COSMIC: A middleware for event-based interaction on CAN

Jörg Kaiser, Carlos Mitidieri, Cristiano Brudna
Dept. of Computer Structures
University of Ulm
James-Franck-Ring
89069 Ulm, Germany
Kaiser@informatik.uni-ulm.de
Carlos.Mitidieri@informatik.uni-ulm.de
Cristiano.Brudna@informatik.uni-ulm.de

Carlos Eduardo Pereira
Dept. of Electrical Engineering
Federal University of Rio Grande do Sul
Av. Oswaldo Aranha, 103
90035-190 Porto Alegre, Brazil
CPereira@eletro.ufrgs.br

***Abstract*— Distributed factory automation systems benefit from field-busses which, in general, provide support for reliable and timely communication. These field-busses, however, provide rather low level communication objects and their features regarding quality aspects of communication are difficult to assess and use for applications. The paper presents COSMIC (Cooperating Smart devices), a middleware architecture which allows to use communication abstractions appropriate for high level applications based on distributed cooperating objects. The middleware supports an event-based communication model which enables spontaneous dissemination of events, maintains control autonomy of objects and allows to specify different real-time and reliability requirements on the application level. The basic abstractions presented at the middleware layer are events and event channels. As an example, the paper describes how these abstractions are mapped to the CAN-Bus which constitutes a widely used field-bus standard.**

I. Introduction

Advanced industrial automation systems increasingly rely on distributed computer architectures. The nodes in these architectures may range from deeply embedded processors to workstations which are interconnected by a large variety of field-bus technologies. The input and output of such a system is performed by intelligent sensors and actuators. These smart components comprise special purpose hardware (sometimes also mechanical elements) for signal reception and conditioning together with a computational element and a network interface. The smart nodes therefore constitute autonomous entities which allow to capture sensor data at the real-world interface, transform it to a machine readable form and spontaneously disseminate it as a standardized message to the distributed system. Because the message already contains preprocessed, application related data, other smart components can directly use it to perform some control action. On a higher level of abstraction, the system can be viewed as cooperating active objects which interact with each other via the communication system.

To ease the interaction of cooperating objects, the communication system also should provide an adequate level of abstraction and the application designer should not be forced to deal with a low level primitive message passing system provided at the raw field-bus level. The communication model should also reflect the requirements of a control system composed from autonomous active components. The model firstly should allow the spontaneous generation and dissemination of events detected at the sensor interface or being triggered by internal state changes. Secondly, many-to-many communication relations are needed to efficiently disseminate sensor and control information. Thirdly, the data transfer should not be coupled with a transfer of control as it is the general case in client server interactions. These properties rule out the remote invocation mechanism usually provided for inter object communication [1], [2]. Additionally, it is desirable to specify and control the quality of event dissemination also on a high level of abstraction.

The paper describes the architecture of COSMIC (Cooperating Smart devices), a middleware which presents an event-based communication model. Events in COSMIC are disseminated in a publisher/subscriber style of interaction. Publisher/subscriber protocols are well known to support spontaneous, many-to-many communication relations and reflect autonomy of communicating entities [3], [4], [5], [6]. Therefore they meet the requirements established above.

COSMIC extends the publisher/subscriber scheme in many respects. Firstly, it takes into account that events disseminated in the system may represent real-world events requiring real-time dissemination in the control system. An event in the real-world is characterized by a context of occurrence which at least is defined by a location and a point in time. Additionally, the event may have quality related properties like a certain fidelity margin reflecting a conversion process or a bounded temporal validity. Therefore, we assign context and quality attributes (sometimes called "functional" and "non-functional") to an event which are complementary to the event data.

Secondly, we introduce event channels which allow to specify the quality of dissemination. An event channel is an abstraction of the communication mechanisms which provides a publisher/subscriber interface. Events can be pushed to an event channel which then ensures the defined dissemination quality. Particularly, real-time and reliability requirements can be specified for event channels.

Thirdly, COSMIC is targeted to low performance systems with a small memory footprint. Therefore, we adopted a number of mechanisms which allow for an efficient system even at the cost of a slight impact on flexibility. This applies in particular to the binding and filtering mechanisms, explained later in the paper.

This paper advances previous work by presenting the architecture of the distributed event middleware. It starts with introducing the notions of events and event channels in Section II. Section III describes the layered COSMIC architecture. Related work is reviewed briefly in Section IV and conclusions and future work are summarized in Section V.

II. Events and Event channels

An event may occur in the physical environment or in the computing system. Hence, both, an observation of a real-time entity and a state transition of a variable are uniformly characterized, represented and disseminated as an event. From an architectural perspective, sensors and actuators are not visible as low level I/O sub-systems, but, as smart networked components used by applications via the event middleware. On the respective abstraction level they are represented as active objects, taking the roles as publishers and subscribers producing or consuming events, respectively. An architecture which well describes and explains this model in general has first been presented by [7] in the context of CORTEX. Veríssimo & Casimiro describe a system architecture to disseminate generic events. In their model, the applications interact only via the event layer. The event layer hides the network as well as the transformation process of the I/O subsystems. The basic objective behind this architecture is: because all events appear at the event layer, it is possible to order the events consistently whether they are generated in the environment or in the system. This can be done by making assumptions about the properties of the respective event channels. The generic event architecture therefore tackles the problem of hidden channels [8] in a consistent model. The event and event channel model introduced by COSMIC constitute a specific way to realize such an architecture.

Events carry information from publishers to subscribers. However, differently from simple messages, events are a typed information carriers which include the context in which such information has been generated and quality attributes defining requirements for dissemination. Thus relevant aspects of the event semantics are explicitly expressed and carried with the event. Similarly to [6], an event instance is specified by a tuple comprising a subject, attributes and contents:

$$Event := \langle Subject, AttributesList, Contents \rangle$$

A subject defines a type of event and thus is related to the event contents. The pro and cons of subject-based addressing are discussed in detail elsewhere [3], [1], [2], [5] and are not reviewed again in this paper. Specifically to our system, a subject is a tag represented by a unique identifier. An event is further characterized by a set of attributes. The attributes related to context may comprise e.g. a location and the time of occurrence. The non-functional attributes include quality aspects as a validity interval (expiration time), a deadline and a tolerated omission degree. Deadlines are employed for scheduling events in the communication system. As explained in Section II-A, an event may miss a (soft) deadline depending on the event channel class. In this case, the validity interval defines the point in time after which an event may be discarded completely. This mechanism is helpful in a real-time system to dispose of outdated events as early as possible. Finally, the events' contents comprise data specified by applications.

Events are propagated from publishers to subscribers through event channels. We conceive event channels as abstractions of network resources and therefore assign QoS related properties attributed to the utilization of these resources. An event channel is an instance of an event channel type characterized by a subject and quality attributes:

$$EventChannel := \langle Subject, AttributesList \rangle$$

An event channel exclusively disseminates events that are compatible with its own subject. As attributes, an event channel may include e.g. a latency, dissemination constraints and reliability parameters. An event channel may handle multiple publishers and multiple subscribers, thus implementing a many-to-many communication channel. A data structure representing an event channel is dynamically created in the local middleware whenever a publisher first announces a publication or a subscriber subscribes to a channel.

A. Real-time event channels

In a system monitoring and controlling a physical environment, timeliness and reliability requirements have to be met. However, not all functions of the system need the same level of quality of service from the underlying communication system. Therefore we introduce events channels with different quality properties to reflect the well known trade-off between highly predictable event channels and the associated costs. COSMIC supports three classes of event channels: hard real-time event channels (HRTEC), soft real-time event channels (SRTEC) and non real-time event channels (NRTEC). HRTC are synchronous, i.e. events sent through a HRTEC are guaranteed to meet their deadlines under an anticipated number of omission faults. Events sent through a SRTEC are scheduled in a best effort manner according to their transmission deadlines. An event pushed to a SRTC may miss its deadline e.g. in situations of transient overload. Events which have no timeliness requirements (e.g. configuration and maintenance events) are sent through a NRTEC.

Hard real-time event channels are synchronous, i.e., the bounds of transmission latency and jitter are known and minimal. These properties are based on a static schedule that reserves network resources to specific event channels at specific time slots, like in a TDMA — Time Division Multiple Access — medium access scheme. Communication traffic flowing through the less stringent channels can not obstruct the transmission of any event that is going through a HRTEC.

This isolation property has been implemented for the CAN-Bus by exploiting the CAN priority scheme. Whenever a hard real-time event has to be sent, it has the highest priority in the system and thus no lower priority message can interfere with its transmission. The mechanism, however, allows to reuse time slots reserved by a HRTC by events of lower criticality classes in the case that no message will be sent through the HRTC. Conflicts between two hard real-time channels are not possible due to the static schedule that commonly rules them all. A detailed description of the mechanism can be found in [9].

The timeliness requirements of soft real-time channels are expressed by deadlines and validity intervals (expiration times). Soft real-time events are scheduled according to their transmission deadlines by an earliest deadline first (EDF) algorithm. The transmission deadline is defined as the latest point in time when a message has to be transmitted. Soft real-time channels enforce pritorities that are always lower than hard real-time channels and higher than non real-time channels. However, because a message can not be interrupted during its transmission and messages may become ready at arbitrary points in time, EDF will not always take the right scheduling decisions (only a clairvoyant scheduler would be able to do so) and hence, situations of temporal conflicts and transient overload may occur. In these situations, messages will still be transmitted at a later time in a best effort manner. An SRT event message eventually will be discarded if its transmission time is delayed beyond its temporal validity specified by the expiration time.

The event channel classes supported by an event system depend on the underlying communication infrastructure. Hard, soft and non real-time event channels have been implemented for the CAN-Bus [10]. Non real-time channels have been implemented also on top of TCP/IP, both for the IEEE 802.3 and 802.11 medium access protocols [11].

B. Routing and filtering

Routing an event from a publisher to a subscriber is based on the subject of a message rather than on a destination address. The subscriber expresses interest in a certain subject and every event which is published on this subject must eventually arrive at the subscriber. Thus the implementation of the publish/subscribe (P/S) model on top of a field-bus infrastructure comprises the issues of getting the event to the right destination and notifying each subscriber only of those events to which it has subscribed. This includes the aspects of routing and filtering. Routing and filtering are tightly related in a subject-based addressing scheme. Conceptually, many publisher/subscriber protocols exploit a broadcast as basic routing mechanism. Then every node receives all messages and then performs the task of selecting an event by applying a filter which passes only those events to which a subscription has been issued. In COSMIC, we take a two stage approach. The subject is exploited for routing by dynamically bind it to a network address. This is done on each announcement of a publication or a subscription transparently to the publisher and subscriber. The mechanism is described in detail in [2], [11]. Binding a subject to a network address (which e.g. can be a multicast address) puts the task of subject filtering to the network controller and thus frees the node from examining every message. This is particularly important for the tiny components encapsulating a smart sensor or actuator which simply do not have enough CPU performance to analyze every message. However, subject filtering alone does not provide an adequate level of filtering. Consider a smart actuator in a robot which only wants to receive information e.g. from local sensors to perform some reactive control. The locality of events is not expressed in the subject but in the attributes. Therefore, an additional level of filtering based on the structural properties of attributes is introduced. This filtering mechanism also is designed to meet the requirements of resource constraint systems with respect to performance and memory demands. With the routing mechanism based on dynamic binding and the attribute-based filtering it is ensured that, with a high probability, the subscriber is notified only about the events for which it actually has subscribed. The attribute-based filter mechanism is further described in Section III-A.1.

III. Architecture

The network architecture to be presented next has been designed in correlation to the event model described in Section II. The architecture is centered on a middleware layer which has been maintained purposefully lean by including only carefully selected semantics and services. The goal of the design is to make quality properties of the communication infrastructure accessible on the abstract level of events. Hence, application objects can explicitly set the temporal and dependability attributes required for event dissemination and notification. The middleware automatically maps the attributes to the technical parameters of the underlying infrastructure.

A comprehensive view of the proposed architecture is shown in Figure 1. The middleware layer (with a grey background) is show on top of an infrastructure comprising the CAN-Bus [10]. The time service provides the abstraction of a global clock that is needed by several layers. As can be observed, the middleware is further subdivided in a pair of layers: the event layer (EL) and the abstract network layer (ANL). This separation allows the event layer to concentrate on the issues to which it is specifically concerned, i.e., addressing, filtering and notification of events. Then, the ANL maps the abstractions of the event layer to the communication infrastructure. This mapping relates to several aspects, e.g., the conversion of events to specific message formats and vice-versa, the enforcement of event channels' quality properties and semantics, etc. On the other hand, the ANL provides awareness on infrastructure related issues to the EL, e.g., on communication failures by rising exceptions.

The burden assigned to the abstract network layer is a tough challenge because of the widely varying characteristics of the existing network infrastructures. In this paper, we tackle the topic of mapping the event layer abstractions to the CAN-Bus

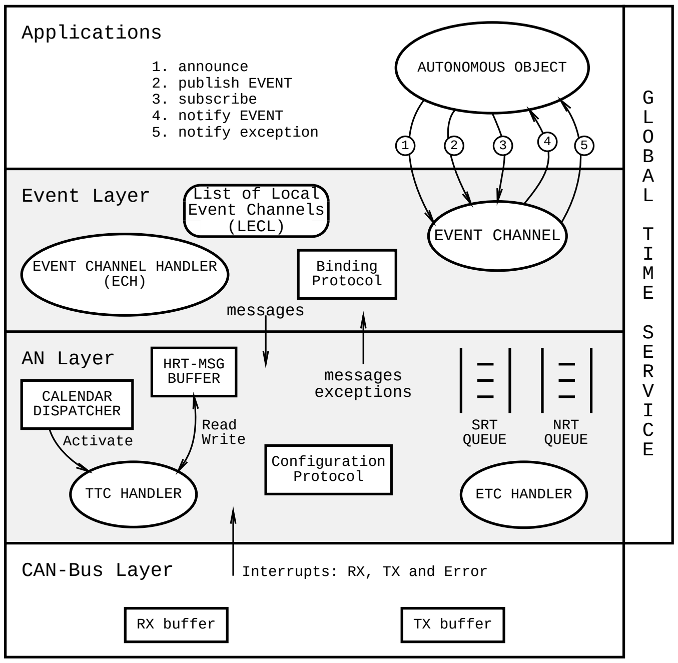

Fig. 1. Middleware layered structure, showing components, data structures and protocols.

interface. Among fieldbus technologies, the CAN-Bus is particularly suited to implement the publish/subscribe protocol. Its message distribution scheme is based on a message identifier, which does not indicate the destination of the message but rather the meaning of the transported data. Therefore, the hardware filtering mechanism can be exploited in order to efficiently implement a subject-based publish/subscribe protocol. Moreover, it provides advanced built-in features that are required by the real-time applications that we intend to support: multiple access without centralized control, priority-based collision resolution, efficient implementation of positive acknowledgment, automatic fail-silence enforcement with different fault levels, etc. In fact, some of these features have been exploited to implement the real-time event channel classes.

The next subsections describe the abstractions, services, components and protocols that comprise the event and the abstract network layers. The CAN-Bus layer is presented for completeness.

A. The Event Layer

The service provided by the event layer is the timely and reliable dissemination of events from publishers to subscribers. This service comprises a set of primitive operations which are accessed through the interfaces of event channels. The specification of the interfaces of hard and soft real-time event channels (EC) is provided in [12]. The implementation of the services provided by the event layer is supported by the Event Channel Handler (ECH), shown in Figure 1. The event channel handler creates, maintains and destroys local representations of event channels on behalf of applications. These local representations are needed to:

- store the dynamic binding data;
- store event dissemination attributes;
- relate the local publishers and subscribers.

The event channel handler maintains the event channels in the "local event channels list" (LECL, see in Fig. 1). Therefore the ECH can retrieve the local configuration data of any event channel when it needs to handle an event. For instance, when an event is pushed to a non real-time channel: the ECH retrieves the corresponding entry, reads the fixed priority assigned to the channel and adds this priority to the event message before passing it to the abstract network layer. At this point, before advancing the discussion on how event channel are represented in the middleware, it is convenient to briefly introduce the interface provided by event channels.

The interface to HRTEC's and to SRTEC's are summarized in Figure 2. A publisher must *announce()* an event channel before publishing to it. The *announce()* operation establishes the local data structures needed to handle an event channel. The *announce()* implementation includes a call to the ECH component, which executes the binding protocol and then creates a local representation of the event channel. During the binding protocol, the ECH exchange some messages through the network. A low priority is assigned to these binding messages, because they are not comprised in a critical control path. In fact, the *announce()* operation was introduced because no assumptions can be made *a priori* about the timeliness of the binding messages. If the dynamic binding was to be performed the first time an object is publishing an event, the timeliness of this event would not be guaranteed. The *announce()* operation enforces the separation between the binding phase and the operation phase of an event channel. The argument *attributes_list* passed in an *announce()* specifies the quality properties of the channel, e.g. whether the publication is periodic or sporadic, reliability requirements and event rates. These informations are needed to configure the access to the required resources.

Because time slots are statically allocated to HRTEC's, the local representations of HRTEC's are also statically created by the ECH. Therefore, the announcement of a HRTEC does not result in the creation of a new event channel data structure. Instead, the ECH searches the LECL for a HRTEC entry with the subject specified by the application. The announcement is accepted if the attributes list declared by the publisher matches the static attributes list of the retrieved HRTEC. In a positive case, the ECH invokes the abstract network layer which creates a hard real-time message buffer (HRT-MB) and sets the time-triggered routine (TTC-Handler) that handles the transmission of HRT messages. The roles of the HRT-MB and the TTC-Handler are discussed in Sections III-B.

After a channel is announced, the application can publish events by calling the *channel.publish(event)* operation. The semantics of the *publish()* operation for hard, soft and non real-time channels can be infered from the discussion presented in Section II-A.

Symmetrically, the *subscribe()* operation allows subscribers to set event filters, QoS parameters of notifications and no-

```
class hrtec {

private:
subject subject_uid;

public:
// constructor and destructor of the class
hrtec(void);
~hrtec(void);
// methods used for publishing
int announce(subject, attributes_list, exception_handler);
int publish(event);
// methods used for subscribing
int subscribe(subject, attribute_list, event_queue, notification_handler, except_handler);
int cancelSubscription(void);
}


class srtec {

private:
subject subject_uid;

public:
// constructor and destructor of the class
srtec(void);
~srtec(void);
// methods used for publishing
int announce(subject, attribute_list, exception_handler);
int cancelPublication();
int publish(event);
// methods used for subscribing
int subscribe(subject, attribute_list, event_queue, not_handler, exception_handler);
int cancelSubscription(void);
}
```

Fig. 2. Interface to a HRTEC class in C++

tification and exception handlers. Then, incoming events that have matched the filters are stored in the *event_queue* specified in the subscription. As is widely known, the selection of a queuing policy is a critical issue in real-time systems [13]. Allowing subscribers to maintain private event queues for receiving events enables them to specify the queuing policy that better matches their needs. The *notification_handler* argument comprises application code that is executed at-the-deadline of an incoming event. The ECH issues the notification, upon which the *notification_handler* retrieves the event from the *event_queue* and executes the required activity.

The event layer must implement its services while enforcing timeliness and reliability. The provided abstractions of events and real-time event channels (Section II) assist on achieving this goal. As already said, events encapsulate a set of intrinsic attributes and application related parameters, which are accessed through specific methods. Hence, a requirement like a deadline can be consistently enforced at any intermediate step when transporting an event through a CAN network. Accordingly, announced and subscribed event channels are explicitly represented in the event layer. Hence their quality properties can be enforced locally. There is one entry in the LECL (introduced above) for each event channel "connected" to a node. That means, even if two or more publisher or/and subscribers have locally announced and/or subscribed to a given event channel, only one EC entry is maintained in the LECL. The fields comprised in each LECL entry are itemized below:

- *Subject UID;*
- *Event tag;*
- *Channel class;*
- *Channel attributes list;*
- *Announcements list;*
- *Subscriptions list.*

The *subject uid — Unique Identifier —* uniquely identifies an event channel[1]. The *event tag* is obtained at runtime after the binding protocol. It represents the subject in a form that is related to the addressing scheme the underlying network[2]. The event tag could be e.g. a multicast address. Hence, the "binding" is actually provided by the event channel, which keeps together these two fields. When an event is published to an event channel, it is tagged and passed to the ANL. Symmetrically, when an event arrival is indicated by the ANL, its tag is matched against the channels' entries whose subscriptions lists are not empty. As already discussed, the *channel class attributes list* describe the quality properties of the channel. This list is either filled with the information provided in the *announce()* and *subscribe()* operations (for SRTEC's and NRTEC's) or statically defined for HRTEC's.

The *announcements* and *subscriptions lists* are further data structures nested in the event channel. Each entry on the announcements list includes the exception handlers. The exceptions supported for announcements are related to missed transmission deadlines and to expiration of validity intervals (Section II-A). Each entry in the subscriptions list includes a notification handler, an exception handler and the specification of an attribute-based filter. The notification handler comprises application defined code which is executed upon the notification of an event. The supported exceptions for subscriptions are missed deadlines of periodic events. Attribute-based filters are briefly introduced in the next section. The reader is referred to [14] for a more comprehensive description.

1) Attribute-based filtering : When striving to filter events there is an intrinsic tradeoff between expressiveness of filter specification and performance of filtering execution. On one extreme, content filters [15] try to match events based on the evaluation of predicates defined over arbitrary contents. This scheme allows a fine specification of events, but induces a certain degree of unpredictability and higher computing overheads. On the other extreme, subject filters [3] are based on the inspection of a single event parameter, the subject, which maps to bounded contents. Subject-based filters provide for better predictability and lower overheads, but are less expressive.

Attribute filters are in an intermediate position between content and subject filters. Attribute filters try to match events based on the presence/absence of pre-defined event attributes. To specify an attribute filter, a programmer must specify the minimum set of attributes that an event must present to be matched. It means that any event presenting all the attributes

[1]I.e., the subject will be the same for the representation of same event channel in any node.

[2]The technical usage of the event tag for filtering messages in the CAN-Bus is explained in Section III-B.

specified for a filter, plus some others, is also matched. In the other hand, an event that lacks any single attribute specified for a filter is not matched by this filter. This filtering scheme is defined formally through the structural conformance relationship [16]. Hence, an attribute filter is formally defined as one that matches all the events conforming to a specific signature. Such signature can be as follows:

$$\mathcal{F}_{att} = \left\{ Name_i : Type_i, ..., Name_j : Type_j \right\}$$

The $Name_k : Type_k$ elements are the formal definitions of the attributes, where $Name_k$ is a variable identifier and $Type_k$ is a primitive type, e.g., integer, float, etc. For example, the attribute filter $F_{att} = \{A : float, B : int\}$ matches the events $E := \langle A : float, B : int\rangle$, $E := \langle A : float\rangle$, $E := \langle B : int\rangle$ and $E := \langle\ \rangle$, but does not match $E = \langle A : float, B : int, C : string\rangle$ neither $E = \langle X : float, Y : int\rangle$.

Attribute filters can be implemented by mapping the event structures to tags (i.e., bit vectors) and then using these tags as the keys for hashing tables. The combination of attribute and subject filters provides a better expressiveness than pure subject filtering, at the price of an additional table look-up.

B. The abstract network layer

Mapping P/S abstractions directly to the underlying network is a tough challenge because the usual abstractions of the underlying communication infra-structure are low-level messages, which do not match the requirements of subject-based addressing and QoS specifications for channels. Therefore, an abstract network layer is introduced which enriches the properties of the network by exploiting the CAN-Bus built-in mechanisms. E.g., the ANL hides the priority-based collision resolution mechanism of the CAN-Bus and in trade it offers the abstractions of hard, soft and non real-time messages which can be handled by the event layer.

The services offered by the ANL are the transport of time-triggered messages, EDF ordered messages and fixed priority messages (i.e., hard, soft and non real-time messages). Hard real-time messages have precedence over soft real-time messages and both have precedence over non real-time messages. There are exceptions to inform the event layer about failures on the message transport. This is needed e.g. when the supported omission degree is exceeded. Exceptions are also supported for missed deadlines and expiration of validity intervals.

The burden put on the ANL has two aspects: the enforcement of quality properties when transporting messages over the CAN-Bus and the mapping of structured messages to the CAN-Bus frame format. Hence we organized the ANL in two sub-layers: the message dispatching sub-layer and the structured CAN transport sub-layer.

The message dispatching sub-layer (MD) ensures that incoming and outgoing messages are dispatched in accord to the required quality properties. Hence, the MD must ensure that soft and non real-time messages (which are event-triggered) do not interfere with hard real-time messages (which are time-triggered). Roughly stating, this goal is accomplished by, firstly, assigning the highest priority to the time-triggered messages, and secondly, by accounting for the worst case durations of priority inversions when defining the static schedule. To implement this mechanism we have introduced two components in the architecture: the Time-Triggered Communication Handler (TTCH) and the Event-Triggered Communication Handler (ETCH) (see in Figure 1). The detailed description of the roughly mentioned mechanism is beyond the scope of this paper and the interested reader is referred to [9].

To guarantee that event-triggered messages do not interfere with time-triggered messages, the TTCH relies relies on the *Calendar Dispatcher* (CD) component. The CD activates the TTCH in synchronism with the static schedule of the arrival and departure of hard real-time messages. Moreover, the CD guarantees that the highest priority in the system is assigned to the TTCH activation. Hence, the precedence for the dispatching of hard real-time messages is assured. E.g., on the time slot assigned for receiving a message, the TTCH reads the message from the CAN-Bus controller and writes to the hard real-time message buffer (HRT-MB). The access to HRT-MB is shared with the event layer. Vice-versa, for an outgoing message the TTCH polls the corresponding HRT-MB on the scheduled slot. If the buffer contains a fresh message (i.e., one published since the last polling), the TTCH handles its transmission over the CAN-Bus.

The Calendar Dispatcher is an active component that implements the dispatching of time-triggered tasks. It supports the handling of hard real-time events within the nodes. The *calendar dispatcher* can be implemented on top of a real-time operating system or as a real-time executive on small computing elements. Its implementation only depends on the availability of the abstractions of a calendar and a global clock. This component can support the programming of time-triggered tasks running in any layer e.g. in the application or in the middleware. It can be exploited to synchronize every step comprised in the transport of hard real-time events, from end to end.

Soft and non real-time messages are handled by the ETCH which passes them to the event layer by means of specific queues. Soft real-time messages are ordered in accord to the EDF discipline. Non real-time messages are ordered by their relative importance. These queues also establish the ordering of notifications in the event layer.

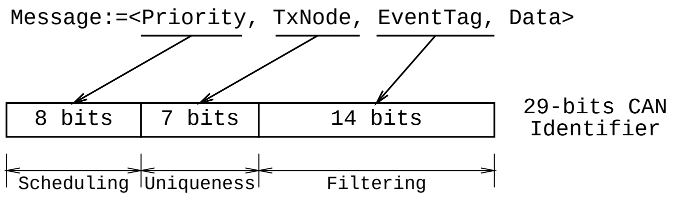

Fig. 3. Mapping of messages to CAN identifiers.

On its turn, the structured CAN transport sub-layer relates specifically to the CAN implementation. It maps the structured message abstractions to the raw CAN-Bus frame format, in a way that all quality properties established in the event layer are finally enforced in the bus. Therefore, the abstraction provided by the structured CAN transport sub-layer are structured messages:

$$Message := \langle Priority, TxNode, EventTag, Data \rangle$$

The mapping of structured messages to a long CAN identifier is depicted in Figure 3. The *Data* field, which may be itself sub-structured, is mapped to the payload part of the CAN message. The *EventTag* field is a short local identifier that designates an event channel in a sub-network. The *EventTag* is maintained in the event layer, after the binding protocol, as discussed in Section III-A. It is passed as an argument when the EL uses the ANL services. The *Priority* and the *TxNode* fields are handled in the ANL.

When mapping middleware messages to the CAN layer, the uniqueness of the CAN identifier must be assured. This is a requirement of the CAN protocol. Uniqueness is assured by stamping the transmitter identity in the message. However, the CAN identifier can not accommodate a global (long) node identifier, due to its limited size. The *configuration protocol* was introduced to overcome this impasse. Here, "configuration" refers to the mapping of the global (long) node identifier to a short one, the *TxNode*, which is local to each CAN sub-network. The ANL executes the configuration protocol on each node, when the node is initialized. As result, the *TxNode* is obtained and subsequently used to stamp every transmitted message.

C. The CAN layer

The CAN Layer is composed by the CAN driver and the CAN controller itself. The driver provides routines to read and write CAN messages and for setting the filters for network addresses. The protocol used in this layer is the CAN 2.0B standard with 29-bit identifiers. "CAN identifier" refers to the header of CAN frames, i.e., the part of the messages used for bus arbitration and addressing purposes. CAN identifiers are structured by the middleware in multiple fields, as depicted in figure 3. The message's priority is mapped to a 8-bits field, which is aimed to enforce the bus access *schedule*. Following, a 7-bit field ensures the *uniqueness* of the CAN identifier. This is a requirement of the CAN-Bus protocol. The remaining 14 bits of the message identifier constitute a short event identifier and are employed to route and filter a message.

IV. Related work

Event channels have been mapped to CORBA services [17]. While CORBA services are inherently centralized, the proposed architecture is distributed. In fact, the CORBA Event Channel retains server semantics. E.g., an application object must have a reference to an object in order to communicate through the event channel. If the event channel is moved from one node to another, the application must be aware of that. In contrast, the presented architecture provides an abstraction of the network. Moreover, the coordination model underlying any CORBA service is the RMI. That means, while an asynchronous interface is offered to applications in the surface, the dissemination of events is effectively accomplished by means of point-to-point synchronous connections; exactly what is supposed to be avoided in the P/S model.

The publish/subscribe model and the abstraction of an information bus have been originally implemented in the *TIBCO Rendezvous* software [18], which integrates some soft real-time features. E.g., it is possible to create several queues and explicitly associate event types with event queues. Furthermore, it is possible to group several queues in a *queue group.* A static priority can be assigned to each queue in a group. The dispatcher thread for a queue group dispatches events in accord to the priority of the queues. Concerning CPU scheduling, this architecture has two drawbacks: 1. dispatched events (callbacks) can be not preempted by events relating to the same queue group; 2. the middleware has no interface for handling threads' priorities. Hence, real-time programming can be not handled at the same abstract level as the event programming.

NDDS is a industrial strenght P/S middleware [19], [20], which is advertised for hard real-time applications. Its architecture explicitly assumes the *Ethernet* as the underlying interface. Therefore, real-time can only be assured on a probabilistic basis. It means that the communication load must be known in advance and that deviations from the load hypothesis are minimally tolerated. Deadlines are specified only for subscriptions. Therefore message scheduling is supported only in the queues maintained in the receivers side.

V. Conclusions and Future Work

This paper has focused on the architectural aspects of COSMIC, a middleware for supporting real-time event-based interaction on top of the CAN-Bus. The design of the architecture has been presented, which separates the issues relating to the event model from the management of the quality properties of communication. An abstract network layer has been introduced to abstract and enrich the raw properties of the basic communication infrastruture. The proposed architecture provides a decoupled interaction model, which supports the development of applications based on autonomous objects. On the publish/subscribe abstract level, distinct event channel classes presenting high level semantics represent a simple programming interface for real-time communication. Future work include the elaboration of a gateway model, for managing the connection of sub-networks presenting different QoS capacities.

Acknowledgments

This work has been supported by the European Union's Information Society Technology Program under contract IST-2000.26031 (CORTEX: Cooperating real-time sentient objects: architecture and Experimental evaluation).

This work has been partly supported by CNPq

References

[1] C. E. Pereira, J. Kaiser, C. Mitidieri, C. Villela, and L. B. Becker, "On evaluating interaction and communication schemes for automation applications based on real-time distributed objects," in *4th Int. Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'01)*, Magdeburg, Germany, 2001.

[2] J. Kaiser and M. Mock, "Implementing the real-time publisher/subscriber on the controller area network (can)," in *2nd Interantional Symposium on Object-Oriented Real-time distributed Computing*, Saint-Malo, France, May 1999.

[3] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen, "The information bus - an architecture for extensible distributed systems," in *ACM Symposium on Operating System Principles*, 1993, pp. 58–68.

[4] R. Rajkumar, M. Gagliard, and lui Sha, "The real-time publisher/subscriber inter-process communication model for distributed real-time systems: Design and implementation," in *IEEE Real-Time Technology and Applications Symposium*. IEEE Real-Time Technology and Applications Symposium, June 1995.

[5] P. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," EPFL, Lausanne, Switzerland, Tech. Rep. DSC ID:200104, 2001. [Online]. Available: citeseer.nj.nec.com/442483.html

[6] R. Meier and V. Cahill, "Steam: Event-based middleware for wireless ad hoc networks," in *International Workshop on Distributed Event-Based Systems*, 2002.

[7] P. Veríssimo and A. Casimiro, "Event-driven support of real-time sentient objects," in *Eighth IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2003)*, Jan 2003.

[8] H. Kopetz and P. Verissimo, "Real time and dependability concepts," in *Distributed Systems*, S. Mullender, Ed. ACM Press New York, 1993, ch. 16, pp. 411–446.

[9] M. Livani, J. Kaiser, and W. Jia, "Scheduling hard and soft real-time communication in the controller area network," *Control Engineering Practice*, vol. 7, no. 12, pp. 1515–1523, December 1999.

[10] *CAN Specification version 2.0*, Robert Bosh GmbH, September 1991.

[11] J. Kaiser and C. Brudna, "A publisher/subscriber architecture supporting interoperability of the CAN-Bus and the internet," in *2002 IEEE International Workshop on Factory Communication Systems (WFCS2002)*, Västeras, Sweden, August 2002. [Online]. Available: citeseer.nj.nec.com/kaiser99implementing.html

[12] J. Kaiser, C. Brudna, and C. Mitidieri, "A real-time event channel model for the CAN-Bus," in *11th Annual Workshop on Parallel and Distributed Real-Time Systems, held in conjunction with the International Parallel and Distributed Processing Symposium IPDPS*, Nice, France, April 2003, pp. 22–26.

[13] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Aplications*. Kluwer Academic Publishers, 1997.

[14] C. Mitidieri and J. Kaiser, "Attribute-based filtering for embedded systems," in *Second International Workshop on Distributed Event-Based Systems (DEBS'03), in conjunction with The ACM SIGMOD/PODS Conference*, 2003.

[15] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Design of a scalable event notification service: Interface and architecture," Department of Computer Science, University of Colorado, Tech. Rep., August 1998.

[16] L. Cardelli, "Structural subtyping and the notion of power type," in *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, San Diego, California, 1988, pp. 70–79.

[17] OMG, "Notification service, version 1.0," June 2000, object Management Group, http://www.omg.org.

[18] TIBCO, "TIBCO Rendezvous Concepts, release 7.0," TIBCO Software Inc., Palo Alto, CA, April 2002.

[19] RTI, "Real-time inovations. network data delivery service," http://www.rti.com.

[20] G. Parado-Castellote, S. Schneider, and M. Hamilton, "Ndds: The real-time publish subscribe network," in *IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, 1997, pp. 222–232.