The RoboCup Soccer Server and CMUnited Clients: Implemented Infrastructure for MAS Research

Itsuki Noda
Electrotechnical Laboratory
1-1-4 Umezono
Tsukuba, Ibaraki 305-8568, JAPAN
noda@etl.go.jp

Peter Stone
AT&T Labs – Research
180 Park Ave., room A273
Florham Park, NJ 07932
pstone@research.att.com

1. Introduction

The field of multiagent systems (MAS) covers a wide variety of research foci and applications, ranging from software-based information processing (e.g. (Sycara *et al.*, 1996)) to robotic control of multiple agents (e.g. (Mataric, 1997)). One common characteristic of multiagent research is that it relies on significant software and/or hardware *infrastructure*: domains that support the simultaneous operation of tens to thousands of agents are needed.

As Gasser (Gasser, 2000) points out, infrastructure both enables domain-specific progress and serves as a "leveling device: it unifies local practices with global ones." He classifies the infrastructure needs of four MAS focus areas: science, education, application, and use. Since each focus area has a wide range of different needs, each has room for several infrastructures. Certainly no single infrastructure can meet the needs of all four focus areas.

One MAS infrastructure that is designed to meet many of the needs of the science and education focus areas is the RoboCup Soccer Server (Noda *et al.*, 1998; Noda and Frank, 1998) and associated client code. The Robot Soccer World Cup, or RoboCup, is an international research initiative that uses the game of soccer as a domain for artificial intelligence and robotics research. Annual international RoboCup events involve technical workshops as well as software and robotic competitions. Soccer Server is used as the substrate for the RoboCup software competitions. Originally released in 1995, Soccer Server has an international user community of over 1000 people.

Soccer Server is a multiagent environment that supports 22 independent agents interacting in a dynamic, real-time environment. The server embodies many real-world complexities, such as noisy, limited sensing; noisy action and object movement; limited agent stamina; and limited inter-agent communication bandwidth. AI researchers have been using the Soccer Server to pursue research in a wide variety of areas, including real-time multiagent planning, real-time communication methods, collaborative sensing, agent/opponent modeling, and multiagent learning (Asada and Kitano, 1999).

In addition to the server itself being publicly available in an open-source paradigm, users have contributed several clients that can be used as starting points for newcomers to the domain. One example is the CMUnited simulated soccer team, champion of the RoboCup-98 and RoboCup-99 robotic soccer competitions. After winning the competitions, much of the CMUnited source code became publicly available, and several groups used it as a resource to help them create new clients for research and as entries in the RoboCup-99 and RoboCup-2000 competitions. As a unit, Soccer Server and the client code comprise a complete infrastructure, allowing researchers to easily focus on a wide variety of issues.

Based on the success of Soccer Server and its associated client code, we are now in the process of creating a new flexible utility for simulation systems (FUSS) that will be designed to support simulations of multiple domains. For example, we plan to use the same underlying simulation for an improved simulator of the game of soccer as well as a disaster rescue simulator for use in the RoboCup Rescue initiative (Kitano *et al.*, 1999). FUSS will also be available as infrastructure for the MAS research community.

The remainder of the paper is organized as follows. Section 2 outlines the science and education needs that are met by this infrastructure. Section 3 gives an overview of the RoboCup Soccer Server. Section 4 presents the CMUnited simulated soccer clients for use with Soccer Server. Section 5 motivates and presents the current state of the development of FUSS and Section 6 concludes.

2. Infrastructure Characteristics

Soccer Server and the CMUnited client code are *widely* and *freely* available over the internet using an *open source* paradigm. The software is *packaged* for easy installation, *supported* both by the developers and by the large *community* of current users.

This infrastructure is a *comprehensive, implemented* MAS designed for *simulation* experiments. It consists of several independent *components*, including visualization, sample client, and coach modules. The coach module is often used as a tool for *experiment construction*. The most natural and compelling form of *measurement* is game results in tournaments with multiple teams, but the infrastructure also includes *data collection and analysis* tools for more rigorous scientific measurement.

Judging by the large user community, this infrastructure is very *usable*; the fact that it has been successfully used for multiple international competitions is a testament to its *robustness*. New users can take advantage of its *progressive complexity* by starting with a single agent and gradually increasing the size of teams and their communicative and organizational capabilities. A recent addition to the infrastructure is the ability to induce *intentional failures* by disabling selected players.

The italicized words above are all characteristics identified by Gasser (Gasser, 2000) as essential or desirable for MAS infrastructures that support science and education.

In addition to meeting these abstract, general needs, Soccer Server has been used to study many concrete research issues and as a basis for several undergraduate and graduate courses (e.g. (Coradeschi and Malec, 1999; Takahashi and Itoh, 2001)).

An IJCAI-97 challenge paper (Kitano *et al.*, 1997) identified three general research challenges that can be addressed within Soccer Server as being

- Multiagent learning;
- Teamwork structures; and
- Agent/Opponent modeling.

As laid out in (Stone, 2000a), other relevant research issues include inter-agent communication in single-channel, low-bandwidth environments; coordination with limited communication, collaboration in a dynamic real-time environment; organizational structures; distributed sensing/sensor fusion; resource management; agent monitoring; and multiagent planning. These research topics are all addressed by various researchers in the continuing series of RoboCup books (Kitano, 1998; Asada and Kitano, 1999; Veloso *et al.*, 2000; Stone *et al.*, 2001).

It also is important to emphasize that Soccer Server is not simply for domain-specific research. It shares characteristics with many other domains, increasing the likelihood that advances will span applications. Specifically, algorithms that have been developed and/or studied in Soccer Server have also been applied to:

Helicopter combat: A generic model of teamwork and opponent modeling has been applied to both robotic soccer and a helicopter combat domain. STEAM, a large number of domain-independent teamwork rules, were defined in a SOAR architecture, reducing the number of domain-specific rules required in each application (Tambe, 1997).

Network routing: Team-Partitioned, Opaque-Transition Reinforcement Learning is an algorithm that enables multiple independent agents to learn to cooperate despite limited communication capabilities. It was originally implemented and tested within Soccer Server, but then generalized and successfully applied to a network packet routing domain (Stone, 2000b).

Disaster rescue: The RoboCup rescue disaster rescue domain, using earthquake rescue as its motivating scenario, has been designed specifically to transfer RoboCup research to a related domain.

> Similar to soccer, non-centralized, efficient control mechanism to assign roles and to share information dynamically among agents are necessary for this domain. Challenges also include scaling up to hundreds of heterogeneous agents (Kitano *et al.*, 1999).

Other potential applications for transferring RoboCup-related technological advances include: intelligent traffic systems; office robots; NASA domains such as multirover or interferometry missions; robotic surveillance; agent communication research; real-time systems research; and market trading. Prokopenko summarizes these applications at `http://www.cmis.csiro.au/iit/Projects/RoboCup/applications.htm`.

Another aspect to show the generality will be the list of new research problems that these infrastructures make clear for researchers. Andou (Andou, 1999) pointed out the importance how rewards are assigned to each agent under reinforcement learning of multiagent systems. They focused especially on autonomous learning by agents who do not play the ball directly when their team get a goal. They also pointed out the issue of ratio of exploitation and exploration in multiagent reinforcement learning. In a multiagent learning, exploration of an agent may disturbs others' exploitation. Therefore exploration is used more carefully in multiagent learning. (Nakashima and Noda, 1998) mentioned an issue of combination of behavior-oriented control and goal-oriented planning, and propose a concept of dynamic subsumption architecture. These issues are motivated by Soccer Server, but include general problems found in various multiagent systems.

With all of these past successes and unrealized potentials, Soccer Server is one of the leading examples of MAS infrastructures appropriate for the science and education communities. The following sections detail the current state and future plans for this infrastructure.

3. The RoboCup Soccer Server

3.1. Soccer Server

Soccer Server enables a soccer match to be played between two teams of player-programs (possibly implemented in different programming systems). A match using Soccer Server is controlled using a form of client-server communication. Soccer Server provides a virtual soccer field such as the one shown in Fig. 1 and simulates the movements of players and a ball. A client program can provide the 'brain' of a player by connecting to the Soccer Server via a computer network (using a UDP/IP socket) and specifying actions for that player to carry out. In return, the client receives information from the player's sensors.

Figure 1. Window image of Soccer Server

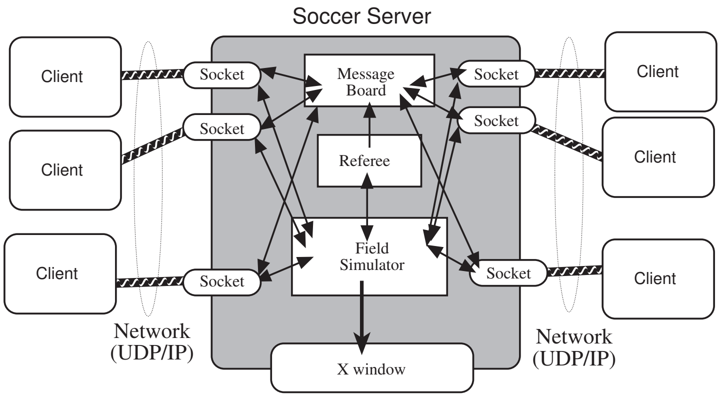

Figure 2. Overview of Soccer Server

The three main modules in the Soccer Server itself are:

1. **A field simulator module**. This creates the basic virtual world of the soccer field, and calculates the movements of objects, checking for collisions.

2. **A referee module**. This ensures that the rules of the game are followed.

3. **A message-board module**. This manages the communication between the client programs.

Fig. 2 gives an overview of the relation of these modules and of how the Soccer Server communicates with clients. A client controls only one player. It receives visual and verbal sensor information ('see' and 'hear' respectively) from the server and sends control commands ('turn', 'dash', 'kick' and 'say') to the server. Visual information gives only partial information about the field from the player's viewpoint, so that the player program must make decisions based on incomplete knowledge. Limited verbal communication is also available, by which the players can communicate with each other to decide team strategy.

All communication between the server and each client is in the form of ASCII strings. Therefore, clients can be realized in any programming environment on any architecture that has the facilities of UDP/IP sockets and string manipulation. The communication protocol consists of:

- **Control commands**: messages sent from a client to the server to control actions of the client's player. The basic commands are `turn`, `dash` and `kick`. Communication is conducted through the `say` command, and a privileged goalie client can also attempt to `catch` the ball.

- **Sensor information**: messages sent from the server to a client describing the current state of the game from the viewpoint of the client's player. There are three types of information, visual (`see`), auditory (`hear`) and bodily (`sense_body`).

Soccer Server is a discrete simulation of continuous time. Thus, both the control commands and the sensor information are processed within a framework of 'simulator steps'. The length of the cycle between the processing steps for the control commands is 100msec, whereas the length of the step cycle for the sensor information ranges from 37–300msec and is controlled actively by the individual clients (frequency is traded off against visible angle and information quality). Note that all players have identical abilities (strength and accuracy of kicking, stamina, sensing) so that the entire difference in performance of teams derives from the effective use of the control commands and sensor information, and especially from the ability to produce collaborative behavior among multiple clients [1].

As a final feature, when invoked with the `-coach` option, the server provides an extra socket for a privileged client (called a *coach* client) that has the ability to direct all aspects of the game. The coach client can move all objects, direct the referee module to make decisions, and announce messages to all clients. This facility is extremely useful for tuning and debugging client programs, which usually involves repeated testing of the behaviors of the clients in many situations. In addition to the `-coach` option, "online coach" feature is implemented to the recent Soccer Server, which allows teams to include a twelfth client that has a global view of the game and can conduct sideline coaching during play by shouting strategic or tactical advice to players.

3.2. As a Research Tool

Soccer Server has been used by researchers to examine MAS. Here we investigate Soccer Server's features as a research tool.

The biggest reason that it is used widely is that it simulates *soccer*, which is very popular world-wide. Similar to chess, popularity is an important factor for research applications, because researchers can share an understanding and intuition about the domain. While individual plays in soccer are relatively simple (this is important in simulations), the variation of team play is very wide. Therefore, we can find many open issues in it, such as opponent modeling, multiagent learning, cooperative actions, multiagent planning, and so on. Thus, researchers in various fields can share a common domain.

The second reason is that it uses a middle-level abstraction for representing the client commands and the sensor information. One possibility was a low-level physical description, for example allowing power values for drive motors to be specified. However it was felt that such a representation would concentrate users' attention too much on the actual control of a player's actions, relegating true investigation of the multiagent nature of team-play to the level of a secondary objective. Further, it is difficult to design a low-level description that is not implicitly based on a specific notion of robot hardware; for example, control of speed by drive motors is biased towards a physical implementation that uses wheels. On the other hand, a more abstract representation, using tactical commands such as `pass-ball-to` and `block-shot`, would produce a game in which the real-world nature of soccer becomes obscured, and in which the development of soccer

[1] Recent Soccer Server versions (starting with version 7.0) include the option of using heterogeneous players, though players are still all homogeneous by default.

techniques not yet realized by human players becomes problematic. Thus, our representation — using basic control commands such as `turn`, `dash`, and `kick` — is a compromise. To make good use of the available commands, clients need to tackle both the problem of control in an incomplete information, dynamic environment and also the challenge of combining the efforts of multiple players. Thus, we believe that Soccer Server achieves our goal of providing a simple test-bed with significant real-world properties.

Several technical issues are also important for Soccer Server's wide-spread use. Soccer Server has the following technical features that help researchers to use it:

- Soccer Server is lightweight. It requires few computing resources so that it can run on entry-level PCs. This enables researchers to start their research with limited resources. Additionally, in order to use it for educational purpose, it is necessary to run on PCs students can use in computer labs in schools.

- Soccer Server runs on various platforms. It supports SunOS 4, Solaris 2.x, Linux, IRIX, OSF/1, and Windows [2]. It also requires quite common tools and libraries like Gnu or ANSI C++ compiler, standard C++ libraries, and X window. They are distributed freely and used widely.

- Soccer Server uses ASCII string on UDP/IP for protocol between clients and the server. This feature enables researchers/students to use any kind of programming language. Indeed, participants in past RoboCup competitions used C, C++, Java, Lisp, Prolog and various research oriented AI programming systems such as SOAR (Tambe *et al.*, 1995). Version control of protocol is also an important feature. It enables us to use old clients to run in newer servers.

As well as supporting research pertaining to player control, Soccer Server also supports several auxiliary research activities. Soccer Server consists of two modules, `soccerserver`, a simulation kernel, and `soccermonitor`, a viewer of simulated soccer field. They are connected via UDP/IP. While this separated structure was introduced only for displaying the field window on multiple monitors, it led to unexpected

[2] Windows versions were contributed by Sebastien Doncker and Dominique Duhaut (compatible to version 2), and now by Mario Pac (compatible to version 4) independently. Information about Mario's versions is available from:

`http://users.informatik.fh-hamburg.de/~pac_m/`

activities in different research fields. Many researchers have made and have been trying to build 3D monitors to display scenes of matches dynamically (Shinjoh and Yoshida, 1998). In addition, some groups are building commentary systems that describe matches dynamically in natural language (Andrè *et al.*, 1998; Tanaka-Ishii *et al.*, 1998). Both kinds of systems are connected with the server as secondary monitors. They get information regarding the state of matches, analyze the situations, and generate appropriate scenes and sentences.

4. The CMUnited Client

As described in Section 3, Soccer Server clients interact with the Soccer Server via an ASCII string protocol. The server supports low-level sensing and acting primitives. However, there are several basic tasks left up to the clients, including

- Managing socket communication with the server;
- Parsing the sensory commands;
- Handling asynchronous sensation and action cycles;
- Maintaining a model of the world; and
- Combining the low-level action primitives into useful skills.

Depending on one's research focus, a newcomer to the domain may not be interested in solving each of these tasks from first principles. Instead, one can look to the growing body of publicly available client code available at `http://medialab.di.unipi.it/Project/Robocup/pub/`.

While there are many possible solutions to each of these tasks, it is often difficult to evaluate them independently. The CMUnited client code (Stone *et al.*, 1999) offers robust solutions to these tasks that have been successfully tested in competitive environments: CMUnited won both the RoboCup-98 and RoboCup-99 simulator competitions. It has already been successfully used by others. For example, the 3rd place finisher in the RoboCup-99 competition, was partially adapted from the CMUnited-98 simulator team code, and the 1st, 2nd, and 3rd place finishers in the RoboCup-2000 competition were all based on CMUnited-99 source code.

We present the client code as a part of the infrastructure, as opposed to as an application. Without this code, creating a substrate team of agents for research purposes is a daunting task, and is likely to yield sub-par agents. The freely available client code enables researchers to immediately focus on any of a wide variety of areas of interest. The remainder of this section gives an overview of the CMUnited client code.

4.1. Agent Architecture Overview

CMUnited agents are capable of perception, cognition, and action. By perceiving the world, they build a model of its current state. Then, based on a set of behaviors, they choose an action appropriate for the current world state.

At the core of CMUnited agents is what we call the locker-room agreement (Stone, 2000a). Based on the premise that agents can periodically meet in safe, full-communication environments, the locker-room agreement specifies how they should act when in low-communication, time-critical, adversarial environments.

Individual agents can capture locker-room agreements and respond to the environment, while acting autonomously. Based on a standard agent paradigm, our team member agent architecture allows agents to sense the environment, to reason about and select their actions, and to act in the real world. At team synchronization opportunities, the team also makes a locker-room agreement for use by all agents during periods of limited communication. Fig. 3 shows the functional input/output model of the architecture.

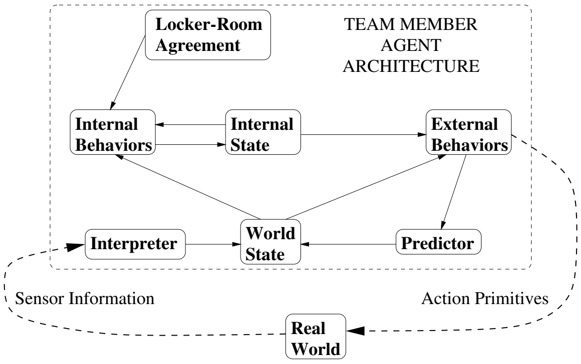

Figure 3. A functional input/output model of CMUnited's team member agent architecture.

The agent keeps track of three different types of state: the *world state*, the *locker-room agreement*, and the *internal state*. The agent also has two different types of behaviors: *internal behaviors* and *external behaviors*.

The world state reflects the agent's conception of the real world, both via its sensors and via the predicted effects of its actions. It is updated as a result of interpreted sensory information. It may also be updated according to the predicted effects of the external behavior module's chosen actions. The world state is directly accessible to both internal and external behaviors.

The locker-room agreement is set by the team when it is able to privately synchronize. It defines the flexible teamwork structure and the inter-agent communication protocols, if any. The locker-room agreement is accessible only to internal behaviors.

The internal state stores the agent's internal variables. It may reflect previous and current world states, possibly as specified by the locker-room agreement. For example, the agent's role within a team behavior could be stored as part of the internal state. A window or distribution of past world states could also be stored as a part of the internal state. The agent updates its internal state via its internal behaviors.

The internal behaviors update the agent's internal state based on its current internal state, the world state, and the team's locker-room agreement.

The external behaviors reference the world and internal states, and select the actions to send to the actuators. The actions affect the real world, thus altering the agent's future percepts. External behaviors consider only the world and internal states, without direct access to the locker-room agreement.

4.2. Asynchronous Sensing and Acting

A driving factor in the design of the agent architecture is the fact that the simulator operates in fixed cycles of length 100 msec, while sensations are sent at different intervals (typically every 150 msec). The simulator accepts commands from clients throughout a cycle and then updates the world state all at once at the end of the cycle. Only one action command (dash, kick, turn, or catch) is executed for a given client during a given cycle.

Therefore, agents (simulator clients) should send exactly one action command to the simulator in every simulator cycle. If more than one command is sent in the same cycle, a random one is executed, possibly leading to undesired behavior. If no command is sent during a simulator cycle, an action opportunity has been lost: opponent agents who have acted during that cycle may gain an advantage.

In addition, since the simulator updates the world at the end of every cycle, it is advantageous to try to determine the state of the world at the end of the previous cycle when choosing an action for the current cycle. As such, the basic agent loop during a given cycle t is as follows:

- Assume the agent has consistent information about the state of the world at the end of cycle $t-2$ and has sent an action during cycle $t-1$.

- While the server is still in cycle $t-1$, upon receipt of a sensation (see, hear, or sense_body), store the new information in temporary structures. Do not update the current state.

- When the server enters cycle t (determined either by a running clock or by the receipt of a sensation with time stamp t), use all of the information available (temporary information from sensations and predicted effects of past actions) to **update the world model** to match the server's world state (the "real world state") at the end of cycle $t-1$. Then **choose and send an action** to the server for cycle t.

- Repeat for cycle $t+1$.

While the above algorithm defines the overall agent loop, much of the challenge is involved in updating the world model effectively and choosing an appropriate action. The remainder of this section goes into these processes in detail.

4.3. World Modeling

When acting based on a world model, it is important to have as accurate and precise a model of the world as possible at the time that an action is taken. In order to achieve this goal, CMUnited agents gather sensory information over time, and process the information by incorporating it into the world model immediately prior to acting.

4.3.1. *Object Representation*

There are several objects in the world, such as the goals and the field markers which remain stationary and can be used for self-localization. Mobile objects are the agent itself, the ball, and 21 other players (10 teammates and 11 opponents). These objects are represented in a type hierarchy as illustrated in Fig. 4.

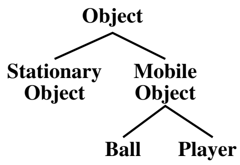

Figure 4. The agent's object type hierarchy.

Each agent's world model stores an instantiation of a stationary object for each goal, sideline, and field marker; a ball object for the ball; and 21 player objects. Since players can be seen without their associated team and/or uniform number, the player objects are not identified with particular individual players. Instead, the variables for team and uniform number can be filled in as they become known.

Mobile objects are stored with confidence values within [0,1] indicating the confidence with which their locations are known. The confidence values are needed because of the large amount of hidden state in the world: no object is seen consistently.

The variables associated with each object type are as follows:

Object :

- Global (x, y) position coordinates
- Confidence within [0,1] of the coordinates' accuracy

Stationary Object : nothing additional

Mobile Object :

- Global (dx, dy) velocity coordinates
- Confidence within [0,1] of the coordinates' accuracy

Ball : nothing additional

Player :

- Team
- Uniform number
- Global θ facing angle
- Confidence within [0,1] of the angle's accuracy

4.3.2. *Updating the World Model*

Information about the world can come from

- Visual information;
- Audial information;
- Sense_body information; and
- Predicted effects of previous actions.

Visual information arrives as relative distances and angles to objects in the player's view cone. Audial information could include information about global object locations from teammates. Sense_body information pertains to the client's own status including stamina, view mode, and speed.

Whenever new information arrives, it is stored in temporary structures with time stamps and confidences (1 for visual information, possibly less for audial information). Visual information is stored as relative coordinates until the agent's exact location is determined.

When it is time to act during cycle t, all of the available information is used to best determine the server's world state at the end of cycle $t-1$. If no new information arrived pertaining to a given object, the velocity and actions taken are used by the predictor to predict the new position of the object and the confidence in that object's position and velocity are both decayed.

When the agent's world model is updated to match the end of simulator cycle $t-1$, first the agent's own position is updated to match the time of the last sight; then those of the ball and players are updated.

4.4. Agent Skills

Once the agent has determined the server's world state for cycle t as accurately as possible, it can choose and send an action to be executed at the end of the cycle. In so doing, it must choose its local goal within the team's overall strategy. It can then choose from among several low-level skills which provide it with basic capabilities. The output of the skills are primitive movement commands.

The skills available to CMUnited players include

- kicking,
- dribbling,
- ball interception,
- goaltending,
- defending, and
- clearing.

The common thread among these skills is that they are all *predictive, locally optimal skills* (PLOS). They take into account predicted world models as well as predicted effects of future actions in order to determine the optimal primitive action from a local perspective, both in time and in space.

One simple example of PLOS is each individual agent's stamina management. The server models stamina as having a replenishable and a non-replenishable component. Each is only decremented when the current stamina goes below a fixed threshold. Each player monitors its own stamina level to make sure that it never uses up any of the non-replenishable component of its stamina. No matter how fast it should move according to the behavior the player is executing, it slows down its movement to keep itself from getting too tired. While such behavior might not be optimal in the context of the team's goal, it is locally optimal considering the agent's current tired state.

Even though the skills are predictive, the agent *commits* to only one action during each cycle. When the time comes to act again, the situation is completely reevaluated. If the world is close to the anticipated configuration, then the agent will act similarly to the way it predicted on previous cycles. However, if the world is significantly different, the agent will arrive at a new sequence of actions rather than being committed to a previous plan. Again, it will only execute the first step in the new sequence.

4.5. Layered Disclosure

A perennial challenge in creating and using complex autonomous agents is following their choices of actions as the world changes dynamically, and understanding why they act as they do. To this end, we introduce the concept of *layered disclosure* (Riley *et al.*, 2000) by which autonomous agents include in their architecture the foundations necessary to allow them to disclose to a person upon request the specific reasons for their actions. The person may request information at any level of detail, and either retroactively or while the agent is acting.

A key component of layered disclosure is that the relevant agent information is organized in *layers*. In general, there is far too much information available to display all of it at all times. The imposed hierarchy allows the user to select at which level of detail he or she would like to probe into the agent in question.

The CMUnited layered disclosure module is publicly available and has been successfully used by other researchers to help them in their code development.

4.6. Summary

In Summary, the CMUnited code hadles several challenges presented by Soccer Server, including managing asynchronous sensing and acting via socket communication with the server; parsing the sensory information; maintaining a world model; and supporting basic skills that can be used to build up a fully functional team. Since it facilitates MAS research in the domain, it forms an important part of this infrastructure.

5. Next Generation Infrastructure

5.1. Problems of Soccer Server

As described above, Soccer Server is a useful infrastructure for research on MAS. While it is used widely for research, several problems of Soccer Server have become clear.

- **Generality:** From 5 years experience of RoboCup activity, we have recognized that many researchers want simulators like Soccer

Server for other domains. For example, some researchers want to modify Soccer Server for hockey or basket-ball. In addition to these ball games, there is growing interest in simulations of rescue from huge natural disasters. Because, Soccer Server itself was designed only for soccer, however, it is difficult to modify it for such purposes.

– **Huge Network Traffic:** Soccer Server communicates with various types of clients (player clients, monitor clients, offline-/online-coach clients) directly. This often makes the server a bottle-neck of network-traffic. In order to avoid such trouble, the server should be re-designed to enable distributed processing easily.

– **Legacy:** In order to keep backward compatibility as much as possible, Soccer Server uses version control for the client-server protocol. Because the current server is a single module, the server must include all protocol versions. In order to solve this problem, the server should have a mechanism that enables it to connect with a kind of filter or proxy that converts internal representations for each version of the protocol.

A possible strategy to overcome these problems is "modular structure over network." In Soccer Server, the monitor module is separated from the simulation kernel. As mentioned before, this modularity brings the following merits:

– It enables the development of systems to show plays in 3D, to describe games in natural language, and to analyze performance of teams from various point of view. These systems are possible because the modules are connected via networks and loosely coupled by a simple protocol. Therefore, each developer can develop their systems independently.

– It enables researchers to develop such monitors on various platforms. This is possible because communication between modules use open and standard protocol (character strings via UDP/IP).

We are now applying a similar technique to other parts of the simulator. In the following sections, we describe the general framework, called FUSS, for distributed simulation based on this strategy, and show the implementation of the soccer simulator as an example.

5.2. Overview

FUSS (Framework for Universal Simulation System) is a collection of programs and libraries to develop distributed simulation systems. It is designed to aid development of systems that simulate complex environments like MAS.

A simulation system in FUSS consists of a few modules, each of which simulates an individual function or phenomenon. For example, we can develop a soccer simulator in FUSS that consists of a field (physical) simulation module, a referee module, and multiple player simulation modules. The modules are combined into a system by a kernel (fskernel) and libraries. Fig. 5 shows the high-level structure of a simulation system built on top of FUSS.

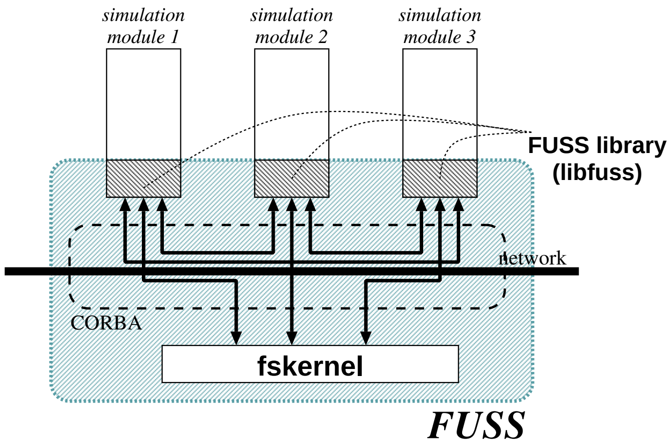

Figure 5. A Simulation System Built on FUSS

FUSS itself consists of the following items:

- `fskernel`: A kernel for a simulation system. It provides services of module database, shared memory management, and synchronization control.

- FUSS library (libfuss): A library to develop modules of the simulation system. The library consists of `FsModule`, `ShrdMem` and `PhaseRef` libraries, which provide a framework of simulation modules, an interface to access shared memories, and facilities to synchronize executions of modules respectively.

- utility library and programs: A collection of utilities.

In order to guarantee open-ness in communication among modules and the kernel, FUSS uses CORBA for the communication layer. This makes users free to select any platform and programming language to develop simulation modules. While the current implementation of FUSS uses C++, we can develop libraries in other languages that have a CORBA interface.

In addition, FUSS uses the POSIX multi-thread facility (pthread) to realize flexible interactions between modules and `fskernel`. Using this facility, users need not manage control of execution of the simulation and the communication.

5.3. Shared Memory and Time Management

In development of distributed systems, there are two major issues, *shared data management* and *time management*. As an infrastructure for distributed simulation systems, FUSS provides two frameworks, *shared memory* and *phase control*, to realize this management.

All shared data in a FUSS simulation system must be defined by IDL of CORBA. The definitions are converted into C++ classes and included by all related modules. The shared data is defined as a subclass of `ShrdMem`, the *shared memory* class, in each module. A module calls the `download` method before using the shared memory, and calls the `upload` method after modifying the memory. Then the FUSS library maintains the consistency of the memory among modules.

In order to make an explicit order of execution of multiple simulation components, FUSS modules are synchronized by *phase control*. In the case of soccer simulation, each cycle of the physical (field) simulation may consist of the following steps:

1. collect players' actions to execute in the cycle,
2. calculate movements of players and a ball,
3. check conditions of the game according to the rules, and
4. reflect changes of movements to a shared memory.

These steps are represented by phases in a FUSS simulation system.

A *phase* is a kind of an event that has joined modules. When a module is plugged into the simulation system, the module sends `joinPhase` messages to `fskernel` to join phases in which it executes a part of the simulation. When a phase starts, the kernel notifies the beginning of the phase by sending an `achievePhase` message to all joined modules. Then, the `cycle` method of the phase, which should be defined by users, is called in each module. The kernel waits until all joined modules finish the `cycle` operations of the phase, and moves to the next phase. In other words, executions of simulation modules are serialized according to sequential order of phases.[3]

The kernel can handle two types of phases: *timer phase* and *adjunct phase*. A *timer phase* has its own interval. The kernel tries to start the phase for every interval. For example, a **field simulation phase** in soccer simulation may occur every 100msec. This phase has the field simulator as a joined module. So, the field simulator receives an `achievePhase` message for every 100msec. Then the simulator module calls `cycle` method of **field simulation phase**, in which it calculates movement of objects.

An *adjunct phase* is invoked before or after another phase adjunctively. For the example of soccer simulation, a **referee phase** will be registered as an adjunct phase after a **field simulation phase**. Then the kernel starts the **referee phase** immediately after the **field simulation phase** is achieved. For another example, a **player phase**, in which player simulators/proxies upload players' commands, will be registered as an adjunct phase before a **field simulation phase**. In this case, the kernel starts the **player phase** first, and starts the **field simulation phase** after it is achieved.

A phase can have multiple adjunct phases before or after it. To arrange them in an order explicitly, each adjunct phase has its own tightness factor. If the factor is larger, the phase occurs more tightly adjoined to the mother phase. For example, a **field simulation phase** will have two adjunct phases, a **referee phase** and a **publish phase**, after it. Tightness factors of the referee and publish phases will be 100 and 50 respectively. So, the **referee phase** occurs just after the field simulation phase, and the **broadcast phase** occurs later. Fig. 6 shows phase-control and communication between the kernel and modules in the soccer simulation described in Sec. 5.4.

The structure of the adjunct relationship among phases is an important feature of FUSS. It serializes executions of simulation of joined modules[4] according to the structure and the tightness. Therefore, users can realize serialization of execution by specifying logical relations (adjunct relationships) between phases rather than an exact order of phases. This feature is useful when users want to add new modules to an existing simulation system.

[3] Execution of modules that join to the same phase are processed in parallel.

[4] Phase control serializes only operations defined as `cycle` methods. Users can invoke other threads of execution, that are performed in parallel with the phase execution, using the multi-thread facility.

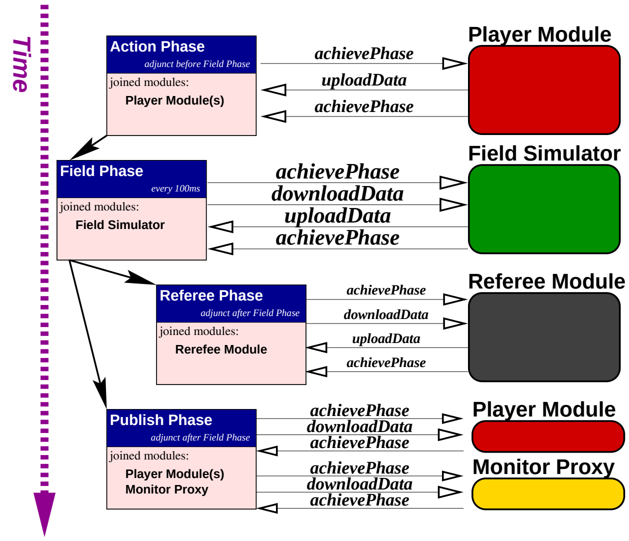

Figure 6. Phase Control and Communication with Joined Modules

5.4. Implementation of Soccer Simulator on FUSS

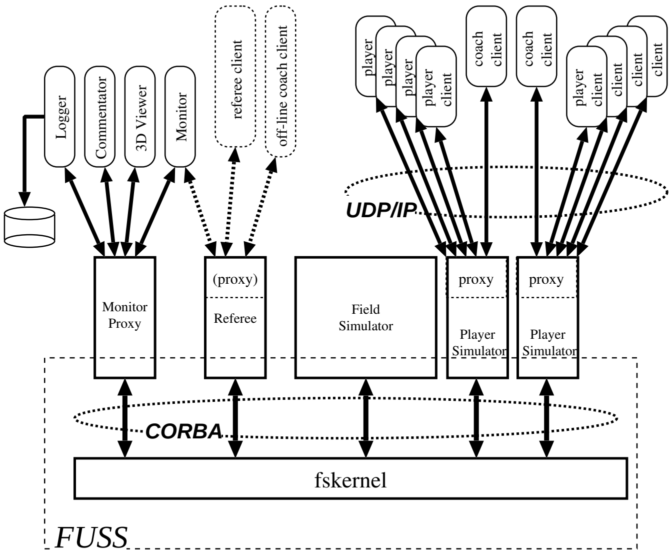

Figure 7. Design of the new soccer simulator

FUSS is designed to be a general tool for development of simulations of arbitrary multiagent systems that run in a distributed way over a computer network. As a proof of concept and to ensure that it can emulate previous successes, we implemented a soccer simulator using FUSS. In the implementation, we divided the functions of the soccer simulator into the following modules:

- **Field Simulator** is a module to simulate the physical events on the field.
- **Referee Simulator** is a privileged module to control a match according to rules. This module may override and modify the results of the field simulator.
- **Player Simulators/Proxies** are modules to simulate events inside of a player's body, and communicate with the player and on-line coach clients.
- **Monitor Proxy** provides a facility for multiple monitors and commentators, as well as the ability to record a game.

The implementation of the referee module is the key of the simulator. Compared with other modules, the referee module should have a special position, because the referee module needs to affect behaviors of other modules directly rather than via data. For example, the referee module restricts movements of players and a ball, that are controlled by the field simulator module, according to the rules of the game. Therefore, the referee module is invoked just before and after the simulator module and checks the data. In other words, the referee module works as a 'wrapper' of other modules. Phase control described in Sec. 5.3 enables this style of implementation. As shown in Fig. 6, **referee phase** is an adjunct phase to **field phase** with a large tightness. Therefore, the referee module can affect the result of the field phase directly. This means the referee module regulates execution of the field module by modifying the result of the simulation.

The advantage of this feature becomes clear when we think of adding a coach module, which will regulate the result of the field simulation in a weaker manner than does the referee module. In this case, a user defines a **coach phase** as an adjunct phase to the field phase, whose tightness is intermediate between those of the referee phase and the publish phase. As a result, the coach phase is invoked after the referee phase and before the publish phase, where the coach module can modify the result of simulation after the referee module.

6. Conclusion

Soccer Server and CMUnited client code provide a robust infrastructure for MAS research using the game of soccer as the underlying domain. A large community has been successfully using it for several years, and it meets many of the science and education needs of the MAS community.

Building on the lessons learned via the Soccer Server, FUSS will provide a utility for creating simulations in a wide variety of multiagent domains. Its modular facilities enable incremental and distributed development of large simulation systems. By using FUSS, Soccer Server's problems are solved as follows:

- Generality: FUSS provides facilities for distributed modular simulation system. We can develop various kinds of simulation systems like rescue simulators and virtual markets using FUSS.

- Huge Traffic: As opposed to Soccer Server, communications with clients are handled with three modules, a monitor proxy and two player simulator/proxies, separately. Therefore, we can distribute the network traffic by invoking these modules on different machines in different network segments.

- Legacy: Communication with player clients is localized by player proxies. This means that we can handle multiple protocols by providing different player proxies for each protocol. This capability makes it much easier to maintain legacy features.

Further information about FUSS is available from `http://www.carc.aist.go.jp/~noda/fuss`.

The infrastructure presented in this paper has many of the characteristics suitable for research in science and education as enumerated in Section 2 (Gasser, 2000). However, there are of course many issues that it does not address. For example, it is not at all intended as an MAS application development tool or as an environment to be presented for general use. In addition, not all MAS research and educational issues can be addressed in this domain. In order to study, for example, web-based multiagent information processing, another infrastructure will be needed.

Nonetheless, Soccer Server and the CMUnited client code provide robust and fun support for disparate research issues such as multiagent learning, sensor fusion, multiagent planning, and agent communication. With the release of FUSS, support for studying these issues across multiple domains will also be introduced. We look forward to continuing research progress in this dynamic multiagent infrastructure.

References

Tomohito Andou. Andhill-98: A robocup team which reinforces positioning with observation. In Minoru Asada and Hiroaki Kitano, editors, *RoboCup-98: Robot Soccer World Cup II*, pages 338–345. Springer, 1999.

E. Andrè, G. Herzog, and T Rist. Generating multimedia presentations for RoboCup soccer games. In H. Kitano, editor, *RoboCup-97: Robot Soccer World Cup I*, pages 200–215. Lecture Notes in Artificial Intelligence, Springer, 1998.

Minoru Asada and Hiroaki Kitano, editors. *RoboCup-98: Robot Soccer World Cup II*. Lecture Notes in Artificial Intelligence 1604. Springer Verlag, Berlin, 1999.

Silvia Coradeschi and Jacek Malec. How to make a challenging AI course enjoyable using the RoboCup soccer simulation system. In Minoru Asada and Hiroaki Kitano, editors, *RoboCup-98: Robot Soccer World Cup II*. Springer Verlag, Berlin, 1999.

Les Gasser. Mas infrastructure definitions, needs, and prospects. In *Proceedings of the Autonomous Agnets 2000 Workshop on Infrastructure for Scalable Multi-Agent Systems*, Barcelona, Spain, June 2000.

Hiroaki Kitano, Milind Tambe, Peter Stone, Manuela Veloso, Silvia Coradeschi, Eiichi Osawa, Hitoshi Matsubara, Itsuki Noda, and Minoru Asada. The RoboCup synthetic agent challenge 97. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 24–29, San Francisco, CA, 1997. Morgan Kaufmann.

Hiroaki Kitano, Satoshi Takokoro, Itsuki Noda, Hitoshi Matsubara, Tomoichi Takahashi, Atsuhi Shinjou, and Susumu Shimada. RoboCup rescue: Search and rescue in large-scale disasters as a domain for autonomous agents research. In *Proceedings of the IEEE International Conference on Man, System, and Cybernetics*, 1999.

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

Maja Mataric. Reinforcement learning in the multi-robot domain. *Autonomous Robots*, 4(1):73–83, January 1997.

Hideyuki Nakashima and Itsuki Noda. Dynamic subsumption architecture for programming intelligent agents. In *Proc. of International Conf. on Multi-Agent Systems 98*, pages 190–197. AAAI Press, 1998.

Itsuki Noda and Ian Frank. Investigating the complex with virtual soccer. In J.-C. Heudin, editor, *Virtual Worlds*, pages 241–253. Ppringer Verlag (LNAI-1434), Sep. 1998.

Itsuki Noda, Hitoshi Matsubara, Kazuo Hiraki, and Ian Frank. Soccer server: A tool for research on multiagent systems. *Applied Artificial Intelligence*, 12:233–250, 1998.

Patrick Riley, Peter Stone, and Manuela Veloso. Layered disclosure: Revealing agents' internals. In *Submitted to the Sixth Pacific Rim International Conference on Artificial Intelligence (PRICAI 2000)*, 2000.

A. Shinjoh and S. Yoshida. The intelligent three-dimensional viewer system for robocup. In *Proceedings of the Second International Workshop on RoboCup*, pages 37–46, July 1998.

Peter Stone, Manuela Veloso, and Patrick Riley. The CMUnited-98 champion simulator team. In Minoru Asada and Hiroaki Kitano, editors, *RoboCup-98: Robot Soccer World Cup II*. Springer Verlag, Berlin, 1999.

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

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

Peter Stone. TPOT-RL applied to network routing. In *Proceedings of the Seventeenth International Conference on Machine Learning*, 2000.

K. Sycara, K. Decker, A. Pannu, M. Williamson, and D. Zeng. Distributed intelligent agents. *IEEE Expert*, 11(6), December 1996.

Tomoichi Takahashi and Nobuhiro Itoh. *Agent Programming using RoboCup (in Japanese)*. Kyoritsu Shuppan, Jul 2001.

Milind Tambe, W. Lewis Johnson, Randolph M. Jones, Frank Koss, John E. Laird, Paul S. Rosenbloom, and Karl Schwamb. Intelligent agents for interactive simulation environments. *AI Magazine*, 16(1), Spring 1995.

Milind Tambe. Towards flexible teamwork. *Journal of Artificial Intelligence Research*, 7:81–124, 1997.

Kumiko Tanaka-Ishii, Itsuki Noda, Ian Frank, Hideyuki Nakashima, Koiti Hasida, and Hitoshi Matsubara. MIKE: An automatic commentary system for soccer. In Yves Demazeau, editor, *Proc. of Third International Conference on Multi-Agent Systems*, pages 285–292, July 1998.

Manuela Veloso, Enrico Pagello, and Hiroaki Kitano, editors. *RoboCup-99: Robot Soccer World Cup III*. Springer Verlag, Berlin, 2000.