A Multi-threaded Approach to Simulated Soccer Agents for the RoboCup Competition

Kostas Kostiadis and Huosheng Hu

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

Abstract: To meet the timing requirements set by the RoboCup soccer server simulator, this paper proposes a multi-threaded approach to simulated soccer agents for the RoboCup competition. At its higher level each agent works at three distinct phases: sensing, thinking and acting. Instead of the traditional single threaded approaches, POSIX threads have been used here to break down these phases and implement them concurrently. The details of how this parallel implementation can significantly improve the agent's responsiveness and its overall performance are described. Implementation results show that a multi-threaded approach clearly outperforms a single-threaded one in terms of efficiency, responsiveness and scalability. The proposed approach will be very efficient in multi-processor systems.

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 [6][7]. 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 [10]. In addition to that, 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 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 can only be guaranteed up to a certain resolution [12].

A real-time system is a system in which the time that output is produced is significant. In other words, a real-time system should be able to respond to stimuli from its environment within fixed time limits [4]. This is particularly true for the RoboCup simulator agents since each of them must react within an interval of 100ms, therefore achieving real-time performance. For a non real-time agent the desired behaviour is focused on the logical correctness of the result. In contrast, a real-time agent requires both logical and timing correctness. There are two broad categories of real-time agents: hard real-time agents and soft real-time agents [1]. For hard real-time agents, timing correctness is of critical importance and should never be sacrificed for other gains. In contrast, timing correctness in soft real-time agents is important but not critical. An occasional failure to meet a deadline should not result in catastrophic consequences.

Another distinction between real-time agents is whether their responses are based on events or clock times. This classification is of particular importance at the implementation level. Event-triggered agents respond to external stimuli by first detecting various conditions and then generating the appropriate reactions dynamically. Time-triggered agents operate in accordance with clock times as shown by an independent clock. In other words, clock pulses are treated as signals that generate certain actions. An appropriate action is selected based on the current state of the environment perceived by the agent.

This paper considers robotic soccer agents as hard, time-triggered real-time agents, and will only focus on how to improve the real-time performance of the soccer agents. Real-time systems are a large topic and a more detailed description of real-time systems can be found in [1][9]. In the rest of this paper, a few design issues regarding the agent architectures are discussed in the next section. Section 3 illustrates how single-threaded implementations have been developed, and then presents the proposed multi-threaded implementation. Section 4 shows experiment results and compares the two approaches. Finally, conclusions are given in section 5.

2. System Design

2.1 Agent Requirements

The robotic soccer simulator is a client/server application in which each client communicates with the server via a UDP (User Datagram Protocol) socket [10]. The server is responsible for executing requests from each client and updating the environment. At regular 150ms time intervals 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. Moreover, 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 (100ms). If no message is sent within this cycle, the client will not perform any actions. If more than one message is send during the same cycle, the server executes only one randomly, which may produce undesired results. The server updates the state of the environment by serially executing each request. The results are projected on a window shown in figure 1.

It is important to mention that 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

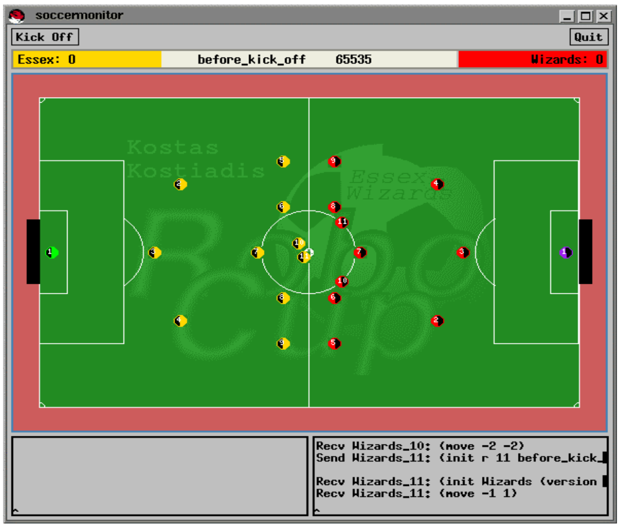

Figure 1 Robot Soccer Simulator

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, leading to undesired effects when the wrong actions are 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 send an action to the server. To be efficient an agent should satisfy the following conditions:

- 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 send the action requests to the server timely, i.e. only one request/cycle. If the agent sends more than one request per cycle, the server will only execute one randomly. Otherwise, it might miss a cycle if it is too slow.
- To allow the maximum time for the thinking process and the minimum time to send or receive data.

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. A brief description of the available I/O models under Unix and the agent architecture is briefly reviewed here.

2.2 I/O Models

Since the client/server communication is done via UDP sockets, there are a number of different ways to handle input and output on these sockets. Unix provides five I/O

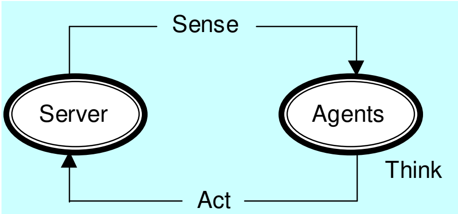

Figure 2 Agent's main loop

models: blocking I/O, non-blocking I/O, I/O multiplexing, signal-driven I/O, and asynchronous I/O. For the shake of completeness each I/O model is briefly introduced here. A more detailed description can be found in [12].

- *Blocking* I/O mode is the default mode for all sockets under Unix, which means that requesting data from a socket will not return until data is available. In other words, the whole process will be put to sleep until new data arrives or an error occurs.
- *Non-blocking* I/O mode avoids putting the process to sleep. If data is not immediately available, the kernel returns an error message, such as EWOULDBLOCK. The process will not be blocked and data can be requested at a later stage.
- *Multiplexing* I/O mode is achieved by using either the *select* or *poll* system calls. This method will block in one of these two calls rather than blocking at the actual I/O system call. When *select* returns, the socket is readable and the *rcvfrom* function is called to copy the data into the application buffer.
- *Signal-driven* I/O mode enables the kernel to notify the process with a (SIGIO) signal when a descriptor is ready. To read the data from the socket the process needs to establish a signal handler for the (SIGIO) signal. To send actions to the server, an interval timer that generates a (SIGALRM) signal when it expires can be used. By setting the timer to 100ms and using a handler that sends actions to the server the accuracy is guaranteed within certain limits.
- *Asynchronous* I/O mode enables the kernel to notify the process when the entire I/O operation is complete. The difference with the signal-driven I/O is that the kernel tells the process when an I/O operation is complete rather than an operation can be initiated. Asynchronous I/O is not very widely used primarily because of support issues.

2.2 Agent Architecture

Given this variety of I/O models supported under Unix, choosing an I/O model heavily depends upon the inner structure of the agent. In this section a novel architecture for building RoboCup agents is presented. Basically the agent contains six different modules as shown in figure 3, namely 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 between the agent and the server in this diagram represent the communication links. The server sends information received by the agent's "Sensors" module and the agent sends actions back to the server through its

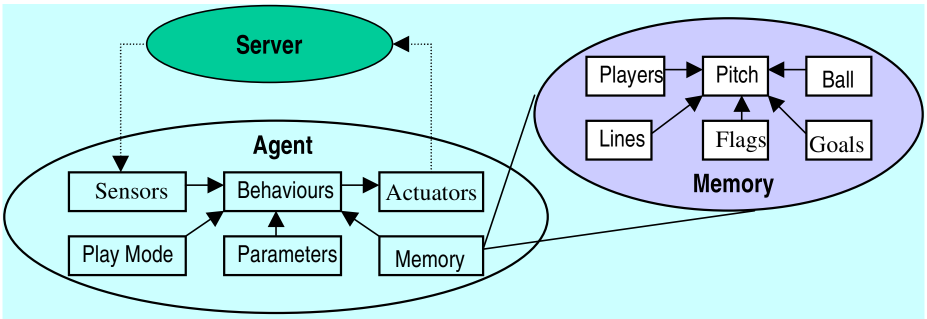

Figure 3 Block Diagram of the Soccer Agent's Structure

"Actuators" module. The solid arrows give dependence relationships between the modules. More specifically, 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 is given below.

- *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 send them 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 that 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. Otherwise, this value is multiplied by a confidence decay constant for every cycle that the object has not been seen. When the confidence value falls under a certain threshold the object is "forgotten".

3. System Implementation

This section is to describe the single-threaded models used so far and the multi-threaded approach being proposed for the RoboCup simulator agents. This paper focuses on POSIX threads that are the thread application-programming interface (API) specified by the standard POSIX 1003.1c-1995 [1][2]. Although POSIX threads are used in this implementation for portability reasons, the higher level design can be implemented in any system with any threads API.

3.1 Single-threaded approach

According to [2] a thread is the set of properties that suggest "continuousness and sequence" within a machine. In other words when a program is executed, a process is created. This process can be thought of as a single "thread" of execution. Of course a process has many additional properties like its own address space, file descriptors and various other data. Up to date, the majority of the implementations in the RoboCup simulation league have been single-threaded (e.g. CMUnited [13], ATHumboldt [3], Andhill). This essentially means that the initial process generated by the executable file does not create any additional threads. All computations are performed in a serial manner. If a given operation requires that the process is put to sleep (e.g. a blocking read when no data is available), then the whole execution is paused.

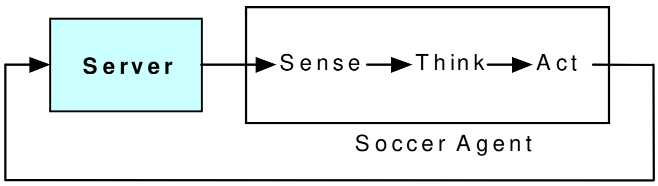

Figure 4 Single-threaded model

At the higher level, the agent is responsible for performing three individual tasks. First it needs to receive sensory information from the server via a UDP socket. Then the agent has to "think" in order to produce a desired action. Finally this action should be sent back to the server via the UDP socket. To perform all these tasks using a single thread the agent has to use a serial-processing loop shown in figure 4.

Given the nature of the simulation and the fact that two out of three operations involve I/O on a UDP socket, it is clear that a single-threaded serial implementation puts a significant limitation to the agent's capabilities. I/O operations can be extremely slow. Although an agent receives visual information every 150ms, it can send actions every 100ms. If an agent is put to sleep until new data arrives, it will miss a number of cycles, which significantly decreases the agent's overall performance. Therefore, both I/O multiplexing and signal-driven I/O are widely used in single threaded implementations.

Using I/O Multiplexing

I/O multiplexing is the capability to tell the kernel to notify the process if one or more I/O conditions are ready (i.e. input is ready to be read, or the descriptor can take more output). For the RoboCup simulator, the agent needs to check if data is available from the server, or if it is ready to send data to the server. To implement multiplexing, after the connection has been established with the server, the agent enters an infinite loop that is normally called the main-loop (or message-loop). In this main loop the *select* call is used to determine whether data are ready to be received or to be sent. If new data has arrived from the server, the agent analyses and stores the information. Alternatively if data are ready to be sent, the agent sends the required action to the server.

```
while (agent is alive)
{
    Call select to check which descriptor is ready.

    if (read descriptor is ready)
        receive and analyse data from the server.

    if (write descriptor is ready)
        send an action to the server.
}
```

Figure 5 Pseudo-code for I/O multiplexing

This approach has several disadvantages. Firstly it is difficult to accurately time each phase of the agent's cycle. The agent can only have an estimate of the time required to analyse the sensory data, perform all the thinking, and produce an action. Hence, timing correctness cannot be guaranteed for the 100ms-interval set by the server. Secondly only one I/O operation will be executed at a time. In other words, although *select* can wait on multiple descriptors, the main-loop can only perform one operation when *select* returns. Hence every time the agent enters the main-loop it will either analyse the sensory data that were received, or send an action to the server. Since only one of the two tasks can be executed, and since this task will not be interrupted before completion, the agent cannot guarantee timing correctness. Finally, the *select* function requires two system calls to receive the data. Initially *select* has to be executed to check whether data is available, and then if data exists on the socket the agent needs to call *rcvfrom* in order to receive that data. Figure 5 presents an example of a pseudo-code implementation using I/O multiplexing.

Using Signal Driven I/O

Signal driven I/O mode is used to instruct the kernel to notify the agent by generating a signal when something happens on a descriptor. To implement signal driven I/O two signals are normally used. One is generated when new data arrives from the server through the UDP socket (SIGIO), the other is generated when an interval timer expires which indicates that an action must be sent.

```
Signal handler for SIGIO
{
    // SIGALRM has just been blocked.

    Receive and analyse data from the server.

    // On exit, SIGALRM will be unblocked.
}

Signal handler for SIGALRM
{
    // SIGIO has just been blocked.

    (Since 100ms have expired…)
    Send an action to the server.

    // On exit, SIGIO will be unblocked.
}
```

```
Agent's Main-Loop
{
    Install SIGIO & SIGALRM handlers
    Set an interval timer to 100ms
    Start thinking process…

    /* The kernel will generate signals for
    the process and hence execute a signal
    handler if either of these two events occurs
    if (100ms expire)
        execute SIGALRM handler.
    If (new data arrives on the socket)
        execute SIGIO handler.
    After the execution of the handlers has
    finished, the program will return
    precisely where it was before the
    signal was generated. */
}
```

Figure 6 Pseudo code for signal driven I/O

Signal driven I/O can only handle one signal at a time. Although it is possible to establish signal handlers for multiple signals (e.g. SIGIO, SIGALRM, etc.) only one signal handler can be executed at any given point. Therefore, when a signal is generated while another signal handler is being executed, certain precautions have to be taken to avoid signal loss. To resolve this problem it is necessary to mask additional signals before entering the signal handler and then unmask them when the handler has finished.

However, great care needs to be taken for masking (SIGIO) signals. When a signal handler is being executed, if two more datagrams arrive on the UDP socket, the (SIGIO) signal will be generated two more times. But since the signal is blocked, when the signal handler returns, the system will only observe one (SIGIO) signal. This will force the agent to read the second datagram but the third one will remain on the queue until another (SIGIO) is generated. This clearly causes a problem since this architecture is unable to guarantee that the number of (SIGIO) generations matches the datagrams on the UDP queue. A solution would be to keep polling the queue until it is empty but this clearly consumes valuable time that could be used to perform other computations. An example of a pseudo-code implementation for signal driven I/O is given in figure 6.

3.2 A multi-threaded approach

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 proposed multi-threaded model can be seen in figure 7.

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 actions (these relationships are

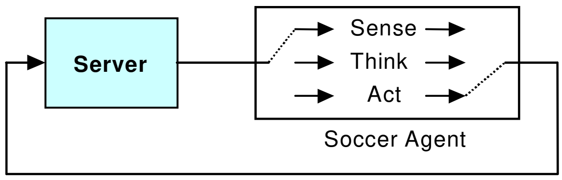

Figure 7 Multi-threaded model

indicated by the dashed arrows). In 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 awaken, execute a *rcvfrom* call to receive the next available datagram from the server, and analyse the new data. 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 impossible.

Secondly, the "Act" thread needs to send any available actions to the server at 100ms intervals. 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.

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. Part of the "Think" thread of the current implementation can be found in [8]. If required, various scheduling policies and different priorities among the threads could be implemented. However, given the nature of the problem, no scheduling is needed. In addition to that, synchronization between separate threads can also be implemented if required. 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 paper. A detailed description on multi-threaded programming can be found in [2]. A pseudo-code implementation of the proposed multi-threaded approach is shown in Figure 8. An alternative multi-threaded approach based on an organic programming language can be found in [11].

Table 1 Performance testing

Cycle loss	Single-thread I/O multiplexing	Single-thread signal driven I/O	Multi-thread approach
1 agent	1651 (27.52%)	24 (0.4%)	0 (0%)
6 agents	1653 (27.55%)	1020 (17%)	0 (0%)
11 agents	1656 (27.60%)	3294 (54.9%)	4 (0.07%)

4. Results and Analysis

4.1 Experimental Results

To compare the two models a simple experiment was carried out using a PC with an Intel Pentium II 450MHz processor, running Redhat Linux 5.2. The purpose of this experiment was to evaluate only the timing correctness of the models described above, since logical correctness is based on the individual AI implementations. The test required an agent to connect to the server and send a turn command in each cycle. In addition to that the agent is required to receive and analyse all sensory information to simulate a real game situation. The sensory analysis functions are identical for all implementations. Initially, a single agent was tested on the field. The same experiment was then carried out using 6 and 11 agents to examine what would happen as computational resources decrease. The results for a 6000 cycle test (a game duration) are illustrated in table 1.

The table entries indicate the number of cycles for which an agent failed to send an action and hence did not meet the timing requirements. Adding more agents has a similar effect as extending the thinking process. In both cases system resources are depleted by adding more computations. This is to test the scalability of each implementation in terms of number of operations per cycle. As can be seen, I/O multiplexing using *select* has over 27% cycle loss even when only 1 agent is used. However multiplexing is highly scalable since this percentage does not increase much more for 6 or even 11 agents. On the other hand signal driven I/O performs significantly better in the single agent trial with only 0.4% cycle loss. However this approach is not scalable since there is 17% loss for six agents and nearly 55% cycle loss for 11 agents.

```
Sensor thread
{
  while (agent is alive)
  {
  receive & analyse data from server
  /* if data is not available the
    will be put to sleep.  When
    arrives on the socket, the
    will wake up, receive and analyse
    that data. */
  }
}
```

```
Actuators thread
{
   while (agent is alive)
   {
      wait for 100ms

      /* Again this thread is
      to sleep until 100ms
      passed. */
      execute an action Agent's
   }
}
```

```
Main-Loop
{
   Create Sensor thread
   Create Actuators thread
   Start thinking process…

    /* Although two threads are
    the main execution is a thread
    Hence there is no need to
    another thread for the
    process, the current one can   used*/
}
```

Figure 8 Pseudo code for the proposed multi-threaded approach

From the results generated by the multi-threaded approach, it can be seen that there is nearly 0% cycle loss for all trials. This proves the superiority of multi-threaded approaches in terms of performance and scalability. It should be noticed that multi-threaded implementations are very efficient in multi-processor hardware. The use of parallel processing is significantly important in obtaining real-time performances even in highly complicated systems such as aerospace control systems [15] and advanced AGVs in industry [5].

4.2 Discussion and Analysis

Although comparing the multi-threaded approach against the single-threaded ones is not an easy task, certain points about the two approaches are briefly presented here. The main advantages of the proposed multi-threaded approach include:

- The multi-threaded model allows more efficient exploitation of the agent's natural concurrency. Various computations can be performed while waiting for slow I/O operations to complete. This provides the agent with a significant advantage considering the number of I/O operations per game.
- Using multiple-threads provides a modular programming model that clearly expresses relationships between independent events within the agent. However, designing a multi-threaded approach clearly illustrates various program dependencies and synchronization requirements between different modules within the agents. Mixing CPU-intensive processes with I/O intensive processes results in better utilization of resources of an uni-processor system. Note that multi-threaded implementations allow an agent to achieve even better performance in a multi-processor environment.
- A multi-threaded approach is highly scalable. In the current implementation, consecutive actions can be sent 100ms apart. A multi-threaded implementation could have one thread performing an expensive search algorithm, and a separate thread executing simple reactive behaviours. By combining the results from both threads, the agent can significantly improve its performance.

However, multi-threaded programming has also a few limitations. First, a multi-threaded model is normally more complicated than a single threaded one. Careful design is required to keep track of synchronization protocols and program invariants. Second, it is necessary to avoid deadlocks, races, and priority inversions. Third, the POSIX standard does not provide an interface for object oriented languages. However a solution does exist and can be found in [14].

5. Conclusions

It has been shown in this paper 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 results becomes a function of both logically correct output and response time. 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 efficiency.

Acknowledgements

We would like to thank Kaz Kylheku for his useful comments on the Unix/Linux systems and the University of Essex for the Research Promotion Fund DDP940.

References

1. A. Burns and A. Wellings, Real-time Systems and Programming Languages, Addison-Wesley, 1997
2. R.D. Butenhof, Programming with POSIX Threads, Harlow, Addison-Wesley, 1997
3. B. Hans-Dieter, et al., AT Humboldt -- Development, Practice and Theory, In Proceedings of RoboCup-97-- Robot Soccer World Cup I, H. Kitano (Editor), 1997, 357-372
4. H. Hu, M. Brady, F. Du, P. Probert, Distributed Real-time Control of a Mobile Robot, Int. Journal of Intelligent Automation & Soft Computing, Vol. 1, No. 1, 1995, 63--83
5. H. Hu and M. Brady, A Parallel Processing Architecture for Sensor-based Control of Intelligent Mobile Robots, Int. J. of Robotics and Autonomous Systems, Vol. 17, No. 4, 1996, 235-258
6. H. Kitano, RoboCup: The Robot World Cup Initiative. Proceedings of the 1st Int. Conference on Autonomous Agent (Agents-97)), Marina del Ray, The ACM Press, 1997
7. H. Kitano, M. Tambe, P. Stone, M. Veloso, S. Coradeschi, E. Osawa, H. Matsubara, I. Noda, and M. Asada, The RoboCup Synthetic Agent Challenge'97. Proceedings of International Joint Conference on Artificial Intelligence, 1997
8. K. Kostiadis and H. Hu, Reinforcement Learning and Co-operation in a Simulated Multi-agent System, Proc. of IROS'99, Korea, October 1999
9. N. Nissanke, Real-time Systems, London, Prentice Hall, 1997
10. I. Noda, Soccer Server: A Simulator for RoboCup. JSAI AI-Symposium 95: Special Session on RoboCup, 1995
11. I. Noda, Kappa: Agent Program by Gaea. Proceedings of the 2nd RoboCup Workshop, July 1998, 387-392
12. W.R. Stevens, Unix Network Programming: Networking APIs: Sockets and XTI (Volume 1), London, Prentice-Hall (1998)
13. P. Stone and M. Veloso and P. Riley, The CMUnited-98 Champion Simulator Team, In "RoboCup-98: Robot Soccer World Cup II", M. Asada and H. Kitano (eds.), Springer Verlag, Berlin, 1999.
14. B. Stroustrup and S. Lippman, Pointers to Class Members in C++. Proc. USENIX C++ Conf., Denver, 1988, 305-326
15. G. S. Virk, J. M. Tahir, P. K. Kourmoulis, Parallel Processing in Aerospace Control Systems, Proc. the 2nd Int. Conference on Applications of Transputers, UK, July 1990