Invocation of Real-Time Objects in a CAN Bus-System

Jörg Kaiser and Mohammad Ali Livani
University of Ulm
{kaiser,mohammad}@informatik.uni-ulm.de

Abstract

The paper focuses on method invocation of real-time objects in a CAN-based distributed real-time system. A simple object model is introduced, which allows the convenient modelling of hardware and software components. Related to the object model, two issues are discussed. Firstly, a model is introduced which allows to form and address object groups. This reflects a basic need in a real-time system to distribute information to multiple clients efficiently. Secondly, we discuss an approach to express timing requirements for object invocations. To achieve distributed consensus on communication resource access, an EDF-like approach is introduced, which takes advantage of knowledge about deadlines, the number of remaining communication activities, and the remaining worst-case execution time for the invoked method at each point of time.

1. Introduction

Future computer systems will, to a large extent, monitor and control real-world processes. This results in an inevitable demand for timeliness and reliability. Distributed systems which inherently provide immunity against single failures are an adequate architecture to meet the goal of reliability. Additionally, because real-world applications often require the spatially distributed control of electromechanical components, a distributed system architecture brings computing power to the points where it is needed. The availability of inexpensive yet powerful microcontrollers supports a distributed solution. This results in a system model which is composed from smart components implementing the instrumentation interface to the real world comprising sensors and actuators. A convenient way to model such an environment is to use an object-based approach where objects encapsulate all kinds of entities necessary to control a physical process.

We can identify at least two layers in systems interacting with the real world. A lower responsive system level tightly controlling the sensor/actuator interface to the world and a higher system level responsible for interpretation and evaluation of the perception of the world as communicated by the instrumentation interface. In this layer, higher level decisions are performed. Although these different levels have different requirements concerning responsiveness, they are sometimes not separated clearly simply because of the hardware architecture. We concentrate on the lower system level and describe an object model which is motivated by a system structure which relies on a modular design and independent computing resources for the components of the instrumentation interface. Rather than having a central powerful processor in which electromechanical components are only modelled as objects from which control signals are generated, the electromechanical components are objects themselves powered by inexpensive microcontrollers. The objects of this layer have a well defined interface to be easily controlled by a higher level instance which can exploit this abstraction rather than dealing with low level control signals.

The hardware modules are connected by a field-bus. We choose the CAN-bus (CAN: Controller Area Network), developed by BOSCH[[28]] because it provides advanced technical features and represents an emerging standard with a wide applicability. Moreover, popular microcontrollers are available with an on-chip CAN controller. CAN is a shared bus designed to connect control systems in a spatially restricted area like cars, robots, tool machines, and other automotive or industrial automation applications. It is targeted to operate in a noisy environment with speeds of up to 1 Mbit/sec, exchanging small real-time control messages. We use the CAN message format to uniformly invoke methods on our objects. This allows a transparent object invocation in the sense that an invoking object must not know where the invoked object resides in the system. Additionally, we can address groups of objects with a single CAN message.

As a common resource, the communication medium has to be shared by all computing nodes. Access to the medium has to be scheduled in a way that distributed computations meet their deadlines in spite of competition for the communication line. Since the scheduling of the

bus cannot be based on local decisions, a distributed consensus about the bus reservation has to be achieved. Clearly, this is only a special case of the more general problem to schedule a cooperative distributed computation.

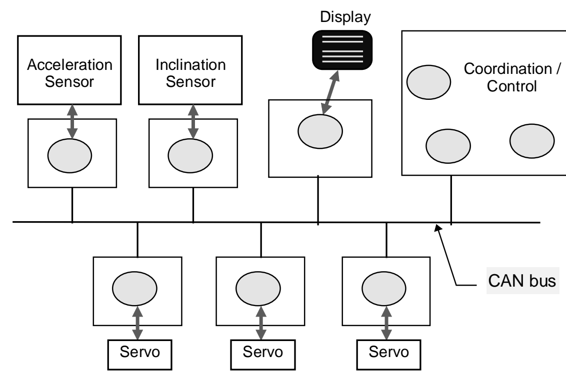
There exist several alternative approaches to solve this problem based on the assumptions about the behaviour of the system and the environment. The first approach, known as time-triggered approach [[18]],[[21]], [[19]] assumes a complete knowledge of all future actions of the system. Hence, during operation these systems exhibit minimal overhead paired with a maximum of predictability and are highly appropriate for safety-critical applications which can be modelled by a periodic behaviour.

A more adaptable and economic way to manage system resources has to be based on run-time information. These real-time systems have to plan their operation at run-time, at least to a certain extent. Although the trade-off generally is between adaptability and predictability, these systems are scalable in the sense that they allow a coexistence of activities with different real-time requirements. These systems rely on run-time guarantees [[30]], [[11]] rather than on a preplanned feasibility guarantee only. A form of calendar-based scheduling [[22]], [[30]] is an approach to run-time guarantees where hard real-time activities have the possibility to reserve resources in advance. The free resources can be used by less important tasks. It should be noted that scheduling policies like e.g. earliest deadline first (EDF), least laxity first (LLF) or rate monotonic scheduling only guarantee optimality, i.e. if a schedule exists, it will be found. But in the dynamic system, there may be conflicting requests for which these scheduling policies will not find a solution. In this case they fail to guarantee anything. A calendar, on the other hand, is a mechanism to guarantee resource availability for reservations which have already been made.

The contributions of the paper are twofold. First, we discuss an object model appropriate for our application scenario. Secondly, this paper describes the enforcement of global scheduling decisions on the shared resource of the communication bus to perform object invocations. It guarantees that firstly, in spite of the restricted local knowledge of the global schedule, hard real-time messages are always sent timely. Secondly, the bus resource can be utilized optimally. We allow hard real-time messages to be sent as early as possible, even before the reserved time slot without affecting the timeliness of other hard real-time messages. Secondly, each node is free to send soft real-time messages or non real-time messages at any time. As described in our paper, an EDF scheduling is used to enforce the reservations in the calendar.

The paper is organized as follows: Chapter 2 briefly introduces a system model and defines the requirements for the objects. Chapter 3 describes the CAN message format and how it is structured for our invocation mechanism. Chapter 4 deals with the guarantees for the timely message transmission and finally chapter 5 concludes the paper with an outlook on our future research.

Fig. 2-1: An application example



2. The System Model

Our system model is influenced by our anticipated hardware basis. We assume a number of different microcontrollers with different performance attributes ranging from 8Bit to 32Bit architectures. All microcontrollers are equipped with a CAN-Bus interface. We exploit the different performance and price characteristics of the microcontrollers to structure the overall task of the system into small packages. Since the simple microcontrollers are in the range of a few $s, it must no longer be a rare resource. In our application example, a simple active suspension system (Fig. 2-1), we use a simple microcontroller for each servo or complex sensor. A servo is composed from a motor and an internal sensor which provides feedback on the current position. This sensor/actuator system is connected to the CAN-bus to cooperate with other similar objects and receive control commands through CAN messages. No direct, low level control signals are visible outside the module. Thus, encapsulation and a well defined interface are supported. On the logical level, the sensor/actuator block is seen as

an object on which certain methods can be invoked, e.g. to position itself to a specific angel, to communicate the angel, etc. A sensor/actuator constitutes the lowest level in the control hierarchy, comparable with a simple reflex loop in a biological system. Higher control instances are available to control groups of objects or eventually the entire system using more powerful microprocessors. Thus, the higher control instances have a well defined instrumentation interface which is composed from objects, each encapsulating a certain functionality.

An object has a unique name and a set of associated operations. The unique name of the object is translated to a short form system name during run-time and maintained by a configuration service. Objects in such an environment must have some extensions to the sequential, passive model known from the field of programming languages. Particularly our model includes:

1. Active, autonomous objects.

Since objects may have a dedicated processor, it is straightforward to assume a model of autonomous, active objects. An object is characterized by a name and a list of methods which can be invoked on the object. Because the object is active, it can also export information. This is equivalent to sending an invocation to another object or to a group of objects without a previous request. As described in chapter 3, we provide a transparent group communication mechanism, i.e. an object itself usually does not know whether it communicates with an object on a remote or on the local node. This group communication mechanism is a basis for remote method invocation of object groups. In fact, communication relations can dynamically be defined during run time. So, objects may join or leave a group, and are autonomous in deciding when to process a request.

The concept of TMO (time-triggered message triggered object) [[16]] derived from the RTO.k (Real-Time Object) [[17]] concept enhances the conventional object model by mechanisms to specify the temporal behaviour of an object. It provides a framework to specify a complex real-time system together with its environment in the same uniform representation. The aim is an integrated design of a distributed real-time system and simulators generating real-time input modelling the target applications. A TMO exhibits active, spontaneous behaviour as well as providing services requested by a method call. Three execution models are derived for executing TMOs [[17]]. Among them, the execution model which provides a dedicated processor for each object (Type II) is very similar to our basic system structure. The difference on the conceptual level is the notion of the object group and the fact that we do not relate spontaneous behaviour to clock events. However, if the spontaneous activity of an object is hard real-time, we also have to map it to some reserved time slot.

CHAOS [[11]] demonstrates that the object model can be tailored to meet the efficiency requirements of real-time applications. The motivation also comes from a robotics application where electromechanical components are encapsulated and controlled by objects. CHAOS provides a spectrum of objects with different weight ranging from light weight passive objects to objects including several processes providing scheduling and synchronization facilities. The concept is targeted to rather powerful parallel machines. The active objects are too heavy weight to implement them with our simple hardware basis.

In Alpha, [[12]], objects are passive abstract data types in which a number of processes can execute concurrently. MARUTI [[22]] provides modules in the design process. However, these modules are broken down into elementary units (EUs) which constitute the atomic entities of sequential execution. Also, EUs resemble to objects, they are not user defined but derived during the compilation process [[25]].

2. Object groups.

In conventional object-oriented languages a method invocation is synchronous and directed to a single object. It is usually not possible to express a request to a group of objects. However, in a real-time control system it is beneficial to provide groups of objects and to use asynchronous multicasts to invoke methods of these groups. The motivation ranges from simple and fast distribution of information, like sensor data and alarm messages, to replicated objects forming a group to achieve fault-tolerance. Because the communication medium, the CAN-Bus, supports consistent multicasts on the low system level, it is beneficial to exploit this feature. It allows to address a group of N objects with one message rather than sending N individual messages. To a certain extent, CAN provides atomicity of message transfer, i.e. either all operational nodes correctly receive a message or none of them.

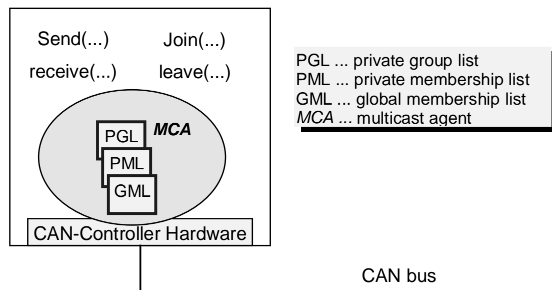
The consistent view of the message status between all nodes is one of the most important features of the CAN bus (cf. chapter 3). It is achieved on the hardware level by a synchronous bit transmission approach. All receivers of a CAN message (including the sender) scan the bus and analyze the current message status during the transmission of every transmitted bit. If a node locally detects a transmission failure, it immediately invalidates the current message by intentionally producing an detectable error on the bus. Every node including the sending node now is aware that the message transfer failed. Automatically, the sender will eventually

retransmit the message. If no transmission error is detetcted, all operational nodes have received and accepted the message. A more detailed description of this feature and its exploitation for atomic group communication is given in [[14]][[28]].

The basic inter-object communication in systems like JAVA and CORBA focus on a point-to-point communication. There are some one-to-many communication methods in JAVA like the observable/observer class [[1]] or recently, the iBus [[24]]. Both mechanisms are based on a explicit registration of clients at the server which differs significantly from a general group communication mechanism which is a many-to-many communication paradigm.

The transparent communication mechanism used e.g. in MARUTI [[25]] enables the objects to use a send primitive without specifying the receiver(s). This is done by specifying the communication channel bindings in the MARUTI configuration language (MCL). The goal is to separate functional and non-functional issues. However, the binding is performed during compile time. No dynamic changes of the communication relations are possible.

Fig. 2-2: The multicast agent



The communication model which we adopt is similar to that of autonomous decentralized systems (ADS) [[26]]. In this model, software subsystems autonomously manage themselves and coordinate their activities with other subsystems. This coordination is achieved by the data field, which represents global information. The autonomous entities can extract relevant information from the data field. Based on this mechanism, groups of objects, sharing a subset of the global information, can be constructed.

In order to support real-time object groups with consistent information, the group communication protocol must deliver real-time messages to all members of a group both *timely*, and in a *consistent order*. In section 4 we describe how to guarantee timely multicast delivery by exploiting the CAN bus arbitration mechanism. Given the guarantee of timely message transmission, we have also shown in [[15]] that the deadline can be used to consistently order hard real-time multicast messages.

3. Use of time information.

Objects must have some notion of time to express execution times of their methods, deadlines, and slack time. In a dynamic real-time system, on-line scheduling decisions are based on these assumptions which therefore must be available at run-time. The temporal behaviour of the object must be specified, verified, and monitored. An approach to bind time information to object classes and monitor the temporal behaviour of the system is described in [[10]]. It is planned to use these mechanisms to specify and monitor the temporal behaviour of the system. In this paper, however, we only describe the use of time information for the dynamic reservation of the bus re-source. This issue is discussed in more detail in section 4.

Group Communication Service

There are many group communication protocols dealing with a reliable multicast[[3]][[4]][[13]]. Some of them also consider real-time properties [[32]]. However, using CAN bus as a basic communication medium, we can relax many assumptions necessary for arbitrary networks, resulting in a rather low overhead protocol. For a detailed discussion the reader is referred to [[14]]. In our system model, an object participating in group communication, does not necessarily have any knowledge about the number and location of other group members. Therefore, the sender of a multicast usually does not know whether it has to send the message locally, or remote. This simplifies the design and implementation of the objects, and minimizes the configuration effort when adding or removing an object.

But consequently, there must be an instance in the system, which knows the configuration. The configuration is given by object groups, where a group contains one or more objects. This configuration information is maintained by distributed multicast agents, one residing on every computing node. Every object wishing to send a multicast message, requests its local multicast agent (MCA) to deliver the message to other group members.

Every MCA (Fig. 2-2) maintains following configuration knowledge to support group communication: firstly, a list of object groups, from which at least one member resides on its host (private group list PGL). Secondly, for each object group, a list of ‘local’ objects, which belong to that group (private membership

list PML). And thirdly, for each object group, a list of nodes, which host at least one member of that group (global membership list GML). The former two lists are used for delivery of incoming messages, and the latter two lists are necessary for sending multicast messages. The communication services of the MCA guarantee reliable delivery and consistent order of messages at non-faulty sites.

The MCA also acts as a broker for the group membership services *Join* and *Leave*, because an object requesting a group membership service does not know where it is located, and hence cannot provide necessary configuration data to the global configuration server.

The MCA is a mechanism to separate the functional behaviour of objects from non-functional configuration structures. Different from MARUTI, the MCA maintains

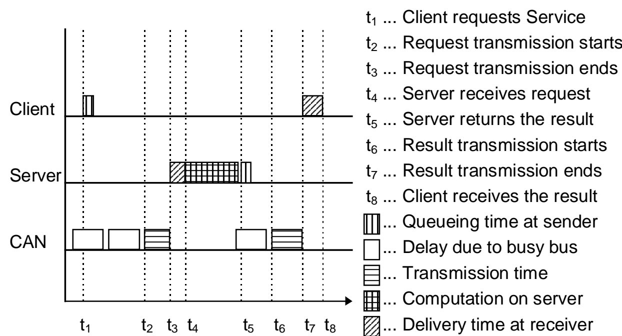


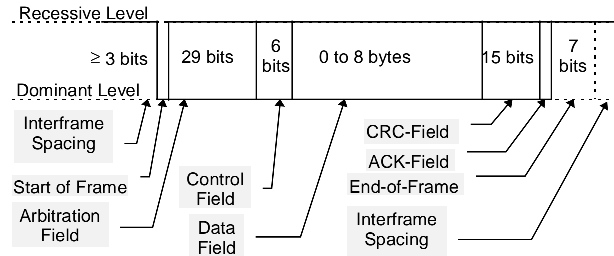
Fig. 2-3: The activities of a method invocation

this information during run-time, allowing dynamic configuration changes.

Communication Delay as a Critical Timing Factor

A real-time activity has to be completed within a given deadline. Even if the client and the server have negotiated a feasible schedule, the bus may become a bottleneck. This is because it is a central resource that can be used asynchronously by other nodes. On a CAN bus, messages are non-preemptive, and the message transmission on a 1 Mbit/s bus takes between 66 and 154 µsecs. Compared to the speed of processors, this a substantial time. If the communication activities are not completed timely, the end-to-end method deadline will be missed in despite of feasible schedule for the client and the server. In chapter 4, we present a mechanism which guarantees timeliness of messages in spite of decentralized bus arbitration. Fig. 2-3 illustrates a timing diagram for the invocation of a method. Here, the time between the method invocation request (t_1) and method response (t_8) depends on

Fig. 3-1: CAN message format (Data Frame)



communication times (t_4-t_1 and t_8-t_5) as well as the method execution time (t_5-t_4) at the server site.

3. Exploiting the CAN Message Format for Method Invocation

Due to the restricted computational power of cheap microcontrollers and the constrained bandwidth of a field bus, one of the very important issues in such an embedded real-time system is efficiency. The available resources have to be optimally exploited. As a consequence, low level features have carefully to be considered to achieve a higher functionality without too much overhead. There are different application-layer communication protocols available for CAN bus [[6]], [[7]], [[8]]. However, non of these protocols supports object groups, and deadline-driven scheduling of the communication resource. Therefore, we discuss the issue of invocation on a rather low system level. Also, the semantics and synchronization of the invocation mechanism is not fixed at this level. Rather, we provide a uniform invocation format and discuss the planning of the necessary communication. Because we assume a homogeneous system without inheritance and dynamic binding, method invocation collapses to addressing and resource planning.

The invocation of an object is a distributed activity consisting of communication and computation actions. To initiate a service, a request message is sent to the server. This message contains the identification of the server, the method name, and the method parameters. If a response is expected, the identification of the client, and optionally the response deadline, are included in the parameter list.

Due to the strictly limited size of CAN messages, the structure of a message must be chosen very carefully. Fig. 3-1 shows the general structure of a CAN Data Frame. Up to 8 bytes of the message are user data (Data field). The remaining fields are used for bus arbitration (Arbitration field), format control (Control field), cyclic redundancy check (CRC field), and fields for

synchronizing the message between transmitter and receivers (Start/End of frame, ACK field, Interframe spacing).

The data field and the arbitration field are under user control. CAN provide a connectionless protocol, where the arbitration field identifies the message. It is also used to resolve collisions on the bus. The value of the arbitration field determines the priority of the message in the bus arbitration process. If the arbitration field is interpreted as a binary number, the priorities are defined in decreasing order, i.e. '0' represents the highest priority, and '2^{29}-1' the lowest priority. Whenever several transmitters compete for the bus, eventually the message with the highest priority wins the arbitration, and can be transmitted without additional delay. As a consequence, two messages which may be sent at the same time, must have different arbitration fields.

The CAN controller has a built-in associative message recognition hardware. If the arbitration field complies to a certain pattern, the message is accepted. This associative filter can recognize groups of messages by masking the respective comparison bits. By exploiting this mechanism and appropriately structuring the arbitration field, we want to achieve:

- applying dynamic priority access control (as described below),
- selectively addressing object groups or individual objects.

The dynamic priority is encoded in the 8 most significant bits of the arbitration field. It is called dynamic, because its value is changed over time by the transmitting node. By relating the priority to the time until transmission deadline, a deadline-driven scheduling can be achieved. The transmission deadline denotes the point of time before which a message has to be sent. In chapter 4, the detailed encoding of the dynamic priority field is described.

The next field (TxNode) identifies the sending node. The ID of the sending node is included to guarantee that different senders may never generate equal arbitration fields, hence no two messages competing for the bus have the same arbitration field. The remaining fields are utilized to address objects as shown in Fig. 3-2. Within this field, we can freely trade the number of groups, specified in the RxGroup field against the number of group members, indicated in the RxObj field. The method name and method parameters are contained in the data field.

Fig. 3-2: the structure of the partitioned CAN arbitration field

	8 bits	8 bits	8 bits	5 bits
Max. 256 groups, group size < 32	Priority	TxNode	RxGroup	RxObj

	8 bits	8 bits	9 bits	4 bits
Max. 512 groups, group size < 16	Priority	TxNode	RxGroup	RxObj

	8 bits	8 bits	10 bits	3 bits*
Max. 1024 groups, group size < 64	Priority	TxNode	RxGroup	RxObj

*) in this large system, for individual addressing of group members, at least 3 bits of addressing information must be placed into the data field.

Multicast Addressing and Message Filtering

CAN is a broadcast medium. In order to implement group communication on CAN, we exploit the feature that the receive buffers can be programmed to receive messages selectively. The group name is used as the key field of the associative filter. All other parts of the arbitration field are masked out. All messages which pass the filter belong to groups, from which at least one member resides on the node. The MCA now uses the RxObj field to decide whether the group message is directed to the entire group (RxObj = '0'), to a local group member, or to a group member not residing on the node. In the latter case the message is discarded.

4. Guaranteeing Timely Method Invocation

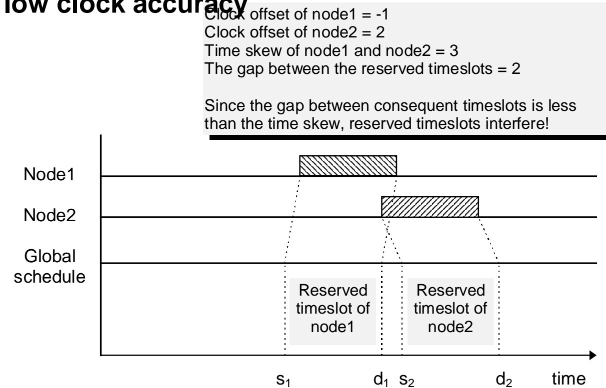
We assume four classes of activities in our system, which have been introduced in [[30]]. Critical activities are hard real-time and their deadlines are absolute in time. To guarantee the timely execution of critical activity, all their occurrences have to be predicted and respective resources have to be scheduled in advance.

Essential activities have deadlines relative to their start times. If the system grants an essential task, it guarantees the resources for a timely execution. However, the system can refuse a guarantee. Approaches dealing with essential tasks are e.g. the Spring kernel guarantees[[30]], and TAFT[[27]]. Soft real time activities have deadlines which are considered by the system but no guarantees are given to meet such a deadline. Soft real-time activities are executed on a best effort basis, however, as shown later, an EDF-like scheduling is used to assign resources. Non real-time activities can only use resources which are not requested by a real-time activity.

The global scheduling in a distributed system requires consensus between all participants about the usage of shared system resources. Particularly, if a joint action will be performed, all local resources must be available and reserved for the respective computation. The global plan has to be enforced by all participants, based on their local information. In a completely static system, a global calendar is available and each participant has its relevant entries referring to its activities in a global time scale. A local activity may only be started according to this schedule. In a more dynamic system where critical, essential, soft real-time and non-real-time tasks coexist things are more complicated. If a computing resource is free it may start computation and request resources. In this case, it must be guaranteed that it does not block an activity with a higher criticality. In this section, the enforcement of the global schedule for the shared bus resource is described.

There are several approaches to achieve a global schedule for real-time communication on CAN bus. The deadline-monotonic approach [[2]], [[31]] guarantees meeting deadlines as planned by an off-line scheduler for a static system with periodic tasks. In [[33]], an EDF-like approach has been proposed to schedule the CAN bus. However, this approach makes unrealistic assumptions about CAN – e.g. 10 Mbits/s – and exhibits a rather restricted scheduling ability due to a short time horizon [[23]].

Fig. 4-1: Collision of reserved time slots due to low clock accuracy



Our scheduling approach for hard real-time communication requires access to a global time reference. Once a time slot is reserved, the respective action can be started locally. To guarantee that it does not interfere with another time slot, the time reference of all nodes must be synchronized. The lower the clock accuracy, the larger the gap between two subsequent time slots in the global bus schedule. Fig. 4-1 shows a situation, where a too low clock accuracy causes a collision of different time slots. In order to provide a global time reference with high accuracy, clock synchronization mechanisms have to be applied, e.g. as described in [[20]].

For hard real-time communication, a deadline is guaranteed by reserving a time slot on the bus. The transmitter enforces the reservation by dynamically increasing the priority of the message according to its laxity relative to the reserved time slot. Thus, the message gains the highest possible priority at the beginning of its reserved time slot.

Soft real-time messages are scheduled by the EDF strategy, and non real-time messages are scheduled by assigning fixed priorities.

All of these scheduling mechanisms share a unique priority scheme. We guarantee timely transmission of hard real-time messages, because firstly, hard real-time messages always have higher priorities than other messages, and secondly, a hard real-time message gains the highest possible priority at the beginning of its reserved time slot. Hence, a hard real-time message is guaranteed to be started at the beginning of its reserved time slot or earlier. We guarantee optimal scheduling of soft real-time messages, because firstly, their priorities are higher than that of non real-time messages, and secondly, the priority of a soft real-time message depends directly on the time remaining until its deadline, thus realizing EDF scheduling. We schedule non real-time messages by

fixed priority assignment, because the importance of a

Fig. 4-2: Encoding message priorities into the arbitration field

Hard RT messages	0	Laxity*	TxNode	RxGroup/Object
Soft RT messages	1 0	Deadline*	TxNode	RxGroup/Object
Non-RT messages	1 1	Priority	TxNode	RxGroup/Object

* due to the wired-AND implementation of CAN, 0 means a higher priority than 1

non real-time message does not depend on the passage of time.

As we already discussed in section 3, the priority of a CAN message is placed into the first byte of its arbitration field. Fig. 4-2 illustrates how the 'time to deadline' and laxity of real-time messages are encoded into the first byte of the CAN arbitration field. This value is decreased by the passage of time, thus increasing the message priority dynamically. Since the dynamic changes to the message priority are applied by periods which are as short as the shortest possible message transmission time (66 bit-times), the priorities of all pending messages are increased by each arbitration round. The prefix '0' of the hard real-time laxities guarantees that hard real-time messages always have higher priorities than other messages. The prefix '10' of the soft real-time deadlines guarantees that soft real-time messages always have higher priorities than non real-time messages.

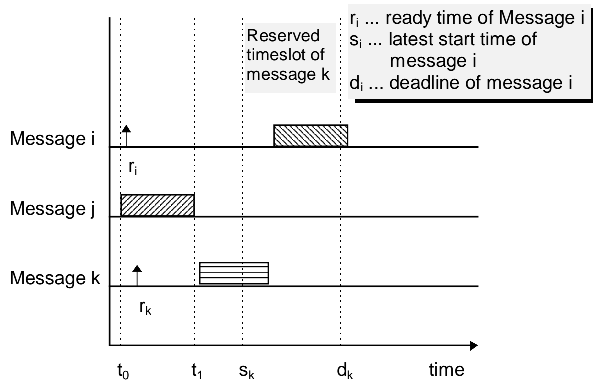


Fig. 4-3: The competition for the bus near a reserved time slot

Our approach supports efficient reservation of time slots for hard real-time communication by defining the beginning of the reserved time slot of a message as its latest start time. Fig. 4-3 illustrates a situation, where several transmitters compete for the bus access, near the reserved time slot of message k. Messages i and k are ready to be transmitted after t_0, where the transmitter of j has started transmission. Because CAN message transfer is non-preemptive, message j is completed regardless of its priority. As the bus becomes idle at t_1, messages i and k compete for the bus according to their priorities. We assume that the next reserved time slot after t_1 begins at s_k, and belongs to the message k. Then message k is guaranteed to have the highest priority at t_1, and to win the arbitration process.

Note that k may be delayed by one message, which is started before the time when k becomes ready. We define ΔT_{max} as the longest possible message transmission time including all overheads. Then a hard real-time message k - with its reserved time slot beginning at s_k - must be ready before $s_k - \Delta T_{max}$, in order to tolerate the non-preemptive transmission of the longest possible message.

In order to guarantee the collective timeliness of hard real-time communication, the following requirements must be met:

(R1) for each hard real-time message, an exclusive time-slot is reserved, which ends before the transmission deadline of the message,

(R2) the reserved time-slot of each message is as long as the worst-case transmission time of the message, including all overheads,

(R3) the reserved time slots of hard real-time messages do not overlap,

(R4) every hard real-time message is ready for transmission at least ΔT_{max} before the beginning of its reserved time slot. This means that the laxity of every hard real-time message at its ready-time is large enough to allow the longest possible message of the system to be transmitted first.

(R5) the priority of a hard real-time message depends on its laxity (Fig. 4-2)

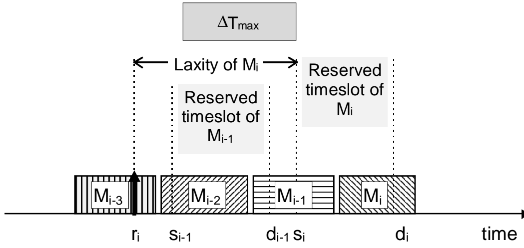


Fig. 4-4: A hard real-time message missing its deadline

Given these assumptions, every hard real-time message will be transmitted timely under fault-free conditions. Assume (Fig. 4-4) that M_i is the first hard real-time message after the system startup, which cannot start at the beginning of its reserved time slot s_i, hence missing its transmission deadline d_i. Since M_i is ready at s_i - ΔT_{max}, it participates in at least one arbitration process before s_i. If M_i does not start before s_i, then another message M_{i-1} wins the arbitration against M_i, and following conditions are true:

(C1) M_{i-1} is a hard real-time message,

(C2) the reserved time slot of M_{i-1} begins at s_{i-1}, where $s_{i-1} < s_i$, because of R5, and

(C3) M_{i-1} is not completely transmitted until s_i.

From R3 and C2 we conclude that the transmission deadline of M_{i-1} lies before s_i, i.e. $d_{i-1} < s_i$. Due to C3, M_{i-1} misses its deadline d_{i-1}, because it is completed after s_i, and $d_{i-1} < s_i$. This is a contradiction to the assumption that M_i is the first hard real-time message after system startup, which misses its deadline.

In order to guarantee timely hard real-time message transfer in the presence of faults, redundancy must be provided. Space redundancy would require multiple CAN busses. If we apply time redundancy to tolerate a single communication failure, we have to schedule two subsequent transmissions plus the fault handling mechanism of the CAN bus, which consumes bounded time [[29]]. This strategy is also used in other real-time communication protocols, e.g. the Time-Triggered Protocol [[19]]. However, in contrast to statically planned communication, we can use the redundant time slot for other messages if the first transmission was successful. In the case that a hard real-time message is not transmitted before its deadline, it will be discarded, and the sending object is notified.

For the soft real-time communication, our mechanism does not guarantee a deadline. This is because it is always possible to dynamically schedule additional hard real-time communication activities. However, optimal utilization of resources is approached to meet soft real-time deadlines, according to EDF which is known to be optimal.

5. Conclusion and Future Research

The paper focuses on the problem of real-time object invocation in a distributed object-oriented real-time system connected by the Controller Area Network (CAN). considering a simple object model, two issues were discussed in detail. Firstly, we introduced a naming approach for the remote method invocations based on CAN-messages, which supports object invocation by multicasting, and hardware message filtering. Secondly, the problem of guaranteeing timely message delivery was discussed. An EDF-like scheduling mechanism for CAN bus was introduced, which expresses transmission deadlines by dynamic priorities. Based on this mechanism, an approach to schedule hard real-time communication under certain fault assumption was given.

Our approach is especially tailored for systems using the CAN bus. It exploits special features of CAN, namely the priority-based CSMA/CA medium access protocol, and the consistent view of the message status by all operational receivers. In CAN, this feature is efficiently realized on a low network level. Logically, the CAN bus acts as a global dispatcher, dispatching the message with the highest priority to the network. This feature is unique to the CANbus. To extend our approach to other communication networks, it is necessary to define a network abstraction which provides a similar functionality which, in general, is difficult to achieve.

Currently, we are working in two main directions:

1. An internet gateway for the embedded system. Here our goal is to visualize and control the embedded system over the Internet. An animated VRML-model of the embedded system is connected to the real system and a JAVA application which models and simulates the objects of the embedded system. The simulation is a front end for the embedded system taking control input from the local workstation and real data from the embedded system, e.g. the current position of the servos. The real-world data is fed to the visualization to give an indication of the current status. The control signals are transferred to the real system and to its visualization. Clearly, all internet communication is considered to be non-real-time. But this may be the communication paradigm for future embedded systems working in distant environments only reachable through a narrow, disturbed channel. Autonomy of the system becomes inevitable.

2. Another field of our future research is the development of robust communication protocols for wireless LAN to support distributed embedded real-time systems composed of physically decoupled objects.

References

[1] K. Arnolds and J. Gosling: „The JAVA Programming Language“, *Addison Wesley, Reading, MA, 1996.*

[2] N. C. Audsley, A. Burns, M. F. Richardson, A. J. Wellings: „Hard Real-Time Scheduling: The Deadline Monotonic Approach“, *Proceedings of 8th IEEE Workshop on Real-Time Operating Systems and Software, May 1991.*

[3] Birman, K.P. and R. van Renesse: „Reliable distributed computing with Isis tool kit“, *IEEE Computer Society Press, 1994, Las Alamitos, USA.*

[4] J.M. Chang and N.F. Maxemchuk: „Reliable broadcast protocols“, *ACM Trans. on Computer Systems, 2(3), Aug. 1984, pp. 251-273.*

[5] S-C. Cheng, J. A. Stankovic, K. Ramamritham: „Scheduling Algorithms for Hard Real-Time Systems – A Brief Survey“, *IEEE Tutorial on Hard Real-Time Systems, J. A. Stankovic and K. Ramamritham, ed., IEEE Computer Society Press 1988.*

[6] CiA Draft Standard 301 version 3.0, „CANopen Communication Profile for Industrial Systems“.

[7] DeviceNet Specification 2.0 Vol. 1, *Published by ODVA, 8222 Wiles Road - Suite 287 - Coral Springs, FL 33067 USA.*

[8] L.B. Fredriksson: „A CAN Kingdom (Rev. 3.01)“, *Published by KVASER AB, Box 4076, S-51104 Kinnahult, Sweden, 1996.*

[9] M. Gergeleit, J. Kaiser, H. Streich: „DIRECT: Towards a Distributed Object-Oriented Real-Time Control System“, *Workshop on Concurrent Object-based Systems, Oct. 1994.*

[10] M. Gergeleit, J. Kaiser, H. Streich: „Checking Timing Constraints in distributed Object-Oriented Programs“, *ACM OOPS Messenger, Special Issue on Object-Oriented Real-Time Systems 7(1), Jan. 1996.*

[11] A. Gheith and K. Schwan: „CHAOS^{arc} - Kernel Support for Multi-Weight Objects, Invocations, and Atomicity in Real-Time Applications“, *ACM Transactions on Computer Systems 11(1):33-72, Feb. 1993.*

[12] E.D. Jensen and J. Northcutt: „Alpha: A non-proprietary OS for large, complex, distributed real-time systems“, 2nd IEEE Workshop on Experimental Distributed Systems, Oct. 1990.

[13] W. Jia, J. Kaiser, E. Nett: „RMP: Fault-Tolerant Group Communication“, *IEEE Micro 16(2):59-67, Apr. 1996.*

[14] J. Kaiser, M. A. Livani, W. Jia: “Membership and Order in a CAN-Bus Based Real-Time Communication System”, *Proc. of Int’l Workshop Advance Parallel Processing Technology*, Sep. 1997.

[15] J. Kaiser and M. A. Livani: „Consistent Message Delivery in CAN Bus under Real-Time Constraints“, *submitted to FTCS’28, Munich, Germany, June 1998.*

[16] K.H. Kim: „Object Structures for Real-Time Systems and Simulators“, *IEEE Computer, Aug. 1997, pp. 62*-70.

[17] K.H. Kim and H. Kopetz: „A Real-Time Object Model RTO.k and an Experimental Investigation of Its Potentials“, *Proc. IEEE Int’l Computer Software & Applications Conf., Nov. 1994, pp. 392-402.*

[18] H. Kopetz and W. Merker: „The Architecture of MARS“, *IEEE Proceedings of the 15th Fault Tolerant Computing Systems Symposium, 1985.*

[19] H. Kopetz and G. Grünsteidl: „TTP - A Time-Triggered Protocol for Fault-Tolerant Real-Time Systems“, *Res. Report 12/92, Inst. f. Techn. Informatik, Technical University of Vienna, 1992.*

[20] H. Kopetz and W. Ochsenreiter: „Clock Synchronization in Distributed Real-Time Systrems“, *IEEE Transactions on Computers, C-36(8):933-940, Aug. 1987.*

[21] H. Kopetz et al.: „The Design of Large Real-Time Systems: The TimeTriggered Approach“, *Res. Report 14/95, Institut für Technische Informatik, Technical University of Vienna, 1995.*

[22] S. Levi and A.K. Agrawala: „Real Time System Design“, *McGraw Hill, 1990.*

[23] M.A. Livani and J. Kaiser: „EDF Consensus on CAN-Bus Access in Dynamic Real-Time Systems“, *Technical Report No. 97-17, University of Ulm, 1997.*

[24] S. Maffeis: „iBus - The JAVA Intranet Software Bus“, http://*www.olsen.ch/export/user/maffeis/publications.*html.

[25] Maruti 3, Design Overview 1st Edition, *System Design and Analysis Group, Dept. Of Comp. Science, University of Maryland, 1995.*

[26] K. Mori: „Autonomous Decentralized Systems: Concept, Data Field Architecture, and Future Trends“, *Proc. International Symposium on Autonomous Decentralized Systems (ISADS 93), Mar. 1993.*

[27] E. Nett, M. Gergeleit, M. Mock: „An Adaptive Approach to Object-Oriented Real-Time Computing“, *Proceedings of ISORC’98, Kyoto, Apr. 1998.*

[28] ROBERT BOSCH GmbH: „CAN Specification Version 2.0“, *Sep. 1991.*

[29] J. Rufino and P. Veríssimo: „A Study on the Inaccessibility Characteristics of the Controller Area Network“, *2nd International CAN Conference 95, Oct. 1995.*

[30] J.A. Stankovic and K. Ramamritham: „The Spring Kernel: A New Paradigm for Real-Time Operating Systems“, *ACM Operating Systems Review 23(3), July 1989.*

[31] K. Tindell and A. Burns, „Guaranteed Message Latencies for Distributed Safety-Critical Hard Real-Time Control Networks“, *Report YCS229, Department of Computer Science, University of York, May 1994.*

[32] P. Veríssimo, L. Rodrigues, J. Rufino: „The Atomic Multicast protocol (Amp)“, in D. Powell ed., Delta-4 - A Generic Architecture for Dependable Distributed Computing, *ESPRIT Research Reports, pages 267-294. Springer Verlag, November 1991.*

[33] K. M. Zuberi and K. G. Shin, „Non-Preemptive Scheduling of messages on Controller Area Network for Real-Time Control Applications“, *Technical Report, University of Michigan, 1995.*