A Distributed Middleware for Automotive Applications

Christian Wawersich wawi@cs.fau.de
Michael Stilkerich stilkerich@cs.fau.de
Ralf Ellner siraelln@stud.uni-erlangen.de
Wolfgang Schröder-Preikschat wosch@cs.fau.de

Friedrich-Alexander University of Erlangen-Nuremberg
Department of Computer Sciences 4
Martensstr. 1, 91058 Erlangen, Germany

Abstract

Modern cars contain a multitude of micro controllers for a wide area of tasks. The micro controllers are connected to controller networks through different bus systems. The development of a connected and cooperating system is difficult and poorly supported by the existing development tools.

With the KESO *system we have implemented a very small and adapted Java middleware for an OSEK/VDX operating system that allows the safe coexistence of tasks on a single micro controller by providing software-based memory protection.*

In this paper we present our approach on extending KESO to a distributed multi-JVM that provides an integrated view on controller networks. KESO will provide one communication mechanism that transparently allows the communication between different tasks on the same as well as tasks on different micro controllers over different communication systems.

1. Introduction

Modern cars contain a multitude of micro controllers for a wide area of tasks, ranging from convenience features such as the supervision of the car's audio system to safety relevant functions such as assisting the braking system of the car. The different micro controllers are connected to a cooperating network.

Every component in the controller network performs a pre-defined task with very specific requirements. Because cars are mass products, choosing the best adapted micro controller for each task allows the engineers to reduce the costs of production. The resulting controller network consists of highly heterogeneous components. There are also various different bus systems to connect the different micro controllers, which differ in properties such as real-time capability or the number of supported nodes in the network.

While the above reasons account for the heterogeneity in the controller network in terms of both, the nodes and the bus systems, this heterogeneity leads to a complicated development process that is poorly supported by the existing tools. The development is component centric, and connecting the nodes requires the specification of communication protocols.

In this paper, we present our prototype system KESO that supports the development of a controller network as an integrated system. KESO is a Java middleware that provides a robust programming model uniform for all participating controllers. KESO implements a restricted multi-JVM architecture running on a traditional OSEK/VDX operating system. Restrictions to the JVM were made where the real-time properties were more important than Java compatibility. For example, we preferred the lock-free synchronization model using a priority ceiling protocol of the OSEK/VDX system over standard Java monitors.

KESO leaves the choice of the components and the bus systems to the system developers. The topology and the components of an integrated systems are, however, contrary to the existing development model, transparent to the application. These properties are configured in a single global configuration file. KESO provides a uniform communication mechanism to the application, which disburdens the developers from specifying and implementing communication protocols. KESO can furthermore enable the migration of tasks to different controllers or the integration of tasks on a single micro controller by only changing the configuration file. The integration of tasks on a controller is safe, because KESO provides software-based memory protection that allows the safe coexistence of multiple tasks on on controller.

The remainder of this paper is structured as follows: In Section 2, we present the core architecture of KESO and the extensions of KESO to a distributed system. Section 3 discusses the core aspects that make KESO a distributed system and ideas on how we plan to implement them. We conclude the paper with a short summary in Section 5.

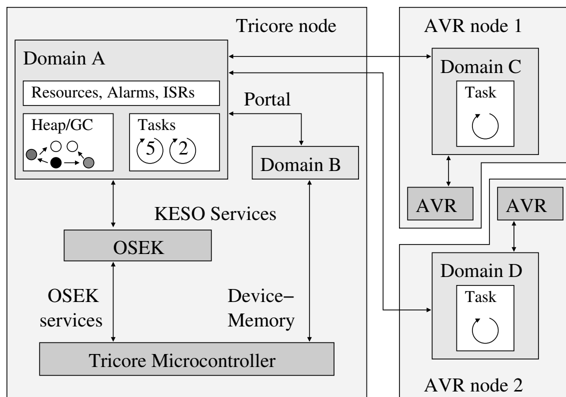

Figure 1. Architecture of Distributed KESO

2. KESO Architecture

Figure 1 shows the architecture of a distributed KESO system. The shown system is a network of two AVR micro controllers and a Tricore micro controller, where the AVR micro controller nodes communicate with the more powerful Tricore micro controller. A distributed KESO system consists of the following components.

Nodes A distributed KESO system is composed by multiple KESO nodes. Each node is represents a micro controller in the network. A KESO node is usually based on a traditional OSEK/VDX [4] operating system and provides the OSEK/VDX scheduling, synchronization and notification concepts to the KESO applications. The structuring of a distributed KESO system in nodes describes the topology of the controller network and is only visible in the configuration. For the applications, the entire KESO system is structured in domains, whose locations are transparent to the applications.

Domains Each system is structured in domains of protection that are strongly isolated from each other. Each domain appears as a self-contained JVM to the application, containing an own set of static class fields and an own object heap.

Each domain is assigned to a node in the configuration file, whereby a node can contain multiple domains. Domains colocated on the same node are isolated by software-based memory protection mechanisms, based on the type-safety of Java and the separation of object heaps and class fields. On OSEK/VDX based nodes, we do furthermore provide access restrictions to the OSEK services to isolate the domains from each other [7]. Domains that reside on different nodes are physically isolated from each other.

Tasks In KESO, threads of control are represented by OSEK/VDX tasks instead of Java threads to the application, which denotes the differences in the execution model. OSEK/VDX tasks are scheduled according to a fixed priority whereas there is a wide range of scheduling policies for Java threads. Every task is assigned to exactly one domain.

Portals The portal mechanism allows the communication among different domains. The interface between the portal mechanism and the application does not differ for intra-node and inter-node communication, which hides the location of a domain to the application.

A *service domain* can provide a portal *service* by exporting the interface of a service object. Another domain can import the service, and obtain access to the service through a global name service.

KESO Services KESO provides a number of services to the Java application. These are comprised by a Java API [7] to the services of the underlying OSEK/VDX operating system and the device-memory, which allows Java applications to access memory mapped device registers. Device-memory furthermore enables the development of device drivers in Java.

Code Generation The user applications are developed in Java and available as Java bytecode after having been processed by a Java compiler. Interpreting or even compiling the bytecode to native code at runtime leads to poor or unpredictable execution times and is not always feasible on deeply embedded devices with very limited resources.

Instead, the bytecode is compiled to C source code ahead of time by the *KESO builder*. This way, the KESO nodes are built using one uniform tool, hiding the differences of the micro controllers and the compilers. The generated C code contains the compiled Java application plus code for services of the KESO runtime environment and additional runtime checks that ensure the properties of a JVM.

The KESO builder creates a highly adapted system that contains only the parts of the KESO runtime system that are required by the application. For instance, many embedded applications do not dynamically allocate memory at runtime. This type of applications does not require a garbage collector, hence a simple heap implementation can be configured that does not provide garbage collection. Garbage collection can also only be used in parts of the system as the heap implementation can be configured individually per domain. For single task systems, that do not require scheduling and synchronization mechanisms, even the underlying OSEK/VDX operating system can be entirely omitted leaving the KESO running on the bare hardware.

3. KESO as a Distributed System

In the following we briefly describe the core aspects of KESO as a distributed system.

(Re-)Configuration There is one global configuration for the entire distributed system. The structuring of a distributed KESO systems into multiple nodes is only visible from within the configuration, i.e. the assignment of a domain to a specific controller in the network is transparent to the applications. KESO thus provides the Java application with an integrated view on the controller network as an integrated system.

As the various nodes in a distributed KESO system are not visible to the applications, functions in the controller network can be migrated to a different micro controller or integrated with other functions on a shared micro controller by only adjusting the global configuration. No changes to the applications are required for this type of reconfiguration.

Because KESO enforces the strict isolation of domains colocated on a micro controller, tasks can safely be integrated on a single controller without the risk of unclear responsibilities in the case of software failures introduced by dangling pointers. KESO thus provides software-based isolation of tasks on a micro controller that would otherwise physically be isolated when deployed on dedicated controllers.

Uniform Programming Model The traditional development process uses different tools for the various architectures. The applications are usually developed in C and Assembler.

In KESO, all applications are developed in Java using one common build tool, the KESO builder. While the KESO builder eventually also generates C code, it hides the differences of the various C compilers to the developers. The rich in semantics Java bytecode enables the builder to make global optimizations and generate highly attributed C code that in turn allows the C compilers to generate more optimized native code than they could achieve from human produced C code. Though programmers could in theory attribute C code like the KESO builder, it is in practice difficult to ensure the correctness of the attributes. Incorrect attributes may cause the compiler to create incorrect native code.

KESO also provides a unified system interface to the applications. While some device drivers are already provided by KESO, others can be implemented using the KESO device-memory service and the OSEK/VDX API. The most notable aspect in a distributed KESO system is the provision of the portal mechanism that allows a uniform way of inter-domain communication, hiding whether the service domain is located within the same KESO system or on a different controller of the network.

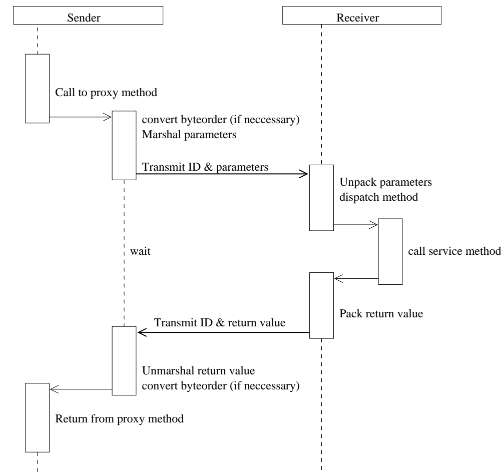

Figure 2. KESO service method invocation

Inter-Domain Communication Two domains can communicate with each other using the portal mechanism. The prerequisite for the communication of two domains is that one domain, the *service domain*, exports a *service* to the other domains in the distributed KESO system. A service consists of an interface of *service methods* that can be invoked by *client tasks* of different domains in the environment of the service domain. This behavior is comparable to remote procedure calls (RPC) or the Java remote method invocation (RMI) [1].

To establish a communication channel between two domains, a task of the client domain needs to acquire a service object (proxy object, stub) of an exported service of the service domain through a global name service. The client can then invoke service methods at this object.

The implementation of a service method invocation depends on whether the service domain and the client domain are colocated on the same micro controller or not. The builder automatically chooses the correct implementation based on the KESO configuration.

In case both domains are located on the same micro controller, the service method invocation is mapped to an extended local method invocation. Because an object reference must not cross a domain boundary for isolation reasons, all objects referenced by parameters of the service method, including the transitive closure, are copied to the heap of the service domain. This overhead is only necessary if object parameters are passed to the service method. Otherwise the parameters are simply passed by value.

In distributed KESO, the same portal mechanism can also be used for communication between different micro

controllers. This is transparent to the application. Figure 2 illustrates the execution of a service method as a remote procedure call. The parameters are marshaled and transported to the target controller over a communication service. The communication service is configurable and can implemented in Java. In a first prototype implementation we used a CAN bus and a serial port driver written in pure Java. The result is transported back to the sender vice versa.

The exchange of data between two micro controllers may require a byte order conversion in case the participating micro controllers use different byte order. KESO pursues a *sender-makes-it-right* strategy on byte order conversion, i.e. the caller of the remote method will convert the byte order before sending and after receiving multi byte values from another controller. The decision whether a conversion is required can be performed at compile time because the architectures of the participating controllers are known for each service method invocation.

We do currently not support references as parameters to service methods invoked on a different controller. Scalar values are passed call-by-value.

Sample Application Our evaluation system is a Robertino [2] robot [1] that can move around and avoid obstacles. Robertino has three wheels that allow the robot to move and turn. Each of the wheels is driven by a motor that is connected to a low-cost AVR micro controller with 512 bytes of RAM and 8 kilo bytes of flash memory. The robot can scan its environment using six infrared distance sensors. Each of the three AVR controllers is connected with the two closest sensors.

The AVR nodes function as sensors and drive controller for the motors. The more complex control and supervision functions are handled by a central more powerful Tricore TC1796 micro controller (2 MB program flash memory, 64 kB data SRAM, 150 MHz CPU) that is connected to the AVR nodes through a CAN bus. The Tricore controller receives the sensor data from the AVR nodes and, based on the sensor information, transmits control commands.

4. Related Work

RTZen [5] is the most notable Java middleware for distributed real-time and embedded systems so far. RTZen is a middleware that is compliant with most of the features defined in the CORBA 2.3 specification, including portions of the Real-time CORBA specification. The design of RTZen is based on many of the patterns, techniques, and lessons learned from the development of The ACE ORB (TAO) [6].

The Open Virtual Machine (OVM) [3] project shows that it is possible to use a Java middleware in a real-time avionic application. OVM uses ahead of time (AOT) compilation from java byte code to C++ and uses global world optimization for code reduction.

In contrast the target platform from KESO is much smaller. While OVM needs a few mega byte of RAM, the AVR micro controllers only provide 512 bytes. Therefore it was important to customize the JVM even more and skip unneeded abstraction layers.

5. Summary

In this paper, we briefly presented our preliminary design of a distributed system that provides an integrated development process for controller networks as they appear in modern cars. Applications can be developed independently of the topology of the controller area network, which can even be changed without the need to adapt the application.

With Java as the programming language, KESO offers a uniform and type-safe way of robust software development for controller networks of heterogeneous components. Structuring the entire distributed system in domains of protection allows the safe integration of multiple applications on the same hardware and represents at the same time the physical isolation for domains that reside on different controllers.

The development efforts on extending KESO to a distributed multi-JVM system are still in progress. We expect first results in the beginning of 2007.

References

[1] Java RMI - Distributed Computing for Java. White Paper, Sun Microsystems Inc.

[2] OpenRobertino. http://www.openrobertino.org/.

[3] J. Baker, A. Cunei, C. Flack, F. Pizlo, M. Prochazka, J. Vitek, A. Armbruster, E. Pla, and D. Holmes. A real-time java virtual machine for avionics - an experience report. In *IEEE Real Time Technology and Applications Symposium*, pages 384–396, Washington, DC, USA, 2006. IEEE.

[4] OSEK/VDX Group. *Operating System Specification 2.2.3*. OSEK/VDX Group, Feb. 2005. http://www.osek-vdx.org/.

[5] K. Raman, Y. Zhang, M. Panahi, J. A. Colmenares, R. Klefstad, and T. Harmon. RTZen: Highly predictable, real-time java middleware for distributed and embedded systems, . In *ACM/IFIP/USENIX 6th International Middleware Conference (Middleware '05)*, pages 225–248, 2005.

[6] D. C. Schmidt, D. L. Levine, and S. Mungee. The design of the TAO real-time object request broker. *Computer Communications*, 21(4), Apr. 1998.

[7] M. Stilkerich, C. Wawersich, W. Schröder-Preikschat, A. Gal, and M. Franz. OSEK/VDX API for Java. In *Linguistic Support for Modern Operating Systems ASPLOS XII Workshop (PLOS '06)*, pages 13–17, San Jose, California, USA, Oct. 2006. ACM.

[1] We replaced the PC104 as the main processing component of the Robertino with a Tricore micro controller.