Implementing the Real-Time Publisher/Subscriber Model on the Controller Area Network (CAN)

J. Kaiser
University of Ulm
kaiser@informatik.uni-ulm.de

M. Mock
GMD - German National Research Center for Information Technology
mock@gmd.de

Abstract

Designing distributed real-time systems as being composed of communicating objects offers many advantages with respect to modularity and extensibility of these systems. However, distributed real-time applications exhibit communication patterns that significantly differ from the traditional object invocation style. The publisher/subscriber model for inter-object communication matches well with these patterns. Any implementation of that model must address the problems of binding subscribers to publishers, of routing and filtering of messages, as well as reliability, efficiency and latency of message delivery. In the context of real-time applications, all these issues must be subject to a rigid inspection with respect to meeting real-time requirements. We argue that for embedded control systems built around smart microcontroller-powered devices these requirements can only be met when exploiting the properties of the underlying network. The CAN-Bus (CAN: Controller Area Network) which is an emerging standard in the field of real-time embedded systems is particularly suited to implement a publisher/subscriber model of communication. In this paper, we present an implementation of the real-time publisher/subscriber model that exploits the underlying facilities of the CAN-Bus. In particular, we introduce a novel addressing scheme for publisher/subscriber communication that makes efficient use of the CAN-Bus addressing method. We provide a detailed design and implementation details along with some preliminary performance estimations.

Keywords: Real-Time Communication Systems, Publisher/Subsrciber Model, Tag-based addressing, CAN-Bus

1 Introduction

Distributed systems, composed from a network of microcontrollers connected via a field-bus network become increasingly popular in process control. Intelligent sensors, actuators and distributed control structures replace the centralized computer. A distributed real-time architecture modeling smart sensors, actuators and control nodes as communicating objects supports in many ways the demands of process control with respect to extensibility, reliability and cost effectiveness. This leads to a modular system architecture in which smart autonomous objects cooperate to control a physical process. There are, however, some properties, which deviate from the conventional object-oriented model of computation. In this paper we will focus on the issues of inter-object communication.

The conventional way of object interaction centers around synchronous object invocation. The model maintains the semantic of a procedure call in which a specific service is requested from another object by transferring control and exchanging parameters via shared memory rather than by explicit communication. Even when extending object-orientation to a distributed environment, this model of object invocation is preserved. Well known examples are the CORBA RPC and Java RMI. In summary, it is a synchronous form of point-to-point coordinating two well-known (by address) objects. In contrast, control applications often require asynchronous coordination (the term “asynchronous“ is used here to characterize an event driven style of communication which does not rely on the (blocking) request-response paradigm described above) between sets of anonymous objects. The reasons for this are the following:

1. In many cases, communication patterns are not one-to-one. A typical situation is that the information gained from a sensor can be used and analyzed in more than one controller, e.g., the output of a vision system on a mobile robot is interesting for reactive collision avoidance implemented on a small microcontroller as well as for long term navigation strategies implemented on a more powerful device. Another typical example is the situation in which control commands issued from a controller address a number of identical actuators; e.g., all motors have to stop in case of emergency.
2. Communication is often anonymous. Consider again the example of stopping a set of motors. When issuing a stop command, it is not of interest to address a specific motor, rather it must be ensured that all relevant motors receive the command. Similarly, when reacting to a stop command, it is not of interest which controller has issued that command. On a more abstract level, a sensor object triggered by the progression of time or the occurrence of an event spontaneously generates the respective information and distributes it to the system. Thus, it can be considered as a producer. The corresponding consumer objects have mechanisms to determine whether this information is useful for them. This interaction leads to a model of anonymous communication in which the producer does not know which consumers will use its information and, vice versa, the consumers only know which information they need independently from which source they receive it. Furthermore, anonymous communication supports the extensibility and the reliability of the system because objects can be added or be replaced easily without changing address information maintained in the other objects.
3. Communication is asynchronous because control systems have to react to external events. These external events are recognized at the sensor interface of an embedded system and lead to internal activities. This is best captured in a generative, event-based communication model. In addition, control applications are highly cooperative. The individual nodes are, from an application point of view, functionally and temporally tightly coupled to perform a complex control task. However, because of extensibility and reliability reasons it is highly desirable to preserve the control autonomy of the individual nodes. Therefore coordination between the cooperating objects should be achieved via asynchronous communication rather than by explicit control transfer.

As already argued in other places, the publisher/subscriber model of inter-object communication meets these requirements [1, 2, 3]. The publisher/subscriber model supports an asynchronous style of anonymous many-to-many communication in contrast to the synchronous style of object invocation. A consumer *subscribes* to a certain event type/subject/channel rather than to a specific producer. A producer *publishes* instances of this type of information. Published information is forwarded eventually to all subscribers, either immediately when being published *(push)* or on demand when a subscriber asks for updates *(pull)*. Popular implementations of this model are the CORBA event service [1] or even web channels [4] Since the publisher/subscriber model is extremely attractive for structuring object-oriented control applications, attempts already have been undertaken to implement the model in the real-time domain [5, 6]. Roughly speaking, these implementations suffer from the uncertainties incurred when mapping the publisher/subscriber model to TCP/IP based communication in a general purpose network (a more detailed discussion of related work is given in section 2). In this paper, we present an implementation of the publisher/subscriber model that exploits the real-time properties of the underlying network, namely the CAN-Bus which is very popular in distributed control applications. We show how the issues of binding subscribers to publishers, of routing and filtering of messages, as well as reliability, efficiency and latency of message delivery are implemented in order to meeting real-time requirements. In particular, we introduce a novel addressing scheme for publisher/subscriber communication that makes efficient use of the CAN-Bus addressing method.

The paper is organized as follows. The next chapter investigates the implementation issues mentioned above, presents the principles of our approach and compares it with the related work. The basic features of the CAN-Bus will be introduced in chapter 3. They are a prerequisite to the understanding of chapter 4 where our addressing scheme and the components of the communication system are described. Chapter 5 gives some preliminary evaluation results and concludes the paper.

2 Design principles

Implementing the publisher/subscriber model requires to map the abstractions of that model (publisher, subscriber, information type, information instance) to the elements provided by the technical infrastructure of the system such as objects, messages and addresses. More precisely, we can identify publishers and subscribers with objects, and information instances with messages that are sent to certain addresses. The information type can be mapped to the address or the content of the message (or a combination of both).

Now, the *routing* problem consists of making sure that a message is sent to all potential subscribers, i.e., each subscriber has a chance to get all messages it is interested in. There are three different approaches to solve the routing problem:

1) send a number of point-to-point messages,
2) send a multicast message,
3) send a broadcast message.

Obviously, alternative 1) is the worst when trying to achieve real-time requirements. This is because implementing a logical one-to-many communication by a number of point-to-point messages induces a higher network load compared to a single broadcast message. This higher load consumes bandwidth resources and makes the delay of message delivery less predictable. It is, however, the only possible choice when the underlying network is assumed to provide point-to-point message delivery only. This is case for the work reported in [5, 6]. In [5], the routing problem is tackled by maintaining a replicated subscription registry at the publishers, i.e., each publisher knows to which subscribers it has to send a message. Real-time properties are supported in so far as the actual sending of messages is given priority to maintaining that replicated information base. However, the principle problem of achieving predictable message delays is not overcome. The solution described in [6] uses an intermediate object (the channel object) which represents the information type. This is comparable to the indirect port based addressing scheme found for instance in the Chorus and in the Mach operating systems. All publishers send their messages to the channel object. All subscribers register at the channel object. Thus, the channel object acts as router between the publishers and the subscribers. [6] describes how some real-time properties such as handling of prioritized message and message correlation can be integrated in the channel object. However, the price to be paid is the extra level of indirection and the increase in the total number of messages needed.

The use of the multicast alternative is advocated for in [2]. Real-time issues are not addressed. Although being attractive for implementing the publisher/subscriber model, the choice of the appropriate multicast protocol is not clear. In general, multicast communication makes group membership explicit and joining or leaving a group is an activity, which creates group wide awareness, and activity. In the publisher/subscriber model, the knowledge about who is in a group is not necessary and subscription is (conceptually) a local activity.

Finally, the broadcast alternative clearly is the premium choice for solving the routing problem given that the underlying network is a physical broadcast medium. This is a main result from [3]. In there, the implementation is based on the Ethernet. Real-time requirements again are not addressed and, in general, are hard to achieve on the Ethernet due to the way collisions are handled. In contrast to Ethernet, the CAN-Bus is a broadcast medium that allows use physical broadcast while its non-destructive collision handling scheme allows support of real-time properties (see section 3 for more details).

Now, we consider the *filtering* problem, which consists in making sure that a subscriber does not receive more messages than it has subscribed for. This is the other side of the routing problem, i.e., if the publisher already knows in advance which subscribers must receive a particular message and if it sends the message only to those subscribers, no filtering must take place. But, as argued above, the most handy approach (and the one that is best suited to achieve real-time properties) to solve the routing problem is the broadcast approach. This induces that, at some level, all receivers receive all messages and that filtering must take place to implement the publisher/subscriber model. The most expressive (and least real-time capable) approach to filtering is introduced in [7]. In this approach, which is also found in [8, 9], subscribers define via predicates over the message contents which messages they are interested in. This is denoted as "content based addressing". In this scheme, the complete contents of each message has to be evaluated in order to filter the relevant messages out. This induces a non-predictable overhead on each node for each message broadcasted in the system. As shown in [3], the overhead can be alleviated by identifying messages by a simple tag (the subject of the message) instead of arbitrary predicates in order to ease the filtering process. Each message is tagged with its subject and filters are specified in terms of a mask for a particular subject rather than using the entire contents of a message. In [3], this technique is termed "subject based addressing". However, it still holds that the contents of each message has to be evaluated by each receiver, only the complexity of the evaluation process in reduced.

Our approach to make the subject-based addressing scheme efficient and usable in an embedded real-time system is to put the subject in the address of the message rather than in the message contents. For simple devices it is substantial that filtering is done at the network controller level because the respective microcontroller would not even be able just to examine every message on the net. By putting the subject in the address we solve the filtering problem already on the level of the communication controller. On the CAN-Bus, all messages are physically broadcasted. However, messages are tagged with addresses that are used for the bus arbitration process. Addresses are also related to the contents of the message rather than to a receiver address. CAN controllers can be configured to selectively receive messages depending on the contents-related address. All other messages are discarded already in the network controller without interrupting the application processes. A masking mechanism is available to realize "wildcarding" and recognize messages with partially identical addresses as members of a subject group. For more details and for a further explanation of our solution of the filtering problem the reader is referred to the description of the Event Channel Handlers in section 4.2.

By tackling the filtering problem by an efficient use of the underlying addressing scheme, we re-introduce a *binding* problem into the implementation of the publisher/subscriber model. The binding problem is that the system has to find out which address it has to use when sending a particular message. In [3], the binding problem is completely eliminated because all messages are sent to the same (broadcast) address. Systems which strictly use the contents of an event to mediate it to the application object, do not need any binding of an event to a name or an address tag. The price which has to be paid for this convenience, however, is as explained above, the costly implementation of the filtering that has to inspect each message. For real-time embedded systems this will be prohibitively expensive. In [6], binding takes place when connecting to the event channel, in [5] binding is implicit in the implementation of the registration functions.

In supporting an object-oriented, modular design approach, we identify system components (intelligent sensors, intelligent actuators) by their external interfaces while allowing their implementation to be hidden. In particular, we do not require components to be re-compiled when being put together to build a system. We even allow components to be added dynamically. Since the subjects being published or subscribed by a component are part of its interface, this subject must be mapped dynamically to an address at run-time. The straightforward approach to directly use the subject as address is not feasible on the CAN-Bus because, firstly, the address is also used for the bus arbitration process and relates to message priorities, and secondly, the address is relatively short compared to global name space of all possible subjects, and thirdly, the CAN-Bus forbids messages with identical addresses to appear on the CAN-Bus. The latter could not be avoided when using subjects directly as addresses because there might several components that publish on the same subject. Thus, our implementation includes a dynamic binding mechanism that binds subjects to addresses at run-time. This mechanism, denoted as Event Channel Broker in the following, supports late binding and local address resolution. It is described in more detail in section 4.3.

Finally, we briefly sketch how the *reliability*, *efficiency* and *latency* problems are solved in our implementation. The first two problems are already addressed by the underlying CAN-Bus. Achieving reliability means to make sure that messages are not lost on the communication medium. As already argued in [10], the use of explicit acknowledgement schemes for detecting message losses is not suitable for real-time systems. This is the more true when considering one-to-many communication as in the publisher/subscriber model. The CAN-Bus, however, provides a sophisticated fault detection mechanism to achieve a consistent view about the status of a message at the end of every individual message transfer. The efficiency problem relates to the effective bandwidth that is achieved on the medium, especially when considering high load situations. Good efficiency in general is hard to achieve especially when broadcasting is used to solve the routing problem. This might overwhelm the medium with unnecessary messages. In particular, it is well known pure CSMA suffers from a low efficiency in high load situations due to the increase of collisions and resends [11]. The CAN-Bus, however, provides a non-destructive collision detection scheme and thus does not suffer from those efficiency penalties. By the latency problem we mean that a predictable message delay must be guaranteed in a real-time system, i.e., each message has a deadline before which it must be sent. This timeliness in message delivery is not achieved by the basic CAN-Bus properties that only provide priority-based message dispatching. For guaranteeing timely delivery, we use a global network scheduling approach [12] that can be implemented efficiently on the CAN-Bus by using a dynamic priority scheme [13].

Due to its specific properties, the CAN-bus is an ideal candidate to support the publisher/subscriber model of communication. Therefore, we will now briefly introduce its basic properties.

3 Basic CAN features

CAN (ISO 11898 an 11519-1) is a broadcast CSMA-network targeted to operate in an automotive or industrial automation environment with speeds of up to 1 Mbit/sec, exchanging small real-time control messages. The CAN-specification [14] developed by BOSCH covers the functionality of layers 1 (physical layer) and 2 (data link layer) of the ISO/OSI protocol stack. CAN is a variant of a polled bus [15]. A polled bus relies on three properties.

P1 every bit of a message will propagate to all connected nodes before the next bit will be sent, thus enabling all nodes to see the same bit value during a certain time window.

P2 the sender monitors the bus at the same time it transmits. For every bit, a sender can check if the bus carries the signal level which was transmitted.

P3 there are dominant and recessive signal levels. A single dominant signal level overrides any number of recessive signal levels.

If two stations start transmission at the same time, a collision will occur. Different from other CSMA-networks, however, collisions always lead to a well-defined signal level on the bus because of the property P1 and P3. In most implementations, the CAN-bus behaves like a wired AND circuit for all bit values sent at the same time. This basic feature of the CAN-bus is exploited for:

- Efficient use of available network bandwidth by providing a non-destructive priority based message dispatching.
- Immediate error detection, signaling and automatic retransmission of messages.

The priority-based non-destructive arbitration scheme assures that a collision does not destroy the messages on the bus, but the message with the highest priority will be transferred without further delay. If a node during the arbitration process sends out a recessive level but monitors a dominant level on the bus, it knows that a message with higher priority is competing for the bus. The node then will switch to a receiving mode. Finally, the node with the lowest message ID will win the arbitration process and send the data. It can easily be seen that by this mechanism, the CAN-bus serves as a priority based global message dispatcher. The non-destructive arbitration scheme overcomes the drawback of general CSMA networks of low predictability and lost bandwidth because of collisions under high load conditions.

The second important feature of CAN is that it provides mechanisms to achieve a consistent view about the status of a message at the end of every individual message transfer. Every message will be accepted or rejected by all participants. If one of the participants (sender or receiver) detects an error locally it will invalidate the ongoing message transfer by sending out a string of dominant bits. This will be detected by the other participants including the sender. The receivers will, as a consequence, discard the current message from their local in-queues. The sender will automatically retransmit the message. Thus, relating a corrupted message with its source and retransmitting it will be done at the controller level. For the host processor the automatic retransmission procedure is transparent which substantially lowers its protocol overhead. There is a very low probability that transmission errors remain undetected [14, 16].

To summarize, CAN supports routing of messages by an efficient broadcasting of messages and reliability by a broad spectrum of error detecting and recovery mechanisms.

4 Implementing a real-time publisher/subscriber protocol

CAN alone does not yet enforce temporal guarantees for real-time communication or a particular model of communication. Moreover, there are some rare failure situations which cannot be handled by the reliability mechanisms of CAN [17, 18]. We need to build additional protocol layers which realize the properties necessary for a real-time publisher/subscriber model of communication. Important, however, is to exploit the CAN basic features as far as possible. Particularly, we have to add:

1. A local run-time component (ECH, see Fig. 1) which performs the filtering of messages based on their event tags. The ECH provides an event channel interface to the application object and exploits the hardware filtering mechanisms of messages on receiver sites efficiently.
2. A binding service (the ECB, see Fig. 1) which supports our enhancement of assigning tags as event identifiers to the messages. This mechanism must guarantee the uniqueness of event tags.

As described above, the CAN identifier and the special treatment of this ID by the CAN controller hardware are the key for priority based message dispatching and for routing and filtering. The CAN standard specifies two different but compatible formats of the CAN-ID. A short 11-Bit form and an extended format (29 bit). However, CAN does not fix any specific use of the identifier. Therefore, higher level protocols are free to structure the identifier according to their needs. There are a number of application level protocols which all use different interpretations of the ID [19, 20, 21]. Their major drawbacks are that they use a short (11 bit) form of CAN identifiers (The CAN standard defines both, 11 Bit and 29 Bit Ids). They argue that this will reduce the overhead for a message. Because messages cannot be preempted and CAN is designed to achieve a short latency for high priority messages, the maximum message length is 154 bits with 64 Bits (8 Bytes) of payload. The overhead is thus around 140% for the longest message. This will be reduced to around 110% when using short IDs.

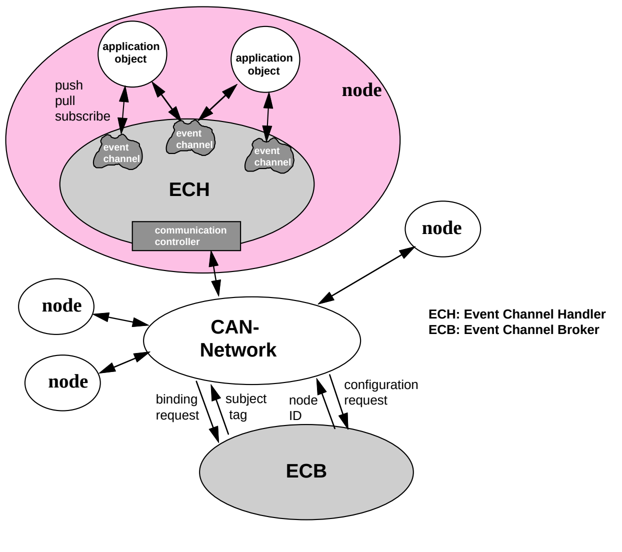

Fig. 1 Overall communication system's architecture

However, these schemes make it almost impossible to implement useful event tags and support a general publisher/subscriber model. Moreover, if the identifier carries useful information, like temporal constraints or subject identifiers, which have to be communicated anyway, more payload is available for additional data. In the protocols mentioned above the payload of the messages is severely reduced by carrying system information.

4.1 The Event Channel Handlers

The Event Channel Handlers (ECH) constitute the local run-time support which provides all functions necessary to manage the event channels on a node. This support ranges from the application objects' interface to the low-level mechanism to setup and control the hardware filters for the reception of relevant events.

The scheduling of messages, routing and filtering expose a certain structure to the extended 29-Bit CAN identifier. The structure of the CAN identifier for our protocol is depicted in Fig. 2. As mentioned in the previous chapter, all 29 bits of the identifier are used in the arbitration process to establish a priority order between the messages ready to be sent. Thus, CAN alone guarantees the timeliness of the highest priority message only. To overcome this drawback, rate monotonic and deadline monotonic analysis is applied to assign priorities in a way that all messages will be properly accommodated [22]. Rate monotonic analysis allows off-line validation of message priorities if the bus has an anticipated load and messages arrive periodically. Thus, it is perfectly suited for static systems which only handle hard-real-time messages. However, we want to support Quality of Service for a more dynamic type of system in which messages may not arrive periodically and hard real-time and soft real-time messages may coexist. We use a dynamic LLF scheme, an idea which first has been proposed for EDF by Zuberi and Shin [23]. However, in the form proposed in [23] it is only suited for a very low number of nodes ($\leq$ 3) and also cannot guarantee the timely delivery of hard real-time messages during transient overload situations. We propose a scheme which is based on reserved time slots for hard real-time messages similar to TTP. The priority scheme of the CAN-Bus is exploited to guarantee that hard real-time messages are properly sent even if they have to compete with soft real-time and non- –real-time traffic. Different from TTP or TDMA schemes, slots not needed by hard real-time messages can be used by lower urgency messages. A discussion of the scheme is beyond the scope of this paper. It is described in detail in [13, 24].

Our scheme uses the 8 most significant bits of the ID to specify the dynamic priority. The remaining bits of the ID are used to carry information which characterizes events. This is referred as static priority field (see Fig. 2).

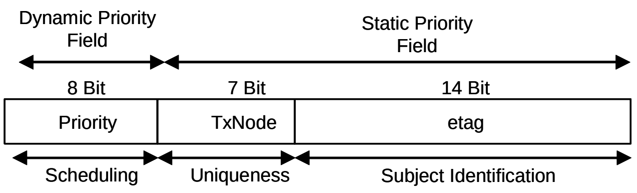

Fig. 2 Identifier structure

The static priority field contains two sections. The first section contains the node identifier. We assume a single local CAN network which may have up to 128 nodes. The number of nodes in a CAN network is not really restricted by the protocol. Rather this is a physical restriction resulting from popular implementations of bus drivers. According to the ISO-DIS 11898 Standard, the physical interface must accommodate 2- 30 nodes. This node ID is needed to guarantee uniqueness of the entire CAN-ID, i.e. that no two identical CAN-IDs compete for the bus access (in this case the protocol would be unable to resolve the arbitration conflict). The partitioning of the CAN-ID has to solve the trade-off between the number of network nodes and the number of events which can be identified by the respective tag in the ID. Because there is only a restricted number of bits, a static assignment of the respective fields in the ID always leads to a restricted extensibility of the system. Therefore nodes and events have some unique long names which are mapped to CAN-IDs when a node is connected to the network. The event channel broker (ECB) and appropriate protocols (described in section 4.3) are implemented for this purpose. Before a new node is allowed to send any regular message, it issues a configuration request to the ECB and obtains a unique 7-bit node identifier. The ECB configuration protocol ensures that this message has a unique identifier which is not used during normal system operation and that two configuration requests do not interfere. Similarly the binding of unique event names and the respective 14-bit event tags has to be performed before an object can use an event channel.

In a system which comprises nodes with limited processing performance it is essential that the mechanism of filtering irrelevant events is embedded in the controller and works without host assistance. The filtering mechanism in CAN applies to the message ID. The event tag as part of the address is a prerequisite to exploit the CAN-ID for event filtering. This low level filter mechanism is now briefly sketched.

Every CAN controller comprises one or more receive registers RxReg. This register holds a 29-bit identifier (TID: template ID). During a message transfer, a message ID on the CAN-bus (CID: current ID) is compared against the identifier in the receive register(s). If a match occurs, the message is fetched from the bus and stored in a buffer. To allow the reception of a group of messages, an additional mask register is available. The bit pattern in this register contains a mask defining which bit-positions of the ID should be used to determine a match. This mechanism is similar to an associative memory which allows the masking of search keys to address a set of memory locations. For each incoming message the controller checks if: TID & MASK == CID & MASK (&: bitwise logical AND).

In our scheme, the mask can be used to select messages according to priority, transmitting node, subject tag, or a combination of them. Fig. 3 gives some examples of selective message reception. Mask 1 specifies the subscription to a specific event. Only the event with the tag "00110110110100" will be accepted by the controller. Mask 2 enables reception of a group of events from a specific node.

dynamic prio				Txnode				etag									
8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1
1	1	1	1	1	1	1	1	0	1	1	0	0	1	1	0	0	1
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	1	1	1
x	x	x	x	x	x	x	x	1	1	1	1	1	1	1	1	1	1

x: don't care

Fig. 3 Examples of selective masking of a CAN ID

The capabilities of the filter mechanism are not specified in the standard. Therefore, they are, to a certain degree, dependent on the CAN controller used. A common feature, however, is that all CAN controllers provide at least one global mask to filter events (some controllers provide additional capabilities which allow a finer grain of filtering events to reduce the load of the host processor. It is however beyond the scope of this paper to discuss these optimizations). If two messages with different identifiers should be received on a node, all differing bit positions must be masked out, i.e. these bits are not used for matching the ID to the contents of the receive register. It is obvious that the filter mechanism can best be exploited if binding is not arbitrary but considers some application dependent assignment of events tags. The ECB, responsible for the binding, can support the low level filtering mechanism by grouping of events and assigning a partly identical event tag.

The efficiency of handling incoming messages is crucial for the applicability of the concept in embedded real-time systems. On an incoming event, the ECH has to execute the following tasks:

1. Handle the controller specific issues like reading and resetting the receive register. This is normally performed in an interrupt handling mode.
2. Determine the respective event channel.
3. Determine the objects which have subscribed to the event channel.
4. Copy the event in the respective queues of the objects.
5. Notify the object of an event occurrence. This may be combined with step 4 depending on the underlying real-time executive.

We assume a controller which is compatible with the Basic CAN standard [14]. The highest possible rate of incoming messages in a 1Mbit/sec network is around 1/90 µsec [26] for messages with no data. Because Basic CAN controllers are equipped with a shadow buffer for incoming messages, the received message is not destroyed while the next message is received. Thus at the worst case, the host has to examine and forward a message every 90 µsec. However, a much lower rate is expected because firstly, only relevant events pass the filtering mechanism and secondly, analysis of requirements for hard real-time communication show, that considering reliable message transfer will reduce the message rate considerably [24, 25]. In chapter 5 we present some preliminary performance figures.

4.2 The Event Channel Broker and the Binding Protocol

The ECB handles binding and configuration requests. Binding refers to mapping subjects to addresses (event tags). Configuration addresses the problem of assigning unique node IDs and guaranteeing that no address conflicts will occur on the CAN-Bus.

As described above, the binding supports the use of tags to identify the contents of a message. This requires that the message contents is related to a message tag or subject by a binding procedure. This binding is performed in two steps. First, the contents of a message has to be related to a subject (or an event channel). The subject is represented by a long unique logical message ID. The assignment is done off-line. A publisher and a subscriber have to know the subject names when they want to participate in a communication over this specific event channel. This subject ID can also be used in higher level protocols that make use of the publisher/subscriber communication implemented by our system. The next step is to assign a short tag which is carried in the CAN ID to a specific subject. This tag is then used to allow filtering at the CAN controller level. Whereas the first step is done off-line the second step is dynamically performed at run time. Thus, a node, which wants to communicate the first time, has to resolve the logical subject names.

The configuration problem which we have to solve for the CAN-bus is to provide unique message identifiers. The same event, provided by multiple sources must have different identifiers to comply with the CAN arbitration scheme. Therefore, the static part of this ID comprises a node-ID and an event tag (see section 4.1). In our current implementation, we support 15-bit unique logical names for nodes and 32-bit logical names for events (the sizes of node names and event names are somewhat arbitrary in our prototype implementation. The size of the node name e.g. was chosen because it fits perfectly in a single configuration request message. The protocol, however, allows for arbitrary long names if multiple messages which are assigned to dedicated event names are used for a configuration requests). The assignment of a logical node name to a 7-bit node ID is a configuration issue which has to be performed when a node is connected to the network. It is a prerequisite for any communication of the node. The binding of logical event names to event tags is the second step. This step can be performed eagerly at node configuration time or lazily, when an event is actually used. Early binding may be the preferred technique to achieve predictability in a real-time system. If late binding is applied, the resolution of the event name has to be considered as an additional scheduling parameter.

Let's start with the configuration protocol. The configuration protocol envisages a bootstrap problem. This results from the fact that if two nodes would try to send messages with the same identifier but different data parts, this would lead to a conflict which could not be resolved by CAN. The two controllers would detect a transmission error, invalidate the message and automatically would start retransmission at the same time with the same messages. Therefore, we have either to assure that the configuration request to the ECB has a unique CAN-ID or the configuration request must not contain any data. In the CAL configuration protocol [19] a configuration request uses a single dedicated CAN-ID to request a unique node-ID. This request can be issued by multiple nodes at the same time, consequently, no data is allowed in this message. Thus, the configuration server does not know which node has issued the request, and hence, it has to poll all nodes in the system. Our solution to this problem avoids the time consuming polling by including the unique long node ID in the message. The protocol uses a specific configuration event (event #1) which is dedicated to a configuration request. The 15 most significant bits of the event contain the long node ID (Because a configuration request should not interfere with a (low laxity = high priority) real-time message, the two most significant bits of the logical node ID are always "11" resulting in an effective name space of 13 bits). This serves two purposes. Firstly, since node IDs are unique, two configuration requests have always different CAN-IDs. Secondly, the ECB knows which node issued the request. The ECB performs the name conversion and returns the short node ID via a second reserved event (event #2). The node which expects an answer of the ECB, sets its receive register exactly to this event. The ECB now pushes the event to the associated event channel with the logical node ID and the short node ID attached as data. The logical node ID allows the receiving node to determine whether the message is a reply to its request. From then, it can communicate using its unique short node ID.

The binding protocol works very similar. Because the node already has a unique ID, the principle of the protocol is straightforward. However, we implemented an optimization. Many local application objects may use the same event channel. It would therefore not be efficient if each of them would issue a (remote) query to the ECB. Instead, the application object always directs its request to the local ECH which caches resolved event names. Only if it is the first local request to resolve the name, the ECH has to communicate with the ECB. The protocol between the ECH and the ECB again uses specific binding events (#3 and #4) for requesting an event tag returning it to the client ECH.

Currently, we only implemented a central ECB which obviously constitutes a single point of failure. Using replication this drawback could be removed. However, it should be noted, that the ECB is only needed if a binding has to be performed, i.e. at system configuration or at the integration of new components.

5 Concluding remarks and preliminary evaluation

The focus of the paper is the efficient implementation of the publisher/subscriber model on the CAN-bus. We argue that the CAN-bus is well suited to implement such a model and that it is necessary to exploit basic network features to enable hosts with restricted processing and memory capacity to use the convenient communication paradigm. The paper addresses two questions:

1. Is, in general, the publisher/subscriber model of communication a good thing for object-oriented distributed control applications?
2. Can such a model be implemented efficiently in a system which incorporate smart devices with limited processing and memory capacity?

To answer the first question, we gave some examples of typical control scenarios in which the conventional style of synchronous object invocation is not appropriate. And in fact, the general idea of anonymous communication has gained credit in control automation environments [3, 8]. The second question which is the focus of the paper cannot be answered easily because it heavily depends on the type of system where the anonymous communication is used. We identified the characteristic requirements of anonymous communication which have to be considered and gave a sketch of how similar systems solved these issues. Of course, a positive answer of the second question is a necessary condition that the model is feasible in a certain environment.

As shown in our state-of -the-art description in section 2, existing protocols for implementing the publisher/ subscriber model use general-purpose networks which are not particularly suited to achieve efficiency in a control application. Hence, it is impossible to use these solutions in a system with 8- or 16-bit microcontrollers. Our solution for a publisher/subscriber protocol for the CAN-bus tries to exploit hardware features as far as possible to free the restricted processing capacity of the host from evaluating every message. We identified the problem of routing messages efficiently to all potential subscribers. This problem is efficiently solved by the CAN broadcast mechanism. The next crucial problem is the filtering of messages. Using a broadcast mechanism leaves the filtering task to the subscriber nodes. It has been argued that it is impossible to examine every message in the respective host processors due to performance restrictions. Instead, hardware filtering at the controller level should be exploited. This requires that the message contents is related to a message tag or subject by a binding procedure. This binding is performed in two steps. First, the contents of a message is associated with a subject and represented by a long logical message ID. The next step is to assign a short tag which is carried in the CAN ID to support hardware filtering. Whereas the first step is done off-line the second step is dynamically performed at run time. Thus, a node which wants to communicate the first time has to resolve the logical subject names of the required message channels. For the sake of efficiency (and feasibility in the controller context) this eager binding sacrifices some of the dynamic properties of the most general approach which has to evaluate the full contents of each message. However, it should be kept in mind that the subject is not related to any physical location, i.e. it is not a network address of a specific node (or a group of nodes) but uniquely identifies the message contents. Although the hardware filtering mechanism of the CAN-Bus is restricted, it particularly helps that processors with restricted capabilities (like a simple smart sensor) only subscribing for a small number of messages are not overloaded with the filtering task.

We did a first implementation of our protocol on the 16-Bit microcontroller SAB 80C167 from Siemens to demonstrate the feasibility of the approach. We were primarily interested in the most basic performance figures of individual event transfers. So far, we only tested the raw performance of the ECHs and the ECB, i.e. we did not evaluate a benchmark or an application. The conditions were set for the highest priority event; i.e. no other event was able to delay the respective message. The times measured included the times for pushing an event to a channel, transferring it to a remote site and notify the subscriber. According to our basic architecture of the embedded system which dedicates a microcontroller to every sensor or actuator, many nodes will only incorporate a single object. Therefore, we have two implementations of the ECH. Both run on the 16-Bit 80C187 embedded controller and are written in C. One is realized on the raw hardware without any operating system support and only supports a single object. The other implementation uses the real-time executive PXROS supporting multiple objects. A task is assigned to every object. For the implementation on the raw machine a complete event transfer needed 169 µsec. The time for message transmission was 95 µsec, thus the local activities on the nodes took 74 µsec. If PXROS was used, the end-to-end time, under the same conditions, was increased to 469 µsec which is mainly because of task switching and the support of multiple objects on a node which results in a more complex assignment mechanism.

We also implemented an ECB. To handle configuration request it simply assigns short node IDs in ascending order and returns it according the protocol. A configuration request to the ECB takes 0,9 ms. To resolve an event name, the ECB first has to check whether an assignment already exists. The current implementation does a simple list search of already assigned names. No grouping of IDs is considered. The protocol needs 1,8 ms adding approx. 1µsec per entry of the list.

Of course, these results are of limited value to assess the behavior of a complex application. Rather, they set the limits of the communication system. E.g. the time to propagate an event under PXROS is much higher than the time it takes to transmit two subsequent messages over CAN. This means, that a node may not be able to receive and inspect every message on the bus. This has to be considered when designing an application system. On the other hand, the implementation showed that a simple sensor or actuator using the stand-alone ECH is feasible. The (ROMmable) code size of the stand-alone ECH is around 3k. To buffer a single incoming event, 12 bytes are required, 8 bytes for the message contents and 4 Bytes to hold the CAN Identifier. Thus, it seems to be feasible that smart elements with limited processing and memory resources can be incorporated in the system using a convenient high-level communication paradigm.

6 References

[1] Object Management Group (www.omg.org), The CORBA Services, OMG Technical Document formal/98-07-05.

[2] S. Maffeis: "iBus - The Java Intranet Software Bus", Olsen&Associates, www.olsen.ch, 1997.

[3] B. Oki, M. Pfluegl, A. Seigel, D. Skeen: "The information Bus®- An Architecture for Extensible Distributed Systems", 14th ACM Symposium on Operating System Principles, Asheville, NC, Dec 1993,pp.58-68.

[4] Pointcast Inc., The Pointcast Network, www.pointcast.com .

[5] R. Rajkumar, M. Gagliardi, L Sha: " The Real-Time Publisher/Subscribe Inter-Process Communication Model for Distributed Real-Time Systems: Design and Implementation", IEEE Real-time Technology and Applications Symposium, June 1995.

[6] T.H. Harrison, D.L. Levine, D.C. Schmidt: "The Design and performance of a Real-time CORBA Event Service", Proc. of the 12th Ann. Conference on Object-oriented Programming, Systems, Languages and Applications, OOPSLA, Atlanta, USA, 1997.

[7] N. Carriero, D. Gelernter: "Linda in Context", Communications of the ACM, 32, 4, April 1989, pp 444-458.

[8] K. Mori. Autonomous decentralized Systems: Concepts, Data Field Architectures, and Future Trends, Int. Conference on Autonomous Decentralized Systems (ISADS93), 1993.

[9] Gradimir Starovic, Vinny Cahill, and Brendan Tangney: "An event based object model for distributed programming." In OOIS (Object-Oriented Information Systems) '95, London, Springer-Verlag, December 1995, pp. 72-86.

[10] H. Kopetz, Real-Time Systems, Kluwer Academic Publishers, 1997.

[11] A. Tannenbaum, Computer Networks, 3rd edition, Prentice Hall, 1996.

[12] M. Mock, E. Nett. Real-Time Communication in Autonomous Robot Systems, Int. Symposium on Autonomous Decentralized Systems, ISAD 99, Tokyo 1999 (to appear).

[13] M.A. Livani, J. Kaiser, W.J. Jia: "Scheduling Hard and Soft Real-Time Communication in the Controller Area Network (CAN)", 23rd IFAC/IFIP Workshop on Real Time Programming, Shantou, China, June 1998.

[14] ROBERT BOSCH GmbH: "CAN Specification Version 2.0", Sep. 1991.

[15] C.M. Krishna, K.G. Shin: "Real-Time Systems", McGraw-Hill, 1997.

[16] J. Unruh, H.J. Mathony, K.H. Kaiser: "Error Detection Analysis of Automotive Communication Protocols". SAE Int. Congrss, No. 900699, Detroit, USA, 1990.

[17] M.A. Livani, J. Kaiser : "Predictable Atomic Multicast in a Controller Area Network (CAN)", Tech. Report, University of Ulm, No. 98-15, December 1998.

[18] J. Rufino, P. Verissimo, G. Arroz, C. Almeida, L Rodrigues: "Fault-Tolerant Broadcasts in CAN", Proc. of FTCS-28, München, Germany, June 23-25, 1998, pp. 150-159.

[19] CiA Draft Standards 201-207: "CAN Application Layer (CAL) for Industrial Applications", May 1993.

[20] D. Noonen, S. Siegel, P. Malony:" DeviceNet Application Protocol", Proc.1st International CAN Conference, CiA, Erlangen, Germany, 1994.

[21] Smart Distributed Systems, Application Layer Protocol Version 2, Honeywell Inc, Micro Switch Specification GS 052 103 Issue 3, USA, 1996.

[22] K. Tindell, A. Burns: "Guaranteed Message latencies for Distributed Safety-Critical Hard Real Time Control Networks", Tech. Report YCS229, Dept. of Comp. Science, University of York, May 1994.

[23] 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.

[24] M.A. Livani, J. Kaiser: "EDF Consensus on CAN Bus Access for Dynamic Real-Time Applications", in José Rolim (Ed.): Lecture Notes in Computer Science, Vol. 1388, pp.1088-1097, Springer, 1998.

[25] J. Rufino, P Verissimo: "A study on the Inaccessibility Characteristics of the Controller Area Network", Proc. of the 2nd International CAN Conference 95, London, Oct. 95. CiA.

[26] M.A. Livani, J. Kaiser: " Evaluation of a Hybrid Real-time Bus Scheduling Mechanism fro CAN", Seventh Int. Workshop on Parallel and Distributed Real-Time Systems (WPDRTS 1999), San Juan, Puerto Rico, April 12 - 16, 1999.