Chapter 2

Substrate Systems

The innovations reported in this thesis are designed primarily for real-time, noisy, collaborative and adversarial domains. As such, simulated robotic soccer—the RoboCup soccer server [Noda *et al.* 98] in particular—has served as an ideal research test bed. However, the positive results achieved are not limited to this domain. Throughout the thesis, the extent to which each result generalizes is discussed. In addition, some of the techniques developed in simulated robotic soccer have been applied in two other domains with some similar characteristics: real robotic soccer and network routing. This chapter compares and contrasts these three domains while specifying their details as they are used for empirical testing.

2.1 Overview

As the main test bed, all the contributions of this thesis are originally developed and implemented in simulated robotic soccer. It is a fully *distributed*, multi-agent domain with both *teammates* and *adversaries*. There is *hidden state*, meaning that each agent has only a partial world view at any given moment. The agents also have *noisy sensors and actuators*, meaning that they do not perceive the world exactly as it is, nor can they affect the world exactly as intended. In addition, the perception and action cycles are *asynchronous*, prohibiting the traditional AI paradigm of using perceptual input to trigger actions. *Communication* opportunities are limited; the agents must make their decisions in *real-time*; and the actions taken by other agents, both teammates and adversaries, and their resulting state transitions are unknown. We refer to this last quality of unknown state transitions as *opaque transitions*. These italicized domain characteristics are appropriate for the development of all four thesis contributions as presented in Section 1.3.

In order to test the generality of our simulator results, we transfer some of our techniques to our real robot system. In particular, portions of the team member agent architecture are implemented in the real robot system as well as in simulation. The real robot system is a completely different domain from the simulator. First, at the most basic level, the I/O is entirely different. While the simulator deals with abstract, asynchronous perceptions and actions, the robotic system processes real-time video images via an overhead camera and outputs motor control commands synchronously (i.e. triggered by perception) via radio. Second, the agents all share the same perception of the world, which makes the robotic system not completely distributed. However, functionally the robots are controlled independently: each is controlled by an independent function call using a turn-taking methodology. They can also be controlled by separate processes with a common sensory input stream. Three other differences of the robots from the simulator are the absence of communication among teammates (which is possible, but not used in our system), the absence of hidden state (agents have a full world view via an overhead camera), and a resulting full knowledge about the state transitions in the world. These domain differences prevent the use of identical agent programs in the two domains, but they do not limit the applicability of the flexible teamwork structure.

While developed within the context of robotic soccer, the multi-agent algorithms presented in this thesis generalize beyond robotic soccer as well. To support this claim, we implement one of our algorithms in a different domain, namely network routing. We believe several other multi-agent domains are also similar to robotic soccer. It is part of our future work to continue to identify other such multi-agent domains (see Chapter 10).

Although network routing differs from robotic soccer in a lot of ways, in an abstract sense it is very similar. Even though our network routing simulator does not involve communication, noisy sensors and actuators, or adversaries, it retains the essential characteristics motivating the development of TPOT-RL: a distributed team of agents operating in real-time with opaque transitions. As reported in Chapter 7, TPOT-RL is successfully implemented in network routing as well as in robotic soccer.

Table 2.1 summarizes the domain comparison. The remainder of this chapter provides the domain specifications of simulated robotic soccer, real robotic soccer, and network routing as used experimentally in this thesis. I use this chapter to describe in detail the aspects of the domains that are not part of the thesis contributions: the substrates upon which the contributions are built.

	Simulator	Robots	Network routing
Distributed perception	yes	no	yes
Distributed action	yes	yes	yes
Asynchronous perception/action	yes	no	no
Teammates	yes	yes	yes
Adversaries	yes	yes	no
Hidden state	yes	no	yes
Noisy sensors	yes	yes	no
Noisy actuators	yes	yes	no
Communication	yes	no	no
Real-time	yes	yes	yes
Opaque transitions	yes	no	yes

Table 2.1: A comparison of the experimental domains used in this thesis.

2.2 The RoboCup Soccer Server

The RoboCup soccer server [Noda *et al.* 98] has been used as the basis for successful international competitions [RoboCup 97] and research challenges [Kitano *et al.* 97]. As one of the first users, I helped to test and tune it over the course of its development, and participated in its first test as the basis for a competition (Pre-RoboCup-96 at IROS-96). Experiments reported in this thesis are conducted in several different versions of the simulator ranging from version 1 to the current version 4. This section describes the current simulator.

The soccer server is a complex and realistic domain. Unlike many AI domains, the soccer server embraces as many real-world complexities as possible. It models a hypothetical robotic system, merging characteristics from different existing and planned systems as well as from human soccer players. The server's sensor and actuator noise models are motivated by typical robotic systems, while many other characteristics, such as limited stamina and vision, are motivated by human parameters.

In this section I describe version 4 of the soccer server in detail. While not as detailed as the soccer server user manual [Andre *et al.* 98a], this section defines all of the concepts and parameters that are important for the thesis.

Section 2.2.1 gives a high-level overview of the simulator. Section 2.2.2 defines the environment including the field and all of the objects recognized by the simulator. Sections 2.2.3 and 2.2.4 specify the perceptions and actions available to agents in the simulator. Section 2.2.5 gives a detailed illustrative trace of the interactions between the server and a client over a period of time. Section 2.2.6 summarizes the challenging characteristics of this simulated robotic soccer domain.

2.2.1 Overview of the Simulator

The simulator, acting as a server, provides a domain and supports users who wish to build their own *agents* (also referred to as *clients* or *players*). Client programs connect to the server via UDP sockets, each controlling a single player. The soccer server simulates the movements of all of the objects in the world, while each client acts as the brain of one player, sending *movement commands* to the server. The server causes the player being controlled by the client to execute the movement commands and sends sensory information from that player's perspective back to the client.

When a game is to be played, two teams of 11 independently controlled clients connect to the server. The object of each team is to direct the ball into one of the goals at the ends of the field, while preventing the ball from entering the other goal.

The server's parameters that are relevant to this thesis are listed, along with their default values and descriptions, in Table 2.4. Refer to this table as the parameters are alluded to in the text of this section.

The simulator includes a visualization tool, pictured in Figure 2.1. Each player is represented as a two-halved circle. The light side is the side towards which the player is facing. In Figure 2.1, all of the 22 players are facing the ball, which is in the middle of the field. The black bars on the left and right sides of the field are the goals.

The simulator also includes a *referee*, which enforces the rules of the game. It indicates

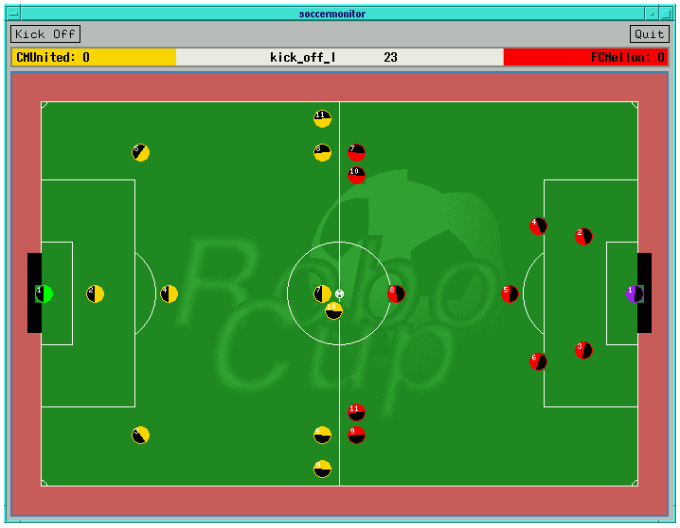


Figure 2.1: The soccer server display. Each player is represented as a two-halved circle. The light side is the side towards which the player is facing. All players are facing the ball, which is in the middle of the field. The black bars on the left and right sides of the field are the goals.

changes in play mode, such as when the ball goes out of bounds, when a goal is scored, or when the game ends. It also enforces the *offsides* rule. Like in real soccer, a player is offsides if it is in the opponent's half of the field and closer to the opponent's goal line (the line along which the goal is located) than all or all but one of the opponent players when the ball is passed to it. The crucial moment for an offsides call is when the ball is kicked, not when it is received: a player can be behind all of the opponent defenders when it receives a pass, but not when a teammate kicks the ball towards it[1].

One of the real-world complexities embraced by the soccer server is asynchronous sensing and acting. Most AI domains use synchronous sensing and acting: an agent senses the world, acts, senses the result, acts again, and so on. In this paradigm, sensations trigger actions. On the other hand, both people and complex robotic systems have independent sensing and acting rates. Sensory information arrives via different sensors at different rates, often unpredictably (e.g. sound). Meanwhile, multiple actions may be possible in between sensations or multiple sensations may arrive between action opportunities.

The soccer server uses a discrete action model: it collects player actions over the course of a fixed *simulator cycle* of length `simulator_step`, but only executes them and updates the world at the end of the cycle. If a client sends more than one movement command in a simulator cycle, the server chooses one randomly for execution. Thus, it is in each client's interest to try to send at most one movement command each simulator cycle. On the other hand, if a client sends no movement commands during a simulator cycle, it loses the opportunity to act during that cycle, which can be a significant disadvantage in a real-time adversarial domain: while the agent remains idle, opponents may gain an advantage. Each cycle, the simulator increments the simulated time counter by one.

[1] In real soccer, an offsides call is actually subjective based on the referee's opinion of whether the player in an offsides position is gaining an advantage by being there. The soccer server artificially operationalizes the offsides rule, making it an objective call.

Figure 2.2 illustrates the interactions between the server and two clients over the course of 3 simulator cycles at times t-1, t, and t+1. The thick central horizontal line represents the real time as kept by the server. The top and bottom horizontal lines represent the time-lines of two separate clients. Since they are separate processes, they do not know precisely when the simulator changes from one cycle to the next. The dashed arrows from the server towards a client represent perceptions for that client. The solid arrows from a client towards the server represent movement commands sent by that client. These arrows end at the point in time at which the server executes the movement commands, namely the end of the simulator cycle during which they are sent.

Client 1 sends movement commands after every perception it receives. This strategy works out fine in cycle t-1; but in cycle t it misses the opportunity to act because it receives no perceptions; and in cycle t+1 it sends two movement commands, only one of which will be executed.

Client 2, on the other hand, successfully sends one movement command every cycle. Note that in cycle t it must act with no new perceptual information, while in cycle t+1, it receives two perceptions prior to acting and one afterwards. Ideally, it would act after receiving and taking into account all three perceptions. However, it does not know precisely when the simulator cycle will change internally in the soccer server and it cannot know ahead of time when it will receive perceptions. Thus, in exchange for the ability to act every simulator cycle, it sometimes acts with less than the maximal amount of information about the world. However, as each simulator cycle represents only a short amount of real time (`simulator_step` msec), the world does not change very much from cycle to cycle, and the client can act accurately even if it takes some of its perceptions into account only before its *subsequent* action.

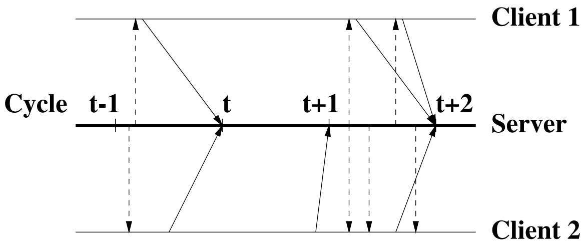


Figure 2.2: A sample period of the server-client interface over the course of 3 simulator cycles at times t-1, t, and t+1. The thick central horizontal line represents the real time as kept by the server. The top and bottom horizontal lines represent the time-lines of two separate clients. The dashed arrows from the server towards a client represent perceptions for that client. The solid arrows from a client towards the server represent movement commands sent by that client. These arrows end at the point in time at which the server executes the movement commands, namely the end of the simulator cycle during which they are sent.

Asynchronous sensing and acting, especially when the sensing can happen at unpredictable intervals, is a very challenging paradigm for agents to handle. Agents must balance the need to act regularly and as quickly as possible with the need to gather information about the environment. Along with asynchronous sensing and action, the soccer server captures several other real-world complexities, as will become evident throughout the remainder of this section.

2.2.2 The Simulated World

This section paraphrases a similar section in the soccer server user manual [Andre *et al.* 98a].

Field and Objects

The soccer server is a two-dimensional simulation. There is no notion of height for any object on the field. The field has dimensions `field_length` $\times$ `field_width` with the goals of width `goal_width`, all of which are parameters defined in Table 2.4. There are up to 22 players on the field at a time along with 1 ball, all of which are modeled as circles. There are also several visible markers, including flags and side lines, distributed around the field as illustrated in Figure 2.3.

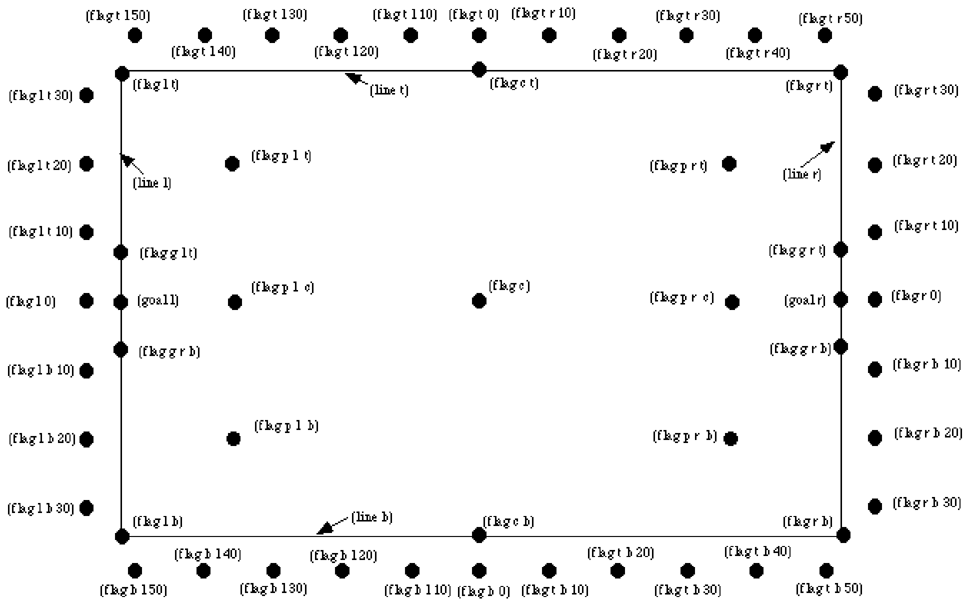


Figure 2.3: The locations and names of the visible markers in the soccer server. This figure originally appeared in the soccer server user manual [Andre *et al.* 98a]

The players and the ball are all mobile objects. At the simulator cycle for time t, each object is in a specific position (p_x^t, p_y^t) and has a specific velocity (v_x^t, v_y^t). Each player is also facing in a specific direction θ^t. These positions are maintained internally as floating point numbers, but player sensations are only given to one decimal place. Thus, the perceived state space with the server parameters we use has more than $10^{9^{22}} = 10^{198}$ states: each of the 22 players can be in any of 680 $\times$ 1050 $\times$ 3600 positions. Taking into account the ball, velocities, and past states, the actual state space is much larger than that.

Object Movement

As described in the user manual:

> In each simulation step, the movement of each object is calculated in the following

manner:

$$\begin{aligned}
(u_x^{t+1}, u_y^{t+1}) &= (v_x^t, v_y^t) + (a_x^t, a_y^t) : \textit{accelerate} \\
(p_x^{t+1}, p_y^{t+1}) &= (p_x^t, p_y^t) + (u_x^{t+1}, u_y^{t+1}) : \textit{move} \\
(v_x^{t+1}, v_y^{t+1}) &= \text{decay} \times (u_x^{t+1}, u_y^{t+1}) : \textit{decay speed} \\
(a_x^{t+1}, a_y^{t+1}) &= (0, 0) : \textit{reset acceleration}
\end{aligned} \tag{2.1}$$

where, decay is a decay parameter specified by `ball_decay` or `player_decay`. (a_x^t, a_y^t) is the object's acceleration, which is derived from the *Power* parameter in `dash` (if the object is a player) or `kick` (if a ball) commands as follows:

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

In the case of a player, its direction is calculated in the following manner:

$$\theta^t = \theta^t + \textit{Angle}$$

where *Angle* is the parameter of `turn` commands. In the case of a ball, its direction is given in the following manner:

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

where θ^t_{ball} and θ^t_{kicker} are the directions of the ball and kicking player respectively, and *Direction* is the second parameter of a `kick` command [Andre *et al.* 98a].

The commands and their parameters referred to above are specified precisely in Section 2.2.4.

Collisions

As described in the user manual:

> 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 [Andre *et al.* 98a].

Noise

The soccer server adds evenly distributed probabilistic noise to all objects' movements. In particular, as described in the manual:

> noise is added into Equation 2.1 as follows:
>
> $$(u_x^{t+1}, u_y^{t+1}) = (v_x^t, v_y^t) + (a_x^t, a_y^t) + (\tilde{r}\text{rmax}, \tilde{r}\text{rmax})$$

where $\tilde{r}_{\mathrm{max}}$ is a random number whose distribution is uniform in the range $[-\mathrm{max}, \mathrm{max}]$. rmax is a parameter that depends on the 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 a *Power* parameter in a `turn` command as follows:

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

[Andre *et al.* 98a]

Stamina

The soccer server prevents players from constantly running at maximum velocity (`player_sp_max`) by assigning each player a limited stamina. The stamina is modeled in three parts.

- `stamina` $\in$ [0, `stamina_max`] is the actual limit of the *Power* parameter of a `dash` command.
- `effort` $\in$ [`effort_min`, 1.0] represents the efficiency with which the player can move.
- `recovery` $\in$ [`recovery_min`, 1.0] represents the rate at which stamina is replenished.

A player's stamina has both replenishable and non-replenishable components. The replenishable components are `stamina` and `effort`; `recovery` is non-replenishable.

The stamina parameters listed in Table 2.4 are used as follows:

- If a player tries to dash with power *Power*, the effective power of its dash is affected by `stamina` and `effort` while `stamina` is subsequently decremented:
  - `effective_dash_power` = Min(`stamina`,*Power*) × `effort`.
  - `stamina` = `stamina` − `effective_dash_power`.
- On every cycle, `effort` is decremented if `stamina` is below a threshold, and incremented if it is above a threshold:
  - if `stamina` ≤ `effort_dec_thr` × `stamina_max` and `effort` > `effort_min`, then `effort` = `effort` − `effort_dec`.
  - if `stamina` ≥ `effort_inc_thr` × `stamina_max` and `effort` < `1.0`, then `effort` = `effort` + `effort_inc`.
- On every cycle, `recovery` is decremented if `stamina` is below a threshold:
  - if `stamina` ≤ `recover_dec_thr` × `stamina_max` and `recovery` > `recover_min`, then `recovery` = `recovery` − `recover_dec`.

- On every cycle, `stamina` is incremented based on the current value of `recovery`:
  - `stamina` = `stamina` + `recovery` × `stamina_inc`.
  - if `stamina` > `stamina_max`, then `stamina` = `stamina_max`.

2.2.3 Agent Perception

Agents receive three different types of sensory perceptions from the server: aural, visual, and physical. This section describes the characteristics of the sensory information, which are summarized in Table 2.2.

Aural information

The soccer server communication paradigm models a crowded, low-bandwidth environment. All 22 agents (11 on each team) use a single, unreliable communication channel. When an agent or the referee speaks, nearby agents on both teams can hear the message immediately. There is no perceptual delay. They hear a message in the format (`hear` *Time Direction Message*) where:

- *Time* is the current simulator cycle;
- *Direction* is the relative direction from whence the sound came;
- *Message* is the message content;

Note that there is no information about which player sent the message or that player's distance.

Agents have a limited communication range, hearing only messages spoken from within a distance specified by the parameter `audio_cut_off_dist`. They also have a limited communication capacity, hearing a maximum of `hear_inc` messages in `hear_decay` simulation cycles. Thus communication is extremely unreliable. Using the parameters in Table 2.4, if more than 1 agent speaks over the course of 2 simulation cycles, a player will miss hearing all but 1 of the messages. Messages from the referee are treated as privileged and are always transmitted to all agents.

In short, the characteristics of the low-bandwidth communication environment include:

- All 22 agents (including adversaries) on the same channel;
- Limited communication range and capacity;
- No guarantee of sounds getting through;
- Immediate communication: no perceptual delay.

Visual information

As described in the user manual:

> Visual information arrives from the server in the following basic format:
>
> (`see` *Time ObjInfo ObjInfo ...*)
> *Time* indicates the current time.
> *ObjInfo* is information about a visible object in the following format:
> (*ObjName Distance Direction DistChng DirChng FaceDir*)

```
ObjName = (player Teamname UNum)
        | (goal Side)
        | (ball)
        | (flag c)
        | (flag [l|c|r] [t|b])
        | (flag p [l|r] [t|c|b])
        | (flag [t|b] [l|r] [10|20|30|40|50])
        | (flag [l|r] [t|b] [10|20|30])
        | (flag [l|r|t|b] 0)
        | (line [l|r|t|b])
```

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

$$
\begin{aligned}
p_{rx} &= p_{xt} - p_{xo} \\
p_{ry} &= p_{yt} - p_{yo} \\
v_{rx} &= v_{xt} - p_{xo} \\
v_{ry} &= v_{yt} - p_{yo} \\
\mathit{Distance} &= \sqrt{p_{rx}^2 + p_{ry}^2} \\
\mathit{Direction} &= \arctan\left(p_{ry}/p_{rx}\right) - a_o \\
e_{rx} &= p_{rx}/\mathit{Distance} \\
e_{ry} &= p_{ry}/\mathit{Distance} \\
\mathit{DistChng} &= (v_{rx} * e_{rx}) + (v_{ry} * e_{ry}) \\
\mathit{DirChng} &= [(-(v_{rx} * e_{ry}) + (v_{ry} * e_{rx}))/\mathit{Distance}] * (180/\pi)
\end{aligned}
$$

> where (p_{xt}, p_{yt}) is the absolute position of a 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. In addition, (p_{rx}, p_{ry}) and (v_{rx}, v_{ry}) are respectively the relative position and the velocity of the target, and (e_{rx}, e_{ry}) is the unit vector that is parallel to the vector of the relative position. *Facedir* is only included if the observed object is another player, and is the direction of the observed player relative to the direction of the observing player. Thus, if both players are facing the same direction, then *FaceDir* would be 0.

> The (goal r) object is interpreted as the center of the goal. (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 inside corner of the penalty box on the left side of the field. 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 there the center line of the player's view crosses the line, and *Direction* is the direction of line.
>
> All of the flags and lines are shown in Figure 2.3. [Andre *et al.* 98a]

The frequency, range, and quality of visual information sent to a particular agent are governed by the integer server parameters `send_step` and `visible_angle`; and the player parameters `view_quality` $\in$ {*high*,*low*}, `view_width` $\in$ {*narrow*, *normal*,*wide*}, and the floating point number `view_angle`. A player can directly control its `view_quality` and `view_width` via its actuators. By default, `view_quality` = *high*, `view_width` = *normal* and `view_angle` = `visible_angle`. In that case the agent receives visual information every `send_step` msec.

However, the agent can trade off the frequency of the visual information against its quality and width via its `view_mode` = {`view_quality`,`view_width`}. By using *low* `view_quality`—in which case it receives angle information, but no distances to object—the agent can get sights twice as frequently: every `send_step`/2. Similarly, the agent can get sights twice as frequently by switching to a *narrow* `view_width`, in which case `view_angle` = `visible_angle`/2. On the other hand, the player can use a *wide* `view_width` (`view_angle` = `visible_angle` $\times$ 2) at the expense of less frequent visual information (every `send_step` $\times$ 2 msec). With both `view_width` = *narrow* and `view_quality` = *low*, visual information arrives every `send_step`/4 msec.

The meaning of the `view_angle` parameter is illustrated in Figure 2.4. In this figure, the viewing agent is the one shown as two semi-circles. The light semi-circle is its front. The black circles represent objects in the world. Only objects within `view_angle`^{o}/2, and those within `visible_distance` of the viewing agent can be seen. Thus, objects b an 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 at roughly -40^o, while object d is at roughly 20^o.

Also illustrated in Figure 2.4, the amount of information describing a player varies with how far away the player is. For nearby players, both the team and the uniform number of the player are reported. However, as distance increases, first the likelihood that the uniform number is visible decreases, and then even the team name may not be visible. It is assumed in the server that `unum_far_length` $\leq$ `unum_too_far_length` $\leq$ `team_far_length` $\leq$ `team_too_far_length`. Let the player's distance be *dist*. Then

- If *dist* $\leq$ `unum_far_length`, then both uniform number and team name are visible.

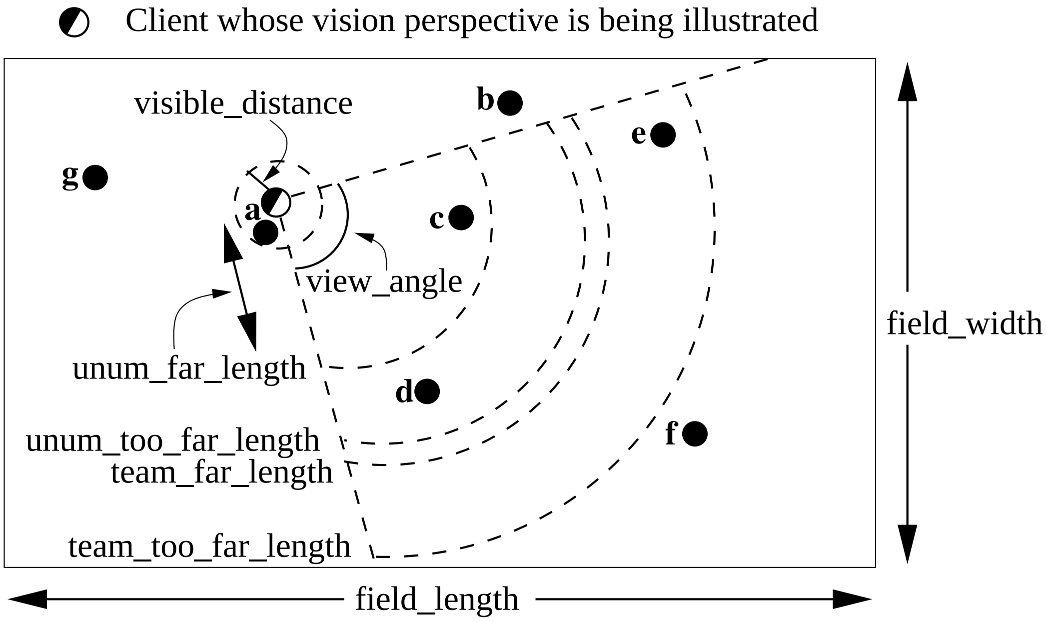


Figure 2.4: The visible range of an individual agent in the soccer server. The viewing agent is the one shown as two semi-circles. The light semi-circle is its front. The black circles represent objects in the world. Only objects within `view_angle`$^o/2$, and those within `visible_distance` of the viewing 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.

- If `unum_far_length` $<$ *dist* $<$ `unum_too_far_length`, then the team name is always visible, but the probability that the uniform number is visible decreases linearly from 1 to 0 as *dist* increases.

- If *dist* $\geq$ `unum_too_far_length`, then the uniform number is not visible.

- If *dist* $\leq$ `team_far_length`, then the team name is visible.

- If `team_far_length` $<$ *dist* $<$ `team_too_far_length`, then the probability that the team name is visible decreases linearly from 1 to 0 as *dist* increases.

- If *dist* $\geq$ `team_too_far_length`, then neither the team name nor the uniform number is visible.

For example, in Figure 2.4, assume that all of the labeled black 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.

Finally, the precision of information regarding an object's distance decreases with its distance. As written in the user manual:

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:

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

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

$$\text{Quantize}(V, Q) = \text{rint}(V/Q) \cdot Q$$

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

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

[Andre *et al.* 98a]

This visual paradigm creates a huge amount of hidden state for each individual agent. In addition, with the parameters identified in Table 2.4, by default, the agents receive visual information less frequently (every 150 msec) than they can act (every 100 msec). Thus, unless they are to miss many action opportunities, the agents must sometimes choose more than one sequential action from a given state.

With these parameters, play occurs in *real time*: the agents must react to their sensory inputs at roughly the same speed as human or robotic soccer players. Our own robots have been able to act between 3 and 30 times per second. Human reaction times have been measured in the 200-500 msec range [Woodworth 38].

Physical information

Upon request from an agent, the server also sends physical information about the agent. Such `sense_body` information includes:

- the agent's current `stamina`, `effort`, and `recovery` values;
- the agent's current speed;
- the agent's current `view_quality` and `view_width` values.

The agent's speed is given as a magnitude only: it must infer the direction of its movement from visual cues.

Table 2.2 summarizes types of sensory information available to soccer server agents.

2.2.4 Agent Action

Agents can send several different types of commands to the simulator as their actuators: they change the world in some way. This section describes the characteristics of the different commands which are summarized in Table 2.3.

Name	When received	Information type	Limitations
see	every `send_step` msec (default)	visual	limited angle, precision decreases with distance
hear	instantaneously when a nearby agent (or the referee) speaks	aural	limited distance and frequency
sense_body	upon request	physical	none

Table 2.2: The soccer server agents' sensors.

Communication

Agents can "say" any text string up to 512 ascii characters in length. Both teammates and opponents will hear the complete message subject to the range and frequency constraints described in Section 2.2.3.

An agent can speak as often as it wants. But since teammates can only hear one message every 2 cycles, it is useless to speak more frequently than that.

Movement

The agent has just four actuators for physically manipulating the world: `turn`, `dash`, `kick`, and `catch`. The server only executes one of these commands for each player at the end of each simulator cycle. If an agent sends more than one such command during the same cycle, only one is executed non-deterministically. Since the simulator runs asynchronously from the agents, there is no way to keep perfect time with the server's cycle. Therefore any given command could be missed by the server. It is up to the agent to determine whether a given command has been executed by observing the future state of the world.

The movement commands are all parameterized with real number arguments indicating the *Angle* and/or *Power* associated with the action. As always, angles are relative to the direction the agent is currently facing (θ^t).

The details of the movement commands are as follows:

Turn(*Angle*): -180 $\leq$ *Angle* $\leq$ 180 indicates the angle of the turn. Rotation efficiency is reduced for moving players. That is, the higher a player's speed, the smaller a fraction of *Angle* the agent will actually turn according to the equation:

$$\text{actual\_angle} = \mathit{Angle}/(1.0 + \texttt{inertia\_moment} * \texttt{player\_speed})$$

Dash(*Power*): -30 $\leq$ *Power* $\leq$ 100 indicates the power of the dash. Agents only move in the direction they are facing, either forwards or backwards. To move in another direction, they must first turn. Dashing efficiency is reduced for players with low stamina as indicated in Section 2.2.2. A dash sets the acceleration in Equation 2.1 for one simulation cycle. It does not cause a sustained run over time. In order to keep running, the agent must keep issuing dash commands. *Power* is multiplied by the server parameter `dash_power_rate` before being applied in Equation 2.1.

Kick(*Power*, *Angle*): $0 \leq Power \leq 100$ indicates the power of the kick, $-180 \leq Angle \leq 180$, the angle at which the ball is accelerated. Like the dash, the kick sets the ball's acceleration in Equation 2.1. There are four points about the kick model of the server that should be understood.

- A kick changes the ball's velocity by vector addition. That is, a kick accelerates the ball in a given direction, as opposed to setting the velocity. Thus multiple kicks on successive cycles can move the ball faster than a single kick.
- An agent can kick the ball when it is within `kickable_area` which is defined as `ball_size` + `player_size` + `kickable_margin`). The ball being within `kickable_area` is a precondition for successfully executing a kick.
- The ball and the player can collide as indicated in Section 2.2.2. Thus in order to smoothly kick the ball, a player must take care to avoid kicking the ball into itself.
- The actual power with which the ball is kicked depends on its relative location to the player. As presented in the user manual:

  > 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:
  >
  > `kick_power_rate` * (1 - .25 * `dir_diff` / 180 -
  > .25 * (`dist_ball` - `player_size` - `ball_size`) / `kickable_area`)

  This number is then multiplied by the *Power* parameter passed to the kick command before applying Equation 2.1.

  According to this paradigm, the strongest kick occurs when the ball is as close as possible to the player and directly in front of it. The power decreases as distance and/or angle increases. [Andre *et al.* 98a]

Catch(*Angle*): $-180 \leq Angle \leq 180$ indicates the angle of the catch. Only the goaltender can catch, and only within its own penalty box. These conditions are preconditions of the catch command. A catch is only effective if the ball is within a rectangle of size `catchable_area_w` × `catchable_area_l` with one edge on the player and extending in the direction indicated by *Angle*. The ball being in such a position is another precondition for successfully executing a catch.

Perception control

The two remaining commands available to agents directly control the type of perception information they receive from the server. The `sense_body` command requests physical information from the server. The `change_view` command specifies a `view_quality` (low/high) and a `view_width` (narrow/normal/wide). As described in Section 2.2.3, higher quality and greater width lead to less frequent visual information.

The `sense_body` command can be executed up to 3 times per simulator cycle (by convention), while `change_view` can be executed once per cycle.

Table 2.3 summarizes all of the actuators available to agents in the soccer server. The types and ranges of the arguments are shown along with when the commands are executed by the server. In the table, "cycle" refers to a simulation cycle of length `simulator_step`. As described in the text, only `kick` and `catch` have preconditions.

Syntax	Argument meaning	Type	Range	When executed	Frequency limit
say(x)	message to be broadcast	ascii text	$\leq$ 512 characters	instantly	teammates only hear 1 every 2 cycles
turn(x)	angle to turn	float	-180 $\leq x \leq$ 180	end of cycle	1 of these per cycle
dash(x)	power to dash	float	-30 $\leq x \leq$ 100	end of cycle	
kick(x, y)	power to kick, angle to kick	float float	0 $\leq x \leq$ 100, -180 $\leq y \leq$ 180	end of cycle	
catch(x)	angle to catch	float	-180 $\leq x \leq$ 180	end of cycle	
sense_body()				instantly	3 per cycle
change_view(x, y)	view quality view width	discrete discrete	high/low narrow/normal/wide	instantly	1 per cycle

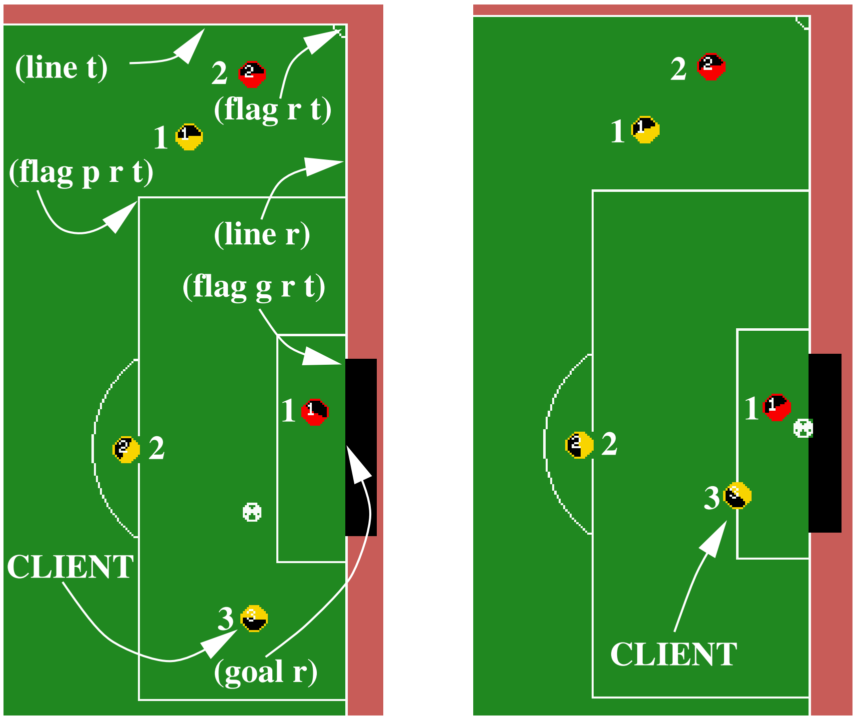
Table 2.3: The soccer server agents' commands. In the table, "cycle" refers to a simulation cycle of length `simulator_step`.

2.2.5 Sample Trace

Figure 2.5 illustrates the format of the interface between the server and a specific client. In the pictures, the white arrow and text are added for reference purposes: they are not part of the actual display. The input/output trace below the pictures is given from the perspective of the client labeled "CLIENT."

In Figure 2.5(a) the client is about to run to the ball and kick it towards the goal. Figure 2.5(b) shows the resulting scene after the kick. In the input/output trace below the pictures, as specified earlier in this section, dashes are followed by a power (they are always in the direction that the player is facing), turns are followed by an angle, and kicks are followed by a power and an angle. Sensory information from the server comes in the form of "hear," "see," and "sense_body" strings representing aural, visual, and physical information respectively. In all cases, the number after the type indicator ("hear," "see", or "sense_body") indicates the elapsed time in the match. Aural information then indicates whether it is the referee speaking or else from what angle the sound came. Visual information includes the distances followed by angles of the visible objects. If an object is close enough, two additional numbers indicate the object's relative velocity. In the case of other players, a fifth number indicates the relative direction that the player is facing. Sense_body information indicates the client's physical parameters.

Since the client's vision is limited to 45^o on either side of the direction it is facing (the black semi-circle is the back of the player, the light semi-circle is the front), not all objects are visible at each sensory step. For example, at the beginning of the trace in Figure 2.5 (time 94), the client sees two teammates and two opponents (one of which is indicated by "`player opponent 1`"). However by the end of the trace (time 112), it is no longer able to



(a) (b)

```
**-> (dash 100.00)
(see 94 ((goal r) 15.3 27) ((flag r t) 47.9 8) ((flag p r t) 34.8 -15) ((flag p r c) 16.4 -34 0 0)
   ((flag g r t) 21.8 19) ((ball) 8.2 0 0 0) ((player CMUnited) 40.4 -8)
   ((player CMUnited 2) 16.4 -37 0 0 117) ((player Opponent 1) 16.4 15 0 0 -148)
   ((player Opponent) 44.7 0) ((line t) 47.5 89))
**-> (sense_body)
(sense_body 95 (view_mode high normal) (stamina 1280 1) (effort 1.0) (recovery 1.0) (speed 0.39))
*-> (dash 100.00)
(see 96 ((goal r) 13.6 31) ((flag r t) 46.1 8) ((flag p r t) 33.1 -16) ((flag p r c) 14.9 -39 -0.298 -0.9)
   ((flag g r t) 19.9 20 -0.398 0.5) ((ball) 6.7 -2 -0.402 0) ((player CMUnited) 36.6 -8)
   ((player CMUnited 2) 14.9 -41 -0.298 -0.9 117) ((player Opponent 1) 14.9 17 -0.298 0.5 -148)
   ((player Opponent) 40.4 0) ((line t) 45.6 89))
**-> (dash 100.00)
**-> (sense_body)
(sense_body 97 (view_mode high normal) (stamina 1120 1) (effort 1.0) (recovery 1.0) (speed 0.44))
**-> (dash 100.00)
⋮
(hear 103 -70 shoot the ball) (see 104 ...((ball) 1.8 6 0.108 5.4) ...)
**-> (say shooting now)
**-> (kick 100.00 65.40)
(hear 104 self shooting now)
**-> (sense_body)
(sense_body 105 (view_mode high normal) (stamina 980 1) (effort 1.0) (recovery 1.0) (speed 0))
**-> (turn 31.76)
(see 106 ...((ball) 4.1 14 1.23 7) ...)
**-> (turn 14.00)
⋮
**-> (dash 100.00)
(see 112 ((goal r) 6.8 12) ((flag r t) 38.5 -32) ((flag g r t) 12.3 -14 -0.246 0) ((ball) 7.4 2 0.74 1.5)
   ((player Opponent 1) 7.4 -18 -0.148 -0.2 107) ((player Opponent) 33.1 -44) ((line r) 8.2 -40))
(hear 113 referee goal_l_1)
(hear 113 referee kick_off_r)
```

Figure 2.5: A trace of the soccer server's input and output to the client controlling player 3 (indicated "CLIENT"). The player starts in the position shown in (a), moves to the ball and shoots it towards the goal, ending in the position shown in (b). Commands from the player are indicated with "**->" preceding them.

see the teammates. Similarly, by the end of the trace, it is no longer able to see the center of the penalty area: (flag p r c).

The method of communication is illustrated by the message from teammate number 2 that is heard at time 103 ("shoot the ball"), and by the spoken response "shooting now." Two messages from the referee at time 113, indicating the successful goal and the subsequent restart, are also present at the end of the trace. The capacity limits on hearing do not apply to messages from the referee.

Both the sensors and the actions in the simulator are noisy. Notice that even though the player begins by facing directly at the stationary ball (the ball's angle is 0) and dashes straight toward it, the ball does not remain directly in front of the player: in the subsequent visual string, the ball's angle is -2. Also notice that the nearby players (CMUnited 2 and Opponent 1) are identified by team and number. The players that are farther away are only identified by team name. Although not apparent from this trace, when players are far enough away, even their team may not be visible.

The trace in Figure 2.5 begins at elapsed time 94 and continues through 113. Each time increment occurs in 100 msec of real time and visual sensor information arrives every 150 msec. The entire trace, from the moment pictured in Figure 2.5(a) until the ball enters the goal in Figure 2.5(b), occurs in about 2 seconds.

2.2.6 Summary

The parameters governing the operation of the soccer server as referred to in this section are summarized in Table 2.4. Their default values are also given.

All of the simulator features described in this section combine to make it a very challenging and realistic environment in which to conduct research. Table 2.5 summarizes the challenges that agents must face.

2.3 The CMUnited-97 Real Robots

While originating from similar motivations, robotic soccer with real robots is quite a different domain than the soccer server. Robotic soccer with real robots is the subject of an entirely separate research challenge from the simulator challenge [Asada *et al.* 98]. Whereas the soccer server provides an abstract sensor and actuator interface, the real robots must perceive the world via real NTSC video images, and they move by controlling the speeds of real motors. Therefore the sources and types of noise differ from the evenly distributed, probabilistic models incorporated into most simulations (including the soccer server). Although it is quicker and easier to get to the point of creating basic behaviors in simulation[2], it is also easy to ignore real-world complexities that may be abstracted away by the simulator. As presented in Chapter 3, the robots are used in this thesis to validate the team member agent architecture which was first implemented in the soccer server.

[2] After downloading the soccer server, it took about 3 days to create a simple go-to-the-ball-and-shoot behavior. It took at least 6 months of development before we were able to create such a behavior using the real robots.

Parameter Name	Default Value	Explanation
field_width	68	Width of the field.
field_length	105	Length of the field.
goal_width	14.02	Width of the goal.
simulator_step	100	Milliseconds in each simulation cycle.
unum_far_length	20	Distance below which players' uniform numbers and the ball's velocity are always visible.
unum_too_far_length	40	Distance above which players' uniform numbers and the ball's velocity are never visible.
team_far_length	40	Distance below which players' uniform colors are always visible.
team_too_far_length	60	Distance above which players' uniform colors are never visible.
player_size	0.8	Radius of a player.
player_decay	0.4	Decay rate of player speed. If this is 1.0, a player keeps its speed, and if this is 0.0, a player loses all of its speed in one simulation cycle.
player_rand	0.1	Amount of noise added in player's movements and turns.
player_speed_max	1.0	Maximum speed of a player during 1 simulation cycle (i.e. the player can achieve a maximum speed of 10 m/sec when `simulator_step`=100).
ball_size	0.085	Radius of the ball.
ball_decay	0.96	Decay rate of the ball speed.
ball_rand	0.05	Amount of noise added in the movement of the ball.
ball_speed_max	2.7	Maximum speed of the ball during 1 simulation cycle (i.e. the ball can achieve a maximum speed of 27 m/sec when `simulator_step`=100).
stamina_max	2000.0	Maximum stamina of a player.
stamina_inc_max	20.0	Amount of stamina that a player gains in a simulation cycle.
recover_dec_thr	0.3	Decrement threshold for player's recovery.
recover_dec	0.0002	Decrement step for player's recovery.
recover_min	0.5	Minimum player recovery.
effort_dec_thr	0.3	Decrement threshold for player's effort capacity.
effort_dec	0.005	Decrement step for player's effort capacity.
effort_inc_thr	0.6	Increment threshold for player's effort capacity.
effort_inc	0.01	Increment step for player's effort capacity.
effort_min	0.6	Minimum value for player's effort capacity.
audio_cut_dist	50.0	Maximum distance a message said by a player can reach.
hear_max	2	Maximum hearing capacity of a player. A player can hear N(= `hear_inc`) messages in M(= `hear_decay`) simulation cycles.
hear_inc	1	Minimum hearing capacity of a player.
hear_decay	2	Decay of hearing capacity of a player.
inertia_moment	5.0	Inertia moment of a player. It affects its turns.
kickable_margin	1.0	The area within which the ball is kickable is: kickable_area = kickable_margin + ball_size + player_size.
catchable_area_l	2.0	Goaltender catchable area length.
catchable_area_w	1.0	Goaltender catchable area width.
catch_probability	1.0	The probability for a goaltender to catch the ball.
dash_power_rate	0.01	Rate by which the *Power* argument in the `dash` command is multiplied.
kick_power_rate	0.016	Rate by which the *Power* argument in the `kick` command is multiplied.
visible_angle	90	Angle of view cone of a player in the standard view mode.
send_step	150	Length of the interval for sending visual information to a player in the standard view mode (milliseconds).

Table 2.4: Soccer server version 4 parameters along with their default values used in this thesis. This table is adapted from the parameter table in the soccer server user manual [Andre *et al.* 98a].

Building a fully functioning robot system can be a frustrating process full of false starts and dead ends. The CMUnited-97 real robots [Veloso *et al.* 98b, Veloso *et al.* 99c] are the culmination of more than two years of development. After building one preliminary version [Achim *et al.* 96], we were able to build CMUnited-97, an autonomous team of robots that won the RoboCup-97 small-robot competition at IJCAI-97 in Nagoya, Japan [Veloso *et al.* 98a]. The CMUnited real robot team is an on-going effort. In 1998, the team entered and won the RoboCup-98 small-robot competion with CMUnited-98, a different team of robots

- Sensing and acting is asynchronous.
- The players' vision is limited, giving them a partial view of the world with lots of hidden state.
- The players can communicate by speaking a message that is audible to all nearby players.
- All players are controlled by separate processes, enforcing a distributed approach.
- Each player has limited stamina;
- There are many sources of noise:
  - Noise in actuator parameters;
  - Noise in object motion;
  - Noise in visual perceptions.
- Percepts sent to the agent or commands from the agent may be lost:
  - There is no guarantee that any sent commands are ever executed;
  - The agent must verify whether commands are executed from the sensory information it receives.
- Everything happens in real time:
  - Visual information arrives at 150 msec intervals (with *high* `view_quality` and *normal* `view_width`);
  - Aural information arrives asynchronously whenever it is issued;
  - Agents can act (physically) at most once every 100 msec.

Table 2.5: Challenges for the soccer server agents.

and mostly different software [Veloso *et al.* 99a]. This section describes the CMUnited-97 robots since they are the ones used for the research reported in this thesis.

The robots are no bigger than 180 cm^2 in area (footprint) and operate on a field with the same size and surface as a ping-pong table. Their object is to push an orange golf ball into one of the goals at the ends of the field. A team consists of up to 5 robots playing at one time. Figure 2.6 shows a picture of the CMUnited-97 robotic agents on the field. One of the 6 robots was used as a reserve in case any of the others temporarily malfunctioned.

2.3.1 Overall Architecture

The architecture of our system addresses the autonomous robotic control task by viewing the overall system as the combination of the robots, the camera, an image processor, and

Figure 2.6: The CMUnited-97 robot team that competed in RoboCup-97. One of the 6 robots was used as a reserve in case any of the others temporarily malfunctioned.

several clients as the minds of the small-size robot players. Figure 2.7 sketches the building blocks of the architecture.

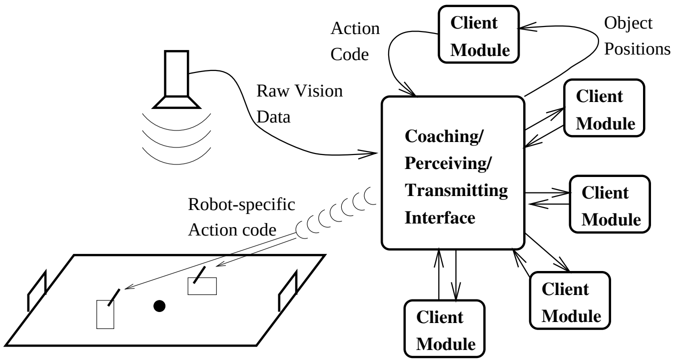


Figure 2.7: The CMUnited-97 robot architecture with global perception and distributed action.

The complete system is fully autonomous consisting of a well-defined and challenging processing cycle:

- The overhead camera with framegrabber grabs still images of the field.
- The image processor finds the ball's and the robots' locations and orientations in the still images.
- The client control modules use the objects' position information to produce control information. Each control module controls one robot. Actions are abstract commands indicating how the robots should move.
- The wireless communication link transmits the control information from the host computer to the robots. Each robot has an identification binary code that is used on-board to detect commands intended for that robot. The abstract commands from the control modules are converted to a sequence of motor control bytes.

- The robot hardware receives the motor commands and moves the physical robots.

Table 2.6 summarizes the inputs and outputs of the five components of the complete robot architecture.

Functionality	**Entity**	**Input**	**Output**
vision	camera/framegrabber	continuous visual data	still frames
perception	image processor	still frames	robots' and ball's coordinates
skills and strategy	client control modules	robots' and ball's coordinates	abstract commands
communication	radio link	abstract commands	motor commands
movement	robots	motor commands	robot movement

Table 2.6: The functional layers of the robot architecture with their inputs and outputs.

The remainder of Section 2.3 is organized as follows. In Section 2.3.2, I describe our development of a preliminary image processor as well as the portions of the current image processor developed principally by Han [Han and Veloso 97] that are relevant to understanding the sensing capabilities of the robot agents. Then, in Section 2.3.3, I precisely specify the perception and abstract actions that are the inputs and outputs of the client control modules. Details of the other hardware components (the camera, framegrabber, radio link and the physical robots) can be found in [Achim *et al.* 96]. The client control modules are part of this thesis work and are therefore described in detail in a later chapter (Section 3.7).

2.3.2 The Image Processor

In this section I describe the preliminary image processor that I developed as part of our initial CMUnited-96 robot implementation [Achim *et al.* 96]. Then I describe the portions of the CMUnited-97 vision system developed principally by Han [Han and Veloso 97] that are relevant to understanding the sensing capabilities of the robot agents.

The robots need sensory information describing the positions of the robots and the ball on the field. The vision system provides this sensory information by way of an overhead camera, framegrabber, and image processor. The fact that perception is achieved by a video camera that overlooks the entire field offers an opportunity to get a complete, global view of the world state. Although this setup may simplify the sharing of information among multiple agents, it presents a challenge for reliable and real-time processing of the movement of multiple moving objects—in our case, the ball, five agents on our team, and five agents on the opposing team.

The CMUnited-96 Vision System

In our initial robot implementation [Achim *et al.* 96] which we entered in the MiroSot-96 competition [Kim 96], I was the principal developer of the image processor. In that system, we use a standard camcorder mounted above the field. It produces NTSC video output that is then processed by the framegrabber.

The framegrabber takes raw, continuous image data and converts it into digital frames at a maximum rate of 30 frames per second. The framegrabber produces still images such as the one shown in Figure 2.8. The format is a 320×240 array of pixels, each having a red, green, and blue value ranging from 0–255.

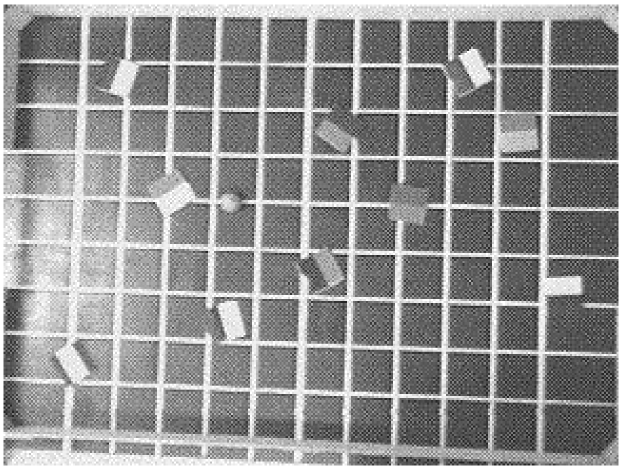


Figure 2.8: An example of a still image of our initial CMUnited-96 robot implementation captured by the framegrabber. Each robot has two different color patches so that we can detect both position and orientation. Each shade in this greyscale image is a different color in reality

Fast image processing to discover the positions and orientations of the objects on the field is a significant challenge. Our method is color-based. By knowing the color of the ball and coding each robot with two different colors, one per half as shown in Figure 2.8 (each shade in the greyscale image is a different color in reality), it is possible to scan the image for pixels that are within a certain threshold distance from these colors.

Once one such pixel is found, the center of a color region is found by computing the center of gravity of all the pixels in a local region that match that color within a threshold. For example, the ball is orange. Thus, to find the ball, the entire image can be scanned for a pixel that is close to orange. Then, in a small region around that pixel, the coordinates of all of the orange pixels are averaged to indicate the center of the ball. For the robots, the centers of the two halves are found in the same way, making it trivial to compute the centers and orientations of the robots.

One important technique for speeding up the image processing is to rely on the assumption that objects will not move very far from frame to frame. Thus we can search for objects in a small region around their previous locations. Our image processing algorithm is summarized in Table 2.7.

While we use several techniques to speed up the CMUnited-96 vision processing, it still does not operate at frame rate: while the images are grabbed 30 times/sec, we can only process them roughly 7 times/sec. The ability to operate reliably at frame rate is a significant improvement in the CMUnited-97 vision system.

To find robot R, consisting of colors $C1$ and $C2$, at time T:

- Find a pixel of color $C1$ near the location of robot R at time $T-1$. (At time 0, the entire image must be scanned.)
- Find the center of gravity of all $C1$-colored pixels near that pixel.
- Find the center of gravity of all $C2$-colored pixels near that pixel.
- Use the two centers of gravity to compute the center and the orientation of the robot.

Table 2.7: A high-level view of our algorithm for locating a given robot in a still frame used by the CMUnited-96 vision-processing software.

The CMUnited-97 Vision System

In contrast to the robots detected by the algorithm laid out in Table 2.7, the CMUnited-97 robots are all marked with the same two colors. Thus there is the added challenge of keeping track of which robot is which. Our current image processor, developed principally by Han, successfully keeps track of five individual identical-looking teammates as well as five opponents. The ball's position and velocity are calculated at frame rate using a Kalman-Bucy filter [Han and Veloso 97].

2.3.3 Agent Perception and Action

This section specifies the perception and action capabilities of the CMUnited-97 robot team which is used as a substrate system in this thesis. The cognition component comprising the client control modules is described in Section 3.7 since it is an implementation of the team member agent architecture.

Perception

While the agents all get the same sensory information from a centralized image processor, the control decisions are made in a distributed manner. The agents are controlled either one at a time by the same process, or by completely separate processes all receiving the same visual information.

Visual information arrives at the framegrabber's processing rate: 30 frames/sec. The vision system sends sensory information to each of the agents in the following form:

- 5 teammates: absolute (x, y, θ) position and orientation coordinates.
- 5 opponents: absolute (x, y) position coordinates. Orientation markers are team-dependent and thus cannot be reliably detected.
- Ball: absolute (x, y) position coordinates as well as (dx, dy) velocity information.

Position information is accurate to within about 1cm and orientation information is accurate to within about 10 degrees, but the noise does not follow any regular distribution. Since the camera always sees the entire field, there is no hidden state.

The field is 274cm in length and 153cm in width. Therefore, each of the 10 robots can be in any of 274*153 possible positions, with the 5 teammates having any of 36 possible orientations. In addition, the ball can be in the same number of positions. Not considering the continuous ball velocity and the past states of the world, the perceived state space has $(274 * 153)^{11} > 10^{49}$ different states. Of course, the actual state space is infinitely large since the robot positions and orientations can vary continuously.

Action

The output of the client modules is abstract commands indicating how the robots should move. Note that the two motors on the robots can be controlled separately. The motors have encoders which enable precise control [Achim *et al.* 96]. Two types of commands are available:

Distance mode: Each motor moves a specified distance in a specified direction. Used for precise on-the-spot-turns.

Velocity mode: Each motor accelerates to a specified velocity. Used for continuous straight or curved movement.

The communication link converts the abstract commands to a stream of motor control bytes which can then be executed by the motor controllers on the robots. The commands are executed reliably and can be sent at the vision system's frame rate: 30 times/sec. Thus perception and action are synchronous with sensations triggering actions.

2.4 Network Routing

Network routing is a domain with enough similar characteristics to robotic soccer that the new TPOT-RL algorithm is effective, but enough differences to validate its generalizability. As a domain for the experiments reported in Chapter 7, we use a modified version of a publicly available packet routing simulator [Boyan and Littman 94].

In this simulator, a network at time t consists of:

- A set of *nodes* $N = \{n_0, \ldots, n_{m-1}\}, |N| = m$. Each node n_i consists of a queue of packets $K_{n_i} \subseteq K$, $|K_{n_i}| = k_{t,n_i}$ at time t. As packets are introduced into and removed from the queue, K_{n_i}, and consequently k_{t,n_i}, changes over time.

- A set of *links* $L \subseteq \{\{n_i, n_j\} | n_i, n_j \in N\}$ connecting pairs of nodes. From any node n_i, $L_{n_i} \subseteq N$ is the set of links from n_i: $L_{n_i} = \{n \in N | \{n, n_i\} \in L\}$. $|L_{n_i}| = l_{n_i}$.

- A set of *packets* $K = \{k_0, \ldots, k_{z_t-1}\}, |K| = z_t$ at time t. Each packet k_i is introduced at a source node $k_{i_{source}} \in N$ and travels towards its destination node $k_{i_{dest}} \in N$. The packet also stores the time at which it left its source, $k_{i_{stime}}$, and when it arrives, the time at which it reaches its destination $k_{i_{dtime}}$. $k_{i_{path}} \subseteq L$ is an ordered list of links along which k_i has traveled from $k_{i_{source}}$ to its current position along with the times at which it has traversed each link. As packets are introduced into and removed from the network, K, and consequently z_t, changes over time.

- A *node capacity* C_{node} indicating the maximum number of packets allowed in a node's packet queue: $\forall i, k_{t,n_i} \leq C_{node}$.
- A *network capacity* C_{net} indicating the number of packets that can be active at one time in the network: $z_t \leq C_{net}$.

Two other parameters affecting the simulation are t_l which is the time it takes a packet to traverse a link and t_n which is the time it takes for a node to process one packet. If at time t, packet k enters the queue K_{n_i} at node n_i, it will stay there for $k_{t,n_i} t_n$ seconds.

As described by its authors, the network routing simulator is ...

> ... a discrete event simulator to model the motion of packets through a local area network. Packets are periodically introduced into the network at a random node with a random destination. Multiple packets at a node are stored in an unbounded FIFO queue; however, we set a limit on the total number of packets active in the network at a time.... In unit time, a node takes the top packet in its queue, examines its destination, and chooses a neighboring node to which to send the packet. A packet sent directly to its destination node is removed from the network immediately [Littman and Boyan 93].

At any give time, the state space can have up to C_{net} packets with a source, destination, and current location all in N ($|N| = m$). Thus the instantaneous size of the state space is:

$$|S| = (m^3)^{C_{net}} \tag{2.2}$$

In addition, each packet has traversed some subset of the $|L|$ links in the network, and the agents internal states' keep track of network statistics over time, thus increasing the effective size of the state space indefinitely.

The packet routing problem can be viewed as a multi-agent collaborative problem by modeling each node as having an independent agent which makes the routing decisions at that node. When a node gets a packet, its agent must decide to which neighboring node it should forward the packet. This decision certainly depends on the packet's destination and might also depend upon the node's perception of the current state of the network. The team's goal is to route packets efficiently so that nodes do not reach their queue capacities and so that packets arrive at their destinations as quickly as possible.

This formulation of network routing is a team problem: all the agents are trying to work together to achieve optimal network performance. While neither the sensors nor the actuators are noisy, there is no explicit inter-agent communication, and there are no adversaries, the overall task is similar to robotic soccer in that unpredictable changes in network traffic, like unpredictable changes in opponent behaviors, can dynamically change the task characteristics. In addition, teammates' actions and their resulting state transitions are unknown. This last quality is that of opaque transitions and is a key characteristic enabling the effective use of the TPOT-RL algorithm.

2.4.1 Agent Perception and Action

Perception

When faced with the decision of where to send a particular packet k_i at time t, the sensory information available to an agent at a given node n_j is:

- $k_{i_{source}}$, the packet's source;
- $k_{i_{dest}}$, the packet's destination;
- $k_{i_{stime}}$, the time the packet left its source;
- $k_{i_{path}}$, the links traversed (with times) by the packet;
- K_{t,n_j}, the queue of packets waiting to be processed.

In addition, the agent can keep track of the traffic along its links. Since it either sends or receives every packet that travels along one of these links, it can keep local traffic statistics over time. Finally, the agent gets periodic overall network performance statistics as the feedback of the team's performance.

Action

The actions the agent can take are straightforward: the agent at node n_j can send packet k_i to a neighboring node along any of the links in L_{n_j}. Sensing and acting are synchronous in this domain: actions are triggered by a perception that the world has changed.

2.4.2 Example Network

Figure 2.9 illustrates our simulation of the packet routing problem. Nodes are represented as circles and links are lines. Suppose that node A receives a packet destined for node B along link 1. It must then decide whether to forward it along link 2, link 3, or possibly back along link 1.

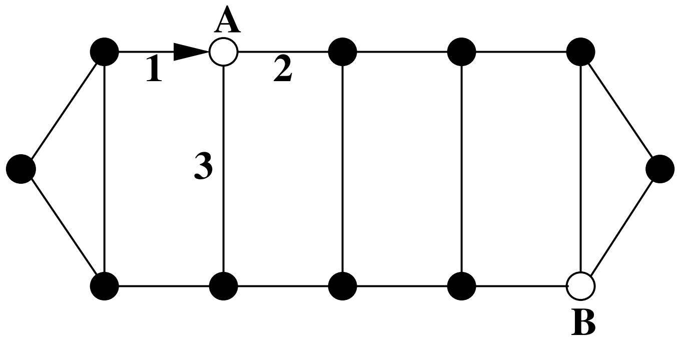


Figure 2.9: An example of the packet routing problem. Nodes are represented as circles and links are lines. In this case, node A is receiving a packet destined for node B along link 1 and must decide along which link to route it.

This simulator is meant to be a test bed for learning experiments rather than a realistic simulation of network routing. Nonetheless, it has been used by researchers other than the simulator's creators to test reinforcement learning approaches [Subramanian *et al.* 97].