Supporting Heterogeneous Applications in the DECOS Integrated Architecture

C. El Salloum, R. Obermaisser, B. Huber, H. Kopetz
Vienna University of Technology, Austria
Email: {salloum, romano, huberb, hk}@vmars.tuwien.ac.at

Neeraj Suri
TU-Darmstadt, Germany
Email: suri@informatik.tu-darmstadt.de

Abstract

The DECOS integrated time-triggered architecture provides a framework for integrating multiple heterogeneous real-time application subsystems within a single distributed computer system while retaining the fault-isolation, fault-containment and complexity-management benefits of a classic federated system. A central issue in the DECOS architecture is the provision of standardized, validated and certified services that facilitate the development of distributed real-time applications. This paper describes how these services are structured within the architecture in order to satisfy the diverse requirements of heterogeneous applications (e.g. different real-time requirements, different criticality levels). In particular we focus on the reuse of legacy subsystems, and show the feasibility of our concept by implementing a Controller Area Network (CAN) application within the integrated architecture.

1 Introduction

Distributed real-time systems are increasingly being used to control safety-critical functions in automotive, aerospace and space applications for civil or military purposes. Examples are steer by-wire, fly-by-wire, missile guidance systems and many other control systems.

At present, the design of large and complex safety-critical systems often follows a *"federated"* design philosophy, where each application subsystem has its own dedicated distributed computer system. Federated systems have been preferred for ultra-dependable applications since the inherent separation of application subsystems simplifies fault-isolation and complexity management. Another reason for the existence of federated systems is that many embedded systems have historically grown. A good example is the automotive industry where different subsystems (e.g. Anti-lock Braking System (ABS), Electronic Stability Control (ESC)) have been subsequently added in order to continuously improve a car's overall functionality.

Contrary to a federated system an *integrated* system is characterized by the integration of multiple application subsystems within a single distributed computer system [13]. Since different application subsystems share the available hardware resources (computational nodes and the communication network) the total number of required nodes and communication resources can be significantly reduced compared to the federated approach (modern luxury cars incorporate up to 75 Electronic Control Units (ECUs) [5]). The reduction of network nodes leads to a higher dependability of the total system with respect to wiring and connectors. Furthermore integrated architectures permit an optimal interplay of application functions, advanced redundancy management and the realization of a system wide homogeneous diagnostic infrastructure.

The objective of the European IST project DECOS (IST-511764) is to devise an integrated architecture that provides a framework for integrating multiple application subsystems within a single distributed computer system while retaining the *fault-isolation*, *fault-containment* and *complexity-management* benefits of a federated system [13].

This paper focuses on the flexible provision of architectural services within the DECOS architecture. It is organized as follows: The challenge of structuring architectural services is stated in Section 2. Section 3 describes our proposed approach. The interface of the DECOS platform is described in section 4. Section 5 describes the assembly of a prototype implementation that was used to evaluate the proposed approach. The developed concepts are finally summarized in section 6.

2 Problem Statement

A central issue in the DECOS integrated architecture is the provision of standardized, validated and certified architectural services that facilitate the development of distributed real-time applications. These architectural services separate the application functionality from the underlying platform technology in order to facilitate reuse and reduce design complexity. This strategy corresponds to the concept of platform-based design [21], which proposes the introduction of abstraction layers that facilitate refinements into subsequent abstraction layers in the design flow. Examples for architectural services are communication, diagnostic or fault-tolerance services. The problem on which we focus in this work is how to structure these services and how to provide them to the applications.

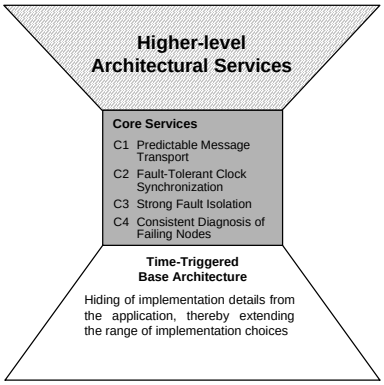

Figure 1: DECOS Architecture

Figure 1 depicts the hierarchical structure of the DECOS architecture. The architecture is based on a minimal set of *core services*. The core services include predictable time-triggered message transport, fault tolerant clock synchronization, strong fault isolation, and consistent diagnosis of failing nodes. The small number of core services eases a thorough validation (e.g., permitting a formal verification), which is crucial for preventing common mode failures since all high-level services and consequently all applications depend on the core services. Any architecture that provides these core services can be used as a base architecture [20] for the DECOS integrated architecture. An example of a suitable base architecture is the *Time-triggered Architecture (TTA)* [11].

Based on the core services, higher-level architectural services can be realized. Since the DECOS architecture is intended to be used in multiple heterogeneous application domains where each domain has its own specific demands, a wide variety of architectural services has to be supported. Specific functional requirements of an application, like dependability or flexibility induce the demand for respective architectural services. Examples for different requirements are the desired properties of communication protocols: Safety-critical applications (e.g. steer-by-wire systems) that have to deliver a given service within a guaranteed time bound require communication services that are highly deterministic in the temporal domain (e.g. time-triggered protocols), whereas many non-safety-critical systems typically have to be optimized for average load in order to optimize cost (e.g. the comfort electronic system of car) and require a highly flexible protocol that adheres to the event-triggered communication paradigm. Additional architectural services have to be provided in order to support the reuse and the integration of legacy applications by emulating the corresponding legacy platform. Examples are emulations of standard communication protocols that are frequently used in specific application domains (e.g. CAN [1] in the automotive sector, ARINC 629 [14] for avionics ...).

Consequently, a *diverse*, *configurable*, *scalable*, and *open* set of architectural services has to be provided. Scalability is an important factor since it is not possible to anticipate how applications and their requirements will evolve in the future. Openness means in this sense that third party developers should be allowed to add additional domain-specific services to the architecture. Since freely added services can not always be assumed to be free from design faults, *error containment* with respect to architectural services is a key issue. In order to be *resource effective*, the set of services has to be configurable, which means that a platform that is used in a given system should be adaptable to contain only those services that are actually required and used by the system.

In this work we propose a layered model for higher-level architectural services that supports the open integration of additional services without compromising the dependability of already existing services.

3 Proposed Model

The proposed approach divides the set of architectural services into two distinct layers: A stable and trusted set of *Generic High-level Services*, and based on these services an extensible and

open set of *Domain Specific Services* (see Figure 2).

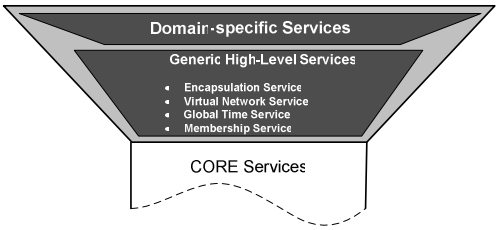

Figure 2: Architectural Services

3.1 Generic High-level Services

In the DECOS architecture, the application services of the integrated system that are provided at controlled object interfaces are divided into a set of nearly-independent *Distributed Application Subsystems (DASs)* where each DAS provides a distinct functionality (e.g. a brake-by-wire DAS or a multimedia DAS in a car). Each DAS is further decomposed into smaller units called *jobs* which are the basic unit of distribution.

A major focus of the architecture is to support independent design and development of separate DASs in an integrated system. The generic high-level services encompass a set of services that support a DAS-centric design flow.

Among the generic high-level services is the *encapsulation service* that establishes so-called *partitions*, where each partition functions as an execution environment for a single job. The encapsulation service guarantees temporal and spatial partitioning between partitions, which prevents faulty jobs from stealing processor time of any other job and from corrupting another job's memory structures [7].

In order to support a DAS-centric design flow also with respect to communication, virtual network services [17] are provided which realize encapsulated communication infrastructures for single DASs. All communication activities of a virtual network are private to the respective DAS. Transmissions and receptions of messages can only occur by jobs of the corresponding DAS unless a message is explicitly exported or imported by a *hidden gateway* [16]. Furthermore, a virtual network exhibits predefined temporal properties that are independent from activities in other virtual networks. The provision of encapsulated execution environments and encapsulated communication infrastructures is a prerequisite for the independent development of DASs and for the integration of multiple DASs with mixed criticality.

Despite the encapsulation service and the virtual network service, the high level-services encompass a *fault-tolerant global time service* for the coordination of distributed activities of the jobs of a DAS, and a *reliable membership service* that provides consistent information about the operational state (correct or faulty) of the network nodes within the distributed system.

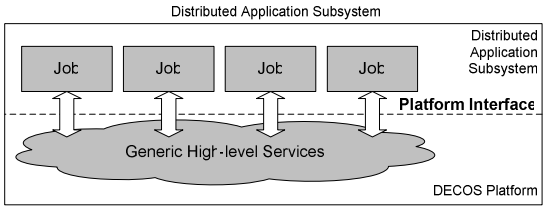

Figure 3: Platform Interface

The generic high-level services are provided via the so called *Platform Interface (PI)* to the jobs of a DAS (figure 3). The specification of the PI hides the details of the underlying platform, while providing all information required for ensuring the functional and meta-functional (dependability, timeliness) requirements in the design of a safety-critical real-time application. It serves as a validated stable baseline that reduces application development efforts and facilitates reuse, because applications are built on a generic interface that can be established on top of numerous platform technologies.

We regard a job as a *Fault Containment Unit (FCR)* for software faults (fault containment is established by the partitions of the encapsulation service), and assume all architectural services that are beneath the PI to be validated and free from design faults. A more detailed fault hypothesis can be found in [13].

3.2 Domain-specific Services

In order to provide a simple and stable baseline for application development only a small set of selected services is provided natively via the PI. For many applications these services will be sufficient, but in some cases they might have to be extended or refined. These domain-specific extensions and refinements should not increase the complexity and the certification effort of the DECOS platform. Therefore these domain-specific services are realized on top of the PI in a dedicated layer within the application job. We call this layer the *application middleware* and its interface the *Application Programming Interface (API)* (see Figure 4).

One common scenario where additional domain-specific services are needed is the reuse of legacy code. When a company decides to port a large federated system to the DECOS architecture, it might be too costly and time-consuming to rewrite the complete application software in order to be compliant with the generic high-level services of the PI. As an example we can think of a distributed application that has initially incorporated the CAN protocol as a communication infrastructure. Instead of rewriting the legacy software, a dedicated application middleware can be incorporated that emulates an application programming interface for CAN, by establishing the CAN protocol on top of the generic virtual network service of the PI.

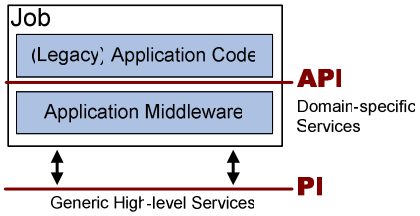

Figure 4: PI versus API

In the following we summarize the advantages that result from moving domain-specific services into the job:

Openness and Configurability: As mentioned above, domain-specific services of a given application middleware are specified and accessed via its API. In contrast to the PI the API is not considered to be single stable interface. Each application middleware provides its own specific set of services, and thus has its own dedicated API which may conform to a proprietary or to a well-established open standard (e.g.TCP/IP or CAN [1]). Since the application middleware is based on the PI which has an open interface specification, third party suppliers are able to develop domain-specific services on their own. Furthermore, the concept of the job-internal application middleware enables the addition and the removal of domain-specific services without changing the PI or the underlying platform.

Error Containment: Since third party developers should be unrestrictedly allowed to provide additional domain-specific services, theses services can not always be assumed to be free of software faults. Therefore non-interference and error containment with respect to domain-specific services is a major issue. As mentioned above, a job is the fault-containment region for software faults [13]. By placing the application middleware into the job, the domain-specific services will become part of the job's fault-containment region, and a software fault within a domain-specific service can have an immediate impact exclusively on the jobs that incorporate the corresponding application middleware. This leads to an architecture where a given application middleware can be certified according to the criticality level that is sufficient for the DAS in which it will be actually used, and does not have be certified according to the highest criticality level within the system.

Intellectual Property Protection: In many cases, the protection of intellectual property is a major concern of potential suppliers of domain-specific services. In the DECOS architecture, jobs are the region of IP protection. This means, that it is not required to supply the source-code of the actual application or of the application middleware to the system integrator. Suppliers of domain-specific functions may decide to deliver their application middleware as precompiled object code that can be linked to the actual application.

4 The Platform Interface - PI

The *platform interface (PI)* constitutes the border line between the DECOS platform and the application subsystems (figure 3). It provides the platform's generic high-level services to the application jobs in a standardized way and abstracts over the actual implementation of the platform. This section describes the services that are provided via the PI.

4.1 Virtual Network Service

A virtual network is an encapsulated communication infrastructure for a single DAS [17]. In order to retain the non-interference properties of federated architectures, where each DAS has its own physical network, each virtual network is logically encapsulated so that communication activities in virtual networks of other DASs are neither visible nor have any effect (e.g. performance penalty) on the exchange of messages in the virtual network. Furthermore each virtual network owns its independent name space. Due to these encapsulation properties, DASs can be independently developed verified and integrated in the system with respect to their communica-

tion activities. A job accesses a virtual network via so-called *ports*. A port through which messages are received is called *input port*, and a port through which messages are sent is called *output port*. The sum of all input ports and output ports through which a job is attached to a virtual network is called *link*.

Multiple virtual networks can be realized as logical overlay networks on top of a single physical network. In the DECOS integrated architecture virtual networks are provided on top of the time-triggered core communication service of the base architecture, where encapsulation is achieved by hierarchic subdivision of the time slots in the communication schedule [17]. In order to support the transmission of information with state semantics and with event semantics, generic time-triggered virtual networks and generic event-triggered networks are provided.

4.1.1 Generic time-triggered virtual networks

Time-triggered virtual networks are designed for the periodic exchange of state messages. Because of the advantages of the time-triggered control paradigm with respect to predictability, error detection, fault tolerance, state synchronization, and replica determinism [8] all virtual networks for safety critical DASs are strictly time-triggered. Jobs interface a virtual time-triggered network via the temporal firewall concept [12] where the sender acts according to the *information push principle* and the receiver according to the *information pull principle* [6].

The time-triggered virtual network service is realized by *state-message ports*, where each state-message port is conceptually a singe buffer that holds a single state message. Whenever the application writes a new state message to an output port, the old state that is stored in that port is overwritten with the contents of the new message. The generic time-triggered virtual network service periodically transmits the actual contents of the state-message output ports of each sender to the corresponding state-message input ports of the receivers, at statically defined points in time. Whenever a new state message is received at a state-message input port, the old content that was stored in that input port is overwritten with the new state message (state messages are not queued). The following functions are used to access a generic time-triggered virtual network:

- void PI_write_TT(link, out_port, state_msg);
  This function updates the state message in the specified port of the specified link.
- PI_state_msg PI_read_TT(link, in_port);
  This function reads the current value of the state message located in the specified port of the specified link.
- PI_temp_acc PI_get_temporal_accuracy(link, in_port);
  This function returns a *temporal accuracy* value which indicates for how long the state message in the specified input port will be still valid to use. Each state message can be associated with a validity time span that indicates for how long the message will be *temporally accurate* after it has been received. If the sender job fails to update the state message within a specified time interval, the state information at the receiver job gets temporally inaccurate. The concept of temporal accuracy is described in [9].

4.1.2 Generic event-triggered virtual networks

Event-triggered virtual networks are designed for the *sporadic* exchange of event messages. Compared to time-triggered networks they offer higher flexibility with respect to resource allocation. The provision of resources can be optimized towards the average demands of an application which results in more cost-effective solutions.

Since each event may be significant and event-messages are generally not idempotent, event messages have to be processed with *exactly-once semantics* (i.e. every event message that has been sent by a sender job has to be received exactly once by each corresponding receiver job). Therefore event-message ports are realized as queues that can hold multiple messages. Whenever the application writes a new message to an event-message output port the new message is enqueued in that output port (the message is discarded if the queue is full). The generic event-triggered network service transports messages from the event-message output ports of each sender job to the corresponding input ports of the receiver jobs with a guaranteed bandwidth. If a new message is received at a event-message input port, it is enqueued in the input port's queue (the message is discarded if the queue is full). Whenever a job reads a message of an event-message input port, the message is removed from the port's queue in order to establish exactly-once processing. The following functions are used to access a generic event-triggered virtual network:

- bool PI_send_ET(link, out_port, event_msg);
  This function tries to enqueue a new mes-

sage in the queue of the specified port of the specified link. If the queue is not full the message in enqueued, and the value *true* is returned. If the queue is full (i.e. an overflow has occurred), the new message is discarded and the value *false* is returned.

- bool PI_read_ET(link, in_port, *event_msg_p);
  This function tries to dequeue the next message from the queue of the specified port of the specified link. If the queue is not empty, the function returns *true*, dequeues the next message of the queue and copies it to the address where *event_msg_p* is pointing to. If the the queue is empty, the function returns *false*.
- bool PI_get_overflow_status(link, in_port);
  The queue of an input port can overflow, if the application job does not call *PI_read_ET* frequently enough. The function *PI_get_overflow_status* returns the status of the specified input port with respect to overflows. *True* is returned when one or more overflows have occurred since the last invocation of the function *PI_get_overflow_status*. Otherwise *false* is returned.

4.2 Global Time Service

In a distributed system, a global notion of time is the prerequisite to coordinate the actions of the nodes in the temporal domain. According to [10] a digital time format can be characterized by the three parameters, *granularity*, *horizon* and *epoch*. The granularity determines the minimum interval between two adjacent ticks of a clock. The horizon determines the instant when the time will warp around, and the epoch determines the instant when the measuring of time starts.

The DECOS integrated architecture uses a uniform 64 bit long time format which is closely related to the GPS time format. It has been standardized by the OMG in the *smart transducer interface standard* [18]. The smart transducer time format has a granularity of 2-24 seconds (about 60 nanoseconds) and a horizon of 240 seconds (more than 30000 years). The epoch starts with the epoch of the GPS time i.e., January 1980. The fault-tolerant global time service can be accessed with the following function:

- PI_global_time_t PI_get_global_time();
  This function returns the actual global time. *PI_global_time_t* is a 64 bit data type that represents an instant of time in the standardized OMG format.

4.3 Consistent Membership Service

The membership service provides consistent information about the operational state (correct or faulty) of the nodes of the integrated system. It is based on the a priori knowledge about the points in time of the time-triggered message exchanges on the core network.

The consistent membership service operates exclusively at node-level. It guarantees that each correct node has a consistent view of the "health-state" of every other node within the cluster ([11]). It is further guaranteed, that all nodes that belong to an agreed membership group have received every message within the cluster consistently. Due to this consistency property each correct node can be sure that every other correct node works exactly on the same data, which is crucial if jobs on different nodes work together in a joint action. The consistent membership service can be accessed with the following function.

- PI_membership_t PI_get_membership();
  This function returns the actual health state of every node within the system. *PI_membership_t* is a bit vector where each bit represents the health state of a specific node.

5 Implementation

In order to show the feasibility of the DECOS integrated architecture, a prototype implementation of a DECOS cluster was developed in [7] (see Figure 5). The focus of this section is on the implementation of the domain-specific services.

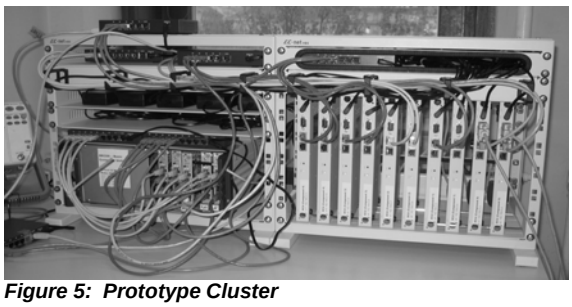

Figure 5: Prototype Cluster

5.1 Core Services

Our prototype implementation uses the *Time-triggered Architecture (TTA)* [11] as a *base architecture* to establish the DECOS core services (time-triggered message transport, fault tolerant clock synchronization, strong fault isolation, and consistent diagnosis of failing nodes). The time-triggered core communication service is pro-

vided by TTTech monitoring nodes [2] which are based on the TTP-C2 controller (AS8202) and are equipped with the Freescale embedded PowerPC processor MPC855T.

5.2 Generic High-level Services

The generic high-level services and the partitions for the jobs of a DAS are executed on a compact communication computer of type Soekris Engineering net4521 [1] which is based on a 133Mhz 468 class ElanSC520 processor from AMD. This compact computer has up to 64 Mbyte SDRAM main memory, uses a CompactFlash module for program and data storage, and is equipped with two 10/100 Mbit Ethernet ports. The interconnection between the Soekris net4521 and the TTTech monitoring node is realized using time-triggerd Ethernet.

Within the Soekris Engineering net4521 the real-time Linux variant *Real-Time Application Interface* (RTAI) [4] [19] is used together with the LXRT extension to provide a *time-triggered execution environment* for temporal and spatial partitioning for the generic high-level services and the partitions for the application jobs.

In order to guarantee temporal and spatial partitioning between jobs with respect to the communication, each job is given a dedicated view of the PI. This is realized by the provision of a dedicated memory area for each job through which the job exclusively accesses its own communication link.

More detailed information about the implementation of the generic high-level services can be found in [7], [16], and [15].

5.3 Domain-specific Services

As an example for a domain-specific service, an application middleware has been realized that provides the CAN protocol layer [1] in order to allow the integration of CAN legacy code. This chapter describes the CAN application middleware with respect to the message-handling capabilities and the implementation. The described application middleware supports BasicCAN [3] and FullCAN [3] transmit and receive behavior, on top of the generic event-triggered virtual network service. The CAN application middleware consists of the CAN middleware task and the CAN API library (see Figure 7) which will be described in the following subsections.

[1] http://www.soekris.com

5.3.1 Message Handling

The CAN application middleware uses its own data structures for message handling. Messages that are to be received by the application are temporary stored in *receive queues* or *receive mailboxes*. In a receive queue, messages are stored in the order of their reception (FIFO principle), whereby the messages can have different identifiers. Each receive queue is equipped with a dedicated configurable message filter for reception filtering. Overflows in a queue occur whenever a new message is received and the considered queue is full. In this case the incoming message will be discarded, and the whole queue will be marked by an overflow flag with can be read by the application.

Contrary to a receive queue, a *receive mailbox* stores only the last instance of a message that matches a particular identifier. Receive mailboxes are recommended for applications where jobs exchange messages with state semantics. A read access to a receive mailbox will return the content of the mailbox, and the number of updates since the previous read access. This feature gives the user the possibility to see whether the data in the mailbox is new and how many updates have been missed.

Messages that are sent by the application are temporary stored in the *transmit queue*. The transmit queue is periodically serviced in order to disseminate the temporary stored messages via the virtual network.

An alternative method for sending messages is the use of *remote mailboxes*. Messages that are entered in a remote mailbox are not sent immediately, but only on request by another node. Other nodes can issue such a request by sending a remote frame.

5.3.2 Underlying Virtual Network

The CAN application middleware is realized on top of the *generic event-triggered virtual network service*, which is provided natively via the DECOS platform interface (PI). The generic event-triggered virtual network service provides multiple encapsulated virtual networks on top of a shared physical network.

In order to be compliant to the CAN protocol, a dedicated virtual network was configured to work in broadcast topology (see Figure 6). In each generic event-triggered virtual network every job has a dedicated input port for each job from which it can receive messages (i.e. for each sender job). The concept of separate input ports

was chosen in the DECOS architecture in order to achieve error containment with respect to the communication channels. A faulty job that sends messages with a higher frequency as defined in its specification can only cause queue overflows in its corresponding input ports at the receivers.

Incoming messages are accessed via the identifier of the input port. Since each input port corresponds to a specific sender job, messages of specific sender jobs can be selectively received (i.e. messages can be filtered with respect to their senders).

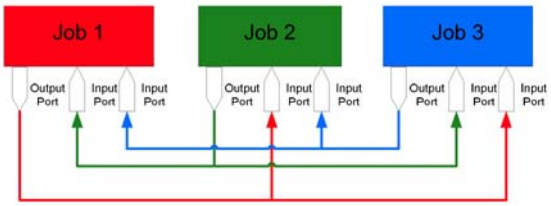

Figure 6: Topology of the Generic Event-triggered Virtual Network for CAN

5.3.3 CAN Middleware Task

In contrast to the generic event-triggered virtual network provided by the PI, the CAN protocol does not perform message filtering with respect to the sender of a message. In CAN, messages are typically filtered with respect to the message identifier. The CAN middleware task maps the filtering method of the generic event-triggered network (message filtering with respect to the sender) to the filtering method typical for CAN (message filtering with respect to the message identifier). It provides the CAN specific receive queues, transmit queues, and mailboxes (receive mailboxes and transmit mailboxes) and layers them on top of the generic event-triggered network (see Figure 7).
The CAN middleware task is periodically activated and scans at each activation each input port of the generic event-triggered virtual network for new messages. Whenever a new message is received via an input port it is handed over to each CAN receive queue and each CAN receive mailbox in case of a CAN data message or to each CAN remote mailbox in case of a CAN remote frame. After each input port of the generic event-triggered virtual network is scanned, the CAN transmit queue is served in order to copy pending messages to the output port of the generic event-triggered network.

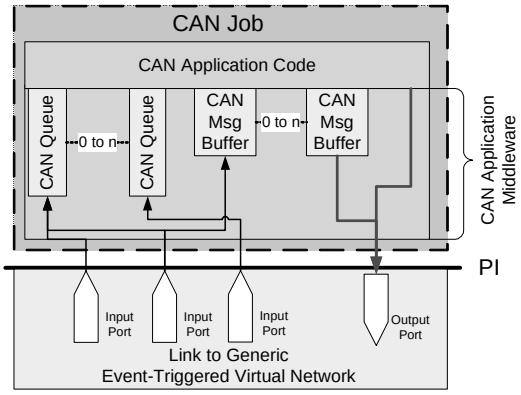

Figure 7: Application Middleware for CAN

5.3.4 CAN API Library

The CAN API library contains all functions that are required to access the queues and mailboxes which are generated by the CAN middleware task, and is linked to the actual application code. The application code plus the linked CAN API are executed as a separate task which is called the CAN application task (see Figure 7). The queues and mailboxes are realized in a shared memory area and can be accessed by the CAN middleware task and the CAN application task. The CAN application task and the CAN middleware task constitute together a CAN job.

5.3.5 Synchronization

This section describes the synchronization between the task realizing the generic event triggered virtual network, the CAN middleware task and the CAN application task.

The middleware task realizing the *generic event-triggered virtual network* and the *CAN middleware task* are implicitly synchronized by *the time-triggered scheduler* of the *execution environment*. Both are periodically activated at statically defined points in time. Based on the specified minimum message inter-arrival times, the appropriate queue sizes of the input ports of the generic event-triggered virtual network and the activation period of the CAN middleware task are dimensioned in a way that overflows at the input ports of the generic event-triggered virtual network are avoided. Overflows at the CAN receive queues have to be handled by the application programmer of the considered CAN job.

The synchronization between the between the CAN Middleware Task and the CAN Application Task is more complex. The CAN middleware task is a time-triggered task, with a worst case execution time that is smaller than its dedicated

time slot. At each invocation, it starts its sequence of operations on the queues and mailboxes and completes this sequence within its dedicated time-slot without being preempted. Thus, all operations on the queues and mailboxes that are performed by the CAN middleware task are inherently atomic.

Contrary to the program flow of the CAN middleware task, the program flow of the CAN application task is generally not synchronized with the points of time of its preemption and resumption that are enforced by the time-triggered scheduler (legacy CAN code cannot be assumed to be written according to the time-triggered paradigm). Due to the missing synchronization, the CAN application task can be preempted while it is performing a write or read operation on the queues or on the mailboxes. Thus, operations on queues and on mailboxes that are performed by the CAN application task cannot be assumed to be atomic.

Accesses to the CAN queues (receive and transmit) are synchronized by means of variables for the write position and the read position of a given queue (explicit synchronization). The CAN middleware task indicates the presence of a new *input message* in a *receive queue* by increasing the write position variable of the receive queue after it has written the message into the queue. The CAN application task indicates the consumption of a message by increasing the read position variable of the receive queue after it has consumed the message from the queue. The same principle is used for the *transmit queue*. The CAN application task indicates the presence of a new *output message* in the *transmit queue* by increasing the write position variable of the transmit queue after it has written the message to the queue. The CAN middleware task indicates the consummation of an output message by increasing the read position variable of the transmit queue after it has consumed the message from the queue.

In order to synchronize accesses to CAN mailboxes, each mailbox incorporates a sequencer variable which can be exclusively written by the CAN middleware task, and a valid flag which can be exclusively written by the CAN application task.

When the CAN application task performs a read operation on a mailbox it can happen, that it is preempted during the read operation and that the CAN middleware task updates exactly that mailbox that the CAN application task is currently reading. In this case the CAN application task would read inconsistent data. In order to detect such a condition, a protocol similar to the Non-Blocking Write protocol (NBW) [9] is used: The sequencer variable of a mailbox is initialized to zero at the system start-up phase. It is incremented by the CAN middleware task each time after it has updated the content of the mailbox. Due to the fact, that the CAN middleware task is a time-triggered task that completes always its whole sequence of operations without preemption, updating the mailbox and incrementing the sequencer variable can be regarded as one atomic operation. When the CAN application task performs a read operation on a mailbox, it always starts by reading the mailbox's sequencer variable. At the end of the read operation the CAN application task checks whether the sequencer variable has been changed by the CAN middleware task during the read operation. If so, the read operation is retried until uncorrupted data is retrieved from the mailbox.

When the CAN application task performs a write operation on a mailbox it can happen, that it is preempted during the write operation and leaves the contents of the mailbox in an inconsistent state until it is resumed again and finishes the write operation. In order to inform the CAN middleware task about the consistency or inconsistency of a mailbox the CAN application task uses the mailbox's valid flag. It sets the flag to invalid before each write operation and resets the flag to valid after the write operation is completed. Thus the CAN middleware task can always verify the integrity of a CAN mailbox. The CAN middleware task treats each invalid mailbox as a temporary non-existent mailbox (the mailbox will not be used for any receive or send operations until the next activation of the can middleware task *after* the can application task has finished its write operation and has set the flag to valid again).

As far as the CAN application task is concerned, all the synchronization activities are performed by the CAN API library and are *transparent* to the actual application code. Thus, legacy application code does not have to be modified in order to establish synchronization with respect to the CAN middleware.

6 Conclusion

The DECOS architecture is intended to integrate multiple application subsystems of heterogeneous application domains into a single distributed computer system. Each application domain has its own specific demands and requirements with

respect to the underlying architecture (e.g. different real-time requirements and communication paradigms, different criticality levels). Consequently a wide variety of architectural services with different characteristics has to be provided in order to support the development and the integration of such heterogeneous applications.

In this work we have devised an architecture that structures the architectural services hierarchically into three distinct layers: The first layer realizes core services which are provided by the underlying time-triggered communication architecture. The second layer provides a fixed and stable set of generic high-level services that supports the independent development of distributed application subsystems. These high-level services include virtual network services that provide a dedicated encapsulated communication infrastructure for each subsystem, a global time service for the coordination of distributed applications and a reliable membership service that provides consistent information about the operational state (correct or faulty) of the nodes within the distributed system. The third layer realizes an open and scalable set of domain-specific services that can be unrestrictedly extended by third party suppliers. Since domain-specific services can be freely added, they can generally not be assumed to be free from design faults. Therefore they are placed into partitions which are fault-containment regions for software faults. In order to support the protection of intellectual property, domain-specific services can be integrated as precompiled object code.

To demonstrate the feasibility of our approach, a prototype of a DECOS cluster was implemented that provides all specified generic high-level services. As an example for a domain-specific service, an application middleware was implemented that provides a CAN protocol layer that supports the integration of legacy CAN application code.

Acknowledgments

We would like to acknowledge the whole DECOS team for valuable discussions on the topics of this paper. This work has been supported in part by the European IST project DECOS (IST-511764) and the European IST project ARTIST2 (IST-004527).

References

[1] CAN Specification, Version 2.0. Robert Bosch Gmbh, Stuttgart, Germany, 1991.

[2] TTP Monitoringnode - A TTP Development Board for the Time-triggered Architecture. TTTech Computertechnik AG, 2002.

[3] CAN history. CAN in automation (CiA), 2004. Available at http://www.can-cia.org/can/protocol/history/history.html.

[4] D. Beal, E. Bianchi, L. Dozio, S. Hughes, P. Mantegazza, and S. Papacharalambous. RTAI: Real-Time Application Interface. Linux Journal, April 2000.

[5] A. Deicke. The electrical/electronic diagnostic concept of the new 7 series. In Convergence International Congress & Exposition On Transportation Electronics, Detroid, USA, Oct. 2002.

[6] W. Elmenreich, W. Haidinger, and H. Kopetz. Interface design for smart transducers. In IEEE Instrumentation and Measurement Technology Conference, volume 3, pages 1642–1647, Budapest, Hungary, May 2001.

[7] B. Huber, P.Peti, R. Obermaisser, and C. E. Salloum. Using RTAI/LXRT for partitioning in a prototype implementation of the DECOS architecture. In Proceedings of the Third International Workshop on Intelligent Solutions in Embedded Systems, pages 3–16, 2005.

[8] H. Kopetz. Why time-triggered architectures will succeed in large hard real-time systems. In Proceedings of the 5th IEEE Computer Society Workshop on Future Trends of Distributed Systems, August 1995.

[9] H. Kopetz. Real-Time Systems, Design Principles for Distributed Embedded Applications. Kluwer Academic Publishers, Boston, Dordrecht, London, 1997.

[10] H. Kopetz, A. Ademaj, P. Grillinger, and K. Steinhammer. The time-triggered ethernet (TTE) design. In 8th IEEE International Symposium on Object-oriented Real-time distributed Computing, May 2005.

[11] H. Kopetz and G. Bauer. The time-triggered architecture. IEEE Special Issue on Modeling and Design of Embedded Software, Jan. 2003.

[12] H. Kopetz and R. Obermaisser. Temporal composability [real-time embedded systems]. Computing & Control Engineering Journal, 13(4):156–162, 2002.

[13] H. Kopetz, R. Obermaisser, P. Peti, and N. Suri. From a federated to an integrated architecture for dependable embedded real-time systems. Technical Report 22, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2004.

[14] J. Moore. Arinc 629, the civil aircraft databus for the 1990s. In IEE Colloquium on Time Critical Communications for Instrumentation and Control, pages 5/1–5/2, 1989.

[15] R. Obermaisser and P. Peti. Specification and execution of gateways in integrated architectures. In Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'05), Italy, 2005.

[16] R. Obermaisser, P. Peti, and H. Kopetz. Virtual gateways in the DECOS integrated architecture. In Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium, pages 134a–134a, 2005.

[17] R. Obermaisser, P. Peti, and H. Kopetz. Virtual networks in an integrated time-triggered architecture. In Proceedings of the 10th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS2005), pages 241–253, Sedona, Arizona, Feb. 2005.

[18] Smart transducers interface v1.0, Jan. 2003. Specification available at doc.omg.org/formal/2003-01-01 as documentptc/2002-10-02.

[19] RTAI Programming Guide, Version 1.0. Dipartimento di Ingegneria Aerospaziale Politecnico di Milano (DIAPM), Italy, September 2000. Available at http://www.rtai.org.

[20] J. Rushby. A comparison of bus architectures for safety-critical embedded systems. Technical report, Computer Science Laboratory, SRI International, Sept. 2001.

[21] A. Sangiovanni-Vincentelli. Defining platform-based design. EEDesign of EETimes, February 2002.