Fast Data Sharing within a Distributed, Multithreaded Control Framework for Robot Teams

Albert SCHOUTE [a,1], Remco SEESINK [b], Werner DIERSSEN [c] and Niek KOOIJ [d]

[a] *University of Twente,* [b] *Atos Origin NL,* [c] *Triopsys NL,* [d] *Qmagic 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, multi-threading, sockets

Introduction

This paper describes the control software framework of the robot soccer team *Mobile Intelligence Twenty* (MI20). Many different types of robot soccer competitions are organized by international associations [1], [2] with varying game and hardware rules. Our team competes in the FIRA MiroSot league, in which small-sized, wheeled robots are controlled based on localization by a central camera system. The application is representative for control systems that heavily rely on globally shared sensor information.

In contrast to the centralized way of robot localization the team control system is designed in a distributed way, where separate single- or multi-threaded programs control distinct parts 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 some tasks are very computationally demanding, for instance robots tracking, path planning or playing strategy, we can run the programs on separate computers.

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.

1. Application

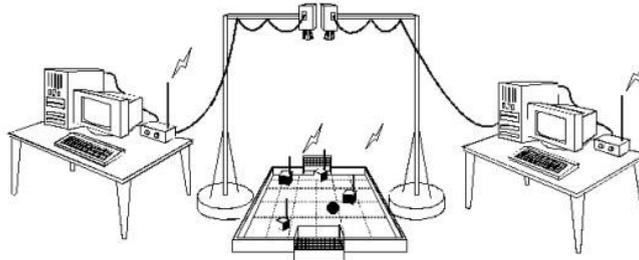### 1.1 The Robot Soccer Game Environment

Figure 1. MiroSot League competition set-up

In the MiroSot league the robot size is limited to cubes with maximum measure 7.5 cm. Competition categories differ with respect to robot team sizes (3, 5, 7 or 11 players) and the matching dimensions of the playground. Our robots have an onboard DSP-processor that takes care of wheel velocity control and wireless communication. A digital camera above the playground captures images that are processed by the team computer(s) that steer(s) the team of robots. Robots are recognized by means of colour patches on their top surface. The game is played with an orange golf ball. Wheel velocity set-points are sent to the robots by a radio link, each team using a different frequency.
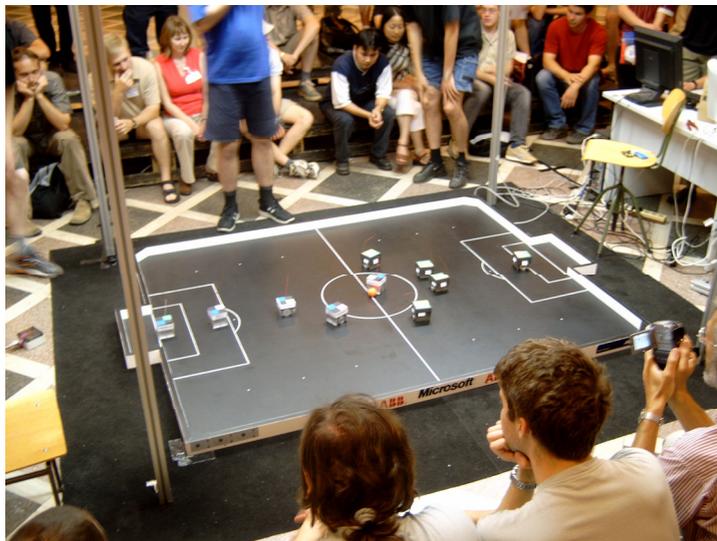

Figure 2. Impression of the real game situation

1.2 Requirements

The design of the data sharing framework has been influenced to a large extend by the requirements of the robot soccer application. Let us consider the main aspects.

First of all, in a game situation it is important to continue under all circumstances even if certain robots are not functioning properly, if disturbing events happen on the play ground or processes deteriorate. Only a human arbiter may interrupt - according to the

playing rules - the fully computer-controlled game. A distributed, concurrent design with independently operating components will contribute to the robustness of the system.

Furthermore, the application must be highly reactive and requires fast responsiveness to the actual situation. Image data need to be processed at the camera frame rate (30 frames per second). Due to many circumstantial influences, for example lighting conditions, data may be unreliable and must be filtered. State information should reflect the real-time situation as close as possible. The rate at which robots receive control data depends on the team size and typically lies in between 10 to 20 set-points per second.

For the application it is important that the most recent sensor data and updated state information is made available throughout the system as fast as possible. State information in this kind of application has a permanent structure and is maintained in globally known data types. Sharing of state information in a flexible way implies that arbitrary many concurrently running threads can access common data structures asynchronously. If the system is distributed over multiple programs, possibly running on different computers, we still want to be able to share common data structures. The data content has to be proliferated to "mirror" the same data at different places.

Of course updating and exchanging shared data must be organized in a sensible way. So, the application programmer is responsible of defining data structures as being common and establishing communication processes that create a "refreshment chain" by which updates are proliferated. We require that the content of shared data structures is "near-time equivalent", which means that reading threads obtain a recently written data content. A reading process may also require getting the next refreshed data instance.

1.3 Solution Approach

The framework presented provides the tools to manage the shared data access and proliferation in an easy, efficient and safe way. Several practical implementation decisions are made to make the data sharing as fast as possible in the context of C++ based programming and Linux based multithreading. The main approach could be stated as a combination of:

1. a shared memory access model within a single multithreaded program
2. a socket communication model to exchange common data structures (in binary form) between program variables of compatible type

Ease of programming is reached by making the access to shared data structures transparent to the fact whether a common data structure is accessed by threads within only one or within multiple programs. In the latter case the same data structure is defined in each program and the content is mirrored in some sense. But the access method of remotely operating threads remains the same.

We do not intend to introduce a new concurrency concept. Nor do we claim that our implementation presents a unique, novel solution. Similar distributed data sharing facilities can be provided by using other programming concepts and tools. In this respect, a comparison with other approaches has to be made yet.

Our purpose is to offer a fast and practical solution in the given object-oriented context for distributed software development while preserving efficient ways of data sharing. In object-oriented programming environments like Java and .Net so-called "object streams" are supported. Complete objects are "serialized" into a string representation to be transported over a network. The associated overhead, however, does not comply with the needs of real-time control applications.

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* [3]. Linux supports multithreading by running pthreads as kernel processes[4]. The pthread-package supplies synchronization functions for exclusive access to class objects according to the monitor concept [5].

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 [6].

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, although it is not enforced automatically. 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 in Table 1 show how a reading thread waits for renewed data to become available and a writing thread signals the renewal of it.

Table 1. Reader / Writer Synchronisation

Reader	*Writer*
`X.lock();`	`X.lock();`
`X.wait();`	`/* writing of X.value */`
`/* reading of X.value */`	`X.signal();`
`X.unlock();`	`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()` instead 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 3 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.
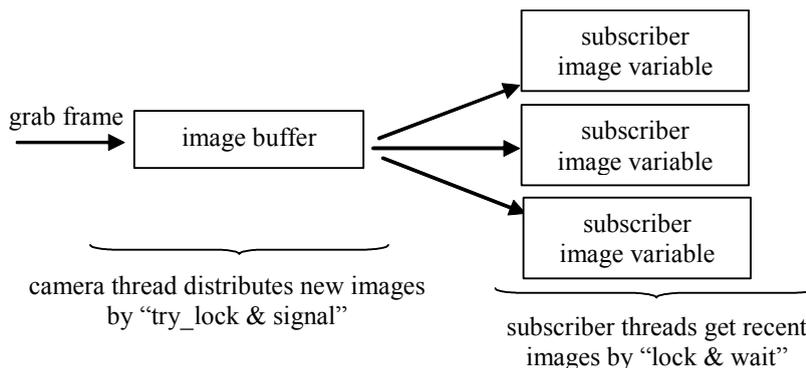

Figure 3. Camera images are copied to multiple *Csafe* variables as an example of safe data distribution

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 in a program 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.

Note that distributed `Csafe` variables are automatically updated by chains of sender and receiver threads. Updating on demand would avoid unnecessary traffic, but induces extra delay time.

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 4. 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.
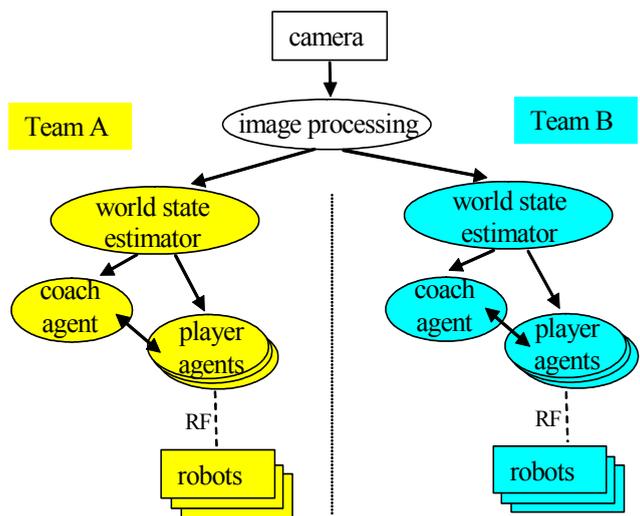
Figure 4. Controlling a complete robot match with two teams using a single camera system

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 – for example:

```
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:

```
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

5.1 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.

5.2 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.

5.3 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 type-specific exception classes derived from the `Cexception` superclass to catch different exception causes and to take 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 [7], the intelligent decision engine [8] and the motion planning subsystem [9]. 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 a simple application example that uses the framework is online available at the author's home page [10].

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.

References

[1] RoboCup: `www.robocup.org`
[2] Federation of International Robosoccer Association: `www.fira.net`
[3] D.R. Butenhof, *Programming with POSIX threads*, Addison-Wesley, 1997.
[4] M. Beck et al., *Linux Kernel Internals*, 2nd Ed., Addison-Wesley, 1997.
[5] A. Silberschatz et al., *Applied Operating System Concepts*, John Wiley & Sons, 2000.
[6] S. Oaks and H. Wong, *Java Threads*, 2nd Edition, O'Reilly & Associates, 1999
[7] N.S. Kooij, *The development of a vision system for robotic soccer*, Masters Thesis, University of Twente, 2003.
[8] R.A. Seesink, *Artificial Intelligence in multi-agent robot soccer domain*, Masters Thesis, University of Twente. 2003
[9] W.D.J. Dierssen, *Motion planning in a robot soccer system*, Masters Thesis, University of Twente. 2003
[10] `wwwhome.cs.utwente.nl/~schoute/ES_files/fc_esi_frame.tar`