cycle. Unit is milli-second. Default value: 100
send_step	150	Length of the interval for sending visual information to a player in the standard view mode. Unit is milli-second. Default value: 150
recv_step	10	Length of period of server's polling sockets to clients. Unit is milli-second. Default value: 20
half_time	300	The length of a half time of a match. Unit is a simulation cycle. The actual length of a match is 2 x half_time x simulator_step [ms]. Negative values mean infinite length. Default value: 6000
say_msg_size	512	Maximum length of a message a player can say. Default value: 256
use_offside	on	Flag for using offside rule [on/off]. Default value: on
offside_active_area_size	9.15	The circular area having the ball oin its center within which the player is considered to be offside. Default value: 9.15
forbid_kick_off_offside	on	Flag which indicates whether kick is forbidden [on] or allowed [off] from an offside position. Default value: on
verbose	off	Flag indicating if supplemental text information concerning errors should be supplied by `soccerserver`. Default value: off
record_version	2	Indicates the type of record log to be created if `record_log` is on. 2 means the new compressed version, 1 the old one. Default value: 2
record_log	off	Flag indicating whether a record of the match is needed [on/off]. Default value: off
send_log	on	Flag for send client command log [on/off]. Default value: on
max_neck_ang	90	Maximum value of *Moment* and *Direction* in **turn_neck** command. Default value: 90
min_neck_ang	-90	Minimum value of *Moment* and *Direction* in **turn_neck** command. Default value: 90
visible_distance	3.0	Maximum distance within which the player can see object out of view cone. Default value: 3.0
wind_none	false	Option that switches the server encounts wind factor in object movements. Default value: false
wind_random	false	Option that switches the server set wind factor randomly. Default value: false
coach	false	Option that switches the server run as (off-line) Coach Mode. Default value: false

2.3.2 Commandline Options of `Soccerserver`

A user can modify parameters described in Section 2.3.3. in the command line as follows.

% `soccerserver` [-*ParameterName Value*]*

`Soccerserver` also accept following options.

-f *ConfFileName*

See Section 2.5.1.

-coach_w_referee

See Section 2.5.2.

2.3.3 Configuration File of `Soccerserver`

A user can also modify the parameters by specifying them in a configuration file usually named `server.conf` as follows.

% `soccerserver` -f *ConfFileName*

In the configuration file, each line consists a pair of name and value of a parameter as follows.

ParameterName : Value

In the files, lines that start '#' are ignored as commands.

With the following two lines it is possible to log a game (saved in game.rec) and the sended commands from all players (game.log).

log_file : game.log
record : game.rec

2.3.4 Parameters of `Soccermonitor`

`Soccermonitor` has the following modifiable parameters.

Parameter Name	Used Value	Explanation
host	localhost	The host name on which `soccerserver` is running. Default value: localhost
port	6000	The port number of UDP/IP of `soccerserver`. Default value: 6000
length_magnify	4.0	Rate of magnification of size of the field. Default value: 6.0
goal_width	14.02	Width of the goal. Default value: 7.32
print_log	on	Flag for display log of communication [on/off]. Default value: on
log_line	6	The size of log window. Default value: 6
print_mark	on	Flag for display marks on the field [on/off]. Default value: on
mark_file_name	mark.xbm	File name to use for displaying the marks on the field. Default value: mark.xbm

Parameter Name	Used Value	Explanation
ball_file_name	ball-s.xbm	File name to use for displaying the ball. Default value: ball.xbm
player_widget_size	12.0	Font to use for displaying the uniform number. Default value: 1.0
player_widget_font	5x8	Size of a player widget. Default value: fixed
uniform_num_pos_x	2	Position (X) of player's uniform number. Default value: 2
uniform_num_pos_y	8	Position (Y) of player's uniform number. Default value: 8
team_l_color	Gold	Left side team color when game starts. Default value: Gold
team_r_color	Red	Right side team color when game starts. Default value: Red
goalie_l_color	Green	Left side goalie color when game starts. Default value: Green
goalie_r_color	Purple	Right side goalie color when game starts. Default value: Purple
status_font	7x14bold	Font used for status line displays [team name and score, time, play_mode]. Default value: fixed
popup_msg	on	Flag for pop up and down "GOAL!" and "Offside!" [on/off]. Default value: off
goal_label_width	120	Goal message width for pop and down "GOAL!!". Default value: 120
goal_label_font	-adobe-times-bold-r-*-*-34-*-*-*-*-*-*-*	Goal message font for pop and down "GOAL!!". Default value: fixed
goal_score_width	40	Score message width for pop and down "GOAL!!". Default value: 40
goal_score_font	-adobe-times-bold-r-*-*-25-*-*-*-*-*-*-*	Score message font for pop and down "GOAL!!". Default value: fixed
offside_label_width	120	Offside message width for pop and down "GOAL!!". Default value: 120
offside_label_font	-adobe-times-bold-r-*-*-34-*-*-*-*-*-*-*	Offside message font for pop and down "GOAL!!". Default value: fixed
eval	off	Flag for evaluation mode [on/off]. Used to allow discarding (disabling) of players for team performance evaluation when handicaped. Default value: off
redraw_player	off	Flag for redrawing player widgets every cycle. The option is needed to avoid a bug of XFree86 server for SVGA and it makes `soccermonitor` heavy. So do not use this option when `soccermonitor` works well without it. Default value: off

2.3.5 Commandline Options of `Soccermonitor`

A user can modify parameters described in Section 1.4 in the command line as follows.

% `soccermonitor` [-*ParameterName Value*]*

2.3.6 Configuration File of `Soccermonitor`

A user can also modify the parameters by specifying them in a configuration file usually named `monitor.conf` as follows.

% `soccermonitor` -f *ConfFileName*

In the configuration file, each line consists a pair of name and value of a parameter as follows.

ParameterName : Value

In the files, lines that start '#' are ignored as commands.

2.4 Rules

This section describes the rule judged by the server and by humans.

2.4.1 Rules Judged by the Server

The soccerserver controls the match according to the following rules.

Goal

When a team scores, the referee announces the goal (broadcasts a message to all clients), updates the score, suspends the match for 5 seconds, moves the ball to the centre mark, and changes the play-mode to *kick_off*. While the referee suspends the match (5 seconds), clients must return to their own side. During this interval, clients can use `move` commands. If a player remains in the opponent side after this period expires, the referee moves the player to a random position within their own side (see 'Kick-off rule below.)

Kick-off

Just before a kick off (either before the game starts, or after every goal), all players must be in their own side. If players are in the opponent side, the referee moves them into their own side to a randomly chosen position. There is a period of 5 seconds for players to go back to their own side after a goal. During this 5-second interval, clients can use the `move` command to move each player to a certain position immediately. After the 5-second period expires, the server moves players located in the opponent side to a random position within their own side.

Out of Field

When the ball is 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.

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 with radius 9.15 meters. The removed players are then 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 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 of 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 (sudden death).

2.4.2 Rules Judged by Human

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 1998 competition.

1. Surrounding the ball
2. Blocking the goal with too many players
3. Not putting the ball into play
4. Intentionally blocking the movement of other players
5. Abusing the goalie catch command (the goalie may not repeatedly kick and catch the ball to move the ball in the penalty area).

Flooding the server with messages

A client 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.

2.5 The Coach-client

2.5.1 Introduction

As in human soccer, a match is played without interruptions from outside the field. So nobody (and no application) except the players and the referee can influence and/or control the match. However, it would be very useful if you could have more control over the game while developing player-clients. For instance, the possibility to perform training sessions in which a certain action, such as dribbling, is tested in an automated way gives you the opportunity to apply machine learning methods.

Therefore, a privileged client called the coach-client has been introduced. The coach-client 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 meaning (and interpretation) is user-defined.
- It can move an object (any player or the ball) to any location on the field irrespective of the current play-mode.
- It can get information about positions of all the (movable) objects on the field.

2.5.2 Coaching With or Without the Soccerserver Referee

By default, an internal referee module is active within the soccerserver that controls the match (see section 2.4.1). If the coach is to have complete control over the match you must ask the soccerserver to deactivate the referee module. In this mode it is the responsibility of the coach-client to regularly check the state of the game and change the play-mode according to its (own) rules.

It is also possible for both the internal referee module and the coach-client to have control over the match. In this mode the coach-client can, for instance, be used to control a training session or to give a player information about its current performance so that it can adjust its behavior.

section 2.5.5 explains how to activate the soccerserver with the coach mode.

2.5.3 Coaching in a Real Match

Until 1998 it was not allowed to connect a coach-client during a real match. It was used only to develop and train the player-clients.
From 1998 on a coach-client will probably also be allowed to participate in a real match. The idea is that such a coach can observe and analyze the game and can give strategic information to its players.
The capabilities the coach can use during a real match will be limited to:

- getting information about the positions of the players and the ball.
- broadcasting and receiving audio messages.

As long as the coach can not guide its players step by step in their actions, allowing a coach at the match does not necessarily violate the principle that players must be autonomous. So the limitations on the capabilities of the coach should reflect the previous remark.

2.5.4 Coach Commands

The items labelled with the symbol ∘ will probably not be allowed during a real match.

∘ `(change_mode` *PLAY_MODE*`)`
Change the play-mode to *PLAY_MODE*. *PLAY_MODE* must match one of the modes defined in section A.7.
Note that for most play-mode requests the soccerserver will indeed only change the play-mode. The position of the ball usually remains unchanged.

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*`])`
This command will move *OBJECT*, which may be a player or the ball, to absolute position (X,Y) and will, if *VDir* is specified, change its absolute direction to *VDir*. See Table 3.2 for information on how to compose the name of a player or the ball.

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 specification was not valid.

∘ `(check_ball)`
Ask the soccerserver to check the position of the ball. Four states 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.

- `(look)`
  Get information about the positions of the following objects on the field:
  - The left and right goals.
  - The ball.
  - All active players.

  Possible replies by the soccerserver:

  `(ok look` *TIME* `(`*OBJ1 X1 Y1*`) (`*OBJ2 X2 Y2*`)...)`
  OBJi can be any of the objects mentioned above. See Table 3.2 for information about the way the names for those objects are composed.

- `(say` *MESSAGE*`)`
  Broadcast the message *MESSAGE* to all clients. The format for *MESSAGE* is the same as for a player-client.

  Possible replies by the soccerserver:

  `(ok say)`
  The command succeeded.

  `(error illegal_command_form)`
  MESSAGE did not match the required format. It must be a string whose length is less than 512 and it must consist of alphanumeric characters and/or the symbols `().+*/?<>_` .

- `(ear` *MODE*`)`
  Turn on or off the sending of auditory information to the coach. *MODE* must be one of `on` and `off`. If `(ear on)` is sent, the server sends all auditory information to the coach-client in the same manner as to player-clients. If `(ear off)` is sent, the server stops sending auditory information to the coach-client.

  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.

2.5.5 Initializing the Soccerserver for Coach Mode

The soccerserver must be informed at start-up time that a coach-client will be used. Add the option '`-coach`' to the command line arguments of the soccerserver application when a coach-client is used and the internal referee module of the server must be deactivated. Use the option '`-coach_w_referee` if the internal referee module should remain active.

If the server is invoked with one of the coach modes, it prepares a UDP socket to which the coach-client can connect. The default portnumber is 6001. If a different portnumber is needed you can set the new port by assigning its value to the `coach_port` parameter (see section 2.3.1).

The messages the server receives through a connection made to this port will be interpreted as commands from a coach-client as described in section 2.5.4. Besides, the connection is used to transmit the information intended for the coach to the coach-client.

Chapter 3

Inside of Soccerserver

3.1 Field and Objects

3.1.1 Field

The soccer field and all objects on it are 2-dimensional. There is no notion of height on any object. The size of the field is 105m $\times$ 68m [1].The width of the goals is 14.64m, which is twice the size of ordinary goals. Experimentally, it was too difficult to score goals with the regular size.

3.1.2 Players and Ball

The players and the ball are treated as circles. All distances and angles used and reported are to the centers of the circles. The action model is discrete (i.e. all moves within one time step occur simultaneously at the end of the step). The length of time for a step is set by the `simulator_step` parameter. At the end of the step, the server takes all action commands received and applies them to the objects on the field, using the current position and velocity information to calculate a new position and velocity for each object (as described in section 3.1.3)

3.1.3 Movements of Objects

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

$$\begin{aligned}
(u_x^{t+1}, u_y^{t+1}) &= (v_x^t, v_y^t) + (a_x^t, a_y^t)\text{: accelerate} \\
(p_x^{t+1}, p_y^{t+1}) &= (p_x^t, p_y^t) + (u_x^{t+1}, u_y^{t+1})\text{: move} \\
(v_x^{t+1}, v_y^{t+1}) &= \text{decay} \times (u_x^{t+1}, u_y^{t+1})\text{: decay speed} \\
(a_x^{t+1}, a_y^{t+1}) &= (0, 0)\text{: reset acceleration}
\end{aligned} \tag{3.1}$$

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 Table 3.1. 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.

[1]The unit is meaningless, because parameters used in the simulation do not be decided based on the physical parameters, but be tuned in order to make matches be meaningful as evaluation of multi-agent systems.

Collisions

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.

Noise and Winds

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

Concerned with movements, noise is added into Eq. 3.1 as follows:

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

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

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

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:

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

3.1.4 Player's Action

Sense Body

In previous versions of the server, there was a `sense_body` command which returned some information about the state of the player. This command no longer exists in versions greater than 5.00. Instead, the following information is sent automatically to every client every `sense_body_step` milliseconds.

```
(sense_body TIME
            (view_mode QUALITY WIDTH)
            (stamina STAMINA EFFORT)
            (speed AMOUNT_OF_SPEED)
            (head_angle RELATIVE_HEAD_ANGLE)
            (kick KICK_COUNT)
            (dash DASH_COUNT)
            (turn TURN_COUNT)
            (say SAY_COUNT)
            (turn_neck TURN_NECK_COUNT))
```

The Basic Actions

A list of all commands is given in Table 3.1 The basic actions require a little more description and are addressed below.

- `turn` The moment argument can be between `minmoment` and `maxmoment` (-180 degrees and 180 degrees by default). 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:

$$\mathrm{actual\_angle} = \mathrm{moment}/(1.0 + \mathrm{inertia\_moment} \times \mathrm{player\_speed}) \tag{3.2}$$

  `inertia_moment` is a parameter with default value 5.0. Therefore (with default values), when the player is at max speed (1.0), the maximum effective turn he can do is ± 30. However, notice that because you can not dash and turn in the same cycle, the fastest that a player can be going when executing a `turn` is `player_speed_max` $\times$ `player_decay`, which means the effective turn (with default values) is ± 120.

- `turn_neck` Each client has a neck which can be turned somewhat independently of its body. The angle of the player's head is the viewing angle of the player. The `turn` command changes the angle of the player's body while `turn_neck` changes the angle of the player's head relative to its body. The maximum relative angle for the player's neck is 90 degrees to either side. Remember that the neck angle is relative to the

Table 3.1: Control Command

`(turn` *Moment*`)` Change the direction of the player according to *Moment*. *Moment* should be between `minmoment` and `maxmoment` (default is $[-180, 180]$. The actual change of direction is reduced when the player is moving quickly. See Section 3.1.4 for more info.
`(turn-neck` *Angle*`)` Adds *Angle* to the clients neck angle (ie angle at which the view cone extends from the player) The neck angle must be between $[-90, 90]$. The neck angle is always relative to the angle of the player, so if a `turn` command is issued, the view angle also changes. See Section 3.1.4 for more info.
`(dash` *Power*`)` Increases the velocity of the player in the direction it is facing by *Power* $\times$ `dash_power_rate`. *Power* should be between `minpower` and `maxpower` (default: $[-30, 100]$). If power is negative, then the player is effectively dashing backwards. See Section 3.1.4 for more info.
`(kick` *Power Direction*`)` Kick the ball with *Power* in *Direction* if the ball is near enough (the distance to the ball is less than `kickable_margin` + `ball_size` + `player_size`.). *Power* should be between `minpower` and `maxpower` (default is $[-30, 100)$. *Direction* should be between `minmoment` and `maxmoment` (default is $[-180, 180]$). See Section 3.1.4 for more info.
`(move` *X Y*`)` Move the player to the position (X, Y). The origin is the center mark, and the X-axis and Y-axis are toward the opponent's goal and the right touchline respectively. Thus, *X* is usually negative to locate a player in its own side of the field. This command is available only in the `before_kick_off` mode, and for the goalie immediately after catching the ball (see the `catch`) command.
`(catch` *Direction*`)` Tries to catch the ball in direction *Direction*. *Direction* should be in $[-180, 180]$. This command is permitted only for goalie clients. The player (goalie) can catch the ball when the ball is in the rectangle with width is `goalie_catchable_area_w` (default=1), length is `goalie_catchable_area_l` (default=2) and the direction is *Direction*. The probability the catch is successful if it is in the rectangle is given by the parameter `catch_probability`. Also note that the goalie can only do 1 catch every few cycles (specified by the `catch_ban_cycle` parameter). If the goalie tries to catch again during the ban cycle, the command is ignored. If the catch is successful, the server goes into free kick mode. Once it has caught the ball, the goalie can move within the penalty box with the `move` command. The ball moves with the agent. However, a catch does not immediatly change the position or facing direction of the goalie.
`(say` *Message*`)` Broadcast *Message* to all players. *Message* is informed immediately to clients as sensor information in the `(hear ...)` format described below. *Message* must be a string with length less than 512 characters, and consists of alphanumeric characters and the symbols "`+-*/_.()` ". There is a maximum distance that messages can be heard. See the section on auditory information for specifics.
`(change_view` *ANGLE_WIDTH QUALITY*`)` Change angle of view cone and quality of visual information. *ANGLE_WIDTH* must be one of `wide` (=180 degrees), `normal` (= 90 degrees) and `narrow` (=45 degrees). *QUALITY* must be one of `high` and `low`. In the case of `high` quality, the server begins to send detailed information about positions of objects to the client. In the case of `low` quality, the server begins to send reduced information about positions (only directions, no distance) of objects to the client. Default values of angle and quality are `normal` and `high` respectively. On the other hand, the frequency of visual information sent by the server changes according to the angle and the quality: In the case that the angle is normal and the quality is high, the information is sent every 150 milli-sec. (The interval is modifiable by specifying `send_step` in the parameter file.) When the angle changes to wide the frequency is halved, and when the angle changes to narrow the frequency is doubled. When the quality is low, the frequency is doubled. For example, when the angle is narrow and the quality is low, the information is sent every 37.5 milli-sec.

player's body so if the client issues a `turn` command, the viewing angle changes even if no `turn_neck` command is issued.

Also, `turn_neck` commands can be executed in 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 $[-180, 180]$ since the resulting neck angle must be in $[-90, 90]$.

- `dash` The `dash` is essentially a small push in the direction that the player is facing. It is *not* a sustained run. In order to have a sustained run, multiple `dash` commands must be sent. The power passed to the dash command is multiplied by `dash_power_rate` (default .01) and the effort (see Section 3.1.5) and applied in the direction that the player is facing. `dash` is a small acceleration and does *not* set the players velocity. With negative power, the agent dashes backwards, but it consumes twice the stamina (see Section 3.1.5).

- `kick` The `kick` is very similar to the dash except that it accelerates the ball instead of the player. If the player tries to kick when the ball is further than the `kickable_area` (which is equal to the `player_size` +`ball_size` +`kickable_margin`), there is no effect. The one important difference between dashes and kicks is how the kick power rate is figured. Let `dir_diff` be the absolute value of the angle of the ball relative to the direction the player is facing (if the ball is directly ahead, this would be 0). Let `dist_ball` be the distance from the center of the player to the ball. Then the kick power rate is figured as follows:

  $$\texttt{kick\_power\_rate} \times (1 - .25 \times \text{dir\_diff}/180 - .25 \times (\text{dist\_ball} - \texttt{player\_size} - \texttt{ball\_size})/\texttt{kickable\_area})$$

  Basically, this means that the most powerful kick can be done when the ball is directly in front of the player and very close to him, and drops off as both distance and angle increase.

3.1.5 Stamina

Each player has its own stamina. There are three relevant parameters:

- **stamina**: used up when dashing and replenished slightly each cycle
- **effort**: determines how effective dashing is
- **recovery**: controls how much stamina is recovered each cycle

The player can `dash` only with *Power* lower than the current stamina. The stamina decreases by the *Power*. The details are as follows: If the client dashes:

$$\begin{aligned} \mathit{Stamina} &:= \max(\mathit{Stamina} - \mathit{Power}, 0) \\ \mathit{Effective\_Power} &:= \mathit{effort} \times \min(\mathit{Power}, \mathit{Stamina}) \end{aligned}$$

The dash that is actually executed uses the *Effective_Power*.

In the equations given above for decreasing *stamina*, no mention was given for the sign of *Power*. A negative power can be given to the `dash` command, causing the agent to run backwards. The difference is that using a negative power causes twice the stamina to be consumed.

Every cycle (whether the client dashes or not), the three parameters can be affected.

- **Stamina** Stamina increases slightly every cycle. As *recovery* decreases, less stamina is recovered.

  $$\mathit{stamina} = \min(\texttt{stamina\_max}, \mathit{stamina} + \mathit{recovery} \times \texttt{stamina\_inc\_max})$$

- **Effort** The basic idea is that if *stamina* gets low, *effort* decreases (with a minimum value given by `effort_min`) and if *stamina* gets high enough, then *effort* increases with a maximum of 1.0. Specifically:

  $$\mathit{effort} = \begin{cases} \max(\texttt{effort\_min}, \mathit{effort} - \texttt{effort\_dec}) & \text{if } \mathit{stamina} \leq \texttt{effort\_dec\_thr} \times \texttt{stamina\_max} \\ \min(1.0, \mathit{effort} + \texttt{effort\_inc}) & \text{if } \mathit{stamina} \geq \texttt{effort\_int\_thr} \times \texttt{stamina\_max} \\ \mathit{effort} & \text{otherwise} \end{cases}$$

- **Recovery** This is similar to effort except that *recovery* never increases.

  $$\mathit{recovery} = \begin{cases} \max(\texttt{recovery\_min}, \mathit{recovery} - \texttt{recovery\_dec}) & \text{if } \mathit{stamina} \leq \texttt{recovery\_dec\_thr} \times \texttt{stamina\_max} \\ \mathit{recovery} & \text{otherwise} \end{cases}$$

One other point is the order which the different values are changed. First the *stamina* value is decreased (if there is a dash). Next, the *recovery* value is changed, then the *effort*. Finally, *stamina* is recovered.

3.2 Sensor Information

3.2.1 Visual Information

Format

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

(*ObjName Distance Direction DistChng DirChng BodyDir HeadDir*)
ObjName ::= (player *Teamname UniformNumber*)
| (goal [l|r])
| (ball)
| (flag c)
| (flag [l|c|r] [t|b])
| (flag p [l|r] [t|c|b])
| (flag g [l|r] [t|b])
| (flag [l|r|t|b] 0)
| (flag [t|b] [l|r] [10|20|30|40|50])
| (flag [l|r] [t|b] [10|20|30])
| (line [l|r|t|b])

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

$$\begin{aligned}
p_{rx} &= p_{xt} - p_{xo} & (3.3)\\
p_{ry} &= p_{yt} - p_{yo} & (3.4)\\
v_{rx} &= v_{xt} - v_{xo} & (3.5)\\
v_{ry} &= v_{yt} - v_{yo} & (3.6)\\
\mathit{Distance} &= \sqrt{p_{rx}^2 + p_{ry}^2} & (3.7)\\
\mathit{Direction} &= \arctan(p_{ry}/p_{rx}) - a_o & (3.8)\\
e_{rx} &= p_{rx}/\mathit{Distance} & (3.9)\\
e_{ry} &= p_{ry}/\mathit{Distance} & (3.10)\\
\mathit{DistChng} &= (v_{rx} * e_{rx}) + (v_{ry} * e_{ry}) & (3.11)\\
\mathit{DirChng} &= [(-(v_{rx} * e_{ry}) + (v_{ry} * e_{rx}))/\mathit{Distance}] * (180/\pi) & (3.12)
\end{aligned}$$

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. (flag c) is a virtual flag at the center of the field. (flag l b) is the flag at the lower left of the field. (flag p l b) is a virtual flag at the lower right corner of the penalty box on the left side of the field. (flag 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, (flag t l 20) is 5 meters from the top sideline and 20 meters left from the center line. In the same way, (flag r b 10) is 5 meters right of the right sideline and 10 meters below the center of the right goal. Also, (flag b 0) is 5 meters below the midpoint of the bottom sideline.

In the case of (line ...), *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.

Currently there are 55 flags (the goals counts as flags) and 4 lines to be seen. All of the flags and lines are shown in Fig. 3.1.

Table 3.2: Sensor Information

(see *Time ObjInfo ObjInfo* ...)

Transmits visual information from the server. *Time* indicates the current time. *ObjInfo* is information about a visible object in the following format:

(*ObjName Distance Direction DistChng DirChng BodyDir HeadDir*)

ObjName ::= (player *Teamname UniformNumber*)
| (goal [l|r])
| (ball)
| (flag c)
| (flag [l|c|r] [t|b])
| (flag p [l|r] [t|c|b])
| (flag g [l|r] [t|b])
| (flag [l|r|t|b] 0)
| (flag [t|b] [l|r] [10|20|30|40|50])
| (flag [l|r] [t|b] [10|20|30])
| (line [l|r|t|b])

where *Distance* is the relative distance and *Direction* is the relative direction of the object, *DistChng* and *DirChng* are respectively the radius- and angle-components of change of the relative position of the object. *DistChng* and *DirChng* are only sketchy values and not very exact. *BodyDir* and *HeadDir* are the direction of the body and the head of the player, they are only included if the object is a player. For more detail, please see section 3.2.1.
The characters "l, c, r, t, b" in flags and lines means left, center, right, top and bottom and indicates where on the field the flag is positioned. "p" means that the flag is indicating the penalty area and "g" means that the flag indicates a goalpost. For more information about the positions of the flags see Fig. 3.1.
In the case of (line ...), *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 line.
As the distance to another player becomes larger and larger, more and more information about the player is lost. The quality of the information also depends on the change_view commands. The exact detail of how this works is described in section 3.2.1

(hear *Time Direction Message*)

Auditory information is delivered to the client in this manner. *Direction* is the direction of the sender. In the case that the sender is the client itself, Direction is 'self'. *Time* indicates the current time. This message is sent immediately when a client sends a (say *Message*) command.
Judgments of the referee are also broadcast using this form. In this case, *Direction* is 'referee'. Possible judgements are as follows:

before_kick_off, kick_off_l, kick_off_r, kick_in_l, kick_in_r, corner_kick_l, corner_kick_r, offside_l, offside_r, goal_kick_l, goal_kick_r, free_kick_l, free_kick_r, play_on, half_time, time_up, extend, foul_*Side_UniformNumber*, goal_*Side_Point*.

(sense_body *Time*
(view_mode *ViewQuality ViewWidth*)
(stamina *Stamina Effort*)
(speed *AmountOfSpeed*)
(head_angle *HeadDirection*)
(kick *KickCount*)
(dash *DashCount*)
(turn *TurnCount*)
(say *SayCount*)
(turn_neck *TurnNeckCount*))

Gives the status of the player's body. The *ViewQuality* is one of high and low and *ViewWidth* is one of narrow, normal, and wide. *AmountOfSpeed* is a rough approximation of the amount of the players' speed. *HeadDirection* is the relative direction of the players head. The *Count* variables are the total number of commands of that type executed by the server. For more information see section 3.2.3.

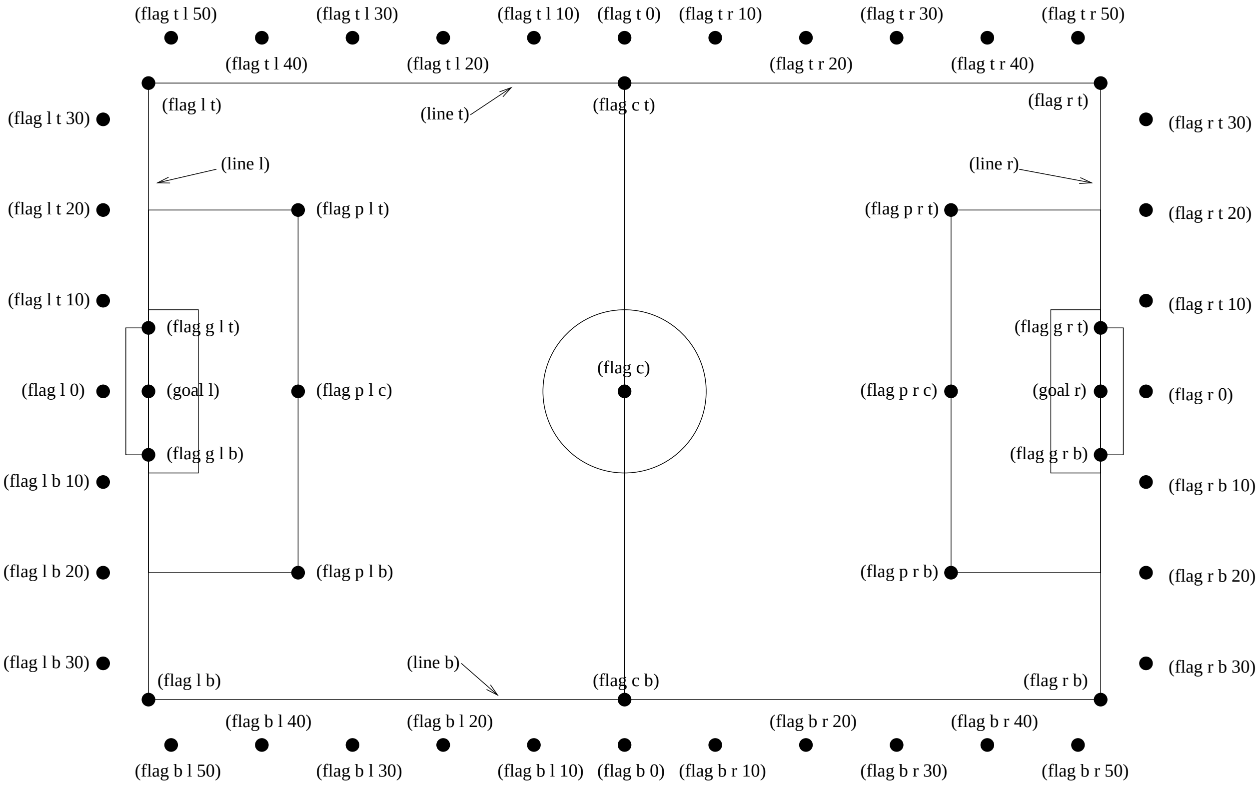

Figure 3.1: The flags and lines in the simulation.

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 players' 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*. How this is done is described in Section 3.1. To calculate the current `view_frequency` and `view_angle` of the agent use equations 3.13 and 3.14.

$$\texttt{view\_frequency} = \texttt{sense\_step} * \texttt{view\_quality\_factor} * \texttt{view\_width\_factor} \tag{3.13}$$

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`.

$$\texttt{view\_angle} = \texttt{visible\_angle} * \texttt{view\_width\_factor} \tag{3.14}$$

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 "`Ball`", "`Player`", "`Goal`" and "`Flag`", is used as the name of the object rather than "`ball`", "`player`", "`goal`" and "`flag`".

The following example and Fig. 3.2 are taken from [5].

The meaning of the `view_angle` parameter is illustrated in Fig. 3.2. 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 wold. 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°.

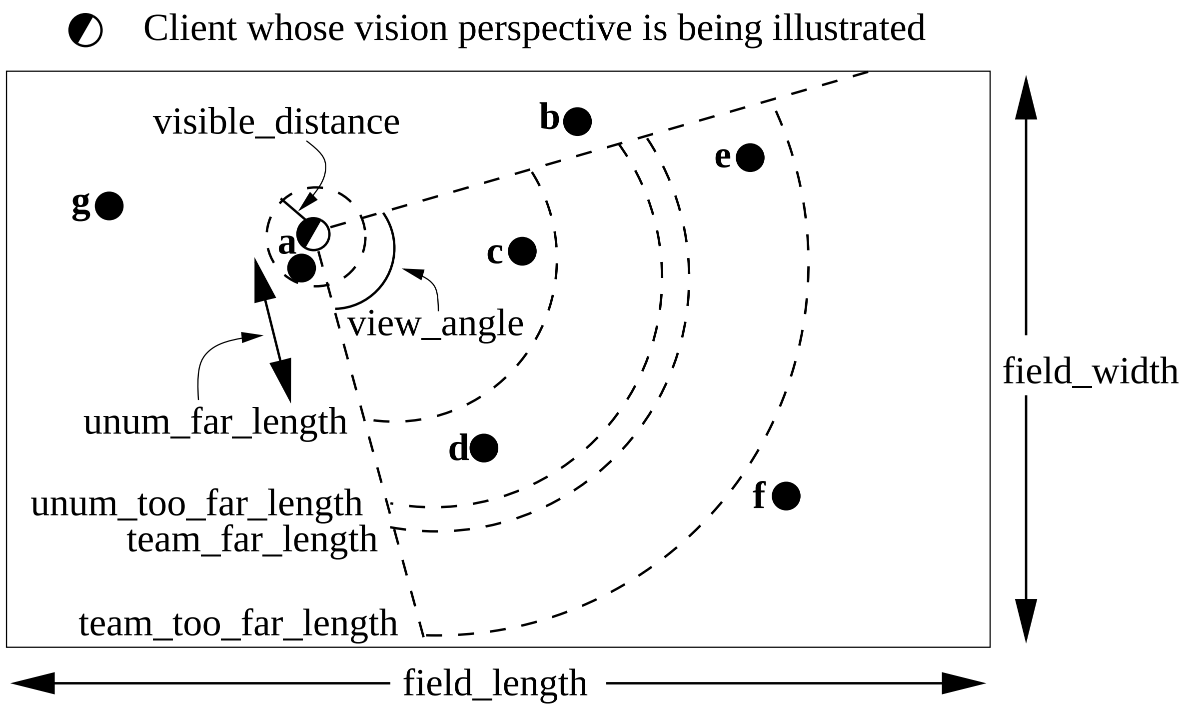

Figure 3.2: The visible range of an individual agent in the soccer server. The viewing agent is the one shown as two semi-circles. The light semi-circle is its front. The black circles represent objects in the world. Only objects withing `view_angle`°/2, and those within `visible_distance` of the viewing 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.

Also illustrated in Fig. 3.2, 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` ≤ `unum_too_far_length` ≤ `team_far_length` ≤ `team_too_far_length`. Let the player's distance be *dist*. Then

- If *dist* ≤ `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* ≥ `unum_too_far_length`, then the uniform number is not visible.
- If *dist* ≤ `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.
- If *dist* ≥ `team_too_far_length`, then the team name is not visible.

For example, in Fig. 3.2, 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.

Unreliability of Information about Far Objects

In the case that an object in sight is a ball or a player, the value of distance to the object is quantized in the following manner:

$$d' \quad = \quad \mathrm{Quantize}(\exp(\mathrm{Quantize}(\log(d), 0.1)), 0.1) \tag{3.15}$$

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

$$\text{Quantize}(V, Q) = \text{ceiling}(V/Q) \cdot Q \tag{3.16}$$

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.

$$d' = \text{Quantize}(\exp(\text{Quantize}(\log(d), 0.01)), 0.1) \tag{3.17}$$

3.2.2 Auditory Information

Format

Auditory information is transmitted by the server in the following format:
`(hear` *Time Direction Message*`)`

Direction is the direction of the sender. In the case that the sender is the client itself, `Direction` is '`self`'. *Time* indicates the current time. This message is sent immediately when a client sends a `(say` *Message*`)` command. Judgments of the referee are also broadcast using this form. In this case, *Direction* is '`referee`'. Possible messages from the referee are as follows:

> `before_kick_off`, `kick_off_l`, `kick_off_r`, `kick_in_l`, `kick_in_r`, `corner_kick_l`, `corner_kick_r`, `goal_kick_l`, `goal_kick_r`, `free_kick_l`, `free_kick_r`, `offside_l`, `offside_r`, `play_on`, `half_time`, `time_up`, `extend`, `foul_`*Side_UNum*, `goal_`*Side_Point*,

Range of Communication

A message said by a player is transmitted only to players within 50 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.

Capacity of Hear

A player can hear only 1 message from each team during each 2 simulation cycles.

If several messages arrive from the same team at the same time one message is chosen 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.

3.2.3 Sensory Information

Format

Starting from version 5.x of the server sends a `(sense_body)` to each player every `sense_step`, currently 100, milli-seconds. The format is:

```
(sense_body Time
    (view_mode ViewQuality ViewWidth)
    (stamina Stamina Effort)
    (speed AmountOfSpeed)
    (head_angle HeadDirection)
    (kick KickCount)
    (dash DashCount)
    (turn TurnCount)
    (say SayCount)
    (turn_neck TurnNeckCount))
```

Gives the status of the player's body. The *ViewQuality* is one of `high` and `low` and *ViewWidth* is one of `narrow`, `normal`, and `wide`. They are important factors in the quality and frequence of the visual information. *AmountOfSpeed* is a rough approximation of the amount of the players' speed. If the direction of the speed is needed it has to be estimated in some other way. *HeadDirection* is the relative direction of the players 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 made 134 `dashes` so far.

3.3 Soccermonitor

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

3.3.1 From Server to Monitor

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

1. `showinfo_t`: information needed to draw the scene
2. `msginfo_t` : contains the messages from the players and the referee shown in the bottom windows
3. `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 21 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
```

3.3.2 From Monitor to Server

The monitor can send to the server the following commands:

(dispinit)

sent to the server as first message to register as monitor (opposed to a player, that connects on port 6000 as well)

(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).

3.4 Logplayer

The logplayer allows you to replay recorded games. If enabled (server commandline: -record LOGFILE, server.conf file: record_log : on), the server writes the information sent to the soccermonitors into a logfile using the specified logfile version (-record_version [1/2], record_version : 2). This files can be read in by the logplayer and sent to connected soccermonitors. To watch logfiles just call logplayer with the logfilename as argument, start a soccermonitor and then use the buttons on the logplayer window to start, stop, play backward, play stepwise, ...

3.4.1 Logfiles

Version 1

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.

Version 2

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 soccermonitor)
> > `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 teamnames 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, ... If someone likes to do, feel free...

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.

Appendix A

Table of Protocols

A.1 Connection

from client to server	from server to client
`(bye)`	

`(init` *TeamName* `[(version` *VerNum*`)] [(goalie)])`
 	
TeamName $::= (-\|\_\|a-z\|A-Z\|0-9)^+$
	`(init` *Side UniformNumber PlayMode*`)`
	Side ::= l | r
	UniformNumber ::= `1` ~ `11`
	PlayMode ::= one of play modes
	`(error no_more_team_or_player)`

`(reconnect` *TeamName UniformNumber*`)`
 	
TeamName $::= (-\|\_\|a-z\|A-Z\|0-9)^+$
	`(init` *Side UniformNumber PlayMode*`)`
	Side ::= l | r
	UniformNumber ::= `1` ~ `11`
	PlayMode ::= one of play modes
	`(error no_more_team_or_player)`
	`(error reconnect)`

A.2 Client Control Commands

from a client to the server	Only one per turn
`(catch` *Direction*`)` *Direction* ::= `-180` ~ `180` degrees.	Yes
`(change_view` *Width Quality*`)` *Width* ::= narrow | normal | wide *Quality* ::= high | low	No
`(dash` *Power*`)` *Power* ::= `-100` ~ `100` Note: backward dash consumes double stamina.	Yes
`(kick` *Power Direction*`)` *Power* ::= `0` ~ `100` *Direction* ::= `-180` ~ `180` degrees.	Yes
`(move` *X Y*`)` *X* ::= `-54` ~ `54` *Y* ::= `-32` ~ `32`	Yes
`(say` *Message*`)` *Message* ::= a message (See section A.8.)	No

`(sense_body)`
The server returns
`(sense_body` *TIME*
`(view_mode {high | low} {narrow | normal | wide} )`
`(stamina` *STAMINA EFFORT*`)`
`(speed` *AMOUNT_OF_SPEED*`)`
`(head_angle` *HEAD_ANGLE*`)`
`(kick` *KICK_COUNT*`)`
`(dash` *DASH_COUNT*`)`
`(turn` *TURN_COUNT*`)`
`(say` *SAY_COUNT*`)`
`(turn_neck` *TURN_NECK_COUNT*`) )`
	No

`(turn` *Moment*`)` *Moment* ::= `-180` ~ `180` degrees.	Yes
`(turn_neck` *ANGLE*`)` *ANGLE* ::= `-90` ~ `90` degrees. turn_neck is relative to to the direction of the body. Can be invoked at 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)`

A.3 Sensor Information

from the server to a client	
`(hear` *Time Sender Message*`)`	
Time	::= simulation cycle of the `soccerserver`
Sender	::= `referee` | `self` | *Direction*
Direction	::= `-180` ~ `180` degrees.
Message	::= a message sent by a `say` command
`(see` *Time ObjInfo 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	::= one of object name (See section A.6.)
Distance	::= positive real number
Direction	::= real number
DistChange	::= real number
DirChange	::= real number
FaceDir	::= real number

A.4 Offline Coach Control Commands

from coach client to server	from server to coach client

`(change_mode` *NewPlayMode*`)`
NewPlayMode ::= one of play modes
	`(ok change_mode)`
	`(error illegal_command_form)`
	`(error illegal_mode)`

`(check_ball)`	`(ok check_ball` *Time Status*`)` *Time* ::= simulation cycle of the `soccerserver` *Status* ::= `in_field` | `goal_`*Side* | `out_of_field`
`(ear` *OnOff*`)` *OnOff* ::= `on` | `off`	`(ok ear` *OnOff*`)` `(error illegal_command_form)`
`(eye` *OnOff*`)` *OnOff* ::= `on` | `off`	`(ok eye` *OnOff*`)` `(error illegal_command_form)`

`(look)`	`(ok look` *Time ObjInfo ObjInfo* `...)`
	Time ::= simulation cycle of the `soccerserver`
	ObjInfo ::= `(`*ObjName Pos*`)` |
	`(`*ObjName Pos Velocity*`)` |
	`(`*ObjName Pos Angles Velocity*`)`
	ObjName ::= one of object names
	Pos ::= *X Y*
	X ::= real number
	Y ::= real number
	Velocity ::= *VelX VelY*
	VelX ::= Velocity in X
	VelY ::= Velocity in Y
	Angles ::= *Body_Angle Neck_Angle*
	Body_Angle ::= The angle of the body
	Neck_Angle ::= Angle of neck relative to *Angle_Body*

`(move` *ObjectName X Y* `[`*Angle*`])`
ObjectName ::= one of object names
X ::= `-54` ∼ `54`
Y ::= `-32` ∼ `32`
Angle ::= `-180` ∼ `180` degrees
	`(ok move)`
	`(error illegal_object_form)`
	`(error illegal_command_form)`

`(say` *Message*`)` *Message* ::= a message (See section A.8.)	`(ok say)` `(error illegal_command_form)`
	`(error unknown_command)`
	`(error illegal_command_form)`
	`(hear` *Time Sender Message*`)` *Time* ::= simulation cycle of the `soccerserver` *Sender* ::= name of the sender *Message* ::= a message

A.5 Online Coach Control Commands

from coach client to server	from server to coach client
(bye)	
(check_ball)	(ok check_ball *Time* *Status*) *Time* ::= simulation cycle of the soccerserver *Status* ::= in_field | goal_*Side* | out_of_field
(eye *OnOff*) *OnOff* ::= on | off	(ok eye *OnOff*) (error illegal_command_form)
(look)	(ok look *Time* *ObjInfo* *ObjInfo* ...) *Time* ::= simulation cycle of the soccerserver *ObjInfo* ::= (*ObjName* *Pos*) | (*ObjName* *Pos* *Velocity*) | (*ObjName* *Pos* *Angles* *Velocity*) *ObjName* ::= one of object names *Pos* ::= *X* *Y* *X* ::= real number *Y* ::= real number *Velocity* ::= *VelX* *VelY* *VelX* ::= Velocity in X *VelY* ::= Velocity in Y *Angles* ::= *Body_Angle* *Neck_Angle* *Body_Angle* ::= The angle of the body *Neck_Angle* ::= Angle of neck relative to *Angle_Body*
(say *Message*) *Message* ::= a message (See section A.8.)	(ok say) (error illegal_command_form) (warning cannot_say_while_playon)
	(error unknown_command)
	(error illegal_command_form)
	(hear *Time* *Sender* *Message*) *Time* ::= simulation cycle of the soccerserver *Sender* ::= name of the sender *Message* ::= a message

A.6 Object Names

(ball)
(player *TeamName* *UniformNumber*)
(player *Side* *UniformNumber*)
(goal *Side*)
(flag c)
(flag g {l | r} {t | b})
(flag p {l | r} {t | c | b})
(flag {l | r| t | b} 0)
(flag {l | c | r} {t | b})
(flag {l | r} {t | b} {10 | 20 | 30})
(flag {t | b} {l | r} {10 | 20 | 30 | 40 | 50})
(line {l | r| t | b})

Object Names Sensed by Short Distance Sensor

(Ball)
(Player)
(Flag)
(Goal)

A.7 Play Modes

before_kick_off
corner_kick_*Side*
free_kick_*Side*
goal_kick_*Side*
kick_in_*Side*
kick_off_*Side*
play_on
time_over

Messages from the referee

foul_*Side*
goal_*Side*
half_time
time_up
time_up_without_a_team
time_extended one of play modes (See section A.7.)

A.8 Message

Restriction of Messages

A *Message* said by clients must be an ASCII string less than 512 characters [1], which consists of numbers, alphabets and symbols in "`().+-*/?<>_`".

[1] The length limitation is modifiable by parameters. See section 2.3.

Appendix B

Robocup Simulation League FAQ

The FAQ is the collective effort of Noda, Gal, Tralvex, Say Poh and Eldwin.

B.1 Background on RoboCup

What is the RoboCup initiative

The Robot World Cup Initiative (RoboCup) is an attempt to foster research in various technological fields by providing a standard problem where wide range of technologies can be integrated and examined. For this purpose, RoboCup chose to use soccer game, and organize RoboCup: The Robot World Cup Soccer Games and Conferences. The idea is to build increasingly advanced physical and synthetic agents that can play soccer at increasing level. Our landmark goal is to have a humanoid robot team play against the best human soccer team under official FIFA rules sometime around the middle of next century.

The RoboCup Federation is an international non-profit organization, registered in Geneva, which was established in 1997 for the purposes of encouraging, promoting and strengthening RoboCup research. This initiative is currently being pursued by approximately 1500 researchers in 16 different countries.

For more information and details, consult the General Resources section below.
[**Gal**]

What technological fields are involved?

At first glance, RoboCup seems to deal only with issues from the field of Robotics. However, the RoboCup challenge actually touches on many fields. Here's a partial list:

- **In Artificial Intelligence and Robotics:** Multi-agent environments, collaborative and individual planning and execution, opponent modeling, distributed and individual learning (both off-line and on-line), adaptation, strategy planning, sensor fusion, knowledge acquisition, real-time reasoning, visions, natural language processing (both for synthetic commentators, and for the players, which need to interact with a human referee), crowd behavior, teleoperation, and more.
- **In electrical engineering**, it touches on fault-tolerant hardware, real-time support, robust storage devices (imagine a hard disk being thrown to the ground due to a tackle...), sensors, control, etc.
- **In Virtual Environments and Distributed Simulations**, it touches on real-time multi-cast protocols (for running simulated games on the Internet - see more on this later in this FAQ), real-time 3D visualization, multi-player protocols, teleoperation, etc.

In addition, there's need for lightweight but tough materials (otherwise the robots will be too heavy, or become dented with every fall), and for batteries/power-sources that actually power a human-size robot for 90 minutes or more. There's need for robust, fault-tolerant joints and gears that can stand some bumps, and the list goes on and on.

The challenge is not limited to soccer-playing robots. Far from it we would like to see more and more applications grow from this domain into other fields. For example, there's already some work on a multi-agent commentator team, in which different synthetic experts work together to produce commentary on the game, much like the human teams seen so often on TV. Some people are considering modeling the fans, and their

influence on the team, to examine issues in crowd behavior. Others still are looking into robot coaches, and so on.
[**Gal**]

How do I participate in the competition?

Any and all teams, whether research-oriented or not, are welcome to participate in the competitions. Some teams just participate for the fun of it (especially in the Simulator League); some are serious research-oriented teams that also publish the results of their research. See below for more details.

Currently, RoboCup competitions have the following leagues. Specific information on each league can be found in the resource lists in the sections describing each league and in the resources section.

- Simulator League for Synthetic (Software) Agents
- Small Robot League (5 robots per team)
- Full Set Small Robot League (11 robots per team)
- Middle Size Robot League

The followings are currently leagues for exhibition games and demonstrations:

- Legged Robot League
- Expert Robot League
- Humanoid League

[**Gal**]

General Resources and Contacts

The RoboCup Web Page:
http://www.robocup.org/
The RoboCup General Mailing List:
Send a subscribe command to majordomo@csl.sony.co.jp with "subscribe robocup" in the body of the message.
National and Regional Committees:

- **Brazil National Committee:**
- **Euro-RoboCup (European Community Committee):**
- **French National Committee:**
- **German National Committee:**
- **Italian National Committee:**
- **Japan National Committee:**
- **Scandinavian National Committee:**
- **Singapore National Committee:**
- **U.S. National Committee:**

[**Gal**]

Simulation-related Resources and Contacts

RoboCup:
http://www.robocup.org/
RoboCup Simulator:
RoboCup Simulation League Resources:
`<http://jsaic.krdl.org.sg/robocup/ftp/resources/>`
RoboCup Simulation Mailing List:
robocup-sim-l@usc.edu
To Join, send a "subscribe robocup-sim-l" request to `<listproc@usc.edu>`.
Or visit: `<http://www.isi.edu/soar/galk/RoboCup/robocup-sim-l>`/ RoboCup Simulation Code Archives: Contain code samples, libraries, etc.
[**Gal**]

B.2 Introduction to Simulation League

What is the RoboCup Simulator?

The RoboCup simulator is a soccer physical simulation system, which allows virtual soccer players to compete in a dynamic, complex, uncertain multi-agent environment. The soccerserverattempts to provide a challenging environment for AI research, by allowing researchers to concentrate on designing brains for the simulated bodies of the players.

The simulation system is run as a client/server system, with a central simulator server communicating with 22 player clients. Each client receives sensory data - visual, auditory, and internal - about its own local view of the world (from its own position), and can send command such as move, run (with power), turn, kick, etc.
[**Gal**]

What platforms does the simulation run on?

The simulator is written in C and C++, and can compile out-of-the-box on many flavors of Unix, including SunOS, Solaris, Linux, and others. It may also be made to run on Windows, but we don't know of any attempts to do so, successful or otherwise.
[**Gal**]

What platforms can I build players for?

The server communicates with player clients via UDP sockets. You can build your clients on any platform you with, as long as they can communicate with the server. Some use Unix, some use Windows. Implementation languages include C, C++, Lisp, Soar (rule-based language), and Java.
[**Gal**]

Can I run a game across the Internet?

Yes, although it may be slow.
[**Gal**]

Whats all this stuff about Gentlemens Agreement?

The simulated environment is very challenging, but like many software systems it has bugs and loopholes. For instance, players are only supposed to communicate with each other via the simulation by using simulated shouts. There is nothing, however, that prevents client programs from using private inter-process communications to bypass the server. However, we expect teams not to use this even though it may practically be feasible, since it works against the aims of the initiative.

The RoboCup Simulation Steering Committee will be releasing ethical guidelines, which explain this further.
[**Gal**]

B.3 Startup Soccerserver

When I attempt to run soccermonitor, I get the message ld.so: libSM.so.6: not found

In this case, please try the following instructions.

- **1.** Change directory to " monitor/ ".
- **2.** Edit setting of "XLIB" in " Makefile " as follows (adding " -lSM -lICE " options).
  XLIB = -L/opt/X11R6/lib -lXaw -lSM -lICE -lXmu -lXt lXext $(XLINK_Dynamic) -lX11
- **3.** Remake soccermonitor.

[**Noda**]

When I attempt to run soccermonitor, I get the message ld.so: lib???.so.???: not found

You must configure LD_LIBRARY_PATH in sserver script by hand. For example, you may change the 3rd lines of sserver as follows:
setenv LD_LIBRARY_PATH /usr/X11R6/lib:/usr/ucblib:/usr/lib:/lib
[**Noda**]

When I start the Soccerserver, I get a message saying can not bind local address.Why?

Can not bind local address usually happens when there is an old soccerserverprocess running. Try
ps -aux || grep soccerserver
If there are old processes running, kill them and try again.
[**Noda**]

I run sserver script, and I got the following messages:

[1] 23967
SoccerserverVer. 4.06
Copyright 1995, 1996, 1997, 1998 Electrotechnical Laboratory.
Itsuki Noda, Yasuo Kuniyoshi and Hitoshi Matsubara.
Wind factor: rand: 0.000000, vector: (0.000000, 0.000000)
Error: Cant open display:
Add "setenv DISPLAY console_name:0.0" to sserver (shell script). Examples of "console_name" are pluto, galileo, etc. Then execute sserver again. If it fails, make sure that you are running on the Unix Server. An alternative is using application program "eXodus", which acts as a server console.
[**Say Poh/Rex**]

What is the recommended directory for X11R6?

This varies from machine to machine. Try "/usr/local/X11R6", if you encounter problems when trying to make soccerserverusing "/usr/openwin/".
[**Say Poh/Rex**]

When I start Soccerserver, why do I get a message: cant bind local address?

Can't bind local address usually happens when there is a previous soccerserverprocess running. Try:
ps -ef || more
If there are previous soccerserverprocess running, kill them and try again. Please note that you must kill the soccerserverand soccermonitor processes after use. Same applies to the client processes.
[**Say Poh/Rex**]

I tried to get Soccerserverto compile on my Linux box.

As with most things, the usual steps of "configure then make" didn't work on the first try for me. I'm running RedHat 5.0 Linux, I've got gcc-2.8.0 installed, I'm using X11R6. It looks to me like the include files `<iostream.h> and <strstream.h>` are missing from my machine. Where can I find these 2 include files? Any other ideas?

- **Suggestion 1:** You have to use g++ instead of gcc. "iostream.h" is a common c++ file, so if you can't find that, you probably can't find any of the c++ files. On some linux box, it can be found in "/usr/include/g++/iostream.h". Try using the 'locate' command if you can't find it.

- **Suggestion 2:** Do a "which g++" command. On most systems, what this returns is a link to where all the gnu goodies reside. 'cd' to the directory pointed to by the link. The libraries reside in ../lib and subdirs. Also, you may need to set your LD_LIBRARY_PATH environment variable to contain paths to the libraries soccerserveruses.

[**Say Poh/Rex**]

B.4 Making Clients

Why does the sample client continuously receive

recv 2035 : (see 0 ((goal r) 73 -7) ((flag r t) 84.8 -31) ((flag r b)
76.7 18) ((flag p r t) 63.4 -28) ((flag p r c) 56.8 -10) ((flag p r b)
56.8 10) ((ball) 22.2 -26 0 0) ((line r) 72.2 -90)
is that just the automatic sensory response from the server to the clients?
Yes. This is visual sensor information sent from the server every 300ms. This means,
Now, TIME is 0 (may be before kick-off),
right goal (goal r) is 73m far away and 7 degrees left
flags of top-right corner is 84.8m far away and 31 degrees left
...
ball is 22.2m far away and 26 degrees left, and its speed is (0,0)
...
The server also sends auditory information that informs messages from referee and other players. These information is sent automatically, so you should make a client that always polls the socket that connects to the server.
[**Noda**]

How do I set up a connection to the server as non-blocking?

Here is a sample of C/C++ code to set up a socket connection as non-blocking.

```
/*
* Open UDP socket.
*/
if( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0){
return sock ; /* Can't open socket. */
}
val = fcntl(sockfd, F_GETFL, 0) ;
/* 5/17/96 Dave found that O_NONBLOCK didn't work for setting up a*/
/* non-blocking socket. O_NDELAY seems to work, though! */
/*     val |= O\_NONBLOCK ;*/
val |= O\_NDELAY;
fcntl(sockfd, F_SETFL, val) ;
```

[**Noda**]

My clients can not get newest visual information, because a couple of messages from the server are on the queue of the socket. How do clients get the newest message from the queue?

Here is a sample of C/C++ code that clean up queue and get the newest "see" information. If there are multiple "see" messages on the socket, this code prints the old ones, but skips them. This makes sure all hear messages are received, though.

```
if(FD_ISSET(sock.socketfd, &readfds)){
/*n = receive_message(buf, MAXMESG, sock, &port) ; */
/*keep receiving messages until the socket is empty*/
while((n = receive_message(tmp, MAXMESG, sock, &port)) > 0)
int copy = TRUE; /* in general, copy tmp to buf */
if (MessageCounter > 0) /* not first time through while */
if ( initialized && tmp[1]=='h' )/* New msg is a hear */
ParseSensoryData(buf, Memry); /* parse the sound */
if ( Memry->HeardNewSound() ) /* respond to it */
Communicate(behavior, Memry, &sock);
Memry->ClearSound(); /* Undo the sound bit */
copy = FALSE; /* Leave the last msg in buf */
#if PRINT_MISSED_MSGS
else if ( initialized )
printf("Player %d-%c skipped a message %.40s ...\n
before %.40s ...\n",
Memry->MyNumber,Memry->MySide,buf,tmp);
#endif
if (copy)
strcpy(buf,tmp);
if ( buf[1] == 'h' ) /* also stop if it's a "hear" */
break;
MessageCounter++;
MessageCounter = 0;
```

[**Noda**]

Im trying to run Ogalets or Sekines clients, but players do not work well. Why?

Unfortunately, Ogalets and Sekine's clients are for sserver-2.78. Protocol of sserver has been changed in sserver-3.xx, so now, these clients can not analyze sensor information sent from the server.

In order to test these clients, please get sserver-2.78.

[**Noda**]

I came across FD_ZERO, FD_SET, FD_ISSET functions in the client program. What do they mean?

Here is the excerpts from "man select" command:

```
select(3C) C Library Functions select(3C)
NAME
    select - synchronous I/O multiplexing
SYNOPSIS
    #include <sys/time.h>
    #include <sys/types.h>
    int select(int nfds, fd_set *readfds, fd_set *writefds,
    fd_set *exceptfds, struct timeval *timeout);
    void FD_SET(int fd, fd_set &fdset);
    void FD_CLR(int fd, fd_set &fdset);
    int FD_ISSET(int fd, fd_set &fdset);
    void FD_ZERO(fd_set &fdset);
MT-LEVEL
```

MT-Safe
DESCRIPTION

select() examines the I/O file descriptor sets whose addresses are passed in readfds, writefds, and exceptfds to see if any of their file descriptors are ready for reading, are ready for writing, or have an exceptional condition pending, respectively. Out-of-band data is the only exceptional condition. nfds is the number of bits to be checked in each bit mask that represents a file descriptor; the file descriptors from 0 to nfds -1 in the file descriptor sets are examined. On return, select() replaces the given file descriptor sets with subsets consisting of those file descriptors that are ready for the requested operation. The return value from the call to select() is the number of ready file descriptors.

The file descriptor sets are stored as bit fields in arrays of integers. The following macros are provided for manipulating such file descriptor sets: FD_ZERO() initializes a file descriptor set fdset to the null set. FD_SET() includes a particular file descriptor fd in fdset. FD_CLR() removes fd from fdset. FD_ISSET() is nonzero if fd is a member of fdset, zero otherwise. The behavior of these macros is undefined if a file descriptor value is less than zero or greater than or equal to FD_SETSIZE. FD_SETSIZE is a constant defined in ¡sys/select.h¿.

If timeout is not a NULL pointer, it specifies a maximum interval to wait for the selection to complete. If timeout is a NULL pointer, the select() blocks indefinitely. To effect a poll, the timeout argument should be a non-NULL pointer, pointing to a zero-valued timeval structure.

Any of readfds, writefds, and exceptfds may be given as NULL pointers if no file descriptors are of interest.
RETURN VALUES

select() returns the number of ready file descriptors contained in the file descriptor sets or - 1 if an error occurred. If the time limit expires, then select() returns 0.
ERRORS

The call fails if:

EBADF One of the I/O file descriptor sets specified an invalid I/O file descriptor.

EINTR A signal was delivered before any of the selected events occurred, or the time limit expired.

EINVAL A component of the pointed-to time limit is outside the acceptable range:

t_sec must be between 0 and $10\hat{8}$, inclusive. t_usec must be greater than or equal to 0, and less than $10\hat{6}$.
SEE ALSO

poll(2), read(2), write(2)
NOTES

The default value for FD_SETSIZE (currently 1024) is larger than the default limit on the number of open files. In order to accommodate programs that may use a larger number of open files with select(), it is possible to increase this size within a program by providing a larger definition of FD_SETSIZE before the inclusion of `<sys/types.h>`.

The file descriptor sets are always modified on return, even if the call returns as the result of a timeout.
[**Say Poh/Rex**]

What is the correct behavior when a player receives the "half_time" message? Should he close the connection with a "bye" command and reconnect after a break? Does the server keep running during the half-time break?

The server keeps running at halftime. Since teams sometimes like to change their behaviors at halftime (keep in mind that halftime only lasts a minute or two - no time for serious code changes), you can disconnect and reconnect the players. But if you would like, you are welcome to keep your team running through the half-time break.
[**Say Poh/Rex**]

What are "quantize_step" and "quantize_step_l" found in the Soccerserver's configuration file (server.conf)?

These are parameters that the server uses to determine how fine values are reported to you, (whether 5.23423423423 should be reported as 5.2, 5.3, 5.234). It should not affect the client program implementation.
[**Say Poh/Rex**]

There are different types of parameters for stamina, recover and effort in the Soccerserver's configuration file (server.conf). How do they work?

Given parameters:

maximum and recovery step of player's stamina (default 2500.0, 50.0)
stamina_max : 2000.0
stamina_inc_max : 20.0
decriment threshold ,decriment step and minimum of player's recovery
(default 0.3, 0.05, 0.1)
recover_dec_thr : 0.3
recover_dec : 0.002
recover_min : 0.5
decriment threshold, decriment step, incriment threshold, incriment step
and minimum of player's effort (default 0.4, 0.05, 0.9, 0.05, 0.1)
effort_dec_thr : 0.5
effort_dec : 0.02
effort_inc_thr : 0.8
effort_inc : 0.02
effort_min : 0.5

There are 3 parameters involed here:

- **1.** stamina [0, max_stamina]
- **2.** recovery [recover_min, 1]
- **3.** effort [effort_min, 1]

recovery and effort are actually multipliers.

Assuming that the stamina scale looks something like this:
— Max stamina (2000.0)
—
- effort_inc_thr (0.8)
—
—
—
—
- effort_dec_thr (0.5)
—
—
- recover_dec_thr (0.3)
—
— 0

Whenever we execute a dash command, our stamina is reduced. If we continue to dash further, our stamina is reduced further. If our stamina is reduced below effort_dec_thr (0.5) * max_stamina (2000.0) = 1000.0, for every simulation cycle we stay below this value, our effort is decreased by effort_dec (0.02). Assuming our stamina was below the 1000.0 mark for 5 cycles, then our effort would be reduced by effort_dec (0.02) * 5 = 0.1. So if we started with an effort of 1, now our effort would be 1 - 0.1 = 0.9. Thereafter, when we dash 100, we are actually dashing a value of 0.9 * 100 = 90. Currently, in the server, our stamina is only deducted 90, the amount that was actually used. So, in fact, assuming we really want to dash 100, we can ask for 111.111, so that 100 is actually applied. I think this is a bug that they haven't removed yet. The original intention was that a player is penalised 100 but actually 90 is used.

At the other end of the scale, we have effort_inc_thr. When your stamina goes above this threshold (effort_inc_thr * max_stamina), for every cycle that you stay above a value of effort_inc is added to your effort.

Similarly, for recover_dec_thr, for every cycle that your stamina goes below the threshold (recover_dec_thr * max_stamina) your recovery is reduced by recover_dec. Once you recovery is reduced, it NEVER gets back to one. It's only replenish at half-time. Your recovery affects how fast your stamina recovers. For every simulation cycle, you are given some stamina.
[Say Poh/Rex]

How much stamina is given to you at every simulation cycle?

recovery * stamina_inc_max
So, you can see that the less recovery you have the slower you recover after an energy-draining sprint.
[**Say Poh/Rex**]

In Soccerserverversion 4.18 (sserver-4.18), there is a part "stamina -= fabs(power)". How does it work?

This is part of the stamina model. The more tired a player is (i.e. effort `< 1`), the more energy it needs to go at the same speed. Another way to look at it is, the same amount of energy produces lesser speed.
[**Say Poh/Rex**]

Human can't assign free_kicks or drop_balls upon holding the mouse click in the soccermonitor. How do I overcome this problem?

Try disabling the Num Lock found on the keyboard.
[**Say Poh/Rex**]

Every Soccerserverrelease has new changes to it. Where can I reference for those changes made?

Refer to the "Changes" file that comes together with the soccerserver.
[**Say Poh/Rex**]

What are the different versions of Soccerserver's protocols that I can use?

Basically, there are 2 different versions of the protocols, version 3 & 4. Below is the comparsion between the 2 versions. You can also refer to question 7 for location of reference.

Version 3 protocol	Version 4 protocol
"(init TEAMNAME)" to initialize an agent	"(init TEAMNAME (version 4.00))" to initialize an agent.
No goal-keeper implementation allowed.	Can init. an agent to be a goal-keeper by "(init TEAMNAME (version 4.00) (goalie))".
Commands like 'catch', 'sense_body' are not available	Commands like 'catch', 'sense_body' are available.
Only 12 different flags in 'see' info.	53 flags altogether, and player has an additional parameter (facedir) in 'see' info.

[**Say Poh/Rex**]

Tell me more about 'sense_body' command.

The following is just Noda's policy (**this means that this is not an official policy**):

- (sense_body) is introduced for help to know rough information about player's body. As same as other sensors like 'see' and 'hear', it does not guarantee to report correct information. So it is not good to assume a mechanism above. If you build a client that assumes such mechanism, it may not work when we change model of simulation in future.
- Another thing is that if you send a command every 20ms, the server becomes too busy to execute the commands on time. In such case, we may change sserver to limit to accept only a sense_body command per 100ms.

And this is simulator committee's policy:

- It is intended that the server promptly responds to all sense_body commands. But it is not intended for clients to send sense_body commands very frequently to the server. Such frequent command execution tends to cause network collisions. Because the soccerserveruses UDP/IP, if collisions occur, some information is lost. The problem is that this loss also affects another team who does not use frequent commands.

- To avoid jamming the server with too many commands, we ask that you limit your clients to no more than 3 or 4 commands per cycle (simulator step = 100 msec) to the server. For example, on a given cycle, each client on your team could send 1 turn/dash/kick, 1 say, 1 sense_body, and perhaps a change_view; or 1 turn/dash/kick and 2 or 3 sense_bodys.
- As usual, this is not a strict rule, but rather a general guideline for reasonable behavior. Nobody will be disqualified if one client sends 7 commands to the server on a single cycle. But if we find that clients are consistently sending too many commands per cycle, the team will be asked to change its code.

[**Say Poh/Rex**]

How do I migrate from version 3 to version 4 protocol?

Make sure that you are using sserver-4.xx, in order to migrate. (In order for goal-keeper's 'catch' command to work properly, please use sserver-4.17 or later).

Now search for "(init TEAMNAME)" in your client implementation, where TEAMNAME is the team name you wish to initialize to. For example:

sprintf(init_mesg,"(init %s)\n",t_name);

where t_name is TEAMNAME. Replace it with "(init TEAMNAME (version 4.00))". For example:

sprintf(init_mesg,"(init %s (version 4.00))\n",t_name);

or "(init TEAMNAME (version 4.00) (goalie))" for goal-keeper. For example:

sprintf(init_mesg,"(init %s (version 4.00) (goalie))\n",t_name);

After that, you have to change your client implementation according with effect to the additional flags and facedir found in 'see' info. The additional flags are:

center mark (flag c)
goal posts (flag g l t) left-top
(flag g l b) left-bottom
(flag g r t) right-top
(flag g r b) right-bottom
flags outside of the pitch (5 meters outside from the edge of the pitch)
(flag TB LR XX)
(flag LR TB YY)
TB ::= [t—b]
LR ::= [l—r]
XX ::= [10—20—30—40—50]
YY ::= [10—20—30]
(flag TBLR 0)
TBLR ::= [t—b—l—r]
As for the facedir, it tells the direction that player is facing. The format is:
((player TEAMNAME UNUM) DIST DIR DISTCHANGE DIRCHANGE FACEDIR)
For example,

```
P=>        <=Q Q's FACEDIR is 180.
P=>         Q Q's FACEDIR is 90.
```

V [**Say Poh/Rex**]

What are the guidelines that will be followed by the human referee at the simulator competition?

So far, the fouls are:

- surrounding the ball,
- blocking the goal with too many players
- not putting the ball back into play after a free kick,
- intentionally blocking the movement of other players,
- the goalie repeatedly kicking and catching to move the ball in the penalty area,

- sending more than 3 or 4 commands per client per cycle to the simulator (checked if the server is being jammed or upon request after a game),
- anything else deemed to be "ungentlemanly" play judged by the referee.

[**Say Poh/Rex**]

B.5 Visual Information

Are the following two values the same?

DirChng=-1 , DistChng=-.334

DirChng=179, DistChng=.334

No. If the client receives

(ObjName Dist Dir DistChng DirChng)

the actual relative movement (Vx, Vy) is:

Vx = DistChng * cos(Dir) + (p /180) * DirChng * Dist * sin(Dir)

Vy = DistChng * sin(Dir) + (p /180) * DirChng * Dist * cos(Dir)

where Vx is a component that is parallel to the facing direction of the player, and Vy is a component that is vertical to the facing direction.
[**Noda**]

Does it always keep ||DirChng|| < 90?

If the player is stationary and the ball moves from relative position (Dist, Ang) (10,15) to position (10,17) in the last simulator step (.1 second), then DirChng would be the angle of the ball motion relative to the player and DistChng would be the ABSOLUTE distance the ball traveled. Even though the balls distance to the player remained constant, DistChng would not be 0?

No. In this case, DistChng should be 0, or near of 0. DirChng may be near of 2.0.
[**Noda**]

If the player moves, the vector of the players ABSOLUTE motion is subtracted from the balls ABSOLUTE motion vector. DistChng is the magnitude of the resulting vector and DirChng is its angle relative to the players direction?

No. Suppose that players motion and position are (Vx,Vy) and (Px,Py) respectively, and balls motion and position are (Ux,Uy) and (Qx,Qy) respectively. Then, relative vector of motion (Wx,Wy) is

(Wx,Wy) = (Ux,Uy) - (Vx,Vy)

The relative position vector of ball (Rx, Ry) and its unit vector (Ex,Ey) are

(Rx,Ry) = (Qx,Qy) - (Px,Py)

R = (R2x + R2y)

(Ex,Ey) = (Rx/R,Ry/R)

Then, radius component (Dr = DistChng) and angle component (Da = DirChng) of the relative movement (Wx,Wy) are

Dr = (Wx * Ex) + (Wy * Ey)

Da = (180/p) * (Wx * Ey - Wy * Ex) / R

The inverse translation is as follows:

Suppose that Dr, Da, Rx, Ry, Vx and Vy are given. Then,

R = (R2x + R2y)

(Ex,Ey) = (Rx/R,Ry/R)

Wx = Dr * Ex - (Da * p/180) * R * Ey

Wy = Dr * Ey + (Da * p/180) * R * Ex

(Ux,Uy) = (Vx,Vy) + (Wx,Wy)

These equations are theoretical and noise-less environment. Actually, the server adds various noises and quantizes values, so that the inverse translation is not so simple.
[**Noda**]

B.6 Inside of Soccerserver

The soccerservermanual says that the magnitude of the noise in the ball motion varies in proportion to the parameter ball_rand.

Does the magnitude of the noise mean the standard deviation of the guassian determines its next position? Or is it some other meaning?

Generally soccerserveruses unformed distribution of noise. This means that, if ball_rand parameter (=magnitude of noise) is R, then the noise value is in [-R,R], and the density of probability is the same in the domain [-R,R]. In other words, the probability that the noise is in [x,x+dx] is,

P(noise is in [x,x+dx]) = (1/2R) * dx ; if x is in [-R,R]

= 0 ; otherwise

Standard deviation s of this distribution is:

s2 = R-R (1/2R) * x2 * dx

= (1/2R) * —x3/3—R-R

= R2/3

[**Noda**]

The scores on the soccermonitor wrap very soon, going into negative numbers if the score gets too big. This is a problem when using the same soccerserverfor a large number of evaluations. Can I suggest using ints or longs for the scores instead of chars or shorts?

Internally, soccerserveruses int for score. So you need not care about it. The referee reports correct score when a team gets a goal.

We use char for communication between soccerserverand soccermonitor in order to reduce the amount of the communication.

[**Noda**]

What are the units of player_size and ball_size? Does these parameters specify radius or diameter?

The units are meters. They specify radius of the objects.

[**Noda**]

What are the units of player_weight and ball_weight? What are the units of wind_force and ball_weight?

There are no correspondence between this unit and physical unit systems. *_weight specifies the stiffness against wind effect. If weight is large, effect of force of the wind is small, and if the weight is small, the effect is large. Actually, wind accelerates the velocity of objects every simulation cycle in the following manner:

vel.x += vel.r * (wind.x / weight * WIND_WEIGHT)

vel.y += vel.r * (wind.y / weight * WIND_WEIGHT)

where (vel.x,vel.y) is objects velocity, vel.r is amount of movement of the object (= `|(vel.x,vel.y)|`), (wind.x,wind.y) is a vector of strongness of the wind, and WIND_WEIGHT is a constant parameter (=10000).

wind_force is amount of force of wind. This is equal to `|(wind.x,wind.y)`— (absolute value of the wind vector) in the above equation.

[**Noda**]

wind_dir is 0 means that wind flows to north?

No. Left-to-right on the display is 0 degree.

[**Noda**]

What are the units of maxpower and minpower?

These do not correspond to the physical parameters. It specifies the degree of power. For example, if the client send (dash 100), then the velocity of its player will change as follows:

vel.x += 100 * dash_power_rate * cos(dir)
vel.y += 100 * dash_power_rate * sin(dir)
where (vel.x,vel.y) is the velocity of the player, dash_power_rate is a parameter specified in file server.conf, and dir is the direction the player is facing.
[**Noda**]

Are the units of maxmoment and minmoment degrees?

Yes.
[**Noda**]

B.7 Miscellanous

How reliable is the message transmission?

Method 1: Half-asychronous transmission. Transmits 76 messages, then stops for 10 cycles.

Test run number	Total number Sent	Total number Received	Total number Lost	Percentage Received
1	2856	1948	908	68.21%
2	2856	1946	910	68.14%
3	2862	1951	911	68.17%
4	2854	1945	909	68.15%
5	2852	1945	907	68.20%

Method 2: Half-asychronous transmission. Transmits 76 messages, then stops for 20 cycles.

Test run number	Total number Sent	Total number Received	Total number Lost	Percentage Received
1	2664	1955	709	73.39%
2	2660	1950	710	73.31%
3	2660	1950	710	73.31%
4	2665	1952	713	73.25%
5	2671	1956	715	73.23%

Method 3: Synchronous transmission, i.e. Send then wait to receive that message

Test run number	Total number Sent	Total number Received	Total number Lost	Percentage Received
1	2836	2836	0	100%
2	2816	2816	0	100%
3	2834	2834	0	100%
4	2812	2812	0	100%
5	2812	2812	0	100%

The above tests were done for one half of the full game, i.e. 3000 simulation cycles, with 2 teams (against CMUnited) of 1 player (goalkeeper) each in idle state (no kicking of ball). One thing to take note of is, it takes one or less simulation cycle to receive sent message.
[**Say Poh/Rex**]

Appendix C

Know How

The creation of an agent is a major undertaking that involves solving a wide variety of problems. In this chapter we try to hint at solutions for a series of common problems that have to be solved independently of the actual scientific approach.

Please note that this is a very incomplete collection of individual experiences.

C.1 Modeling

In this section we want to give useful hints that concern the modeling aspect of a client.

C.1.1 Controlling Stamina

The soccerserver's stamina model requires agents to save and spend energy as a partially replenishable resource. Since the stamina parameter *recovery* is non-replenishable, agents are best off if they avoid depleting it (the only exception is when the game is about to end).

To avoid depleting their recovery, agents should monitor their stamina using the sense_body command. If the stamina is within 100 of the recovery threshold (`recover_dec_thr * stamina_max`), then the power of a dash should be limited to the difference between the current stamina and the recovery threshold. While the player may be limited to dashing with a power of 20 (`stamina_inc_max`) for some time, it will still be able to recover stamina at the maximum possible rate.

Another way of conserving stamina is by moving slowly (dash with power 40 or so) when adjusting positions on the field. In general, it is only necessary to dash at full speed when trying to get to the ball before an opponent.

C.1.2 Determining the Position on the Field

Most RoboCup agents try to figure out their absolute position on the field from the lines and markers they see. There is a variety of ways to do this. Unfortunately the example implementation of the triangulation is not very accurate and sometimes even incorrect. The following approach has given fairly accurate positions.

In a first step you calculate the absolute view direction of the agent from a line. Since the intersection angle between the line and the agent's view direction is given without any added error, the result is very accurate. If the agent sees more than one line, it can ignore the line that it sees from the outside (this is the closer line). If the agent doesn't see any line, some alternative algorithm must be used to determine the view direction (such as taking it from an internal world model or using markers).

Now the agent can calculate a vector from its position to the marker by adding its view direction to the polar coordinates of the flag position it receives. By subtracting it from the (known) global position of the marker, the agent gets its absolute position. You can compute one such position per seen flag and then take some sort of weighted average to further increase the accuracy.

An example implementation can be found in the file `Field.cc` of the AT Humboldt 98 source code, which is publicly available at
`http://www.ki.informatik.hu-berlin.de/RoboCup/RoboCup98/ATH98.tar.gz`.

C.1.3 A Memory of Seen Objects

Since clients have limited vision angles, objects on the field can come in and out of view to a particular client. While mobile objects may not stay in the same position when they are out of view, they also cannot move more than a certain distance per simulator cycle. Therefore, agents should "remember" the last known position of an object while becoming less and less confident in the accuracy of this information.

One simple implementation is to keep a confidence variable for each mobile object. Set the confidence in an object's position to a maximum value when it is seen and then multiply the confidence by some fraction every cycle that it is not seen. When the confidence reaches some threshold, it can be considered forgotten.

Since the ball can move faster than players, confidence in the ball's location should decay more quickly. Nevertheless, an agent should be able to turn away from the ball to move to a new position without forgetting the ball's location entirely.

C.1.4 "Ghost Objects"

A frequently made mistake in the implementation of such a memory of seen objects results in what we call "ghost objects". During RoboCup-98 it seemed several times that agents were following a ghost ball.

What happens is this: When merging the memory of seen objects with the current visual information, it is very tempting to simply keep all objects the agent currently doesn't see. Now if, for example, the agent's memory says the ball is in front of the agent, but the agent doesn't see it there (because, actually, the ball is behind the agent), the agent still keeps in its memory the old, obviously faulty position of the ball. And it keeps running towards this imaginary ball somewhere in front of it.

Of course, the right thing to do is check if the agent should have seen the object before deciding to keep the object's position. If the agent should have seen it but didn't, the position is obviously incorrect and should be discarded.

C.1.5 The Client's View of the World

It can be very time consuming and cumbersome to debug the agent's internal model of its environment. One approach is to visualize it either using an adapted version of the soccer monitor or to write a separate program. The individual belief of an agent can then be easily compared to the "real" world as visualized by the soccer monitor.

C.1.6 Some Words About Time

In general, the server time increments by 1 on every simulator cycle. However, there are occasions (after goals and after offsides calls) during which the simulator clock stops. If your clients base their behaviors on time in any way (e.g. sending a sense_body command every 10 cycles), they may become confused when the server time stops.

One possible solution is to represent time as an ordered pair (t,s) where t is the server time and s is the number of cycles since the clock has stopped. When the clock is moving, s is always 0. Thus, t always matches the server time, but with appropriate arithmetic functions defined, agents can still reason about the number of simulator cycles between events.

For an example of such an implemented time structure, see the CMUnited-98 source code available at `http://www.cs.cmu.edu/~pstone/RoboCup/CMUnited98-sim.html`.

C.2 Synchronization

Synchronization between client and server is an important and very intricate issue. The server cycles may be longer than you think, commands may be lost, and information coming from the server may be inconsistent.

C.2.1 A Method to Check the Synchronization

The Soccerserver produces a log file that records every incoming command if its config file contains the line:

```
log_file: game.log
```

Of course, you can use any other file name instead of `game.log`. Using the UNIX command `grep` you can easily extract all the commands the server received during one cycle and/or those coming from a particular agent.

If you find more than one main command like `dash` or `kick` per agent per cycle, the agent is too fast and commands were lost. If you don't find any commands from an agent for a particular cycle, the agent is too slow and thus not as effective as it could be.

C.2.2 Inconsistent Information From the Server

The soccerserver does not guarantee that its internal information is always instantaneously consistent: it reaches a consistent state at the end of every simulator cycle. For example, when the server receives a turn command, it updates the player's angle instantaneously. However, dashes don't take effect until the end of a cycle.

One problem that this inconsistency causes relates to estimating an agent's angle. You may want your agent to update its belief of the angle that it's facing based on a turn command sent. In this way, it can keep a correct world view even if it doesn't receive visual information. However, it is important to be careful to notice whether the previous visual information already reflected the turn or not. Otherwise, your agent may wrongly incorporate the effects of the same turn twice. An agent may determine whether the server already incorporated the turn before sending visual information by checking whether its facing angle (determined by the angle of the closest line) is closer to the previous vision angle or closer to the expected vision angle after executing the turn.

Another problem occurs when you try to evaluate `sense_body` results. If you have sent a `dash` in the current cycle, the `sense_body` answer already reflects the reduced stamina value; the number of dashes is also already increased. But the speed is not updated until the end of the cycle, so you still get the old value.

C.2.3 Tracking Down Lost Commands

If your agents aren't responding as well as you think they should, one thing to check is whether or not the server is executing all the commands your agents are sending. You can check this by seeing if the counts of turns, dashes, and kicks returned in response to sense_body commands match up with the number of commands actually sent.

Under normal network conditions, the server should be able to execute all commands if only one is sent every simulator cycle. Therefore, you may want to have your agents print out error messages if several messages are being missed. You will then know to check the network or to slow down the simulator (see Section C.3.1).

While you may consider re-sending messages that have been missed, it is difficult to do so in a reliable and sensible way. Instead, we recommend slowing down the simulator, running fewer players, or moving to a faster network.

C.3 Debugging

Debugging a client in a pseudo-real-time environment is difficult. There are different approaches. Probably you will have to combine several. Don't underestimate the debugging time in your project time plan.

C.3.1 Slowing Down Server and Clients

If you don't have the resources to run all clients and the server at full speed, you can train under accurate conditions by slowing down the server and clients. For example, by slowing down the simulation to half the normal speed, we were able to run the server, the monitor, and all 22 clients on a single 266 MHz Pentium without having any commands missed. If you slow down the server, be sure to slow down the clients as well.

To slow down the server, you should change two parameters in server.conf: `simulator_step` and `send_step`. Be sure to keep them in the same 2:3 ratio. For example, to run the simulator at half-speed, set `simulator_step` to 200 and `send_step` to 300. If you still want your match to have the same number of cycles, you also have to increase the duration of the match in seconds as given in the `half_time` parameter accordingly.

If you run the server significantly slower than real-time, you may want to record your trials while you are away from the computer and then play them back at full speed using the logplayer program.

A convenient way to ensure that the simulator and the clients are running at the same speed is to have them both read the same configuration file (see Section C.4.5).

C.3.2 Usage of the Log Player for Debugging

Use the logplayer to play back sequences and see, cycle by cycle, the actions taken. Particularly kicks and catches can be noticed. Every kick is shown by changing the border of the circle that represents the kicking player. A catch changes the color of the whole circle.

You can make the soccerserver create a log by adding the lines

```
record_log: off
record: game.rec
```

to the config file of the server. The first line means that the server should not include the log info in the record of the game. The second line defines the name under which the game record is saved.

Output time-stamped debugging info of your agents to compare with what you see on the logplayer. Be aware that writing a lot of debugging info to disk may slow down your agents so much that they can't keep up with the server. You may have to slow down the server and/or reduce the amount of debug info your agents produce. Also, you might just turn on debugging for some or even just one agent of your team.

C.3.3 Organization of Log Files

When debugging, it is often useful to know exactly what sensor information was received and actuator commands were sent by each client. A convenient way to do that is to have each client open a file named according to its team name and uniform number. It can then write its input and output strings to the file as it goes.

This information is especially useful when stepping through a recorded sequence one cycle at a time. However, take care to turn this function off when you want to run at maximum speed: all the file writing can slow things down significantly.

For an example of clients that save all of their input and output, see the CMUnited-98 team available at `http://www.cs.cmu.edu/~pstone/RoboCup/CMUnited98-sim.html`. If clients are started with the flag -save_log, they will save their input and output to files.

C.4 Software Technology

The development of a reasonably successful RoboCup-Agent is a considerable software project. For example, the team AT Humboldt 98 has more than 20,000 lines of code. Getting any program of that size to work within a given time span is a non-trivial task. Having someone on the team that has at least some software construction experience is extremely helpful.

Besides experience, the use of certain tools and practices can improve the efficiency of your development process tremendously. This section is a loose collection of topics related to the general software development aspect of creating a RoboCup agent.

C.4.1 Sourcecode Management

Unless you have a lot (more than one year) of time, you will probably want to develop the agent as a group. This means that you have to coordinate concurrent development between more several developers. A source code management system is very helpful here.

There are several non-commercial source code management systems available. During the development of AT Humboldt 98 the system CVS was used. It is much less restrictive than the system RCS it is based on. For example, it allows several developers to edit the same project file and merges the changes. For more information, see the CVS manual. Another such system is PRCS. It is more tailored towards smaller projects and is simpler to use. One the other hand it is newer and less thoroughly tested than CVS.

The CVS manual can be found at `http://www.loria.fr/~molli/cvs/doc/cvs_toc.html`.

The PRCS home page is located at `http://www.XCF.Berkeley.EDU/~jmacd/prcs.html`.

C.4.2 Useful Resources

Even if this is not your first software development project, you might find some of these pointers helpful

- **GDB** is THE source-level debugger for C and C++ code. Use it to step through your program, find out why it core-dumped, or attach it to a running process to find the endless loop where it got stuck. Gdb should be part of most UNIX installations. You can download it from one of the many FTP sites for GNU software; a list of these sites can be found at `http://www.gnu.org/order/ftp.html`. An online-manual is available, for example at `http://www.cs.tu-bs.de/softech/info/gdb_toc.html`.

- **DDD** (Data Display Debugger) is a graphical front-end to several source-level debuggers, among them gdb. Move the mouse over a variable of your source-code and DDD displays its current value. More information can be found on the DDD homepage at `http://www.cs.tu-bs.de/softech/ddd/`.
- **Code Complete** is THE practical guide to software development. It is full of hints to write elegant, self-documenting, maintainable code. Especially the sections "Naming Data" and "Layout and Style" can be extremely helpful.
  Code Complete: A Practical Handbook of Software Construction, Steve McConnell, Microsoft Press, 1993. Reviews can be found e.g. at `http://www.amazon.com` (search for "code complete").

C.4.3 Quitting Clients Automatically

Although it's not so hard to write a script to kill your clients when you are done with a run, you may find it more convenient if they exit automatically when the server is killed. You can make this happen by keeping track of how many timed action cycles go by between sensor information messages from the server. If a client hasn't heard from the server in several seconds, it is probably safe to exit.

C.4.4 Getting Communication Right

Communication between the client and the server is done via UDP/IP, the so-called connectionless IP protocol. This means that data is exchanged in individual packets, and the protocol doesn't guarantee that the packets arrive in the order they are sent or that they arrive at all.

In order to send data to the server, you have to create a communication end point, called socket. The system gives you a number, the socket descriptor, that you use from then on to work with the socket. You have to use the following system functions to set up a UDP communication in the UNIX environment:

1. *gethostbyname*, *inet_addr()*, and *inet_ntoa* are Internet address manipulation and lookup functions that you use to initialize the client and the host address structures.
2. *socket()* creates a new socket and returns the socket descriptor.
3. *setsockopt* sets socket characteristics such as the receive buffer size.
4. *bind()* connects the socket to a port of the local machine.
5. *sendto()* is used to send data to a host.
6. *select()* can be used to check if data from the server is available at the socket.
7. *recvfrom()* is used to receive the data.
8. *close()* is used to close the socket when the agent terminates.

The detailed description of these commands can be found in the UNIX man pages or in a book about network programming such as [4].

For an implementation example look at the files `Communicator.h` and `Communicator.cc` of AT Humboldt 98, which are available at `http://www.ki.informatik.hu-berlin.de/RoboCup/RoboCup98/ATH98.tar.gz`.

C.4.5 Using `server.conf` in the Client

A good deal of the server's behavior is governed by its configuration file, `server.conf`. The speed of the simulation, the size of the player, the maximum ball speed, etc. are all server parameters.

Just as the server reads in the configuration file, your clients should read the very same file. In fact, you can use the very code from the server (the function Stadium::GetOption() in field.C). If you use these parameters rather than any hard-coded numbers, it will be easy for you to adapt your clients to different simulator conditions.

For example, it is tempting to write your clients such that they will kick the ball whenever it is within 1.885 meters. However, it will probably be worth the effort to write a BallIsKickable() function which returns TRUE when the ball is within `kickable_margin + ball_size + player_size` meters.

Not only will you be able to adjust to changes in the server defaults, but you will also be able to customize your clients to react differently when the offsides rule is not used, etc.

While it is generally good to parameterize your code, in this case it is particularly good to tie your client parameters to the server parameters.

Bibliography

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

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

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

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

[5] Peter Stone. *Layered Learning in Multi-Agent Systems.* PhD thesis, Carnegie Mellon University, December 1998.