Essex Wizards'99 Team Description

H. Hu, K. Kostiadis, M. Hunter, M. Seabrook

Department of Computer Science, University of Essex
Wivenhoe Park, Colchester CO4 3SQ, United Kingdom
Email: {hhu,kkosti,mchunt,seabmn}@essex.ac.uk

Abstract: This paper describes the Essex Wizards team participated in the RoboCup'99 simulator league. It is mainly concentrated on a multi-threaded implementation of simulated soccer agents to achieve real-time performance. Simulated robot agents work at three distinct phases: sensing, thinking and acting. POSIX threads are adopted to implement them concurrently. The issues of decision-making and co-operation are also addressed.

1. Introduction

In the RoboCup simulator environment, the response time of a soccer agent becomes significantly important since the soccer server operates with 100ms cycles for executing actions and 150ms cycles for providing sensory data [12]. Moreover, auditory sensory data can be received at random intervals. It is vital that each agent has bounded response times. If an action is not generated within 100ms, then the agent will stay idle for that cycle and enemy agents that did act might gain an advantage. On the other hand, if more than one action is generated per cycle, the server will only execute one of them chosen randomly. An additional constraint is that Unix is not a "true" real-time system and hence real-time performance and response times can only be guaranteed up to a certain resolution [13]. A more detailed description of real-time systems can be found in [2,11].

In addition to the responsiveness, the ability to cope with changes in the agent's environment provides a significant advantage, especially when the environment is noisy, complex, and changes over time [5]. One of important issues for an agent therefore is to learn from its environment and past experience in order to autonomously operate without the need of human intervention. Another important issue in multi-agent systems is co-operation. It has been shown that groups of agents can derive more efficient solutions in terms of energy, costs, time and quality [6,7]. A common feature in co-operative frameworks is that of distribution of responsibilities and multiple roles. Each agent in a group has an individual role and therefore a set of responsibilities in the team [4]. In this article a form of emergent co-operation through reinforcement learning is presented. Using the back-propagating nature of Q-learning, co-operation is achieved by linking the intermediate local goals.

In section 2, a description of the agent requirements is presented. The agent architecture for Essex Wizards is illustrated in section 3. Then how multiple threads have been implemented to improve the agent's responsiveness is explained in section 4. Section 5 illustrates how machine learning is used in our team for decision making and co-operation of multiple agents. Finally conclusions and future work are briefly presented in section 6.

M. Veloso, E. Pagello, and H. Kitano (Eds.): RoboCup-99, LNAI 1856, pp. 558–562, 2000.

2. Agent Requirements

The robotic soccer simulator is an instance of a client/server application in which each client communicates with the server via a UDP socket [12]. The server is responsible for executing requests from each client and updating the environment. At regular time intervals (150ms) the server broadcasts visual information to all clients depending on their position, the quality and size of the field of their view, and their facing direction on the field. In addition to that, the server sends auditory information to various clients at random time intervals.

After processing the sensory data, the clients respond by sending action requests to the server from a set of primitive actions available to them. To avoid message congestion on the server, the clients are allowed to send one request per cycle. A cycle in the current implementation is 100ms. If no message is sent within this interval, the client will not perform any actions. If more than one message is send during the same cycle, the server executes only one, chosen at random, which might produce undesired results. The server updates the state of the environment by serially executing each request.

Since UDP sockets have a limited receive buffer, messages arriving on a UDP socket will be queued until the receiving buffer is full in which case additional messages will be discarded. A client that fails to retrieve the messages at the rate that they arrive is in danger of receiving older information from the server, since newer data will be further back in the queue. This will cause the client to create the wrong representation about the current state of the pitch, which will lead to undesired effects since the wrong actions might be executed. The term “client”, used in the client/server application context above, is the real-time agent to be built. For each cycle, the agent receives data from the server, and produces an action. When new data is available, the agent should receive this data and update the current state of the environment. For efficiency the following conditions should be satisfied:

- To receive the newest sensory data that arrives on the socket as quickly as possible and no data queue up. This enables agents to have the most recent representation of the environment, and execute the most appropriate action.
- To time the execution of requests to the server accurately. If more than one request is send by an agent per cycle, the server will only execute one at random, which might be non-optimal. If the agent is too slow, it might miss a cycle and then give an advantage to the enemy agents.
- To allow the maximum time and resources for the thinking process. Since an agent has a fixed amount of time per cycle, the longer it waits to send or receive data, the less time it has to think.

Given the frequency of the message exchange and the timing constraints, building an agent that will satisfy the conditions described above becomes a challenging task.

3. Agent Architecture

Given the variety of I/O models supported under Unix, it becomes difficult to choose the most suitable one for the soccer agents. In addition to that, choosing an I/O model heavily depends upon the inner structure of the agent. As it can be seen in figure 3,

the agent contains six different modules, including the agent's sensors, a set of behaviours, the actuators, the current play mode, a set of predefined parameters and a memory module. The dashed arrows in this diagram represent the communication links between the agent and the server. The server sends information received by the agent's "Sensors" module, and the agent sends information back to the server through its "Actuators" module. A brief description is given below to show the relationships among these modules:

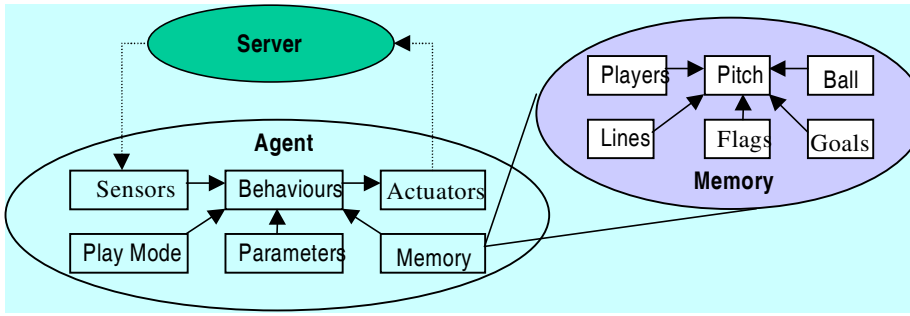

Figure 1 Block diagram of the soccer agent's architecture

- Sensors -- are responsible for receiving and analyzing the visual or auditory information transmitted by the server.
- Behaviours -- is the most important module within the agent. It is responsible for generating actions according to the current state of the environment.
- Actuators -- are used for timing and sending actions to the server every 100ms.
- Play Mode -- holds and updates the current play mode using the information received by the "Sensors" module. The current play mode directly affects the behaviour of the agent
- Parameters -- hold information regarding various settings both for the server and the agent.
- Memory -- is a representation of the whole pitch, rather than part of it.

4. Multi-Threaded Implementation

Instead of a single-thread, a process can have multiple threads, sharing the same address space and performing different operations independently. This architecture allows the agent to use a separate thread for each of the three tasks. Inside the agent, the three main tasks are running concurrently (or in parallel in multi-processor hardware) minimizing delays from the I/O operations. Only the "Sense" thread is responsible for waiting data from the server, and only the "Act" thread is responsible for timing and sending the. In this way the agent can dedicate the maximum amount of processing power available by the processor(s) to the "Think" thread.

Firstly the "Sense" thread is dedicated to receiving data using a blocking I/O mode connection in which the *rcvfrom* call will put the "Sense" thread to sleep until new data arrives on the socket. Putting the "Sense" thread to sleep does not affect the

execution of the other two threads. When data arrives on the socket, the "Sense" thread will be awaken, analyze the data, and repeat a *rcvfrom* call for the next available datagram from the server.

Secondly, the "Act" thread needs to measure 100ms intervals, and sends any available actions to the server. The *gettimeofday* function and a conditional variable are used to implement the "Act" thread. In other words, the current absolute time is incremented by 100ms and then the thread is also put to sleep to wait for the conditional variable to return. Since no other thread will signal this conditional variable, it will only return when the 100ms have passed, in which case the "Act" thread can send an action to the server. This method provides highly accurate timing and enables the agent to guarantee certain levels of timing correctness.

Thirdly the "Think" thread is the only one that stays permanently awake, and consumes the majority of the available resources to perform most of the computations. The details on the "Think" thread in the current implementation can be found in [8]. More details on the multi-threaded implementation and the results are presented in [9]. The issues on multi-threaded programming can be found in [3].

5. Reinforcement Learning

Reinforcement learning (RL) addresses the question of how an agent that senses and acts in its environment can learn to choose optimal actions in order to achieve its goals. There are many systems that use RL for learning with little or no a priori knowledge and capability of reactive and adaptive behaviours [1,10,14,15]. The main advantage of reinforcement learning is that it provides a way of programming agents by reward and punishment without needing to specify how the task is to be achieved. On each step of interaction the agent receives an input *i* which normally provides some indication of the current state s of the environment. The agent then chooses an action to generate as output. The action changes the state of the environment and also provides the agent with a reward of how well it performed. The agent should choose actions that maximize the long-run sum of rewards.

To fully utilize the power of the learning scheme used (Q-learning), the state space is divided by assigning different roles for each individual agent. For example, the goalkeeper need not worry about how to score a goal. This task is indeed numerous stages away given the goalkeeper's responsibilities. It would take numerous iterations before the goalkeeper's training can yield acceptable levels of performance. On the other hand, a goalkeeper can easily learn to pass the ball safely to a nearby defender since this task has a goal-state that is near the goalkeeper's region. In a similar manner a defender can learn to clear or pass the ball to a midfielder and so on.

The agents in the current implementation not only have different roles and responsibilities, but also have different subgoals in the team. Hence, every individual agent in turn tries to reach its own individual goal state without worrying about the performance, or the goals of the other agents. By linking the different goals of each agent, co-operation emerges. Although each agent tries to optimize its actions and reach its own goal-state, since these goal-states are related, the agents co-operate. The ultimate goal, which is to score against the opposition, becomes a joint effort that is distributed between the members of the team. A detailed description of the Essex Wizards RL implementation can be found in [8].

6. Conclusions and Future Work

To satisfy all the necessary timing constraints for real-time agents in general, football-playing robots in particular, a single-threaded implementation will not suffice. This is mainly due to the low speed of network I/O operations, and the limiting serial nature of such architectures. Therefore, a multi-threaded implementation is presented in this article to overcome this problem. Based on this approach, the agents can perform various computations concurrently and avoid waiting for the slow I/O operations. A multi-threaded model clearly outperforms a single-threaded one in terms of responsiveness and efficiency. A decision-making mechanism based on reinforcement learning is briefly described, which can also be used to enable co-operation between multiple agents by distributing their responsibilities. By gathering useful experience from earlier stages, an agent can significantly improve its performance.

The future work for the Essex Wizards team is to focus on cooperative behaviours, team formations, sensor fusion and machine learning capability.

Acknowledgements: We would like to thank the University of Essex for the financial support to the project by providing the Research Promotion Fund DDP940.

References

1. Balch T. R., Integrating RL and Behaviour-based Control for Soccer: Proc. IJCAI Workshop on RoboCup, 1997.
2. Burns A. and Wellings: A., Real-time Systems and Programming Languages, Addison-Wesley, 1997.
3. Butenhof R.D: Programming with POSIX Threads, Harlow, Addison-Wesley, 1997.
4. Ch'ng S., Padgham L.: From Roles to Teamwork: A Framework and Architecture, Applied Artificial Intelligence Journal, 1997.
5. Hu H., Gu D., Brady M.: A Modular Computing Architecture for Autonomous Robots, Int. Journal of Microprocessors and Microsystems, Vol. 21, No. 6, pages 349-362, 1998.
6. Hu H., Kelly I., Keating D., Vinagre D.: Coordination of Multiple Mobile Robots via Communication, Proc. SPIE'98, Mobile Robots XIII, Boston, pp. 94-103, Nov. 1998.
7. Kitano H., RoboCup: The Robot World Cup Initiative, Proceedings of the 1st International Conference on Autonomous Agent (Agents-97), Marina del Ray, The ACM Press, 1997.
8. Kostiadis K. and Hu H.: Reinforcement Learning and Co-operation in a Simulated Multi-agent System, Proc. of IEEE/RJS IROS'99, Korea, Oct. 1999.
9. Kostiadis K. and Hu H.: A multi-threaded approach to simulated soccer agents for the RoboCup competition, IJCAI'99 workshop on RoboCup, 1999.
10. Mataric, J. M., Interaction and Intelligent Behaviour, PhD Thesis, MIT, 1994.
11. Nissanke N.: Realtime Systems, London, Prentice Hall, 1997.
12. Noda I.: Soccer Server: A Simulator for RoboCup, JSAI AI-Symposium 95: Special Session on RoboCup, 1995
13. Stevens W.R.: Unix Network Programming: Networking APIs: Sockets and XTI (Volume 1), London, Prentice-Hall International, 1998.
14. Stone Peter and Veloso Manuela: Team-Partitioned, Opaque-Transition Reinforcement Learning, Proc. 15th Int. Conf. on Machine Learning, 1998.
15. Uchibe E., Asada M., Noda S., Takahashi Y., Hosoda K.: Vision-Based Reinforcement Learning for RoboCup: Towards Real Robot Competition, Proc. of IROS 96 Workshop on RoboCup, 1996.