Towards a Middleware Approach for a Self-configurable Automotive Embedded System

Isabell Jahnich[1], Ina Podolski[1], and Achim Rettberg[2]

[1] University of Paderborn/C-LAB, Germany,
[2] Carl von Ossietzky University Oldenburg, Germany
isabell.jahnich@c-lab.de, ina.podolski@c-lab.de,
achim.rettberg@informatik.uni-oldenburg.de

Abstract. In this paper a middleware architecture for distributed automotive systems that supports self-configuration by dynamic load balancing of tasks is presented. The inclusion of self-configurability is able to offer reliability within the multimedia network of the vehicle (Infotainment). Load balancing of tasks could be applied if an error occurred within the network. The error detection in the network and the load balancing should run automatically. Therefore, the middleware architecture has to deal on one hand with the error detection and on the other hand with the migration of tasks. Additionally, to enable the migration it is important to identify the requirements of all electronic control units (ECU) and tasks within the network.

1 Introduction

Future application scenarios for vehicle electronic systems include on one side the access to mobile devices that build ad-hoc networks with the built-in devices of the vehicle and on the other side the support of robustness, redundancy and dependability within the vehicle network. Modern electronic vehicle networks have to be as flexible as possible to cope the actual requirements. The idea to build up a self-configurable system could help to overcome these requirements. A mobile device could automatically be attached and integrated in the existing system if the system supports self-configurability. If an ECU has a failure all task could be migrated to other ECUs inside the vehicle network by a self-configurable system middleware. Self-configuration could be applied to distributed networks. In modern vehicles three types of networks are built in. That is namely the *Powertrain*, *Body* or *Chasis* and *Infotainment* network. Within the *Powertrain* safety critical tasks like the anti-blocking system and the motor management are located. The *Body* or *Chasis* network contains also critical tasks, but the vehicle will run if a failure occurs. The window opener is a typical task of this network. The *Infotainment* network consists of more or less media based tasks, like the radio or navigation system. Due to safety critical reasons our approach will focus on the *Infotainment* network.

To increase the quality of the vehicle it is important to built in fault-tolerant systems in the network. In a distributed system fault-tolerance can be include in three ways: Replication, redundancy, and diversity. While the former provides multiple

U. Brinkschulte, T. Givargis, and S. Russo (Eds.): SEUS 2008, LNCS 5287, pp. 55–65, 2008.

identical instances of a system, the tasks and requests are directed to all of them in parallel, and the choosing of the correct result is based on a quorum, redundancy is characterized by multiple identical instances and a switching to one of the remaining instances in case of failure. Diversity provides different implementations of a system that are used like replicated systems.

A self-configurable system is able to provide redundancy, diversity and replication of tasks, therefore, it helps to make the system more stable.

In the context of self-configuration of automotive systems redundancy of data, applications, and tasks can be used to get an increased fault-tolerance in case of ECU failures. Crucial data, applications or tasks are distributed as backup components on the ECUs of the vehicle system that they can be used of or executed by other ECU if their originally ECUs failed.

The way of distribution and the number of replicas and the decision which components are replicated depends on an adequate algorithm. At this costs of replication and migration and load of other ECUs will be considered.

In the following a vehicle middleware architecture is presented that supports self-configuration by load balancing strategies for non-critical tasks within the *Infotainment* network. In our case we enable a dynamic reconfigurable system by load balancing. In existing approaches self-configuration is enabled, by including redundancy and replication of tasks during design time. This is a static system reconfiguration. Furthermore, our middleware offers services to realize a load balancing based on different strategies for the *Infotainment* network. This work is part of the DySCAS project (ref.). The main objective of the DySCAS project is the elaboration of fundamental concepts and architectural guidelines, as well as methods and tools for the development of self-configurable systems in the context of embedded vehicle electronic systems. The reason is the increasing demand on configurational flexibility and scalability of the systems imposed by future applications which will include simultaneous access to a number of mobile devices and ad-hoc networking with the built-in devices.

The rest of the paper is organized as follows: Section 2 will describe the related work in the field of research where our architectural approach is located. As a motivation for this paper Section 3 motivates and describes a use case scenario. Afterwards our middleware architecture is presented, see Section 4. In Section 5 we describe the load balancing strategy we use within the middleware. A short description of the simulation and some early results are discussed in Section 6. We conclude the paper with a summary and give an outlook for future work.

2 Related Work

In this section we will give a short overview of existing load balancing approaches to support self-configuration and on middleware approaches in automotive systems.

There are several publications regarding load balancing and extensive research has been done on static and dynamic strategies and algorithms [6].

On the one hand, load balancing is a topic in the area of parallel and grid computing, where dynamic and static algorithms are used for optimization of the simultaneous task execution on multiple processors. Cybenko addresses the dynamic load balancing for distributed memory multiprocessors [3]. In [5] Hu et. al. regard an optimal dynamic algorithm and Azar discusses on-line load balancing. Moreover Diekmann et. al. differentiate between dynamic and static strategies for distributed memory machines [4]. Heiss and Schmitz introduce the Particle Approach that deals with the problem of mapping tasks to processor nodes at run-time in multiprogrammed multicomputer systems solved by considering tasks as particles acted upon by forces.

All these approaches have the goal of optimizing the load balancing in the area of parallel and grid computing by migrating tasks between different processors, while our approach focuses the direct migration of selected tasks to a newly added resource. Furthermore we regard load balancing that is located on the middleware-layer.

Moreover there are static approaches, like [11], that address a finite set of jobs, operations and machines, while our approach deals with a dynamic set of tasks and processors within the vehicle system.

Balasubramanian, Schmidt, Dowdy, and Othman consider in [7], [9], and [8] middleware load balancing strategies and adaptive load balancing services. They introduce the Cygnus, an adaptive load balancing/monitoring service based on CORBA middleware standard. Their concept is primarily described on the basis of a single centralized server, while decentralized servers that collectively form a single logical *Load Balancer* is not explained in detail.

Moreover the topic of dynamic reconfigurable automotive systems is regarded in [2], [1], [13] and [14]. In the following paragraphs we discuss several middleware approaches for automotive systems.

The Autosar consortium [19] suggested a middleware approach based on a runtime environment (RTE). The RTE is developed to support a common infrastructure for automotive systems. The self-configurability developed in our approach will enrich the Autosar RTE especially by dynamic reconfiguration management through load balancing.

In [15] a formal specification for developing distributed, embedded, real-time control systems is described. The middleware supports dependable, adaptive dynamic resource management based on replicated services.

An additional approach according fault-tolerance and dynamic reconfiguration is discussed in [16]. Again replicated services are used in this model. In [17] a middleware architecture for telematics software based on OSGi and AMI-C specification is presented. An application manager is introduced for telematic applications. The architecture enable in-vehicle terminal to provide various telematics services to increase driver's safety.

The authors of [18] describe trends for automotive systems. They give an overview of requirements for middleware systems in this area. Especially what industry demands for such middleware services. Hiding the distribution and the heterogeneity of such platforms is demanded as well as providing high-level services (e.g. mode and redundancy management) and ensuring QoS.

3 Motivation

In this section we give a motivation for our approach. We identify three use cases:

- Task fails on an ECU and have to migrate to another one
- ECU has a defect - all task will be migrated
- New device is attached to the network

As an example we will use the second use case. If an ECU of the vehicle *Infotainment* system failed, a migration to another ECU within the vehicle that is able to execute the applications or tasks should be possible. Thus it is possible to migrate for example tasks of the ECU with the radio system to the ECU running the navigation system.

After the failure occurred within the vehicle the system starts a self-reconfiguration without avoiding overloading ECUs. The self-reconfiguration is surely based on specific characteristics from the tasks and the ECUs. That means, it has to be ensured that a task could only run on an ECU that is able to execute it.

In consideration of all running processes and the resources situation within the vehicle network appropriate services decide on a possible load balancing according to different strategies and initiate the task migration where required. Thus in our example where an error occurred inside the radio system the appropriate tasks migrate from the radio to the navigation system. Let us assume that the navigation system respectively the ECU is able to run the tasks from the radio system.

4 Proposed Middleware Architecture

To realize the use case scenario (failure in the radio system) described above and other possible services for example device detection a middleware architecture is required that fulfills several requirements. We introduce four sub-modules to handle self-configuration in the middleware. The *Event Management* detects failures in the vehicle network and it is responsible for detection and removal of additional ECUs. Detailed information and capabilities of existing ECUs as well as the registration of newly added devices is realized within the *Device Registration* module. All status information and the resource load of each ECU within the vehicle are stored by the *Resource Management*. Finally, the Load Balancing initiates the task migration based on specific characteristics and requirements of the tasks and ECUs. In the following we give a more detailed view of the middleware.

The operating system builds the interface between the hardware and the middleware (see Figure 1). Additionally, device drivers are necessary for specific hardware parts. The tasks run on top of the middleware. Middleware is a software layer that connects and manages application components running on distributed hosts. It exists between network operating systems and application components. The middleware hides and abstracts many of the complex details of distributed programming from application developers. Specifically, it deals with network communication, coordination, reliability, scalability, and heterogeneity. By virtue of middleware, application developers can be freed from these complexities and can focus on the application's own functional requirements.

Before explaining the design of our automotive middleware and the specific services, we enumerate the five requirements of automotive middleware. These requirements are resource management, fault-tolerance, and specialized communication model for automotive networks, global time base, and resource frugality. These requirements are derived from the distributed, real-time, and mission-critical nature of automotive systems and differentiate automotive middleware from conventional enterprise middleware products.

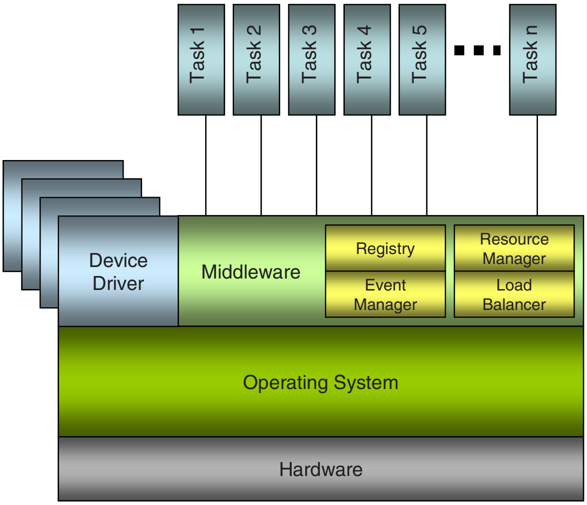

Fig. 1. Self-configurable architecture

A vehicle has a real-time nature. It is a system in which its correctness depends not only on the correctness of the logical result, but also on the result delivery time. Since a vehicle is subject to various timing constraints, every component in a vehicle should be designed in a way that its timing constraints are guaranteed a-priori. At the same time, the timing constraints of a vehicle should be guaranteed in an end-to-end manner since an automobile is a distributed system and its timing constraints are usually specified across several nodes. For example, let us consider a typical timing constraint of an automobile. If pressing a brake pedal is detected at the sensor node, then the brake actuator node must respond to it within 1 ms. To meet this constraint, there must be a global *Resource Manager* that calculates the required amount of resources on each node and actually makes resource reservations to network interface controllers and operating systems on distributed nodes. Automotive middleware is responsible for such resource management.

The middleware in our approach includes four components that offer specific services: *Registry*, *Event Manager*, *Resource Manager* and *Load Balancer*.

The *Event Manager* is responsible for the failure detection and the device discovery. If a failure occurred the Event Manger triggers the *Load Balancer* to initiate a feasible migration of tasks. Additionally, if a new device is added to the automotive system via technologies like Bluetooth or WLAN for example, it is recognized by the *Event Manager* component. Vice versa the *Event Manager* also notices the detaching of the device. In both cases it will inform the *Registry* of the middleware about the availability or the detaching of the additional device.

Existing and new devices are registered and detached devices are unsubscribed within the *Registry* service. During the registration the specific characteristics of the device (like memory, CPU, etc.) are stored within the *Registry*. Due to the distributed system the Registries of each vehicle ECU (Electronic Control Unit) communicate with each other to guarantee that each *Registry* of an ECU knows the actual status of all devices within the network inclusive of the newly added devices.

The *Load Balancer* spread tasks between the vehicle ECUs in order to get optimal resource utilization and decrease computing time. It evaluates possible migration of tasks based on different load balancing strategies. To guarantee a suitable migration the *Load Balancer* considers the current resource situation on the ECUs with aid of the *Resource Manager*. If a failure is occurred the *Load Balancer* tries to find based on the characteristics of the tasks and ECUs a feasible migration. Once a load balancing on an additional device is started, and this device is detached while the migrated tasks are executed, they will be re-started on the original ECU again. In this case the *Event Manager* is responsible to inform the *Load Balancer* to initiate this re-start.

The *Resource Manager* supervises the resources of the local ECU. To be aware of the complete network resource situation all *Resource Manager*s synchronize with each other. Thus the *Load Balancer* gets the current resource situation of the complete vehicle infrastructure with aid of its local *Resource Manager*.

In our approach, the middleware is located on each ECU in the vehicle. Every ECU has a unique ID. The ECU with the lowest ID is the master. Thus it is responsible for the control of the entire vehicle network, and newly connected and the detaching of additional devices are discovered by its *Event Manager*, device information is registered by its *Registry*, and its *Load Balancer* is responsible for the evaluation of the possible migration with the aid of the local *Resource Manager*. If the master ECU fails a new master will be chosen with the aid of the Bully-Algorithm [10].

The failure detection if a node fails will be handled by a hardware interrupt. It initiates an error correction in our middleware. That means, to correct the error, tasks of the omitted node are migrated to other ones, which are able to execute them. In this paper we will not focus on the failure detection but on error correction. Therefore, our middleware must be able to migrate tasks. A detailed knowledge of the task characteristics is needed. It is important to know if it is a real-time task or not.

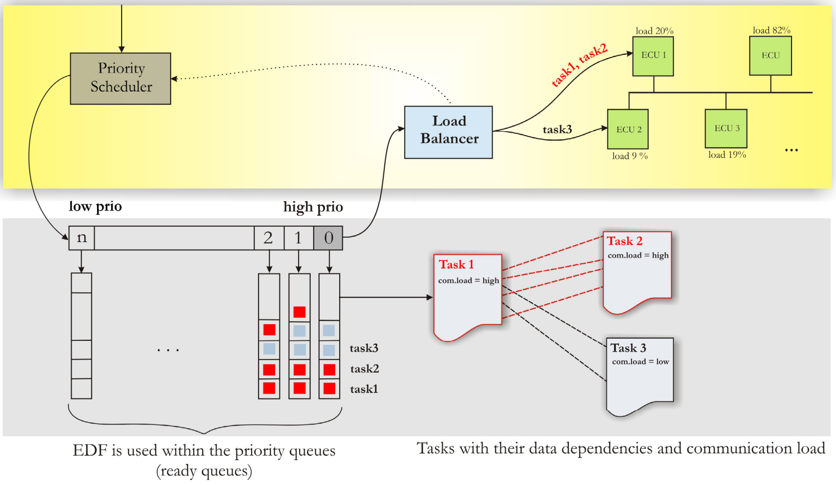

Fig. 2. Failure correction handling - the task migration mechanism

Figure 2 presents our approach for task migration. We assume that each task has a priority and we have a detailed knowledge about their hardware requirements. Additionally, the data dependencies between the tasks are known. As we can see from figure 2 we start with a priority scheduler. He will schedule the tasks according their priority in priority queues. That means, we have for each priority an own task queue. Within the queues the tasks are scheduled by a simple earliest deadline first (EDF) scheduler to ensure a flexible schedule [12]. Real-time (RT) tasks have a high priority. The *Load Balancer* works on the priority queues beginning from the queue with the highest up to the lowest priority. For each selected task a possible set of ECUs who are able to execute the task is evaluated. After that a data dependency check will be done. That means, we look at those tasks that interact with the inspected one. In that case the interaction is weak the *Load Balancer* selects an ECU from the previously evaluated set of ECU and finally migrate the task to that one and delete the task in the priority queue. In case of a strong interaction the *Load Balancer* will try to avoid unnecessary busload, by selecting an ECU from the ECU set that is able to execute both tasks. Afterwards both tasks will be deleted in the priority queue. If the *Load Balancer* could not find a possible ECU for migration the task will be deleted from the queue with the outcome that a migration is not possible.

The previous paragraph give an overview of the migration, but there are still some open issues we will discuss in the following. If an ECU with more than one task running on it fails we will migrate the tasks to one or more ECUs according the classification of the tasks (see Figure 2). That means tasks with high-priority will migrated first followed by the other ones. During the migration phase the timing of the tasks are taken into account. After a task migration we have to decide to start the task new or from that state before the ECU fails, but how to recognize this state? Therefore, we need the context of the task. Our solution is the following, if we have a context available (e.g. store in an external flash memory of the ECU and still available) we will invoke the task with the context, otherwise not. This gives a brief overview how our middleware migrate tasks. Finally the decision which tasks are migrated is done by the *Load Balancer*, see section 6.

Figure 3 shows a sequence diagram where a failure occurred in the radio system. We assume the tasks from the radio system can migrated to the navigation system.

As we can see in Figure 3 the *Event Manager* detects the failure of the radio system, this is done by the function failure_detection(error_code). Afterwards the *Event Manager* triggers the *Load Balancer* with the initialize() function. The *Load Balancer* ask for all device information from the *Registry* (req_loads(*device[0..n])). Then the *Resource Manager* runs the schedule() function to calculate all possible schedules. The *Load Balancer* will get the device information back from the *Resource Manager* with ack_loads(*device[0..n]). Finally the *Load Balancer* will calculate (initiate_load_balancer()) which tasks could be move from device with the failure to another one based on the information of the schedules, the load of each processing element in the car-network, the communication costs and regarding the feasibility. In our case he will decide to move tasks from the radio to the navigation system.

In the last paragraph we describe the interactions between the four tasks, which are necessary to support load balancing. Now we will discuss the internal data structure of our middleware. The *Event Manager* triggers the *Registry* and initialize the *Load Balancer*. The *Registry* itself interacts with the *Resource Manager* and the *Load Balancer*. The *Resource Manager* hands over the actual status of the entire system to the *Load Balancer*.

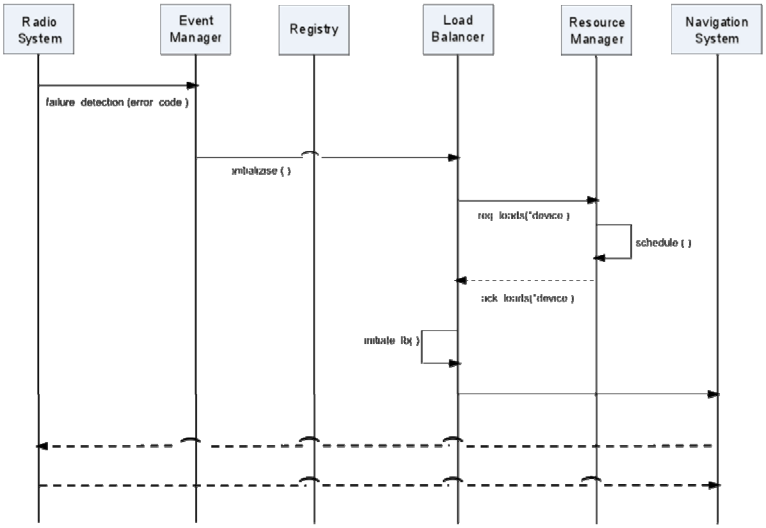

Fig. 3. Failure detection of the radio system

To perform the scheduling in the *Resource Manager* we can select between different scheduling strategies. They are instantiated within the scheduling mechanism class of the internal data structure.

The *Registry* as well as the scheduling mechanism needs information's of all tasks and devices. This is handled by the so called list class. It contains linked lists of devices and tasks and offers functions to manage the lists. As described before list offers all functions to manage the task list, but additionally functions to set the status of the tasks are needed. The status of the task is running, waiting or sleeping. Besides this the task manager is able to create a new task. The information of a task is stored in the data structured provided by the task control block. The parameters of the generated structure are set by the task manager with functions from the list class. The list class uses the functions from the task control block to get information's from tasks.

For the devices we have the same functions available as for the tasks. This is realized in the device control block. Each device has a list containing the task-id's that are running on the device. By setting the global variables of our middleware we can initialize the system and can set it in running mode.

5 Load Balancing Strategy

There are several possibilities to balance the load after an error happened inside the vehicle *Infotainment* network. Initiated by the *Load Balancer* component the new resources can be used and applications or tasks can be migrated to the additional device.

In the following the cost-based load balancing strategy is briefly described. Within the cost based strategy the *Load Balancer* evaluates possible migration of tasks from one ECU to another. He evaluates a set of ECUs where the task could be migrated. Hence that the migration is only a useful option if

- the cost of migrating is lower than the cost of keeping tasks with their original device and
- it is feasible to migrate a task or a set of tasks from one ECU to another one (*feasibility*).

The cost benefit ratio for tasks of busy devices is computed which helps the *Load Balancer* to form the decision of whether to migrate or not. The calculation of migration costs of task is realized according to the priority list of the Most Loaded strategy. Most Loaded generates a priority list which ranks the tasks from the busiest processor. In that way the tasks with the highest priority will be migrated to the resources of the additional device.

Let us assume we have tasks t_i with $i = 1 \text{ to } n$, and the utilization of the task running on an ECU is u_i. Additionally, let U_j the maximum utilization of ECU e_j with $j = 1 \text{ to } m$. Then the upper bound for the *utilization* of an ECU e_j is:

$$\sum_{i=1}^{n} u_i \leq U_j$$

For the communication we can make the following assumptions. Let c_k with $k = 1 \text{ to } r$ the communication channels in the vehicle and C_k the maximum costs a channel c_k has. Furthermore, let $m_{i,k}$ the cost task t_i produce on channel c_k. Then we can define the following bound for the *communication cost* a channel c_k:

$$\sum_{i=1}^{n} m_{i,k} \leq C_k$$

Now our *Load Balancer* has to find an optimal balancing for all tasks within in the vehicle network regarding the utilization, communication cost and the feasibility. This can be done with integer linear programming (ILP) or other optimization methods. This is ongoing work right now and in the final version of the paper we will show some simulation results.

6 Simulation and Results

In this section we will describe the implementation status of our middleware w.r.t. simulation and results. Our middleware was implemented in C code. We choose C, because it is more or less the language used for ECUs. Therefore, the code transfer from a PC based simulation to a real target platform doesn't need too much effort.

The implementation follows the class diagram structure presented in the previous section, see Figure 5. Within the PC based simulation we are able to parameterize our virtual software tasks and virtual ECUs with real values to achieve a software simulation of the entire system. The simulation is due to the fact that we use real values, near to the real system behavior.

As we figured out from the simulation that the migration time, to start a task on an ECU needs more time as our implemented scheduling and load balancing approach. Therefore, the time our middleware needs is dominated by the task migration of the underlying hardware (ECUs).

7 Conclusion and Outlook

We presented a middleware architecture for automotive systems that enables dynamic load balancing within the *Infotainment* network. The integration of load balancing is a step towards a self-reconfiguration within the vehicle and to integrate redundancy by task migration. We focus on a specific use case scenario whereby an error occurred within the vehicle network. Tasks running on the ECU with an error are migrates to another ECU by regarding the so-called feasibility, utilization and communication costs. With the help of the requirements, we described the middleware architecture and their enrichment with new services to support the distribution and exchange of tasks. Furthermore, we present briefly a cost-based load balancing strategy we will use for our approach.

Future work will be a detailed evaluation of the already existing load balancing strategies in the context of automotive systems. Additionally, the extension of existing or the development of new load balancing strategies will be done together with the implementation of the proposed architecture.

Acknowledgements

This project was funded by the EU Commission within the project DySCAS (Dynamically Self-Configuring Automotive Systems).

References

[1] Athony, R., Ekelin, C., Chen, D., Törngren, M., de Boer, G., Jahnich, I., et al.: A future dynamically reconfigurable automotive software system. In: Proceedings of the Elektronik im Kraftfahrzeug, Dresden, Germany. LNCS. Springer, Heidelberg (2006)

[2] Athony, R., Rettberg, A., Jahnich, I., Ekelin, C., et al.: Towards a dynamically reconfigurable automotive control system architecture. In: Rettberg, A., Dömer, R., Zanella, M., Gerstlauer, A., Rammig, F. (eds.) Proceedings of the IESS 2007, Irvine, California, USA. Springer, Heidelberg (2007)

[3] Cybenko, G.: Dynamic load balancing for distributed memory multiprocessors. J. Parallel Distrib. Comput. (1989)

[4] Diekmann, R., Monien, B., Preis, R.: Load balancing strategies for distributed memory machines. In: Satz, H., Karsch, F., Monien, B. (eds.) Multiscale Phenomena and Their Simulation, pp. 255–266. World Scientific, Singapore (1997)

[5] Hu, Y.F., Blake, R.J.: An optimal dynamic load balancing algorithm, vol. DL-P-95-011 (1995), http://citeseer.ist.psu.edu/hu95optimal.htm

[6] Hui, C.-C., Chanson, S.T.: Improved strategies for dynamic load balancing. In: IEEE Concurrency (1999)

[7] Jaiganesh, B., Douglas,: Evaluating the performance of middleware load balancing strategies (2004), http://citeseer.ist.psu.edu/635250.html

[8] Othman, O., Schmidt, D.: Optimizing distributed system performance via adaptive middleware load balancing. In: Othman, O., Schmidt, D.C. (eds.) ACM SIGPLAN Workshop on Optimization of Middleware and Distributed Systems (OM 2001), Snowbird, Utah, June 18, 2001 (2001)

[9] Othman, O., Schmidt, D.C.: Issues in the design of adaptive middleware load balancing. In: LCTES 2001: Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems, pp. 205–213. ACM Press, New York (2001)

[10] Stoller, S.: Leader election in distributed systems with crash failures. Technical report, Indiana University, April 1997, p. 169 (1997)

[11] van der Zwaan, S., Marques, C.: Ant Colony Optimisation for Job Shop Scheduling. In: Proceedings of the Third Workshop on Genetic Algorithms and Artificial Life (GAAL 1999) (1999)

[12] Buttazzo, G.C.: Hard real time computing systems. Kluwer Academic Publishers, Dordrecht (2000)

[13] Jahnich, I., Rettberg, A.: Towards Dynamic Load Balancing for Distributed Embedded Automotive Systems. In: Rettberg, A., Dömer, R., Zanella, M., Gerstlauer, A., Rammig, F. (eds.) Proceedings of the IESS 2007, Irvine, California, USA. Springer, Heidelberg (2007)

[14] Jahnich, I., Podolski, I., Rettberg, A.: Integrating Dynamic Load Balancing into the Car-Network. In: 4th Proc. of the Electronic Design, Test and Application (DELTA 2008), Hong Kong, January 23–25 (2008)

[15] Ravindran, B., Welch, L.R., Kelling, C.: Building Distributed Scalable Dependable Real-Time Systems. In: Proceedings of the IEEE Conference on Engineering of Computer-Based Systems, March 24-28 (1997)

[16] Chaaban, K., Shawky, M., Crubillé, P.: A Distributed Framework For Real-Time In-Vehicle Applications. In: Proceedings of the 8th International IEEE Conference on Intelligent Transportation Systems, Vienna, Austria, September 13–16 (2005)

[17] Kim, M., Choi, Y., Moon, Y., Kim, S., Kwon, O.: Design and Implementation of Status based Application Manager for Telematics. In: The 8th International Conference on Advanced Communication Technology (CACT), February 20-22 (2006)

[18] Navet, N., Song, Y., Simonot-Lion, F., Wilwert, C.: Trends in Automotive Communication Systems. In: Proceedings of the IEEE (June 2005)

[19] `http://www.autosar.org`