SICE-ICASE International Joint Conference 2006
Oct. 18-21, 2006 in Bexco, Busan, Korea

Designing Real-Time and Fault-Tolerant Middleware for Automotive Software

Jiyong Park, Saehwa Kim, Wooseok Yoo, and Seongsoo Hong

School of Electrical Engineering and Computer Science, Seoul National University, Seoul, Korea
(Tel : +82-2-880-8370; E-mail: {parkjy, ksaehwa, wsyoo, sshong}@redwood.snu.ac.kr)

Abstract: Automotive software development poses a great deal of challenges to automotive manufacturers since an automobile is inherently distributed and subject to fault-tolerance and real-time requirements. Middleware is a software layer that can handle the intrinsic complexities of distributed systems and arises as an indispensable run-time platform for automotive systems. This paper explains the concept of middleware by enumerating its functions and categorizes middleware according to adopted communication models. It also extracts five essential requirements of automotive middleware and proposes a middleware design for automotive systems based on the message-oriented middleware (MOM) structure. The proposed middleware effectively addresses the derived requirements and includes many essential features such as real-time guarantee, fault-tolerance, and a global time base.

Keywords: Middleware, Real-time, Fault-tolerance, Message-based

1. INTRODUCTION

Due to severe competition in automotive industry, it is required for an automobile to be equipped with sophisticated electronic devices for improved driver assistance and passenger safety. According to an industry analysis, 90% of automotive innovations will be created for electronic devices in 2010 [1]. The number of ECUs (Electronic Control Unit) in an automobile is rapidly increasing as the portion of automotive electronic devices increases. ECUs are used in most automobile modules such as power train, chassis control, and transmission modules. They are connected via various automotive network protocols such as CAN, LIN, TTP, and FlexRay. As a result, an automobile becomes a complex and heterogeneous distributed system. For example, in Mercedes Benz S class 2003, there are over 50 ECUs interconnected via three different types of networks. They exchange approximately 600 signals using about 150 different types of messages [2].

As the portion of electronic devices in an automobile increases, the amount and complexity of automotive software increase as well. It is predicted that 80% of innovations related to automotive electronics will be achieved in the software area [1]. In the case of the S class, 500,000 LOC (lines of code) is currently used. As the LOC of software increases, the software complexity gets increased more drastically.

Increased software complexity can only be solved by adopting a systematic run-time software platform. Middleware is such a platform that can handle these intrinsic complexities and thus enables programmers to concentrate on their own business logic.

In this paper, we present a design of an embedded middleware for automotive software. First, we explain the concept of middleware by enumerating its functions and then categorize middleware according to adopted communication models. We then extract five requirements for automotive middleware. Finally, we propose the design of our automotive middleware that effectively satisfies these derived requirements.

2. CONCEPT OF MIDDLEWARE

Middleware is a software layer that connects and manages application components running on distributed hosts. As shown in Fig. 1, it exists between network operating systems and application components. Middleware hides and abstracts many of the complex details of distributed programming from application developers [3][4]. 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.

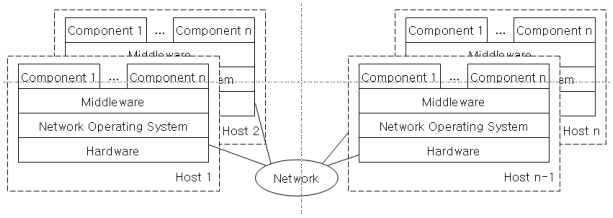

Fig. 1 Middleware in a distributed system

2.1 Functions of middleware

Followings are the main functions that middleware should provide for application components [3].

- Network communication: Middleware provides high level network communication primitives such as remote procedure calls and/or the publish/subscribe model of communications. Using these primitives, application developers can program application components as if they were running on the same host.
- Coordination: Middleware manages synchronization between application components. Synchronization is a crucial function when applications are running in a

distributed environment. This is because there are multiple threads of control which cause deadlocks and dangerous system inconsistency without proper synchronization.

- Reliability: Middleware performs error detection and correction in a transparent manner to mask out the unreliability of communication networks.
- Scalability: Middleware makes it possible to add new components without modifying the existing system architecture or already implemented code.
- Heterogeneity: Middleware hides the heterogeneity of hardware, operating systems, network protocols, programming languages, and middleware itself.

2.2 Categories of middleware

Middleware can be effectively classified into four different categories if we consider the communication models that are adopted into middleware, as discussed in [3]. The four categories are (1) transactional middleware, (2) message-oriented middleware, (3) procedural middleware, and (4) object middleware. We characterize these middleware categories and analyze their strengths and weaknesses.

First, transactional middleware supports transactions as its communication model. A transaction is a series of information exchange that must be performed atomically, consistently, isolated, and durably. It thus simplifies the construction of a transactional distributed system that must guarantee the atomic consistency of operations. However, it produces an undesired overhead where transactions hardly occur as in an automobile.

Second, message-oriented middleware (MOM) supports a publish/subscribe mechanism as its key communication model. This model provides a global data space that is shared by all application components. Components can communicate with each other by publishing and subscribing to data using the global data space. In this model, communications are done asynchronously. This means that a publisher and a subscriber run independently of each other and do not wait for their peer.

MOM has both strengths and weaknesses. It allows for complete decoupling among application components thanks to the asynchronous property of the publish/subscriber model. As such, MOM is particularly well-suited for a distributed event notification system. MOM also achieves fault-tolerance by implementing a global data space on persistent storage. Even though a component is unavailable due to a temporary failure, its data are retained in the permanent storage until the component is available again.

Third, procedural middleware supports remote procedure calls (RPC) as its communication model. In this model, communications can be made by using primitives similar to local procedure calls. Specifically, a server exports several procedures for communication and a client invokes these procedures. Then, procedural middleware marshals an RPC into several messages and vice versa. Since its programming interface is easy to understand, it is available on most operating systems. However, procedural middleware does not support reliability and scalability very well and supports only quite limited synchronous invocations.

Lastly, object middleware is an evolution from procedural middleware. It adds to middleware the concept of object-oriented programming such as object identification through references and inheritance. This type of middleware enables independent development and distribution of each component since each interaction of components is defined by interfaces. Object middleware has weaknesses as well, particularly in performance. It incurs a fairly large overhead when maintaining coherency among distributed objects.

3. REQUIREMENTS FOR AUTOMOTIVE MIDDLEWARE

Before explaining the design of our automotive middleware, we enumerate the five requirements of automotive middleware. These requirements are (1) resource management, (2) fault-tolerance, (3) specialized communication model for automotive networks, (4) global time base, and (5) 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.

3.1 Resource management

An automobile has a real-time nature. It is a system where its correctness depends not only on the correctness of the logical result, but also on the result delivery time. Since an automobile is subject to various timing constraints, every component in an automobile should be designed in a way that its timing constraints are guaranteed a priori.

At the same time, the timing constraints of an automobile 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.

3.2 Fault-tolerance

Fault-tolerance is a property that enables a system to keep operating in a required manner even in the presence of a failure of some of its components. An automobile is operating in a harsh environment where there exist wide variations of temperature and moisture and electronic and physical shock. It is often subject to

permanent or transient failures. Since such failures can not be easily fixed on the fly while the automobile is operating, fault-tolerance is one of the essential properties of automotive software.

However, it is very difficult for application programmers to implement fault-tolerance features by themselves when they design automotive software. The fault-tolerance features require complex underlying mechanisms that involve component replication, error detection, replica determinism, and membership management and re-integration. Implementing such features would excessively delay the development process and force developers to learn related skills. This will increase the possibility of software defect. Therefore, it is preferable that fault-tolerance is provided by middleware and hidden from application programmers.

3.3 Specialized communication model for automotive networks

While TCP/IP and high speed network protocols such as Ethernet form a communication basis for enterprise middleware, they are rarely adopted in an automotive system due to limited physical space, pricing, and resistance to electro-magnetic and physical damages. Instead, CAN, TTP, LIN, and FlexRay are representative automotive network protocols. The common properties of these network protocols are that they transmit data using broadcast; that the size of a message is very small, usually a few byte; and that the transmission speed is rather slow compared to the Ethernet.

Therefore, it is inevitable to specialize the communication model of automotive middleware for these automotive networks. Otherwise, the resultant system would experience unallowable overhead. As an example, let us consider a communication model that allows for the transmission of arbitrary long messages. The associated middleware has to divide the long message into small packets to transmit the long message over the network. Later, the middleware has to reassemble received packets into a complete message. Furthermore, it needs to incorporate an error recovery mechanism similar to TCP's packet loss handling. Integrating all of these features into middleware surely yields excessive complexity and causes long transmission delays.

Clearly, automotive middleware should avoid complex multi-level protocol stacks and adopt a simple communication model similar to a hardware-level message transport mechanism. The use of such a simple and efficient communication model also helps automotive manufactures translate their legacy automotive software into component-based software.

3.4 Global time base

A global time base is a shared logical clock that is used at each node in a distributed system. If there is no global time base, then each node has to rely on its own local clock. Since local clocks have different drift rates, different nodes see different clock readings. This makes it extremely difficult to write distributed programs. Since a global time base can free application developers from the task of handling time differences among distributed nodes, automotive middleware should provide it.

3.5 Resource frugality

The last requirement for automotive middleware is resource frugality. This requirement means that the middleware has to achieve desired performance with only limited memory space and CPU bandwidth. Compared to PCs and workstations where traditional enterprise middleware are used, ECUs in an automobile contain low-end microcontrollers and a small amount of memory. Specifically, ECU microcontrollers usually operate on only a few tens of MHz and memory is also only a few tens of Kbyte. Consequently, automotive middleware are required to provide only necessary functionalities and be implemented with effective and simple algorithms and data structures.

4. DESIGN OF AUTOMOTIVE MIDDLEWARE

In this section, we explain the structure and core mechanisms of the proposed automotive middleware in detail. We also explain how we address each of the previous requirements.

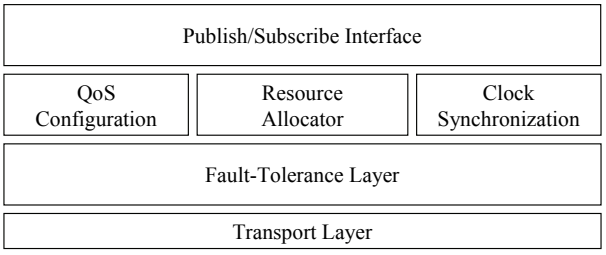

Fig. 2 Structure of the proposed middleware

Fig 2 shows the structure of the proposed middleware for automotive systems. At the top level, there exists the publish/subscribe interface that is used by application components. Beneath the interface, there are QoS Configuration module for specifying real-time requirements, Resource Allocation module for guaranteeing timeliness operations, and Clock Synchronization module for providing a global time base. All incoming and outgoing messages pass through Fault-Tolerance Layer in order to guarantee reliability. Finally, messages are transmitted and received by Transport Layer. We proceed to explain the details of these mechanisms.

4.1 Message-based communication model

Among the four categories of middleware, we argue that the MOM is best suited for the automotive environment because it can be easily implemented on automotive networks and also supports the required

level of fault-tolerance. We further explain why MOM is selected as the base of our automotive middleware.

First, MOM uses the publish/subscribe model as its communication model. DDS [7] and JMS [8] are two representative middleware products that support this model. Fig 3 pictorially depicts the concept of the publish/subscribe model of communication where middleware provides a global data space. A distributed component may write (publish) data into the global data space or read (subscribe to) data from the global data space independently of other components. When multiple components subscribe to the same data, then the publisher of the data must be able to broadcast the message that contains the data to all the subscribers. This broadcast is a basic transport mechanism that is commonly used by many automotive network buses. Thus, the publish/subscribe model can be easily implemented on those networks.

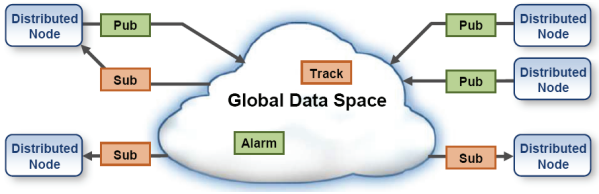

Fig. 3 Publish/subscribe model of communication

Second, in MOM, data publishing and data subscription are done independently. This means that a publisher and a subscriber do not have to wait for its peer to subscribe to or publish data. This gives MOM a fault-tolerance capability. Even though a publisher malfunctions or it does not produce any data, its corresponding subscriber can still operate normally.

4.2 QoS configuration and static resource allocation

Using MOM alone can not satisfy all the requirements for automotive middleware. To guarantee real-time data transmission, we use the QoS configuration mechanism and its associated resource allocator. QoS configuration is a way of describing the timing requirements of an application component to the middleware. Such timing requirements include response time, period to name a few. A resource allocator is a middleware module that guarantees the requirements described in QoS configuration. For example, the amount of resources (CPU time, memory, network bandwidth, etc.) to satisfy given requirements is calculated and reserved before run-time. At run-time, it monitors and schedules the resource usage to make sure that the actual resource usage of a component does not exceed the reserved value. We decide to use static resource allocation although it is less effective than the dynamic one. Nevertheless, we do not choose the dynamic resource allocation since it renders middleware more complex and incurs a more run-time overhead. This should be avoided to satisfy the resource frugality requirement.

4.3 Clock synchronization

The proposed automotive middleware makes use of a clock synchronization mechanism provided by the network hardware. This mechanism makes the local clock of each distributed node to get synchronized within a specific time drift. Clock synchronization can be achieved through periodically exchanging local clock values among local clocks or by resetting local clocks to a central master clock. Using this mechanism, the clock synchronization module inside the middleware provides a synchronized global time base with which distributed real-time scheduling is performed.

4.4 Fault-tolerance service

Since the message-based communication model offers only the preliminary level of fault-tolerance, it is necessary for the middleware to provide additional fault-tolerance mechanisms for automotive applications. For example, a fault-tolerance mechanism that detects a faulty node is required. Suppose that there are four nodes, each of which controls a wheel brake. If one of the nodes fails, then other three nodes must be notified of the failure so that they can adjust the pressure of brake drums to prevent the automobile from being overturned. This module also provides a membership service that separates normal nodes from failed nodes, and a re-integration service that handles nodes which are recovered from failure.

4.5 Resource frugality

We rule out quite a few functions that are not needed in the automotive environment even though are used in traditional enterprise middleware. First, marshalling and un-marshalling are removed from our middleware. These are the functionalities that resolve difference in data representation of heterogeneous networks and convert complex data structures into a bit stream so that it can be transmitted through network. However, in an automobile, the number of heterogeneous networks is fixed and complex data structures are rarely transmitted over the network.

Second, dynamic loading and load balancing are also removed. The former is a functionality that enables loading or unloading of application components at run-time. The later is a functionality that dynamically selects appropriate components according to the system load. Since an automobile is a closed system in which new tasks are not created at run-time, these two functions are not needed.

Lastly, a naming service is removed. This is a service that matches the location of a component to a unique name so that application programmer does not need to know about the exact location of components while programming. However, in an automobile, the existence and location of each component is fixed when designing the entire system. Thus, it is easy to know the location of a component without using the naming service.

5. CONCLUSION

Our contributions in this paper are two folds. First, we have identified five essential requirements that must be satisfied by automotive middleware. These are resource management, communication model specialized for automotive network, fault-tolerance, a global time base, and resource frugality. Second, we have presented a middleware structure satisfying these requirements. The middleware we have designed supports the publish/subscribe communication model based on message transmission. It also includes a QoS configuration and static resource allocation mechanism, a clock synchronization mechanism, and various fault-tolerance algorithms to achieve real-time data transmission, fault-tolerance, and a global time base.

For future work, we are looking to extend our work by implementing the prototype of the automotive middleware and applying it to the development of actual automotive software to show its utility.

ACKNOWLEDGEMENT

Our research is sponsored by the Ministry of Commerce, Industry and Energy (MOCIE).

REFERENCES

[1] "Reuse of Software in Distributed Embedded Automotive Systems," Audi, 2004

[2] Klaus Grimm, "Software Technology in an Automotive Company – Major Challenges", Proceedings of the 25th International Conference on Software Engineering, pp. 498-503, IEEE, 2003.

[3] W. Emmerich, "Software Engineering and Middleware: A Roadmap. In the Future of Software Engineering", A. Finkelstein ed. pages. 76-90. ACM Press, 2000.

[4] R. Schantz and D. Schmidt, "Middleware for Distributed Systems - Evolving the Common Structure for Network-centric Applications. In the Encyclopedia of Software Engineering", John Wiley & Sons, December 2001.

[5] J. Stankovic, "Real-Time Computing", BYTE, invited paper, pp. 155-160, August 1992.

[6] J. Stankovic, "Misconceptions about Real-Time Computing", IEEE Computer, 21(10), pages 10-19, Oct. 1988.

[7] "Data Distribution Service for Real-Time Systems Specification", OMG, December 2004

[8] "Java Message Service", Sun Microsystems, April 2002