Framework of Distributed Simulation System for Multi-agent Environment

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

Electrotechnical Laboratory, Tsukuba 305, Japan

Abstract. Simulation systems are important tools for researches on Multi-agent systems. As a research tool, a simulation system should have features of: (1) light-weight and small requirement of computational resources, and (2) scalability to add new functions and features. I investigate experience of development of Soccer Server, the official soccer simulator used in RoboCup Simulation League, and propose a new framework for distributed simulation system for general purpose of multi-agent researches.

In the new framework, a simulation is divided into several modules and executed in parallel. These executions are combined by a kernel module via a computer network. Because of the modularity over networks, users easily maintain the development of simulation system.

I also discuss about the relation to HLA, another framework for distributed military simulation system.

1 Introduction

"Simulation" is an important research tool on multi-agent systems [2]. Most of multi-agent systems, in which each agent behaves intelligently and adaptively, are complex systems. Generally, it is hard to apply simple mathematical analysis to such complex systems, so that computer simulation is an indispensable method to investigate behaviors of multi-agent systems.

I developed Soccer Server[6], a soccer simulation system, as a tool for researches on multi-agent systems. It has been used as an official platform in RoboCup Simulation League. 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. In the same time, it becomes clear that Soccer Server has many design problems. It was originally developed just as a prototype of the simulator, and modified again and again to add new features according to requirements from various viewpoints of researches. Therefore, the system became complicated and difficult to maintain.

P. Stone, T. Balch, and G. Kraetzschmar (Eds.): RoboCup 2000, LNAI 2019, pp. 229-238, 2001.

Based on these experience, I investigate requirement to a platform for simulation of multi-agent systems, and propose a new framework for it, which will provide:

- capability to execute the simulation in distributed way,
- facility to add new functions easily to the simulation.

In the following sections, I investigate features and problems of the current Soccer Server, and describe the proposed framework.

2 Soccer Server and its Problems

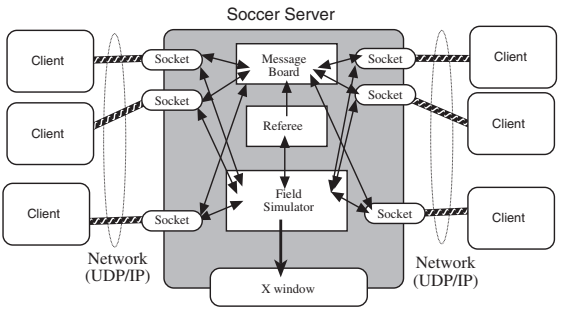

Fig. 1. Architecture of Current Soccer Server

2.1 Soccer Server

Soccer Server [6, 5] 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 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

Soccer Server has been used as an official platform for RoboCup Simulation League. In addition to it, it is used widely for research, education and entertainment. 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 [1] . 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. 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 [10]. 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 [8]. In addition to it, a couple of groups are building commentary systems that describe situations of matches in natural language dynamically [11, 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 with various clients (player clients, monitor

[1] 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/`

clients, offline-/online-coach clients) directly, so that the server often becomes a bottle-neck of network-traffic. Especially, communication with 22 player clients is a big issue. If a term use many communication, it causes network collisions and slow down of the server, so that another team may be influenced. This problem will be solved by distributed processing over the network.
- maintenance problem:
  Though Soccer Server was maintained by a small number of developers, 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.

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

- This evolved many activities to develop systems to show plays in 3D, to describe game status in natural language, and to analyze performance of teams from various point of view. This is possible because the modules are connected via networks. Therefore, each developer can develop these systems independently.
- This makes 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).

I apply the similar technique to other part of the simulator. In the next section, I describe the general framework, called FUSS, for distributed simulation based on the idea, and show the implementation of the soccer simulator as an example.

3 FUSS: An Universal Simulation System for Multi-Agent System

3.1 Overview

FUSS (Framework for Universal Simulation System) is a collection of programs and libraries to develop distributed simulation systems. It consists of the following items:

- `fskernel`: A kernel for a simulation system. This provides services of:

  - module database
  - shared memory management
  - synchronization control
- FUSS library (libfuss): A library to develop modules of the simulation system. The library consists of:
  - `FsModule` library: provides a framework of a simulation module.
  - `ShrdMem` library: provides an interface to access shared memories.
  - `PhaseRef` library: provides a facility to control synchronization of simulation.
- utility library and programs: A collection of utilities.

Generally, a simulation system on FUSS consists of a kernel (fskernel) and a couple of simulation modules. The modules are combined into a system by the kernel via the FUSS library. Fig. 2 shows a brief structure of an example of a simulation system built on FUSS.

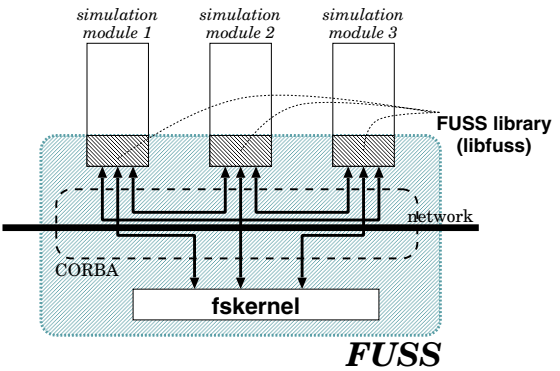

Fig. 2. A Simulation System Built on FUSS

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

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

Currently, FUSS is implemented by using C++ and OmniORB2[7]. It is available from the FUSS homepage:

`http://ci.etl.go.jp/~noda/soccer/fuss/`

3.2 Shared Memory Management

A shared memory is defined as a sub-class of `ShrdMem` class in each module. The memory is registered to the kernel before the simulation. Then, a module becomes an owner of the memory, who has an original data, and other module have its copy. The ownership can be transfered by calling `takeOwnership` method explicitly.

Modules need not know who is owner of the shared memory. A module calls `download` method before using the shared memory, and calls `upload` method after modifying the memory. Then the FUSS library maintain the consistency of the memory.

Each shared memory must be defined by IDL of CORBA. The definitions are converted into C++ classes and included by all related modules. Because FUSS uses CORBA, we can develop modules or utilities by other CORBA-compliant programming systems.

3.3 Synchronization by Phase Control

In a simulation on FUSS, modules are synchronized by `fskernel` by *phase*s.

A *phase* is a kind of an event that have joined modules. When a module is plugged into the simulation system, the module send **joinPhase** messages to `fskernel` to joins a couple of phases in which it executes a part of simulation. When the 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 tries to start the phase for every interval. For example, suppose that we develop a soccer simulator, in which a field simulator module calculate objects' movements every 100ms. In this case, we will define a timer phase (named as **field simulation phase**) that has the field simulator module as a joined module. The interval of the phase is set to 100ms. Then, the field simulator receives an **achievePhase** message for every 100ms, and executes its operation and sends an **achievePhase** message back to the kernel.

An *adjunct phase* is invoked before or after another phase adjunctively. In the example of soccer simulation, suppose that referee's judgments, which is operated by a referee module, should be performed just after the field simulation. In this case, 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 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** may 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. 3 shows phase-control and communication between the kernel and modules in the soccer simulation example. 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.

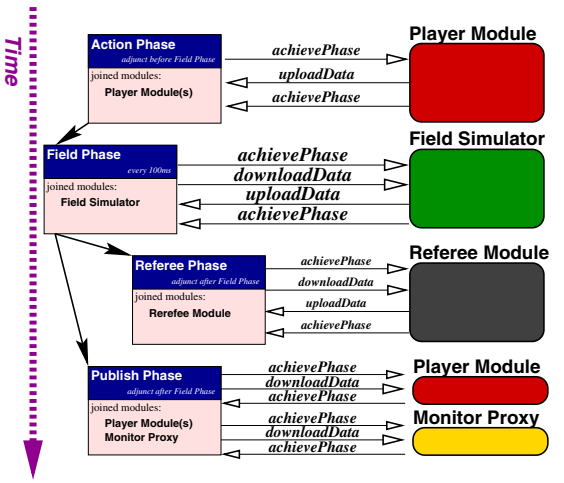

Fig. 3. Phase Control and Communication with Joined Modules

4 Implementation of Soccer Simulator on FUSS

4.1 Overview of the Design

I implemented Soccer Simulator using FUSS. In the implementation, 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.

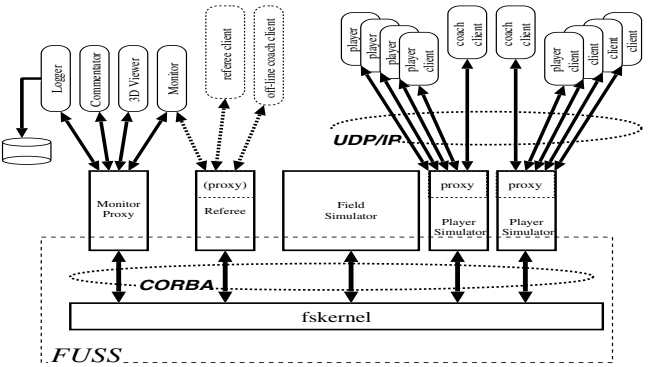

Fig. 4. Plan of Design of New Soccer Server

- **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.

Execution of these modules are controlled by phases shown in Fig. 3. Note that the execution and communication with clients are processed in parallel using the multi-thread facility.

4.2 Player Simulator/Proxy

As mentioned in Sec. 2, 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.

In the new design, on the other hand, 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.

4.3 Referee Module

The implementation of the referee module is the key of the simulator. Compared with other modules, the referee module should have a prerogative, because the referee module needs to affect to behaviors of other modules directly. For example, the referee module restricts movements of players and a ball, which are controlled by the field simulator module, according to the rule.

We may be able to realize the prerogative by it that the referee module only controls flags that specify the restrictions, and simulator modules runs according to the flags. In this implementation, however, it is difficult to maintain the referee module separately from other modules.

In the implementation using FUSS, 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. 3.3 enables this style of implementation in a flexible manner.

5 Related Work and Discussion

Researches on distributed simulation systems are done mainly for military simulation and training [9, 12]. DIS (Distributed Interactive Simulation) [3] and HLA (High-Level Architecture) [4] have been developed for this purpose. These works focuses on connecting stand-alone simulators like fright simulators over networks. In such simulation, accuracy of simulation of behaviors of each planes and vehicles is more important than interaction between them. Therefore, DIS and HLA is designed mainly for synchronizing relatively independent simulators.

In simulation of multi-agent systems like Soccer, on the other hand, interactions among agents occur more frequently. Therefore, execution of each simulation module should be controlled more carefully. Moreover, the simulation may be regulated by complex rules that related the whole simulated world. For example, the soccer simulation have to be carried out according to rules of soccer that is related to total condition of the field. In order to implement such regulation, each simulation module needs to have a function to deal with it in HLA, in which it is difficult to maintain the regulation. Compared with this, it is easy to implement such regulation using FUSS, because FUSS has a strong ordering mechanism by phase control. In other words, FUSS has an advantage in implement tightly coupled simulation with supervised modules.

6 Summary

I investigated problems of current Soccer Server, and figured 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

framework for general purpose multi-agent simulation called FUSS. FUSS enables for us to develop simulation systems that run in a distributed way over computer networks.

The proposed framework is 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.

Moreover, FUSS uses CORBA as a base of communication among the kernel and modules. While the current implementation provides only a library for C++. it is possible to develop libraries for other languages.

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. J. L. Casti. *Would-be Worlds: how simulation is changing the frontiers of science.* John Wiley and Sons, Inc., 1997a.
3. Thomas L. Clarke, editor. *Distributed Interactive Simulation Systems for Simulation and Training in the Aero Space Environment*, volume CR58 of *Critical Reviews Series.* SPIE Optical Engineering Press, April 1995.
4. 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.
5. 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.
6. 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.
7. omniorb homepage. WWW home page `http://www.uk.research.att.com/omniORB/omniORB.html`.
8. 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.
9. Stow 97 - documentation and software. WWW home page `http://web1.stricom.army.mil/STRICOM/DRSTRICOM/T3FG/SOFTWARE_LIBRARY/ST% OW97.html`.
10. 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.
11. 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.
12. Warfighters' simulation (warsim) directorate national simulation center. WWW home page `http://www-leav.army.mil/nsc/warsim/index.htm`.