FIRA Robot World Congress 2003
October 1 – 3, 2003, Vienna, Austria

FAST DATA SHARING WITHIN A DISTRIBUTED, MULTITHREADED CONTROL FRAMEWORK FOR ROBOT TEAMS

**Remco Seesink, Werner Dierssen, Niek Kooij,
Albert Schoute, Mannes Poel,
Erik Schepers, Thijs Verschoor**

*University of Twente
Department of Computer Science
Postbox 217
7500 AE Enschede – Netherlands
email: a.l.schoute@utwente.nl, robotsoccer@cs.utwente.nl*

Abstract

In this paper a data sharing framework for multi-threaded, distributed control programs is described that is realized in C++ by means of only a few, powerful classes and templates. Fast data exchange of entire data structures is supported using sockets as communication medium. Access methods are provided that preserve data consistency and synchronize the data exchange. The framework has been successfully used to build a distributed robot soccer control system running on as many computers as needed.

Keywords: robot soccer, control software, distributed design, data sharing, multithreading, sockets

1. Introduction

This paper describes the control software framework of our robot soccer team MI20. This system is designed in a distributed way, where separate single- or multi-threaded programs control each part of the system. The big advantage of this design is that we can run our system on as many computers as we think is necessary. So if one of the threads is very demanding, the robots tracking thread for instance, we can run the program exclusively on one computer. Distributed software design has many more advantages, but also one big disadvantage: it complicates data sharing. Because many threads have to share common data, they will communicate quite intensively. Therefore we need to find a very fast way of exchanging data. We chose to use sockets as a communication medium, because they can provide fast communication. The second important issue in our design is that we exchange entire data structures. Because the layout of the data structure is known on both sides of the communication channel, we can address members of the structured data without using functions, which provides good speed performance. Functionality is added to automatically maintain data consistency between application threads that access the data structures and communication threads that exchange the data. The application programmer can use safe access methods without having to bother about thread interference. This way we have achieved a fast and reliable system, that we can expand or change, without the need of redesigning the system.

2. Framework components

The distributed data sharing framework is realized in C++ by means of only a few, powerful classes and templates:

- a super-class *Cthread* that enables threads to start, stop, pause and resume
- a class *Cmutex* to exclusively lock data and wait on or signal data presence or renewal
- a template class *Csafe* usable for any type of shared, structured variable to enforce safe access
- a super-class *Csocket* to instantiate threads that operate on sockets
- template classes *Ccommunication_sender* and *Ccommunication_receiver* to instantiate communication threads that send or receive the content of a “safe variable” over a socket
- a super-class *Cexception* to keep error management simple while acting appropriately on different sorts of exceptions

Thread instances of *Cthread* are actually based on Posix compliant threads, known as *pthreads* [1]. Linux supports multithreading by running *pthreads* as kernel processes [2]. The pthread-package supplies synchronization functions for exclusive access to class objects according to the monitor concept [3].
The power of the framework doesn’t result from each of the classes alone. It results from their combined

use by fully exploiting all the nice features of C++ like function inheritance, type-independent template-programming and function overloading.
For example the template declaration *Csafe*, being a derived class of *Cmutex*, creates exclusive access to arbitrarily typed variables. Basically it adds a "value" of any type to an instance of class *Cmutex*. This "mutex" instance functions as exclusion and synchronization monitor for the added "value". The template declaration of *Csafe* is contained in nothing more than a two-page header file. This provides the basic locking mechanism to preserve data consistency of shared variables accessed by multiple threads. Moreover, the wait and signal functions of class *Cmutex* (again based on the pthread-package) automatically take care of condition synchronization between asynchronously reading and writing threads. In the *Cmutex* class a single private condition variable is defined on which threads will block when calling the wait function. The solution resembles object synchronization as made implicitly available in Java [4].
By defining any variable as "Csafe" and obeying the usage protocol as shown in the next section, the programmer can rely on the mechanism to guarantee safe and synchronized access.
The safe access mechanism is applied to the framework itself to extend its power even more. Thread instances of class *Cthread* are represented by underlying pthreads that can be created, paused or stopped. Their state can be dynamically changed by other threads and hence the state variable is implemented as a *Csafe* object. Only if a thread of class *Cthread* is executing its "run-function" the underlying pthread is needed and actually present as a Linux process. The introduction of a function *run()* as "actual body" of a thread is borrowed from Java.

3. Framework usage

When reading or writing a *Csafe*-variable X exclusive access needs to be established by explicitly calling locking and unlocking functions as follows:

```
X.lock();
/* now X.value can be read or written safely */
X.unlock();
```

It has been considered to perform locking implicitly and hide it from the programmer. However, this is rather a burden than an advantage if accesses are more complex. A mixture of both explicit and implicit locking would be even more confusing. So explicit locking is required as being the most transparent, flexible and efficient solution. If it is important to keep the locking period short the programmer can make a local copy.
In the context of a dynamic application like robot soccer fast asynchronous updating of state information is an important issue. The synchronization properties inherited from the *Cmutex* class make the signaling of and waiting on data renewal very straightforward. The program sequences below show how a reading thread wait for renewed data to become available and a writing thread signals the renewal of it:

Reader	*Writer*
X.lock(); X.wait(); /* reading of X.value */ X.unlock();	X.lock(); /* writing of X.value */ X.signal(); X.unlock();

On this schema many variations are possible. If multiple threads possibly wait on reading the same variable X the writer should issue *X.broadcast()* instead of *X.signal()*. In the former case any waiting thread is signaled, in the latter case only a single one is signaled. In fact, the most robust way of programming is to use always *X.broadcast()*.

A thread may also read or write a new value only if the variable X is not locked by using the function *X.try_lock()* in stead of *X.lock()*. This could be desirable in order to avoid locking delays when data has to be captured and distributed in real-time. Figure 1 reflects the case where a camera thread distributes images to multiple "subscriber threads" by writing a new image to each of their "safe" image variables. By using "try_lock" the variable is refreshed only if the reading thread is not yet busy with processing an earlier image.

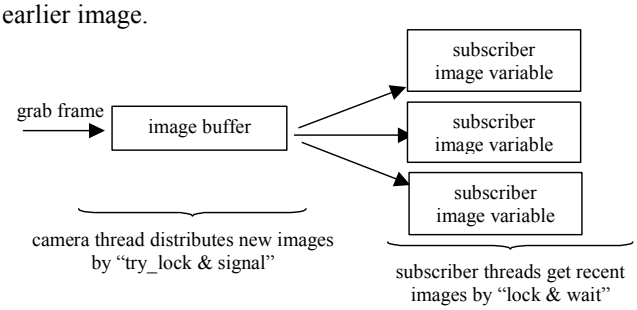

Figure 1. Camera images are copied to multiple *Csafe* variables as an example of safe data distribution

The simple data exchange concept provided by *Csafe* variables has been extended to a distributed environment by means of the communication classes *Ccommunication_sender* and *Ccommunication_receiver* of the framework. As these classes are derived from the classes *Csocket* and *Cthread* respectively, instances of the communication classes become sender and receiver threads capable of communicating through sockets. If a thread has modified a *Csafe* variable on one computer, it has only to signal this variable to activate an associated chain of sender and receiver threads to transport the modified content to another computer. Finally, the receiver thread will update a similar variable in a program that runs on the other

computer. Any thread waiting on this variable is notified. Dedicated sender and receiver threads have to be defined to couple a pair of distributed *Csafe* variables. An example related to the robot soccer application is given in the next section.
Due to the general nature of sockets the framework allows for interoperability between Linux, Solaris or Windows. There is however a prerequisite to be made with respect to compatibility of the compilers used. Apart from byte-order conversion (big / little endian) that is automatically detected and corrected, the variables must be mapped on memory identically on all machines.

4. MI20 software architecture

The framework facilities have been used extensively in the MI20 control software. Due to the distributed design there is no essential difference in controlling a single robot team or controlling both teams of a robot game. In the latter case the global vision system tracking the robots consists of a single program for image processing and two separate programs for the state estimation as viewed by each team. The image-processing program contains multiple threads interpreting the images: for each team a vision thread together with threads that display images on the user interface. A camera thread distributes images to all of these image processing threads in a way as described in the previous section.
Also the "soccer playing intelligence" of the system is distributed over multiple agent threads. Each team consists of player agents, one for each robot, and a single coach agent. When controlling two teams the system has the multi-agent architecture as shown in figure 2. Each of the robots is steered by its player agent. This agent actually sends control commands to a thread that drives the radio-frequency link.

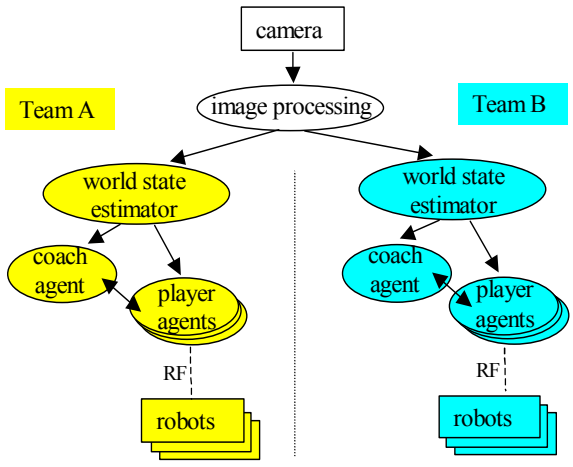

Figure 2 Controlling a complete robot match with two teams using a single camera system

Let us take the player agent as an example to see how data is exchanged in the system. State information is maintained in several globally known data structures like "world data", "player data", "coach data", "wheel data", etc. In the main program of the player agent *Csafe*-variables are defined for all of the data structures needed, e.g.:

```
Csafe<Tworld_data> world_data;
Csafe<Tplayer_data> player_data[PLAYERS];
```

The player agent will typically read the world data produced by the state estimator and write player data and wheel data. The interconnection structure for a player agent is established by defining its communication servers, for example to receive world data and send player data to the coach agent by the following lines:

```
Ccommunication_receiver_thread<Tworld_data>
        Iworld(&world_data, P2W_PORT[robot_id]);

Ccommunication_sender_thread<Tplayer_data>
        Iplay(&player_data, P2C_PORT[robot_id]);
```

Then the communication threads only have to be started by calling *Iworld.start(); Iplay.start();* etc. Thereafter the distributed data exchange will proceed automatically through the locking and synchronization protocol as described in the previous section.
The distributed approach forces the separate control parts to communicate through well-defined interfaces. This has the additional advantage of modular design making independent development and testing easier. For example, the coach and playing agents can be tested by using a simulator without changing any of the interfaces. The simulator used even runs on a Windows machine, whereas all the MI20 control software runs on Linux.

5. Implementation features

Coupled exclusion
In certain cases it is desirable to access multiple *Csafe*-variables within a single exclusion regime. For example to read the speed values of both robot wheels consistently. This has been made possible by the option to supply a common *Cmutex* variable as argument of the constructor function of the *Cmutex* class. Without this argument *Csafe*-variables use their private mutex, with this argument given an indirect link is made to the *Cmutex*-variable supplied.

Pausing and resuming threads
For efficiency reasons only thread instances that are actually running have underlying pthreads in operation. Non-running thread instances only exist as class instance, but do not consume further system resources. The idea is that threads are started or resumed through the user interface only when necessary and paused or stopped when not needed

anymore. By this way for instance, the actual number of running player threads can be configured dynamically to match the real world. A drawback is the requirement that threads have to poll regularly their status to see if they should pause or stop.

Automatic connection recovery
Socket connections may become broken for several reasons. Any sender thread will try to re-establish the connection. It makes use of the type specific exception classes derived from the *Cexception* superclass to catch different exception causes and to take the appropriate action.

6. Conclusion

In this paper we focused on the additional software "infrastructure" that supports the distributed design of the robot soccer system MI20. The MI20 system consists of three major parts that have been designed by master thesis students, e.g. the global vision system [5], the intelligent decision engine [6] and the motion planning subsystem [7]. These parts could not have been developed and glued together so easily without the distributed data sharing framework. This framework has been designed and implemented at the beginning of the project to serve as common starting environment. It has been extended gradually during the subsequent integration stages. The source code of the MI20 system is online available [8]
The main objective of the framework was to make distributed system composition easy without suffering from the overhead, which has been realized successfully. The result proofs that in a dedicated application like robot soccer both distributed processing and fast and easy data sharing can go together. Fast data communication is reached by the exchange of complete, commonly known data structures using sockets. Easy data access is the result of full exploitation of today's software facilities as offered by the C++ (template) class concept, multithreading packages and socket communication. The flexibility of the distributed control framework has resulted in many blessings not planned in advance. As mentioned, the system was easily expanded with a duplicate playing team (and duplicate user interface), allowing us to control a complete robot soccer match. Whereas the system was initially set up to play with teams of 5 robots, the system is equally capable of handling larger teams, which made it possible to participate in the large Mirosot league with 7 against 7 robots.

7. References

[1] D.R. Butenhof (1997). "Programming with POSIX threads", Addison-Wesley.
[2] M. Beck et al. (1997). "Linux Kernel Internals", 2nd Ed., Addison Wesley.
[3} A. Silberschatz et al (2000). "Applied Operating System Concepts", John Wiley & Sons.
[4] S.Oaks and H.Wong (1999). "Java Threads", 2nd Edition, O'Reilly & Associates.
[5] N.S. Kooij (2003). "The development of a vision system for robotic soccer", Masters Thesis, University of Twente.
[6] R.A. Seesink, (2003). "Artificial Intelligence in multi agent robot soccer domain", Masters Thesis, University of Twente.
[7] W.D.J. Dierssen (2003). "Motion planning in a robot soccer system", Masters Thesis, University of Twente.
[8] "MI20 robot soccer website", http://parlevink.cs.utwente.nl/robotsoccer