Modular Simulator: A Draft of New Simulator for RoboCup

NODA, Itsuki[1,2]
noda@etl.go.jp

[1] CSLI, Stanford University, Palo Alto CA 94306, USA
[2] Electrotechnical Laboratory, Tsukuba 305, Japan

Abstract. Soccer Server has been used as the official simulator for RoboCup Simulation League last three years. Based on this experience, I investigate the feature of Soccer Server and figure out the issue to building such kind of open simulator. Then, I propose a new design of simulator that will provide more flexible version up, easiness of maintenance, and wide application.

1 Introduction

Soccer Server has been used as the official simulator for RoboCup Simulation League last three years. The reasons why Soccer Server is chosen are open system, light weight, and widely supported platforms. These features enable many researchers to use it as a standard tool for their research. And now, we have a large community of simulation league, in which we discuss new rules, share ideas and information, and cooperate with each other to develop libraries and documents.

However, problems of Soccer Server become clear in recent years. Most of them are lied on design of Soccer Server itself. Originally, it was built just as a prototype of the simulator and modified again and again to add new features and to fix bugs. Therefore, the system became complicated and difficult to maintain.

So, it is the time we re-design a new system of simulator. In this paper, I investigate features and problems of current Soccer Server, and propose a new design of Soccer Server.

2 Issues on Soccer Server

2.1 Soccer Server

Soccer Server [5, 4] enables a soccer match to be played between two teams of player-programs (possibly implemented in different programming systems). The match is controlled using a form of client-server communication. The server (Soccer Server) provides a virtual soccer field and simulates the movements of players and a ball. A client (player program) can provide the 'brain' of a player by

M. Veloso, E. Pagello, and H. Kitano (Eds.): RoboCup-99, LNAI 1856, pp. 400–410, 2000.

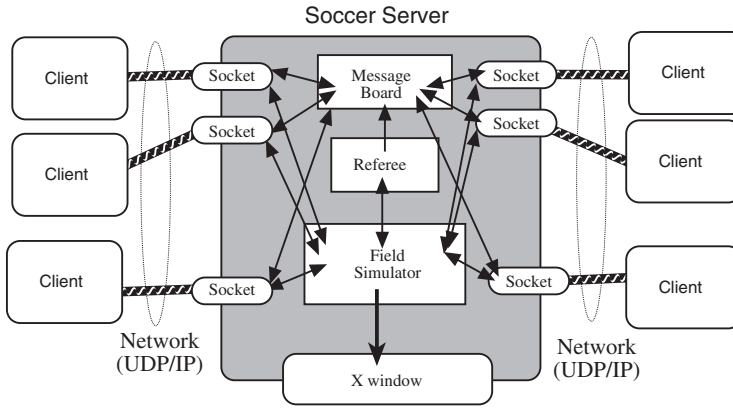

Fig. 1. Architecture of Current Soccer Server

connecting to the server via a computer network and specifying actions for that player to carry out. In return, the client receives information from the player's sensors.

A client controls only a 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. Sensor information tells only partial situation of the field from the player's viewpoint, so that the player program should make decisions using these partial and incomplete information. Limited verbal communication is also available, by which the player can communicate with each other to decide team strategy.

2.2 Features of Soccer Server

Here, I like list up features of Soccer Server that are reason why it is used widely.

- Soccer Server is light. It can run entry-level PCs and requires small resources. This enables researchers to start their research from small environment. And also, 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. Finally, it supports SunOS 4, Solaris 2.x, Linux, IRIX, OSF/1, and Windows [3]. 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.

[3] 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/`

- Soccer Server uses ascii string on UDP/IP for protocol between clients and the server. It enables researchers/students to use any kind of program language. Actually, participants in past RoboCup competitions used C, C++, Java, Lisp, Prolog and various research oriented AI programming systems like SOAR [8]. Version control of protocol is also an important feature. It enables us to use old clients to run in newer servers.
- The system has a separated module, `soccermonitor`, for displaying the field status on window systems. Simulation kernel, `soccerserver`, permits to connect additional monitors. While this mechanism was introduced only for displaying field window on multiple monitors, it leads unexpected activities in the different research field. Many researchers have made and have been trying to build 3D monitors to demonstrate scene of matches dynamically [6]. In addition to it, a couple of groups are building commentary systems that describe situations of matches in natural language dynamically [9, 1]. Both kinds of systems are connected with the server as secondary monitors, get information of state of matches, analyze the situations, and generate appropriate scenes and sentences.

2.3 Open Issues of Soccer Server

During past three yeas, Soccer Server was modified again and again in order to add new functions and to fix bugs. From these experience, it became to be clear that Soccer Server have the following open issues.

- huge communication:
  Soccer Server communicates various clients (player clients, monitor clients, offline-/online-coach clients) directly, so that the server often becomes a bottle-neck of network-traffic. In order to solve this problem, the server should be re-designed to enable distributed processing easily.
- maintenance problem:
  Though Soccer Server is maintained only at ETL, the source code became so complicated that it is difficult to figure out bugs and to maintain the code. The reason is that structure of classes of C++ program became not to reflect a hierarchy of required functions. Therefore, it is the time to re-design modules of the server according to required functions.
- version control:
  In order to keep upper compatibility as much as possible, Soccer Server uses version control of protocol between clients and the server. Because the current server is a single module, the server must include all version of protocols. In order to solve the problem, the server should have a mechanism that enable to connect with a kind of filter or proxy that convert internal representation and each version of the protocol.

In order to overcome these problems, I introduce a modular architecture into Soccer Server. In this architecture we divided Soccer Server into a couple of modules, which are loosely coupled via networks. These modules can run in a

distributed way, so that we will be able to avoid bottle-neck problem of huge communication. Modularity also provides the way of distributed maintenance of the system. Also, it makes easy to version control by swapping modules to communicate player's clients for each version.

In the next section, I propose a new design of the simulator based on this idea.

3 A Design of New Simulator

3.1 Overview of the Design

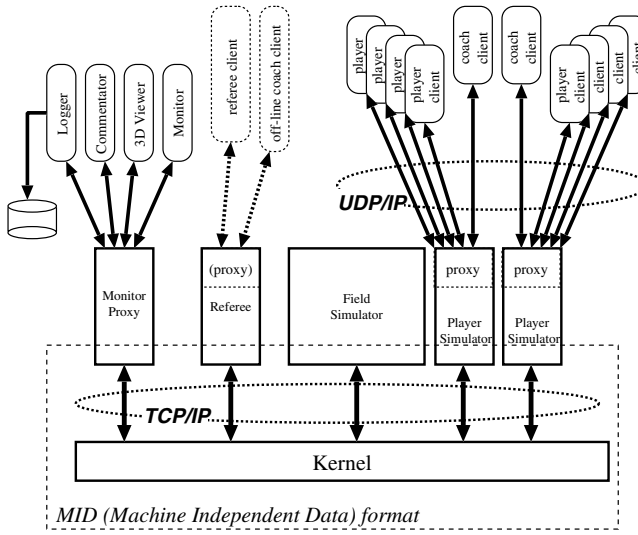

Fig. 2. Plan of Design of New Soccer Server

As mentioned in the previous section, I divided the functions of Soccer Server into the following modules:

- **Field Simulator** is a module to simulate the physical events on the field respectively.
- **Referee Module** is a privileged module to control a match according to rules. This module may override and modify the result of field simulator.
- **Player Simulators/Proxies** are modules to simulate events inside of player's body, and communicate with player and on-line coach clients.
- **Monitor Proxy** provides a facility of multiple monitor, commentator, and saving a log.

These modules are combined by a **kernel** (Fig. 2). The **kernel** manages shared data and synchronization among the modules. Each module communicates only with the kernel rather than with each other directly. In order to guarantee to run modules in various platforms, the system use **MID**, a platform-independent format, in the communication. [4]

The kernel keeps the primary data, and each module has its copy. When a module changes the data, it uploads the change to the kernel. Then other modules refer the changed data in the kernel.

The kernel treats all modules in a uniformed way. So, we can add additional modules for the system. Also, we can apply this kernel to different purposes like a rescue simulator.

3.2 Kernel

Kernel is a back born of whole system. The kernel will provide the following services to the modules:

- **Management of Shared Data**: All shared data are kept in the kernel as the primary data. Each module should have a copy of the primary data. When a module modify the data, the module must upload the data to the kernel. Other modules download the data when they use it. The kernel also provides *automatic* download mechanism. If a module is registered as a *watcher* of the data, the kernel *automatically* downloads the data when it is modified.
- **Control of Synchronization by Phase**: In order to synchronize executions of modules, the kernel provides *phase* facilities. The kernel begins a *phase* when a certain condition is satisfied. Then it notifies the beginning of the phase to all modules join the phase. Each module notifies the end of operation of the phase to the kernel. The kernel ends the phase when it receives the notification from all joined modules.
  The *phase* mechanism provides the way to give a priority to a certain operation of a certain modules. The kernel can begin a phase adjunctly before or after another phase. Therefore a user can put phases in order using the relation of adjunctness.

At the beginning of the simulation, each module connects to the kernel, and registers shared data and joining phases. Then the module start to receives services phase controls and automatic download of data. Also, the module can upload/download the data when it needs.

As mentioned above, the kernel treats all modules in a uniformed way. So there are no restriction on the number and the type of modules. This means that we can connect additional simulation units like an auditory simulator or global coaching modules. Also, we can swap the (2D-) field simulator to a 3D-simulator easily. (In this case, we may have to change other modules to adjust new data

[4] Currently **MID** is defined originally. However, I will move to use more common way like CORBA.

format of 3D.) The kernel is also applicable to other domains. For example, the kernel can be used in a rescue simulation. In this case, a couple of simulation modules like fire simulator, traffic simulator, and so on.

3.3 Player Simulator/Proxy

One of major problems of the current Soccer Server is management of protocol. In the Soccer Server, the protocol is implemented in various point of the whole system. Therefore, it is difficult to maintain and version-up the protocol.

On the other hand, in the new design, A player simulator/proxy receives whole information about data from the kernel, and convert it to the suitable protocol. As a result, maintainers may focus only to this module when we change the protocol.

This style brings another merit. The current system communicates with clients directly, so it the server tends to be a bottle neck of network traffic. On the other hand, this module works as a proxy that connects with multiple clients. Therefore, when we run two proxies for both teams on two machines placed in separated sub-networks, we can distribute the traffic. This also equalizes the condition for each team even if one team uses huge communication with the server.

3.4 Referee Module

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 to behaviors of other modules directly rather than data. For example, the referee module restricts movements of players and a ball, that are controlled by the field simulator module, according to the rule.

One solution is that the referee module only controls flags that specify the restrictions, and simulator modules runs according to the flags. The problem of this implementation is that it is difficult to maintain the referee module separately from other modules.

Another solution is that the referee module is invoked just before and after the simulator module and check the data. In other words, the referee module works as a 'wrapper' of other modules. The merit of this implementation is that it is easy to keep simulator modules independent from referee modules. Phase control described in Sec. 3.6 enables this style of implementation in a flexible manner.

3.5 Protocol Between Kernel and Modules

Protocol between the kernel and modules uses TCP/IP, not UDP/IP. The reason is that:

- Communication between the kernel and modules should be reliable. For example, if communication of the phase control is not reliable, the execution of

the whole system may fall into deadlock because of fault of communication. Because of the nature of UDP/IP, it is difficult to avoid such fault completely in UDP/IP during a match.
- The number of packets will be smaller than communication between Soccer Server and its clients, so that overhead of TCP/IP will be negligible.

Of course, the low-level protocol layer is designed to be independent from high-level layer. We can replace this level if more efficient protocol will be available in the future.

In order to guarantee independence of data format from machine architecture, I designed MID (Machine Independent Data) format. All data are reformed into network byte order (big-endian). In addition to it, we can use fixed decimal point format to transfer float values. It is useful because most of physical value has a limited domain of value and fixed decimal point format can reduce the size of data.

All conversion methods of data to/from MID format are defined shared header files of C++ in an object oriented manner. For example, 'position' and 'player' classes will be defined as Fig. 3.

Note that we can keep that protocol between the server and player clients is UDP/IP. The conversion of TCP/IP and UDP/IP is done by the player simulator/proxy modules.

3.6 Phase Control

The kernel controls synchronization of execution of modules by *phases*.

A *phase* is a kind of an event that have joined modules. When a phase starts, the kernel notifies the beginning of the phase by sending an **achievePhase** message to all joined modules. Then the kernel waits until all joined modules finish operations of the phase. Each module must inform the end of the operation of the phase by sending an **achievePhase** message to the kernel.

The kernel can handle two types of phases, *timer phase* and *adjunct phase*.

A *timer phase* has its own interval. The kernel try to start the phase for every interval. For example, a **field simulation phase** should occur every 100ms [5].This phase has the field simulator as a joined module. So, the field simulator receives an **achievePhase** message for every 100ms. Then the simulator executes its operation and sends an **achievePhase** message back to the kernel.

An *adjunct phase* is invoked before or after another phase adjunctively. For example, 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.

[5] This interval is based on the interval of the original soccer server.

```
class FsPos {
  public:
    FsFloat x ;
    FsFloat y ;
       ...
    FsBool writeMID(FsBuffer buffer) {
      writeMID(buffer,x,10) ;
      writeMID(buffer,y,10) ;
    } ;
      // precision of x and y are 3 decimal
      // places under decimal points.
    FsBool readMID(FsBuffer buffer) {
      readMID(buffer,x,10) ;
      readMID(buffer,y,10) ;
    } ;
      ...
} ;
```

```
class FsPlayer {
  public:
    FsSide  side ;
    FsUInt  unum ;
    FsPos   pos ;
       ...

    FsBool writeMID(FsBuffer buffer) {
      writeMID(buffer,side) ;
      writeMID(buffer,unum) ;
      writeMID(buffer,pos) ;
      ...
    } ;
    FsBool readMID(FsBuffer buffer) {
      readMID(buffer,side) ;
      readMID(buffer,unum) ;
      readMID(buffer,pos) ;
      ...
    } ;
    ...
} ;
```

Fig. 3. Example of definition of data structures and their conversion to/from MID format

A phase may have two or more adjunct phases before/after it. To arrange them in an order explicitly, each adjunct phase has its own tightness factor. 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 last.

Fig. 4 shows phase-control and communication between the kernel and modules in the soccer simulation. Note that the implementation of the phase control mechanism is general and flexible, so that there is no limitation on the number of phase, the duration of the interval of timer phase, or the depth of nest of adjunct phases.

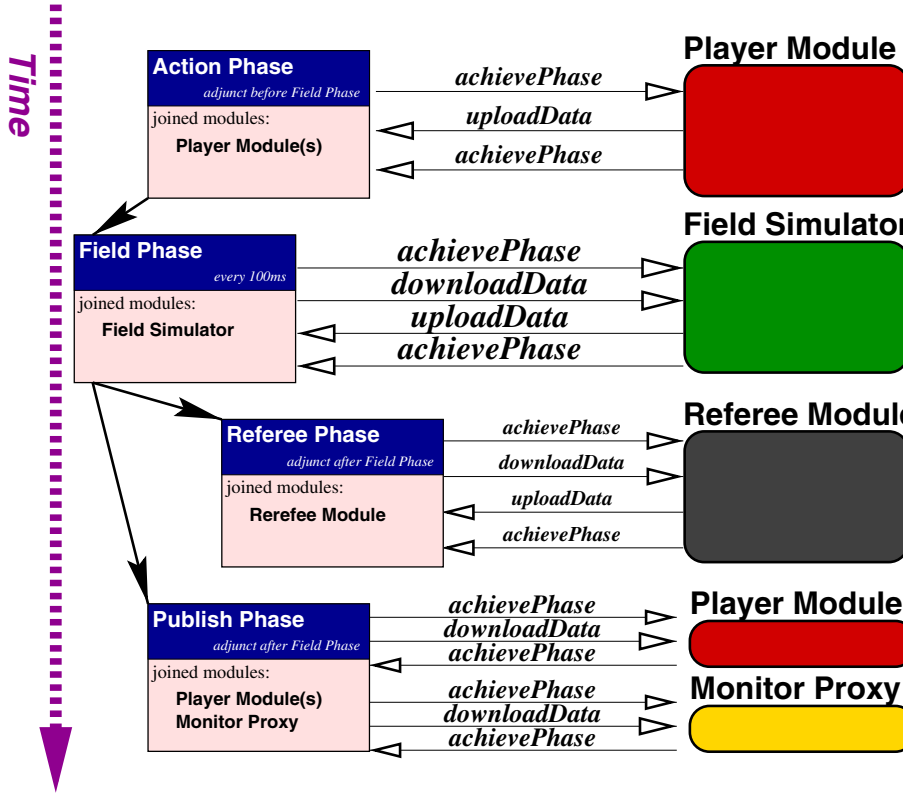

Fig. 4. Phase Control and Communication with Joined Modules

4 Related Work

4.1 Distributed Interactive Simulation

There is a series of development of distributed interactive simulation systems for military training and simulation [3, 2, 7, 11]. The most significant features of such kind of simulations is that the area of the field is very wide and objects are located relatively sparse. They use the similar architecture of the new simulator proposed here. Their main purpose is to connect simulators developed individually via network, and to enable integrated training of pilots simulate war in large scale in real-time. In addition to it, the system can connect with machine-intelligented pilots [8].

Major differences between the new simulator and these military simulators are:

- Existence of referee module is significant compared with other simulator. Referee module is tightly coupled with the field simulator, so the kernel is required to control these two modules in sequential manner. On the other hand, military simulators listed above suppose interaction between each simulation modules are localized, so that it is possible to build loosely coupled system.
- The new simulator should be light weight. One of important features is light weight and ability for researchers to run it with low cost of computer resources. In the military purpose, more realistic simulation is required even if it needs more expensive computational resources.
- The new simulator will be released under Gnu GPL. Open source policy is important in RoboCup community.

4.2 Hybrid Simulation

Hybrid simulation systems also have been investigated. [10] shows a core architecture to enable a hybrid simulation of embedded systems. Compared with the military simulations and the proposed systems, this hybrid simulation aims to simulate a system consists of more tightly coupled elements like inside of a circuit, rather than to simulate events happen in widely spread area like combat/soccer field. However, when we move to more accurate and multi-modal simulation (for example, rescue domain), we must take care the similar problem they attacked.

5 Conclusion

I investigated problems of current Soccer Server, and figure out issues that should be solved in the new simulator. Two main points are modularity and possibility of distributed simulation. Base on this investigation, I proposed a design of the new simulator. In which, simulation and communication are divided into deferent modules.

The proposed design is relatively general and is not restricted to simulation of Soccer. So, it is possible to use this design as a prototype of the kernel of other simulation of complex environment like rescue from huge disasters.

There still remain many open issues. For example, the following issues are still open:

- tradeoff between reality flexibility and computational power
- tradeoff between tight coupling and loose coupling
- generality of interaction between modules
- timing control over networks

References

1. 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.
2. Judith S. Dahmann. High level architecture for simulation: An update. In Azzedine Bourkerche and Paul Reynolds, editors, *Distributed Interactive Simulation and Real-time Applications*, pages 32–40. IEEE Computer Society Technical Committee on Pattern Analysis and Machine Intelligence, IEEE Computer Society, July 1998.
3. Xin Li, Kien A. Hua, and J. Michael Moshell. Distributed database designs and computation strategies for network interactive simulations. *Journal of Parallel and Distributed Computing*, 25:72–90, 1995.
4. Itsuki Noda and Ian Frank. Investigating the complex with virtual soccer. In Jean-Claude Heudin, editor, *VW'98 Virtual Worlds (Proc. of First International Conference)*, pages 241–253. Springer, July 1998.
5. Itsuki Noda, Hitoshi Matsubara, Kazuo Hiraki, and Ian Frank. Soccer server: A tool for research on multiagent systems. *Applied Artificial Intelligence*, 12(2–3):233–250, 1998.
6. 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.
7. Stow 97 - documentation and software. WWW home page `http://web1.stricom.army.mil/STRICOM/DRSTRICOM/T3FG/SOFTWARE_LIBRARY/ST% OW97.html`.
8. 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.
9. 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.
10. Werner van Almsick, Thorsten Drabe, Wilfried Daehn, and Christian Müller Schloer. A central control engine for an open and hybrid simulation environment. In Azzedine Bourkerche and Paul Reynolds, editors, *Proc. of Distributed Interactive Simulation and Real-time Applications*, pages 15–22. IEEE Computer Society, July 1998.
11. Warfighters' simulation (warsim) directorate national simulation center. WWW home page `http://www-leav.army.mil/nsc/warsim/index.htm`.