

Essex Wizards

Essex-Wizards

Kostas Kostiadis, Huosheng Hu

KK, HH: University of Essex

Abstract. *This article is to describe the Essex Wizards team attending the simulation league of RoboCup'99 in Stockholm. A long-term research goal of this team is to focus on cooperative behaviours, team roles and formations, sensor fusion and machine learning capabilities. Therefore, the initial efforts for participating at RoboCup'99 are mainly concentrated on a multi-threaded implementation to simulated soccer agents for the RoboCup competition in order to meet the timing requirements set by the RoboCup soccer server simulator. Since robot agents work at three distinct phases: sensing, thinking and acting, POSIX threads are adopted to break down these phases and implement them concurrently. Implementation results have shown that it outperforms traditional single-threaded approaches in terms of efficiency, responsiveness and scalability. To handle a complex, dynamic, adversarial environment like the one of a football game, this article also describes how machine learning techniques and agent technology have been used in the current implementation, to tackle the decision-making and co-operation problems. By gathering useful experience from earlier stages, an agent can significantly improve its performance and by distributing the responsibilities among the agents, an efficient way of co-operation emerges.*

1 Introduction

The creation of the robotic soccer, the robot world cup initiative (RoboCup), is an attempt to foster AI and intelligent robotics research by providing a standard problem where wide range of technologies can be integrated and examined [8,9]. Some of the fields covered include multi-agent collaboration, strategy acquisition, real-time planning and reasoning, sensor fusion, strategic decision making, intelligent robot control, and machine learning. Given the nature of the RoboCup 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 visual sensory data [14]. Moreover, auditory sensory data can be received at completely 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, which might result to a

non-optimal solution. 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 [15]. Real-time systems are a large topic and only certain properties of real-time systems that improve the performance of the soccer agents are investigated in this article. A more detailed description of real-time systems can be found in [2,13].

In addition to the responsiveness of an agent, the ability to cope with changes in its environment also provides a significant advantage, especially if the environment is noisy, complex, and changes over time [6]. In multi-agent environments the need for such adaptability becomes necessary. The first important issue is for an agent 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 [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 [4,5]. In this article a form of emergent co-operation through reinforcement learning is presented. In other words, using the back-propagating nature of Q-learning, co-operation is achieved by linking the intermediate local goals each agent is trying to achieve.

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

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 [14]. 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 sent 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. The results are projected on a window similar to that shown in figure 1. Figure 1: Robot Soccer Simulator

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 environment, 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 (if new data is available), processes this data, and produces an action. A very basic model of an agent’s loop can be seen in figure 2. When new data is available, the agent should receive this data and update the current state of the environment. It should then “think” and produce an action that it will send back to the server. For the agent to be efficient the following conditions need to be satisfied:

- To receive the newest sensory data that arrives on the socket as quickly as possible, and do not let data queue up. This enables the agent to have the most recent representation of the environment, and hence execute the most appropriate action.
- To time the execution of requests to the server accurately. If the agent is too fast and more than one request is send per cycle, the server will only execute one at random, which might have undesired effects. If the agent is too slow, it might miss a cycle and then give an advantage to the enemy agents.
- To allow the maximum amount of time and resources for the thinking process. Since an agent has a fixed amount of time per cycle, the longer it spends waiting 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. These modules include 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. The solid arrows in this diagram represent dependence relationships between the modules. For example, the “Actuators” module is dependent upon the “Behaviours” module. This essentially means that the output produced by the “Actuators” module, is directly dependent upon the input it will receive from the “Behaviours” module. The rest of the modules within the agent affect the “Behaviours” module. A brief description for each of these modules will provide a better understanding of the relationships between them.

- **Sensors** – are responsible for receiving and analysing the visual or auditory information transmitted by the server. After receiving the data for each cycle, the agent creates a representation for the current state of the environment. Due to the limited field of view, the agent updates only part of the environment in each cycle. When new information arrives from the server, the old information is passed to the memory module, which holds a probabilistic representation for the whole environment.
- **Behaviours** – is the most important module within the agent. It is responsible for generating actions according to the current state of the environment. The state of the environment is determined using all the modules within the agent apart from the actuators. Normally the agent needs to gather information from all other modules before an action is generated, including data regarding the agent’s position and role in the team, the current formation, the position of the ball and other agents, the current play mode and so on. The “Behaviours” module should process all this information and determine the best course of action.
- **Actuators** – are responsible for timing, and sending actions to the server. As mentioned earlier the server accepts one action each 100ms. The actuators receive an action (or a set of actions) from the “Behaviours” module, and they send these actions to the server.
- **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. For example the agent is expected to act differently if a free kick has been won for its team or if the opposition has won a corner kick.
- **Parameters** – hold information regarding various settings both for the server and the agent. The parameters affect the behaviours of the agent since they include information regarding the current formation, the role of the agent in the team, and most of the server parameters including use of offside rule, player’s size and maximum speed.
- **Memory** – is a representation of the whole soccer pitch, rather than a partial representation like the one provided by the sensors. The “Pitch” contains players, lines, flags, goals and a ball. Each of these objects is associated with a confidence value. This value represents the agent’s confidence that an object is at the current co-ordinates. If an object was seen in the last cycle, its confidence value is 1. It is decreased by a certain value if the object has not been seen in the current cycle. When this value falls under a certain threshold the object is “forgotten”.

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 and without affecting each other. This architecture allows the agent to use a separate thread for each of the three tasks. The multi-threaded model of the proposed implementation can be seen in figure 4. Inside the agent, the three main tasks are running concurrently (or in parallel in multi-processor

hardware) minimising 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 actions (these relationships are indicated by the dashed arrows). This way the agent can dedicate the maximum amount of processing power available by the processor(s) to the “Think” thread.

Firstly it is necessary to specify the I/O model that is going to be used. This now becomes much simpler since there are separate execution threads for input and output. The “Sense” thread can now use a blocking I/O connection. Since this thread is now dedicated to receiving data, it does not need to waste processing time querying the socket as to whether data is available. With a blocking connection 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 that can proceed as normal. When data arrives on the socket, the “Sense” thread will be awoken, analyse the current data, and then repeat a `rcvfrom` call for the next available datagram from the server. This approach does not allow for datagrams to be lost, or queue up. This would only happen if the server transferred data faster than the thread could analyse it, which is not possible.

Secondly, the “Act” thread needs to measure 100ms intervals, and send any available actions to the server. By having a dedicated thread to perform this task, the accuracy of the timing is only limited by the resolution of the operating system’s clock. 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 waits for the conditional variable to return. Assuming spurious wake-ups will not occur, and 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 comparing to the single-threaded approaches described earlier. This enables the agent to guarantee certain levels of timing correctness, which is something single-threaded approaches failed to do. In addition to that, this thread is also put to sleep while waiting for the conditional variable to return. This provides the other threads with the maximum amount of resources available.

Finally 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. Part of the “Think” thread of the current implementation can be found in [10]. If required, various scheduling policies, and different priorities between the threads could be implemented. However, given the nature of the problem, no scheduling is needed. In addition to that, synchronisation between separate threads can also be implemented if required. A more detailed description of the Essex Wizards multi-threaded implementation, as well as experimental results, can be found in [11]. Multi-threaded programming is a large topic itself. Describing issues such as how threads compete for resources, or how often pre-emption occurs falls outside the scope of this article. A detailed description 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 [1,12,16,17,18]. 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 a 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 maximise the long-run sum of rewards.

To fully utilise 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 current implementation introduces agents that not only have different roles and responsibilities, but also have different goals in the team. In other words, although the overall goal of the team is to score against the opposition, each agent's local goal is different. 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 optimise 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, as well as experimental results, can be found in [10].

6 Conclusions and Future Work

It has been shown in this article that the behaviour of real-time agents is not founded only on the logical correctness of their actions. Timing correctness becomes an equally important factor especially in applications where response times can significantly affect the result. In such cases the quality of the result becomes a function of both logically correct output and response time. In fact there are cases where response time is much more important than the actual logical correctness of the output.

To satisfy all the necessary timing constraints for a real-time agent, 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 proposed in

order to overcome this problem. Based on this approach, the agents can perform various computations concurrently (or even in parallel on multi-processor hardware) and hence avoid waiting for the slow I/O operations to complete. This allows the agents to guarantee a certain degree of timing correctness that is only limited by the resolution of the given operating system. In addition, the experimental results have shown that a multi-threaded model clearly outperforms a single-threaded one in terms of responsiveness and hence efficiency.

In addition to that, a decision-making mechanism based on reinforcement learning has briefly being described. This same mechanism can also be used to enable co-operation between multiple agents by distributing their responsibilities. Part of the future work for the Essex Wizards team involves designing and implementing a global positioning scheme, using machine learning. This way the team can adapt to changes in the opposition's formations and strategies. This global positioning scheme can be used both in offensive and defensive situations. When in offensive mode, each agent will try and become a good candidate for a pass, and when in defensive mode, each agent will try and block the opponents from advancing.

7 Acknowledgements

We would like to thank the University of Essex for the financial support to the project by providing the Research Promotion Fund DP940.

8 Appendix

The appendix presents some of the key low-level behaviours, namely kicking and passing. Additional behaviours like dribbling, localisation, ball-interception etc. can be provided upon request. **(a) Kicking behaviour**

As mentioned earlier in [4], the server provides a primitive kicking command. An agent can kick the ball with a specified power towards a specified direction as long as the ball is within kicking distance. Unfortunately there are certain occasions where a kick command cannot provide the desired result. Imagine for instance the case where the ball is behind the agent (i.e. ball's direction is 180 degrees) and the agent wants to kick the ball straight ahead (i.e. kicking direction is 0 degrees). Unfortunately this will not be possible because the agent's body blocks the motion of the ball. As can see in figure 5 (a) below that if the ball (dark circle) is in collision with the agent's body (white circle), all kicking vectors within an angle q_0 will result in further collisions and are therefore infeasible. It can be shown that given the diameter of the ball (0.085 meters) and the diameter of the agent (0.8 meters), θ_0 is approximately equal to 42 degrees. (These are the parameters used in RoboCup-98)

To resolve this problem a more advanced kicking behaviour had to be implemented. If the ball is very close to the agent's body, the agent initially kicks the ball further away. In doing so the agent has to ensure that the ball stays within a kicking distance. Figure 5 (b) shows how an additional buffer to the agent's size is used to provide a guideline as to where the ball

should stay. Using this buffer also reduces the number of infeasible vectors as seen by figures 5 (a) and (b) since θ_1 is smaller than θ_0 .

If the desired kicking direction belongs to the set of infeasible vectors, the agent kicks the ball in a path tangential to the agent's body (figure 5 (c)). This rotates the ball round the agent until the desired kicking direction is feasible. When the desired kicking vector becomes feasible, the agent performs the final kick with the specified power. The ball is rotated in the direction that is closer to the desired kicking vector. For example in figure 5 (c), if the agent wanted to kick the ball to the direction it is facing (i.e. downwards) it would rotate the ball clockwise. (Other player's are ignored when calculating the direction of the rotation). Figure 5: Kicking behaviour

(b) Passing behaviour

Before an agent passes the ball to a teammate, it should first determine who is the best candidate for a pass. The passing behaviour is not guaranteed to succeed. There are certain occasions where an unsuccessful code is returned and the behaviour is aborted. This normally happens when no friendly agents are visible, or when no agents satisfy the criteria for a pass described below.

The passing behaviour first invokes two other behaviours, namely "find friends" and "find foes". When "find friends" is invoked, it examines the visual data received by the server, and flags teammates within a certain radius as feasible candidates for a pass. Similarly the "find foes" behaviour flags all opponents within a certain radius as dangerous candidates that could intercept a pass. By filtering out agents that are far away, valuable processing time is saved in further stages of the passing behaviour. Figure 6 below will be used to ease the explanation of the passing behaviour. The grey circle in the middle represents the agent in possession of the ball. The dotted line represents the agent's field of view (in this case set to wide i.e. 180 degrees). All objects above that line are visible to the agent. The small white and black circles represent the friendly and enemy agents respectively. The letters next to the circles will be used to refer to certain agents in the figure. 'B' stands for bad candidate, 'G' stands for good candidate and 'E' stands for enemy.

The maximum distance the agent can kick the ball is defined by the outer circle. Hence all agents outside that region are considered as bad candidates (since the ball cannot reach them). This makes B3 and E3 to be neglected from further calculations. Any friendly agents within the inner circle are also rejected since a pass at such a close distance is meaningless. Therefore after calling the "find friends" and "find foes" behaviours the passing module will consider all agents apart from B3 and E3. Figure 6: Passing behaviour

From the remaining agents, those that are within a certain radius (larger white circles) from enemies are also rejected. In other words B1 is dangerously close to E1 so it is not a good candidate for a pass. Finally if an agent is in the same direction and within a given arc of an enemy agent (i.e. agents E2 and B2) then this pass can be intercepted. In other words although B2 is not close to any enemies, a pass to B2 can be easily intercepted by E2. Hence B2 is also a bad candidate for a pass. The remaining agents (i.e. G1 and G2) are both good candidates for a pass. The ball will go to the one closer to the enemy goal. In other words if the enemy goal is to the right the pass will go to G2 and vice versa. If no good candidates are found at the end of the parsing, the passing behaviour returns an error code.

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 Ch'ng S., Padgham L.: Team description: Building Teams Using Roles, Responsibilities, and Strategies, Proc. IJCAI-97 on RoboCup, 1997.
- 6 Hu H., Gu D., Brady M.: A Modular Computing Architecture for Autonomous Robots, in the Int. Journal of Microprocessors and Microsystems, Vol. 21, No. 6 (March 1998), pages 349-362.
- 7 Hu H., Kelly I., Keating D., Vinagre D.: Coordination of Multiple Mobile Robots via Communication, Proc. SPIE'98, Mobile Robots XIII Conference, Boston, pages 94-103, Nov. 1998.
- 8 Kitano H., RoboCup: The Robot World Cup Initiative, In proceedings of The First International Conference on Autonomous Agent (Agents-97)), Marina del Ray, The ACM Press, 1997.
- 9 Kitano H., Tambe M., Stone P., Veloso M., Coradeschi S., Osawa E., Matsubara H., Noda I., and Asada M.: The RoboCup Synthetic Agent Challenge, 97. Int. Joint Conference on Artificial Intelligence (IJCAI97), 1997.
- 10 Kostiadis K. and Hu H.: Reinforcement Learning and Cooperation in a Simulated Multi-agent System, Submitted to IROS'99, Korea, October 1999.
- 11 Kostiadis K. and Hu H.: A multi-threaded approach to simulated soccer agents for the RoboCup competition, Submitted to RoboCup '99 Workshop, 1999.
- 12 Mataric, J. M.: Interaction and Intelligent Behaviour, PhD Thesis, Massachusetts Institute of Technology, 1994.
- 13 Nisanke N.: Realtime Systems, London, Prentice Hall, 1997.
- 14 Noda I.: Soccer Server: A Simulator for RoboCup, JSAI AI-Symposium 95: Special Session on RoboCup, 1995.
- 15 Stevens W.R.: Unix Network Programming: Networking APIs: Sockets and XTI (Volume 1), London, Prentice-Hall International, 1998.
- 16 Stone Peter and Veloso Manuela: Team-Partitioned, Opaque-Transition Reinforcement Learning, Proc. 15th Int. Conf. on Machine Learning, 1998.

- 17 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.
- 18 Uchibe E., Asada M., Hosoda K., **Strategy Classification in Multi-Agent Environment - Applying Reinforcement Learning to Soccer Agents**. ICMASS'96 workshop 2: RC Workshop, 1996.