Fourth Euromicro Workshop on Real-Time Systems
Athens, Greece, June 3-5, 1992
Proceedings pp. 68-73
IEEE Computer Society Press

ALBATROSS
The Communication Scheme as a Key to Fulfil Hard Real-Time Constraints

Ewald von Puttkamer, Christopher Wetzler & Uwe R. Zimmer

University of Kaiserslautern - Computer Science Department - Research Group Prof. E. v. Puttkamer
P.O. Box 3049 - W6750 Kaiserslautern - Germany
e-mail: uzimmer@informatik.uni-kl.de

Based on experiences from an autonomous mobile robot project called MOBOT-III, we found hard realtime-constraints for the operating-system-design. ALBATROSS is "A flexible multi-tasking and realtime network-operating-system-kernel", not limited to mobile-robot-projects only, but which might be useful also wherever you have to guarantee a high reliability of a realtime-system. The focus in this article is on a communication-scheme fulfilling the demanded (hard realtime-) assurances although not implying time-delays or jitters on the critical information-channels.
The central chapters discuss a locking-free shared buffer management, without the need for interrupts and a way to arrange the communication architecture in order to produce minimal protocol-overhead and short cycle-times. Most of the remaining communication-capacity (if there is any) is used for redundant transfers, increasing the reliability of the whole system.
ALBATROSS is actually implemented on a multi-processor VMEbus-system.

1. Motivation & Introduction

Why are realtime-aspects so important for our group? Before trying to find an answer to this question, we would like to give a really short survey of the goals in our autonomous mobile robot project called "MOBOT-III" in the form of a definition[1].

An autonomous mobile robot (AMR) is a system which perceives information about its environment in order to use this information for solving a given task. It has to be equipped with an onboard computer-system to do all computations independently and without external intervention. It must be able to explore unknown environments while building maps (of various kinds). Based on these maps the AMR navigates to specified goals while avoiding collisions with fixed or moving obstacles and performs local tasks like identifying and manipulating objects.

From the domain of realtime-problems in this area we will shortly discuss two as a motivation for our following strategies.

a. Guarantee for actual information - Most of the decisions have to be based on actual data. This implies the two classical realtime-demands. First, the information-packets have to reach their destination at all (☞ "Reliability") and second they have to reach their destination within a certain time-limit (☞ "Timeliness"). In our project any violation of these two demands would lead to a robot acting in a "world before our time".

b. Graceful degradation - In lots of situations the breakdown of one component makes life too risky for the robot and the only answer will be an immediate stop of all motors. But on the other hand it is not the best idea to stop the robot because one part, e.g. of the object-recognition-component is beginning to hallucinate. In this kind of degradation, the other parts must be able to decide that the output of this component does not make sense any longer and (very important) the crashed part must be isolated, in order to be not able to disturb the whole system. So we have to demand modularity as well as loose and secure coupling between the modules.

The above points should be enough to show our motivation to build a realtime-system fulfilling some hard realtime-constraints.

2. Assurances for each task

In this proceeding it is assumed that the underlying hardware is a distributed system with several processors, each of them used by a single task. Each of these tasks has to fulfil a constraint, which is simple to formulate but quite hard to assure: The job must be finished in time! There are a lot of different techniques to construct tasks in order to

1. for further information about the MOBOT-III project see [Edlinger 6/91], [Hoppen 4/90], [Knieriemen 3/91].

predict (to a certain degree) their timing behaviour - but these techniques are beyond the scope of this article. The interesting point here is: What are the basic requirements for the environment to enable the task to meet a certain time-limit. Seen from the perspective of a single task communicating with the whole (complex) system, what will be the necessary assurances a realtime-system has to give to this task?

a. Full CPU-power - All the CPU-time (which is known in advance and constant) is exclusively reserved for the local task. This sounds easy but there are two problems raised by this restriction:

a-1 No system -interrupts - The OS has access to the CPU only if the local task is explicitly calling the OS. There must not be any background operations at all (e.g. interrupts for the system time, etc. pp.)!

a-2 No communication-interrupts - If any information is arriving via the communication-system, the local task can not be disturbed by interrupts. So the necessary transports must be done by another processor. A local task may inform itself if it is interested in new data and is not informed when new data are arriving.

b. No direct connection to tasks on other processors
All the tasks are synchronised implicitly via the flow of data, i.e. a task will start (again) when it gets a new set of input data - an explicit trigger is not necessary! The decision to look for and to use new data depends on the internal state of the local task. Assuming a task in the system is crashed - so the only effect seen by other tasks is the lack of newer data. There is no direct connection to this crashed task. In most cases such a connection would be disastrous.

c. Non-blocking access to the communication-system
Every call for new data or for an export of new outputs must successfully end within a fixed (and of course short) time - even if one communication partner is broken down or in any other possible constellation.

d. Guaranteed actuality of all received information
This implies guaranteed transfer times. Because loss of data happens in the reality of a moving system, this restriction may only be approximated by redundant transfers.

Three and a half out of these four assurances refer to the communication-system. So it looks like the communication-scheme is playing an important role in the design process of a realtime-system. All the constraints above are fulfilled in ALBATROSS even if they are not mentioned again in the rest of the proceeding.

3. Communication-Scheme

When trying to satisfy all of the assurances above, there are two (apparently) contradictory directions of the way to design the communication. Each of the two dispositions leads to standard solutions, when trying to fulfil them isolated. Before discussing the combined solution, there are the two conventional paths.

a. Couple two processes as loosely as possible
Seen from the viewpoint of one processor for one task, this assumption guarantees in a natural manner a kind of graceful degradation. The typical implementations of this philosophy cover the whole bandwidth of message-passing-systems.

b. Assure restrictive transfer times - The closer two processes are coupled (the fewer communication-layers are between them), the better are the time-assurances for the communication-accesses. So if you like to get short transfer times, you will fit the two processes close together. Further (if this is still too slow) you will introduce restrictions in the communication-phases (e.g. no interrupts are allowed while reading in a communication-buffer).

It is obvious that solving the two requirements individually will not lead to the optimum. Up to here the proceeding contains only problem-descriptions, so it is time to show some solutions in the following chapters.

3.1 Realtime Ports

Before talking about protocols, we have to define the hardware-environment. There are two independent processors connected via a dual-ported-RAM, i.e. the memory-domains of the processors overlay. What is the definition of a dual-ported-RAM in this context? Both processors may access the same memory-area *simultaneously.* A conflict at the level of a byte-access has to be solved with some special hardware, in a way that neither side is being blocked and a byte will stay an indivisible element.

The realtime-transfer in ALBATROSS follows a realtime-philosophy which can be described by three short demands:

a. Consistency - Information may only be transferred in consistent units.

b. Actuality - Newer information has priority over an older one. This is a contradiction to the usual demand of keeping order, because you have to destroy old information at the level of the communication-system, if there is a newer one (of the same class) available.

c. Availability - Information must be available at any time.

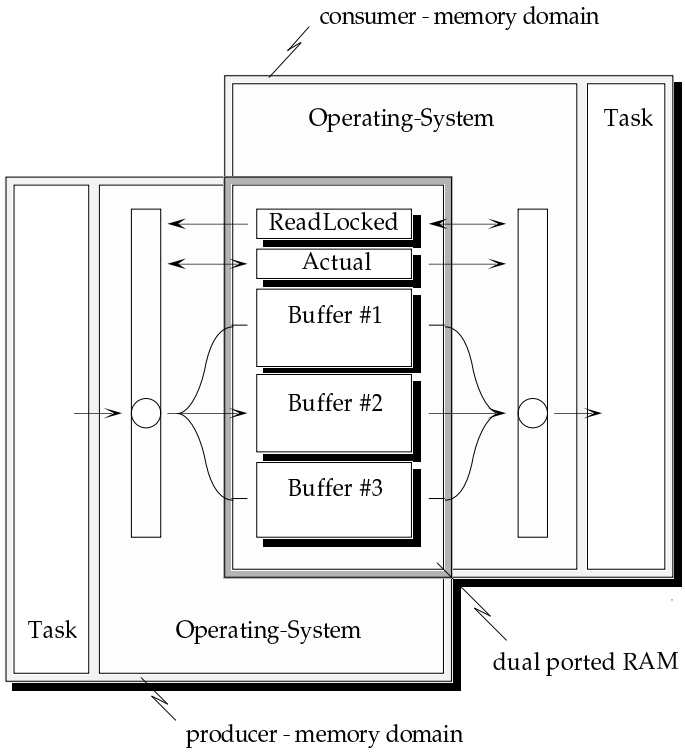

figure 1: **memory domains**

```
Type    BufferIndex  = (Buffer1, Buffer2, Buffer3);

Var     Actual,             {may only be written by the producer}
        ReadLocked          {may only be written by the consumer}
                : BufferIndex;

            {   Both variables have to be one byte long and }
            {   are located in the dual-ported-RAM area.    }
            {   The initial value of both variables         }
            {   should be "Buffer1".                        }
```

figure 2: **common data-structures**

Following these demands we have designed a simple implementation of the buffer-access as seen from the operating-system. The common base of collision-free and locking-free access with two asynchronous partners is the three-buffer structure. In figure 1 the connection between two processors at the hardware-level and the location of the buffers for the communication are shown.

In figure 2, figure 3 and figure 4 the implementation fragments (in a Pascal-like syntax) for reading from and writing to the communication-area are shown. The critical accesses to the variables "ReadLocked" and "Actual" are marked (**Bold** for a critical writing; *italic* for a critical reading).

Variables not under local control may change their values at *any* time. This fact seems to make any formal proof of the correctness quite hard. Fortunately there is only a

```
Type    Actuality = (Old, New);

Function Import (Var ImportedData: Datatype): Actuality;

    Begin
        Import:= Old;

    {--- Phase 1: select a buffer for read-access}
        While ReadLocked ≠ Actual Do
            ReadLocked:= Actual;
            Import:= New
        EndWhile;

    {--- Phase 2: read-access to the selected buffer}
        ReadFrom (ReadLocked, ImportedData)
    EndFunction Import;
```

figure 3: **buffer-access for reading**

```
Procedure Export (ExportedData: Datatype);

    Var Buffer, WriteBuffer, CopyOfReadLocked: BufferIndex;

    Begin
    {--- Phase 1: select a buffer for write-access}
        CopyOfReadLocked:= ReadLocked;
        For Buffer:= Buffer1 to Buffer3 Do
            If      (Buffer ≠ Actual) and
                    (Buffer ≠ CopyOfReadLocked) Then
                WriteBuffer:= Buffer
            EndIf
        EndFor;

    {--- Phase 2: write-access to the selected buffer}
        WriteTo (WriteBuffer, ExportedData);

    {--- Phase 3: assign completely written buffer as actual}
        Actual:= WriteBuffer
    EndProcedure Export;
```

figure 4: **buffer-access for writing**

small number of values, the critical variables may change to. So you are able to proof all the possible cases step by step.

To assure the correctness of the whole realtime-transfer you have to proof the following four points in detail.

- Mutual exclusion
- Definite results
- Termination
- Actuality

Up to here, only the access-routines at the operating-system level are mentioned. But how does this appear to the task? The syntax is simple and the semantic is much like an electric wire. There are two special functions for each variable, so all the possibilities of range- and type-checking may be used. For a consuming task the buffer access is masked by the following interface function:

```
LookForNew<VarName> (Var <VarName>: <VarType>): Boolean;

e.g.: LookForNewRadarMap (Var RadarMap: RadarShot): Boolean;
```

The boolean result signals the actuality of the read information (Is this information ever being read before?). For a producing task the buffer-access is hidden by the following interface procedure:

Make<VarName>Available (<VarName>: <VarType>);

e.g.: MakeRadarMapAvailable (RadarMap: RadarShot);

As the final remark for this chapter once again we would like to emphasize that reading or writing in this communication-scheme is free of blocking even when the communication partner has crashed in a critical phase!

3.2 Need for a communication-controller

In a real system one processor (or a small number of processors) will be implemented on one board. So one of the communication-partners can only access the dual-ported-RAM via some kind of communication-system (normally a short range bus-system). This means a break in the real-time-communication scheme shown so far, because the communication is not symmetric at the physical level. One processor may access the dual-ported-RAM much like the local RAM, while the other processor has to use a communication-system to access the same dual-ported-RAM.

A new aspect appears from here on. What happens if the access to the (far) dual-ported-RAM fails, because of a disturbance on the communication-system? In the above discussion a memory-access was a local transfer and therefore without any aspects of a failed communication.

With this problem in mind we are running into a contradiction. On the one hand it is necessary to finish a transfer in a short predefined time, but on the other hand a failed access via the communication-system must be repeated. The processor for the local task is not available any longer after the first failed trial. So which processor will initiate the second trial? The problem can only be solved by introducing a third processor as a host for the *communication-controller* as shown in figure 5.

From the view of the tasks any buffer-access looks like an access to the local RAM, i.e. it can be successfully done in a well defined time. The communication-controller may read from or write to the (far) dual-ported-RAMs several times, without disturbing the local tasks at all.

3.3 Cyclic transfers

Cyclic transfers means that there is a pre-scheduling of the communication-slots instead of transient transfers while the system is running. The pre-calculated scheduling plan is then executed in a cyclic manner. We will highlight the aspects of this kind of transfer in the form of three questions.

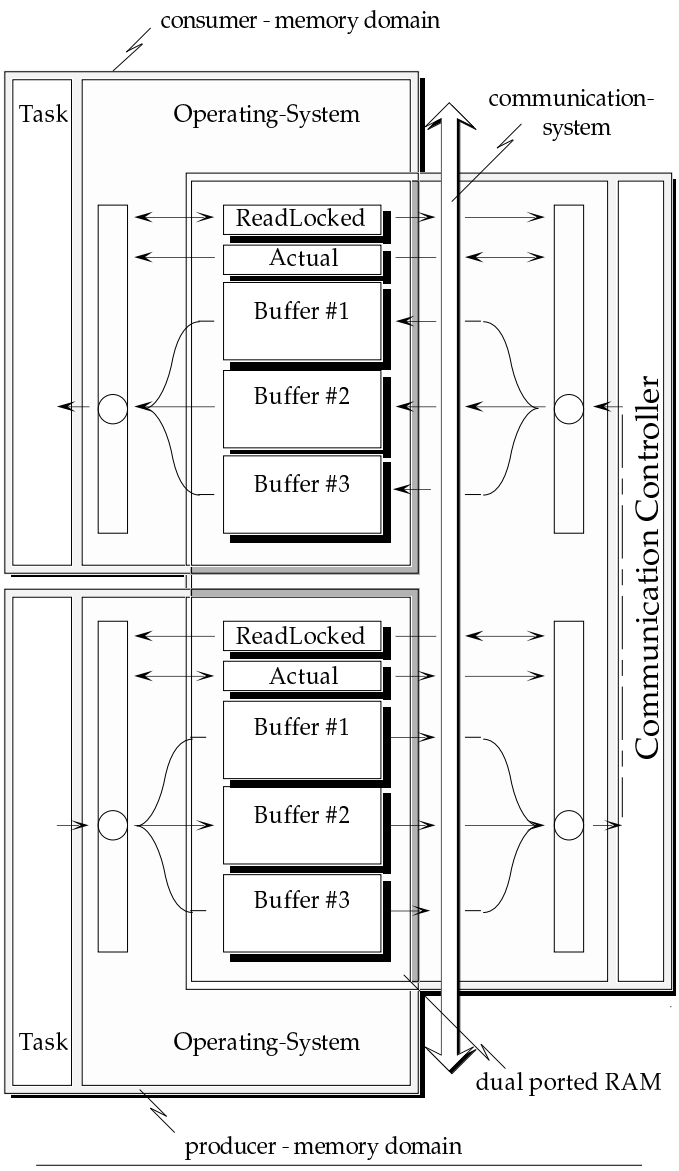

figure 5: **the communication-controller**

What is the major problem with transient transfers?

Whenever a producer wants to distribute it‘s results (this happens completely asynchronously) it runs through an arbitration-phase on the communication-system. It is quite difficult to calculate the time, the process has to spend in this phase.

What is the restriction to pre-scheduling?

The restriction is really simple - you have to know the maximal communication times in advance. In a realtime system the worst case conditions must be calculated. From the worst case conditions you have to deduce the highest needed sampling frequency.

What are the advantages of a cyclic-transfer-system?

The construction is based on worst case conditions, i.e. the worst case may happen without any confusion. Additionally it is possible to do something more, a kind of oversampling. If the maximal needed sampling frequency is known, three possibilities are implied here. First the communication-system is not able to transfer data with such a frequency. So you have to look for some quicker hardware, or you have to relax your realtime-constraints. Second, the communication capacity just corresponds to this highest frequency, so congratulations for the configuration department. But the normal case will be, that there is some extra capacity of the communication-medium, even in the worst case. In this context worst case means, the constraints of the physical system are considered, but all the computer hardware is assumed to be without any failure, especially the communication-system. So why not using this extra capacity for a number of redundant transfers? In a physical environment this may be interpreted as oversampling. From the computer scientist‘s point of view this means we are using these extra resources for redundancy. The best we can do to avoid the risk of transient failures, is to use all the communication-capacity for as much redundant transfers as possible.

3.4 Transfer synchronization

Up to here we have introduced a communication-controller and a cyclic-transfer strategy, but some problems are still left. These problems will become obvious when we examine the timing-behaviour of the communication-system as seen by the local tasks. Two important points should be listed:

- The communication-channels are served with a guaranteed frequency or even more often.
- The exact time of the update-event (i.e. arrival of new information) on any communication-channel is not explicitly known by the local task. (And the update-event does not force any interrupt!)

Assuming this communication environment, what will be the behaviour of a typical task? At some state the task will look for new input-data. If there is actual (i.e. not yet processed) information already available, the task will start calculating. Otherwise the task will do some busy-waiting loops on the requested input-channels, until the new information has arrived. After finishing the specified work on this new information and having made some output-data available for the rest of the world, the task will check the needed input-channels and the procedure starts again. So, there is an implicit synchronization on the input-data. An explicit specification of the update-time is not necessary.

So far with the nice part, but there are some other tasks in a realtime-system also. First, for correlation purposes, some tasks will need a global time. For example you might think of a module composing data from different outer sensors to one representation. Here you have to be able to use the sensor-information from different sources (transmitted via the communication-system), sampled at the same global time.

Theoretically a mechanism for distributing a global time is sufficient, but in our environment we have made the following experience: In most of the cases the global time is used only indirectly for correlation of some outer incidents. The correlation is mostly done by the position in space at which the sensor-data is sampled. (Of course, our realtime system is moving around.)

In the following chapters we will introduce two different strategies solving the “correlation-problem”.

3.4.1 Time-rigid transfers

The execution of the scheduling-plan (in this first model) is only in the responsibility of the communication-controller. So the only way to make a global time available is to produce the global time by the communication-controller itself and embed this global time as an ordinary realtime-transfer in the scheduling-plan.

What are the advantages of this strategy?

a. The communication-capacity might be used nearly optimal, because the timing of the communication is done by one processor and all the execution-times are exactly predictable.

b. This model implies a high reliability because of two facts:

b-1 The implementation of the communication-controller is trivial. (Here the execution-phase is addressed, *not* the calculation of the scheduling-plan.)

b-2 The communication-controller is completely independent from the other processors.

c. A global time is available and completely embedded in the whole communication-structure. The global time is exact just in the moment of each update-event on this global-time-channel.

But there is a price to pay when using this model:

The global time is quite inaccurate, because the distribution frequency cannot be higher than the maximal frequency from the scheduling-plan. In order to enlarge the accuracy of the global time the local tasks need local timers synchronized to this global time. Because no interrupt is triggered at the update-event on the global time channel,

the tasks are forced to do some busy-waiting loops on this update-event from time to time (depending on the accuracy of the local timers).

The direct and easiest way to overcome this problem is to implement an asynchronous way of making the global time available. Either by an extra hardware or by spending some amount of communication-capacity for this asynchronous time transmission. The second version would lead to a massive violation of the introduced communication paradigm and is not examined here any deeper, for this reason. The hardware version is proposed by [Kopetz 9/87]. Assuming that we have not any access to a special hardware and we do not want to use the questionable version of asynchronous transfers in the background, we might introduce an alternative synchronization scheme.

3.4.2 Position-synchronized transfers

Assuming that the different correlations between the sampled informations in the system are either done by a global time or by a globally available position in space (depending on the task), the above synchronization scheme implies two main problems:

- The position-producing task must be synchronized on the global time.
- The position information itself is subjected to an unpredictable time jitter.

In order to overcome these problems two changes in the synchronization-mechanism are necessary:

a. The global time and the global position have to be produced by the same task.

b. The whole communication system must be synchronized with the combined time-position information.

When the time and the position is produced by the same task they are correlated automatically, and in the best case this is all done by the communication-controller itself. If the global position cannot be produced by the communication-controller itself for some reasons (e.g. no direct connection to the outer world), the communication-controller has to be synchronized on the position-producing task and is therefore not completely independent.

4. Conclusion

Finally we will give a short collection of the main strategies used in ALBATROSS.

a. Worst case as normal case - Calculating of the worst case conditions, as given by the outer world. These conditions deduce a highest sampling frequency needed for the control of the physical system.

b. Dividing the problem - Splitting the whole problem in a number of (to a certain degree) independent tasks.

c. No hard synchronization - The divided tasks should only by synchronized via the flow of data, not via a “hard” synchronization scheme.

d. Pre-Scheduling of the communication phases - Generating a pre-calculated (i.e. not calculated at runtime) scheduling-plan, based on the known highest sampling frequency and the available capacity on the communication-media, using redundancy, i.e. over-sampling.

e. Prefer correlation-critical information - Data needed for the various correlations in a realtime-system have to be transferred without time jitter.

Summarising the realtime-aspects as shown in this article, the key to realtime reliability seems to be the communication-scheme. So perhaps the often mentioned interrupt response times are not the whole truth in a realtime-world.

ALBATROSS

References

[Edlinger 6/91]
Thomas Edlinger, Ewald von Puttkamer, Thomas Strauch
An efficient navigation strategy for an autonomous mobile Robot
14th IASTED International Symposium “Manufacturing and Robotics”; Lugano, Schweiz, June 1991, 60-63

[Hoppen 4/90]
Peter Hoppen, Thomas Knieriemen, Ewald von Puttkamer
Laser-Radar based Mapping and Navigation for an Autonomous Mobile Robot
IEEE International Conference on Robotics and Automation 1990, Cincinatti, Ohio 13-18 May 1990 pp. 948-953

[Knieriemen 3/91]
Thomas Knieriemen, Ewald von Puttkamer
Realtime Control in an Autonomous Mobile Robot
International Workshop - Information Processing in Autonomous Mobile Robots - University of München, Germany, March 6th to 8th, 1991

[Kopetz 9/87]
Kopetz H., Ochsenreiter W.
Interval Measurements in Distributed Real Time Systems
Proc. 7th Intern. Conf. on Distributed Computer Systems Berlin, Sept. 1987, IEEE Press, pp. 292-298

[Northcutt 87]
Northcutt J. Duane
Mechanisms for Reliable Distributed Real-Time Operating Systems
Perspectives in Computing, Vol. 16, Rheinboldt W., Siewiorek D. (eds.), Academic Press, Orlando, 1987