

AN OPEN AUTOMOTIVE DEVELOPMENT PLATFORM

Ir M.W. Nelisse
TNO TPD

PO Box 155, 2600 AD Delft, the Netherlands

Tel: +31 (15) 269 2323, Fax: +31 (15) 269 2111, Email: nelisse@tpd.tno.nl

Automotive components are steadily using more and more electronic parts and software applications and show an in-creasing level of interaction, not only within a car but also between a car and its environment. To be able to assist in the development of these components, to simplify system integration and allow better testing, TNO defined a generic development, simulation, visualization, test and evaluation platform. This open automotive development platform is based on the concept of a distributed computing system using off-the-shelf interconnection components and standardized communication protocols. Existing and proprietary system parts can be interfaced to this platform by means of gateways.

INTRODUCTION

The increasing complexity of in-car electronics and the rapidly growing amount of sensors, actuators and electronic control units, places higher demand on in-vehicle data communication protocols. Safety critical systems need deterministic protocols with fault-tolerant behavior, the need for on-board diagnostics calls for flexible use of bandwidth and an ever-increasing number of functions necessitate a flexible means of extending the system.

Furthermore the new area of vehicle telematics extends the range of in-vehicle data communication to remote data access. Vehicle telematics introduces safety, security, information and entertainment to vehicles via wireless telecommunications networks. It creates a link between the vehicle and a wide range of information or service providers, which can be situated in both static and dynamic locations (e.g. centralized locations, road beacons and even other vehicles).

The combination of these new developments in information technology and telecommunication networks will enable more intelligent vehicle solutions and cost efficient telematics applications. However in recent years an increasing part of the development costs of new cars is associated with electronic components and software applications. In fact in some cases these costs already overshadow the costs related to the more traditional parts.

Furthermore there is a clear distinction regarding the life cycle of standard in-car components and vehicle telematics components. Standard in-car components are normally mounted in the car during production and under normal circumstances will function over its entire life span. Vehicle telematics however are generally after-sale market products, which are delivered by third parties. They also tend to have a much shorter life span, due to the fast developments and product updates in this market segment.

This different life cycle requires a different approach in the development of telematics applications ('open' and 'standard') and the way in which car manufacturers could offer new features ('regular updates') to customers. The development of vehicle telematics applications showed a transition from 'stand-alone devices' (with virtual no integration), via 'combined devices' (combining several services in a single device without a common integration standard) to '*open systems*' in which applications are delivered through a single standard platform, where several service applications are run in a standard programming environment, which offers possibilities for subscription to new services, which are made available to the on-board computer by a service provider.

The main question is how these developments can be integrated in a manageable and cost efficient manner. The obvious solution is to strive for an open multi-service platform in the vehicle, but the industry is still juggling with the desire to create a multi-company standard (in order to drive down costs), while it is also competing for advantages over its competitors. Furthermore the automotive industry

has also questions like "can this solution work intrinsically safe" and "in case of a malfunctioning telematics application will that damage our customer relationship". Therefore the automotive industry has a cautious attitude and unfortunately has not yet decided on a single open standard.

BACKGROUND

TNO is an independent contract research organization, not only handling large strategic R&D projects together with R&D-intensive companies, but also targeting various projects geared towards small and medium-sized enterprises. As such TNO provides a link within the innovation chain between fundamental research as a source of knowledge and practical application as the use of knowledge, which can be commercially exploited. To anticipate the need for future innovative knowledge, TNO also participates in several international research programs for innovative knowledge development.

STATEMENT OF THE PROBLEM

In many of the multidisciplinary projects TNO carries out in the automotive field, it became apparent that the current development process in this field is often too limited for state of the art vehicle control applications.

Current vehicle control applications are no longer limited to the stand-alone control of a vehicle, but also address the interaction between the vehicle, its surrounding vehicles and its environment. The result is such that it is often not feasible anymore to test new control algorithms under real-life situations where multiple cars interact on a real road. In stead it becomes important to perform these tests in a well-defined test environment and using simulations where necessary (either full software, or partly hardware in the loop, or partly software in the loop).

Another concern during such developments is the lack of true and widely used open standards within the automotive field. This lack of standardization not only increases development time (in a field where time-to-market is becoming more important), but also increases development costs (due to the dedicated interfaces which have to be developed) and often reduced functionalities (due to a lack of openness).

Considering the aspects mentioned above the following important issues were identified:

- How to define a generic architecture that can be efficiently implemented for commercial products, but is also flexible enough for development purposes.
- How to handle proprietary components in an otherwise open system.
- How to define a distributed in-vehicle ICT platform that enables information sharing, while guaranteeing undisturbed (real-time) behavior of each and every application and service.
- How to support hardware-in-the-loop (HIL) and software-in-the-loop (SIL) approaches in a flexible way during the development phase.

GENERIC COMPONENT ARCHITECTURE

The generic component architecture for TNO's open automotive platform is shown in Figure 1.

The generic component architecture is based on 6 groups of application computers, a vehicle services interface connecting add-on and OEM equipment and an interconnection network. The following paragraphs will discuss in more detail these separate parts.

First of all the figure shows a clear separation in those parts which will only be present during development phases (left hand side) and those which will/could be present within the vehicle itself (right hand side). The center of the picture shows the interconnection network, it is a key part in this open platform and should therefore always be present.

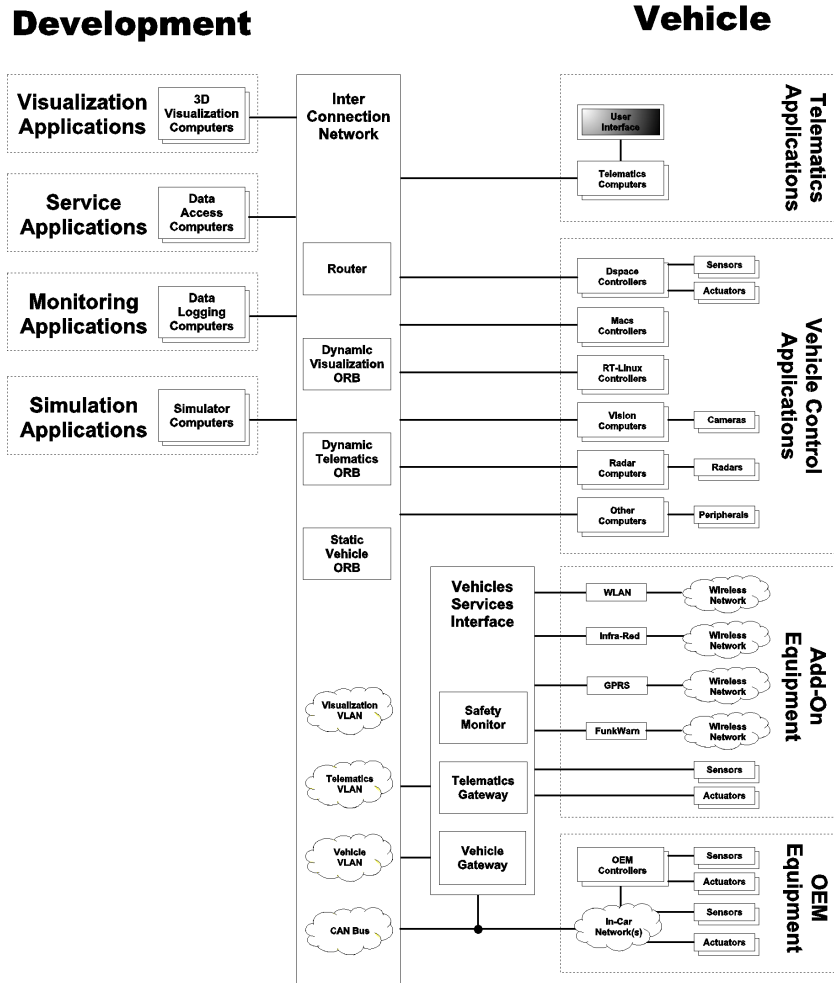


Figure 1: Generic component architecture

The *Telematics* services offer the driver a wide range of new services in the areas of safety, security, information and entertainment. These services can be available off-line (like a CD-ROM based navigation systems), but also on-line (like navigation systems which use up-to-date traffic information). This however forces an extension from in-vehicle data communication to remote data access and requires the possibility to access a wide range of information or service providers via wireless telecommunication networks.

The *Vehicle control* applications are the programs running on the in-car electronic control units (ECUs). The rapidly growing amount of sensors, actuators and electronic control units, places high demands on the complexity of the algorithms on these electronic control units and especially on their interaction. This complexity also forces the developer to use automated development, optimization and test tools.

The *OEM equipment* are those sensors, actuators and controllers, which are typically mounted in the vehicle by the car manufacturer. The sensors and actuators are often directly connected to electronic control units. In cases where they are used by multiple electronic control units, they are sometimes connected to an in-car network, just like the electronic control units themselves. Most modern cars now use one or more CAN-busses as their interconnection network. Newer car designs also use other busses (like FlexRay, ByteFlight) for safety-critical applications like x-by-wire applications. Although these in-car networks are based on open network standards for the physical bus layer, the higher layer communication protocols are still OEM proprietary and therefore cannot be considered as really 'open'.

The *add-on equipments* are typically those peripheral sensors and actuator, which are not (yet) mounted in the vehicle by the car manufacturer during production. These peripherals are often added together with an after-sale product, because of specific requirements of the application (like a GPS for

a navigation system). Although these peripherals are typically not tied to a specific vehicle, they often still have dedicated interfaces.

The *vehicle services interface* provides an open interface (or access point) to the legacy components found in the OEM equipment and add-on peripherals. The vehicle service interface should also provide gateway functionalities between the different networks running on both sides of the vehicle service interface to be able to make the location of legacy components on the vehicle side transparent for applications on the open interconnection network. Furthermore the vehicle service interface should function as a safety monitor for the legacy components, providing the same level of security for the legacy components, as is the case for the components in the open system.

The *visualization applications* are used in a development environment to show the behavior of (parts of) the system under test. Especially when using simulated system parts, the visualization can make it much more comprehensible for a developer what is happening within the overall system. State of the art visualization tools can even show what a user would see when it would be present at a certain position in the system (e.g. the driver seat) or serve as a virtual camera sensor in a simulation environment.

The *service applications* play an important role in the optimization phase of the development process. Especially the control algorithms in the vehicle control applications might require an extensive phase where settings of the algorithms need to be changed based on internal variables and signals from sensors, actuators and/or other controllers. The servicing applications will make it possible to monitor these internal and external values and get and set the parameters of the control algorithms. Typical implementations will work together with the tools, which were used to generate the control algorithms of the vehicle control applications in the first place.

The *monitoring applications* play an important role during the whole development process and maybe even during later times when vehicles are brought to for example a garage for scheduled (or unscheduled) maintenance. With these applications it should be possible to monitor any data, which is exchanged between other applications. This data should be visualized in a way, which is suitable for that specific data and may depend on the actual needs. It can vary from simple ASCII message logs, to full graphical diagrams showing the actual state or the behavior over a certain time-span. Special implementations of these monitoring applications may take the form of black boxes, allowing playback and visualization of signals that were recorded in a prior phase.

The *simulation applications* can significantly speed up the development process. At times when physical sensors/actuator/controllers are not yet available, it is possible to proceed with software models of the missing physical part ('software in the loop'). At times when physical sensors/actuator/controllers are available, but their performance is questionable, it is possible to place them in a virtual environment made of software models, to investigate their behavior under well-defined environmental constraints ('hardware in the loop').

The *interconnection network* is what links all the different parts in the system together and makes the various sources of information in the system available to all other parts. The next sections will discuss this open communication approach in more detail.

Although the generic component architecture shows a large number of components (e.g. computers) it does not mean that every implementation will always have all these components. In many cases not all components are necessary for a specific implementation. Furthermore it is often possible to map several component functions on the same piece of hardware. At the end of this paper some concrete examples of actual implementations of this generic component architecture will be shown.

PHYSICAL NETWORKS

Within TOAP we have chosen to base the open interconnection network on a high-speed full-duplex Ethernet network and optionally on a CAN-bus network. For the future extensions to TTCAN (Time Triggered CAN) and FlexRay are foreseen.

Ethernet has gained popularity in automation environments due to the fact that it is readily available, subsequently low in cost and fast when compared to existing field busses. Ethernet has however a

name of being largely non-deterministic, since the older Ethernet implementations did not have the ability to predict when information would be delivered due to the underlying CSMA/CD bus arbitration scheme. Recent advancements have enabled Ethernet networks to be largely deterministic. The usage of switching devices instead of hubs prevents data collisions and the full-duplex operation mode of Ethernet offers full-wire throughput for all peer-to-peer connections. With these state of the art technologies Ethernet can already approach the guaranteed delivery as provided by typical field bus solutions like CAN-bus.

Therefore the Ethernet networks will use the full-duplex operation mode and be configured in a star topology with a switch acting as the network concentrator for connecting multiple computers. The Ethernet switch will be configured in such a way that multiple virtual local area networks (VLANS) will be available. Each VLAN will be dedicated to network traffic with specific characteristics. Currently the following VLANs are foreseen:

- Visualization VLAN. This VLAN will be used for the exchange of data between visualization applications. The data will consist of large blocks, which are exchanged on a non real-time basis.
- Telematics VLAN. This VLAN will be used for the exchange of data between telematics applications and to and from the vehicle service gateway. The data will consist of blocks with highly varying sizes and which are exchanged on a non or soft real-time basis.
- Vehicle VLAN. This VLAN will be used for the exchange of all data that needs to be accessed on a real-time basis. The data will consist of small blocks, which are updated on a regular time basis.

Finally the CAN-bus will serve as the interconnection network for dedicated controllers, which do not have an Ethernet interconnection possibility. Since the CAN-bus uses a bus instead of point-to-point topology, no special interconnection facilities are necessary. This does however mean that careful design checks have to be made to guarantee correct real-time behavior.

A router can provide functions that pass messages between different and potentially physically dissimilar networks, like the CAN and Ethernet networks. Furthermore it can restrict the access from one network to another depending on certain rules ('security') or limit the flow of communication between networks ('safety'). Using the Quality of Service functions of the Ethernet switch it is possible to guarantee that communication on one VLAN will not interfere with communication on another VLAN, while both are using the same physical switch. Using the routing functions of the switch (or a dedicated computer) it is also still possible to allow certain computers on different networks to exchange data. This not only means that the network efficiency will increase since no data collisions will occur, but also that some sort of data security can be offered since the access of data can be restricted to well defined computers.

COMMUNICATION PROTOCOLS

The UDP-TCP/IP protocol suite is a well-known set of communication services in Ethernet networks. TCP/IP is inherently a point-to-point communication mechanism and therefore suitable for explicit addressing schemes. These types of messages are often used for device configuration and diagnostics and are typically highly variable in both size and frequency. Many field bus protocols however, assume a bus topology and heavily depend on broadcast transmission possibilities for their application level messages. TCP/IP is not very well suited for these implicit addressing schemes. Implicit messaging is however very well feasible by using UDP/IP. Here the data field contains mainly real-time I/O data, while the meaning of the data is predefined at the time the connection is established. This way the processing time in the node can be minimized during runtime. Such messages are low overhead, short and provide the required, time-critical performance needed for control.

The most reasonable way to implement control based networking on top of Ethernet is to use the TCP/IP protocol for best-effort and guaranteed delivery and UDP/IP protocol for temporally guaranteed (but not delivery guaranteed) messaging.

Although CAN and UDP-TCP/IP provide services for data transfer and network management it does not guarantee that the applications on the devices can communicate. It only guarantees that application level messages will be successfully transferred between the devices. For interoperability a common application-layer is needed on top of the data transfer and network management layers. It are

these upper layer protocols that determine the level of functionality the network support, which devices may connect to the network, and how devices interoperate on the network.

MIDDLEWARE

Typical vehicle control applications may work with a fixed configuration of components. However telematics services and development applications require a more dynamic configuration. This not only means that it should be easy to add components to one of the physical interconnection networks, but also that these applications should be able to establish communication channels to other applications or services without knowing the physical whereabouts of other application programs or services.

A middleware solution can provide a common communication mechanism among platforms with different hardware, software and operating systems. The middleware uses an object-oriented framework hiding the implementation details of the system architecture and inter-processor communication. When a distributed object is accessed, the object request broker ('ORB') of the middleware receives the local method call, locates the processor on which the object is implemented, encodes and sends the operation and its parameters, and does all the work necessary to turn the call back into a local invocation on the receiving processor.

From a system design point of view it would be advantageous to consider the complete system as distributed object instances and to have their services available via a middleware layer. However these "object wrappers" around the modules and the usage of an ORB also have potential performance and real-time behaviour drawbacks. Therefore the following approach has been chosen:

- Dynamic ORB. The dynamic ORB will support services that can come and go. It will offer a mechanism to dynamically discover and use these services. Because of this it will be hampered by some sort of performance penalty, making it better suited to non or soft real-time applications.
- Static ORB. The static ORB is basically a compile/build time configuration: all interactions are defined and automatically inserted into the service application code. Therefore there will be no performance degradation and it will be better suited for true real-time processes.

The advantage of this "Janus-faced" solution is that on component level it is irrelevant whether the component will be used in static or dynamic environment. While on the other hand on system level the trade-off between flexibility and performance can be managed.

DISTRIBUTED CONTROL AND SAFETY

Distributed real time control applications classically consist of one or more algorithms or modules, which will run on one or more processors and typically follow a scheme of functionally layered control as is shown in Figure 2. This layered scheme has a clear signal flow (exceptions bottom up, commands top down) and it is relatively easy to make the implementation modular (component based). The controlled process (the vehicle) is at the bottom and layers of control components are built on it. Each control component covers a well-defined control function, and this control function becomes more and more complex ("intelligent") as climbing up in the hierarchy (e.g. actuating, motion, tracking).

Each control component has its own (sufficient) knowledge about the process and environment to generate control commands based on observations. The observations (i.e. sensory readings and derived quantities/signals) are "freely" available for control components via the observation bus. It should be mentioned explicitly that beside the control components other signal/data processing components could also be added to the scheme (for example to calculate estimators and other derived signals/data).

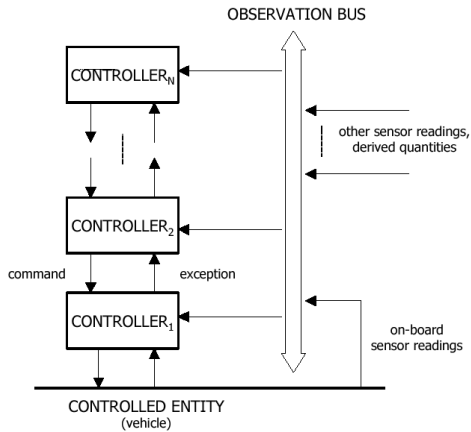


Figure 2: Functionally layered control

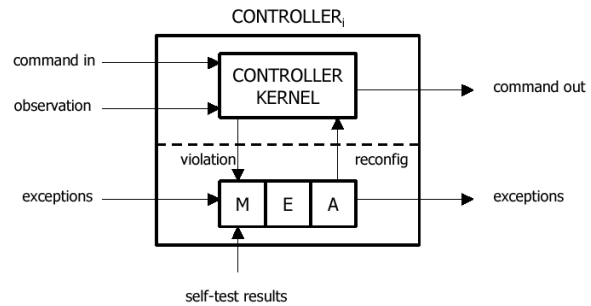


Figure 3: Controller exception handling

Beside the control functionality (Controller Kernel) each component executes a monitor-evaluate-act (MEA) loop to track its own and related subsystems' health state, as is shown in Figure 3. If a component receives an exception from the layer below or the exception is generated locally (by the monitor) the evaluator decides how it should be handled. If the exception can be handled locally (reconfigure) it will be done and the exception disappears from the system. If the component cannot handle the exception fully (or cannot handle at all) the exception is propagated to higher control layers. It should be noted that the local handling of exception could result in substantial restructuring in the controller (i.e. (structurally) adaptive control schemes).

A control layer may use dedicated hardware components (e.g. sensors, communication links, etc.) to carry out its functionality. Should this component fail the controller should adjust its operation to this new situation or – if not capable of doing this – should emit exception(s). Safety critical dedicated components have to incorporate diagnostics (self-test) functionalities and in case of component fault the monitoring functionality has to be informed. Since the safety of the operation of a distributed embedded system is an extremely complex issue with heavily interacting application, software and hardware aspects, we choose an approach, as shown in Figure 4, where a minimum set of system services are defined, which are greatly independent of any applications built on the top. These services merely provide a way to the application layer to express and implement the dedicated safety functionalities required.

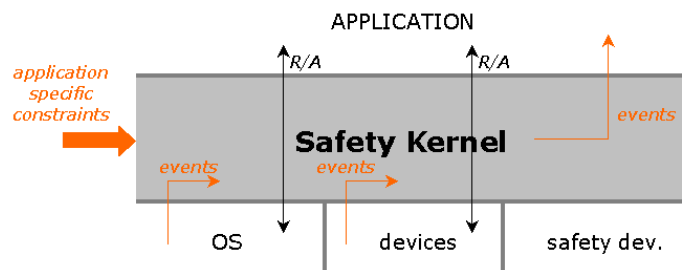


Figure 4: Safety kernel approach

The application independent safety functionalities are concentrated in the Safety Kernel. The application layer will access to operating system services and devices always via the Safety Kernel (Request/Answer, R/A in the figure). The Safety Kernel continuously monitors the access pattern. Any violation of access constraints (e.g. invalid sequence, time overrun, etc.) will be reported to dedicated event handlers and/or resolved by the kernel if possible. The access constraints can be described using special constraint language or can be built into the Safety Kernel code itself ("hard wired" solution). The Safety Kernel may use dedicated devices (e.g. watchdog hardware) and could be a verified system component (at least theoretically). Consequently verified safety aware application programs

can be built on not verified software layers (e.g. operating system) – and this is something very important.

In our approach the algorithms of the modules will (at least during the development phase) be developed in Matlab/Simulink. These Matlab/Simulink models can then be compiled to an executable running within a real-time Linux runtime environment. The chosen implementation will allow at most one module per processor to run in kernel-space, in order to guarantee the hard real-time execution of this module. Other modules on the same processor run as processes in user-space. This approach is shown in Figure 5.

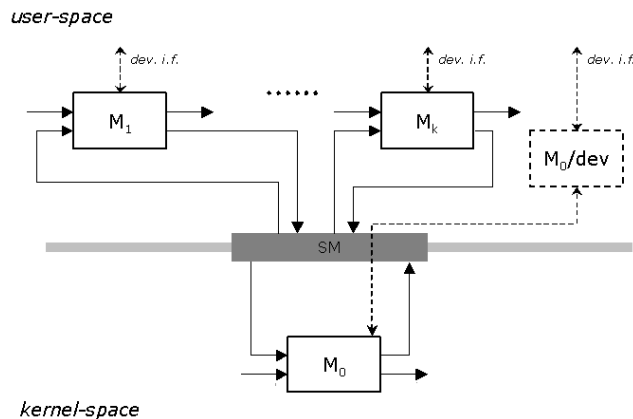


Figure 5: Module allocation on processor

Modules receive input signals ('in/DCN': input from Dedicated, Configurable and/or Network interfaces; or 'in/M' input from other modules) and send output signals ('out/DCN': output to Dedicated, Configurable and/or Network interfaces; or 'out/M' output to other modules). For service applications, modules implement a dedicated interface ('dev. i.f.': development interface). Communication between a user space module (M_i) and kernel space module (M_0) can be established via network interfaces (restriction: only UDP/IP) or via shared memory (SM).

In general the runtime environment (RTE) follows a minimalist approach: mechanisms will be established to feed-out OS exceptions and other device events to the application software, but the structured and adequate handling of those are passed to the application designer. The module code itself can be generated by the Real-Time Workshop from a Simulink/Stateflow model using a proprietary framework, or by hand written C/C++ code provided that the code conforms with the framework API if communication with other module(s) has to be established. The development interface of the kernel-space module is implemented via a user-space "middleman" (M_0/dev block) relying on the shared memory communication scheme. This way the hard real-time operation of M_0 can be assured independently of the eventual temporal jitter on the development interface. Furthermore the development interface is equivalent with the Simulink external mode interface, consequently Simulink/Stateflow originated modules can be debugged/tuned via Matlab/Simulink interface. For other modules this interface is optional, but if it is implemented an "external mode like" functionality is available to facilitate debugging, tuning, visualization, etc.

RESULTS

This open automotive development platform approach has been and is being used in a number of ways in several projects within TNO. Three of these projects will be discussed here in more detail.

VEHIL

The first example where this open automotive architecture has been applied is the VEHICLE Hardware In the Loop (VEHIL) project. The VEHIL concept (see Figure 6) allows for conducting experiments on full-scale intelligent vehicles and their infrastructure in a laboratory. First, a virtual environment is defined in which the vehicles, the infrastructure and their interactions are simulated. Next, a full-scale Vehicle Under Test (VUT) is placed on a test bench (e.g. a roller bench) and interfaced with this virtual

CarTALK 2000

The second example where this open automotive architecture has been applied is the European Project CarTALK 2000. This project is focusing on new driver assistance systems, which are based upon inter-vehicle communication. The main objectives are the development of co-operative driver assistance systems and the development of a self-organizing ad-hoc radio network as a communication basis with the aim of preparing a future standard.

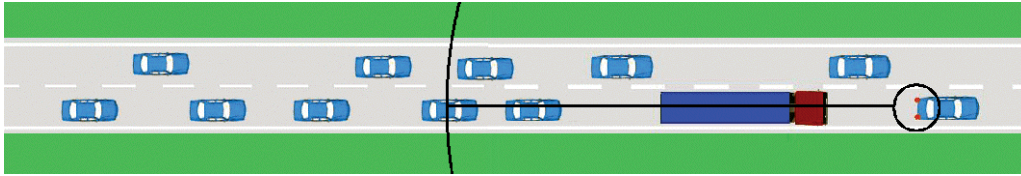


Figure 8: CarTALK example for Communication-based Longitudinal Control (CBLC)

The TNO CarTALK demonstrator will consist of three in-vehicle computers as shown in Figure 9. In-vehicle communication is done using IEEE 802.3 Ethernet, while inter-vehicle communication is done using IEEE 802.11b WLAN.

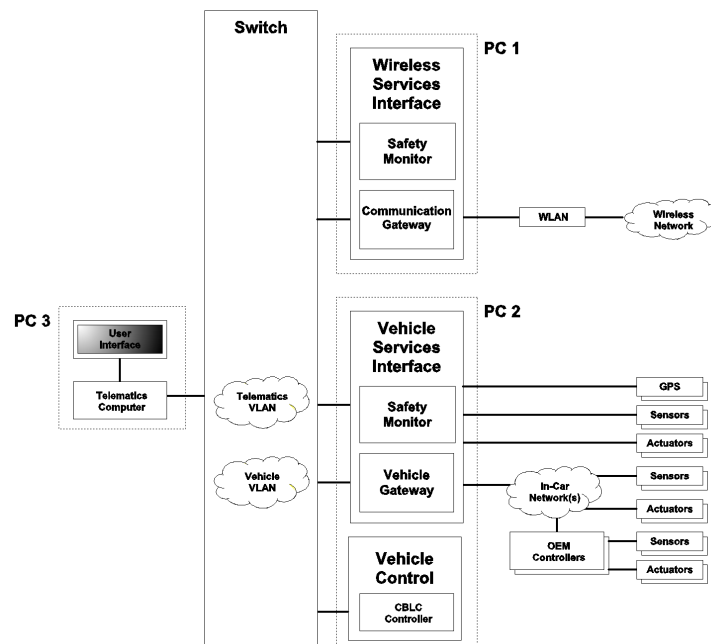


Figure 9: CarTALK hardware architecture for TNO demonstrator

Computer 1 is the *CarTALK services interface*. It functions as an access point to the other cars and it provides a safety monitor observing the proper status of the system. The CarTALK services interface software will run as a user process under the Linux operating system on a laptop PC.

Computer 2 is the *vehicle services interface & vehicle control*. It functions as an access point to the sensors, actuators and controllers within the car, it provides a safety monitor observing the proper status of these vehicle components and it executes the CBLC co-operative driver assistance system. The combined software will run as a real-time process under the RT-Linux operating system on an embedded PC.

Computer 3 is the *telematics computer*. It provides a graphical user-interface to the driver. The services will run within the Matlab/Simulink environment under the Microsoft Windows 2000 Professional operating system on a laptop PC.

CoDrive

The third example where this open automotive architecture has been applied is the CoDrive project. This project provides an intelligent Human Machine Interface (HMI) framework channeling the different flows of information from the growing number of in-vehicle services and driver assistance systems, taking into account driver workload and attention levels. For example under high traffic conditions, the user needs to pay lots of attention to it's surrounding, so feedback from unimportant services/sub-systems should be suppressed/postponed, while feedback from more important services/sub-systems should be possible if they provide a simplified HMI.

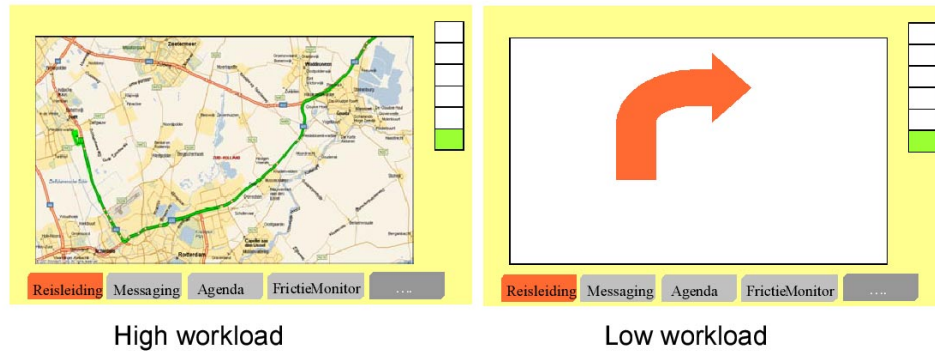


Figure 10: CoDrive example of workload dependable UI

The CoDrive system will consist of three in-vehicle computers and one computer-system located somewhere outside the vehicle as shown in Figure 11.

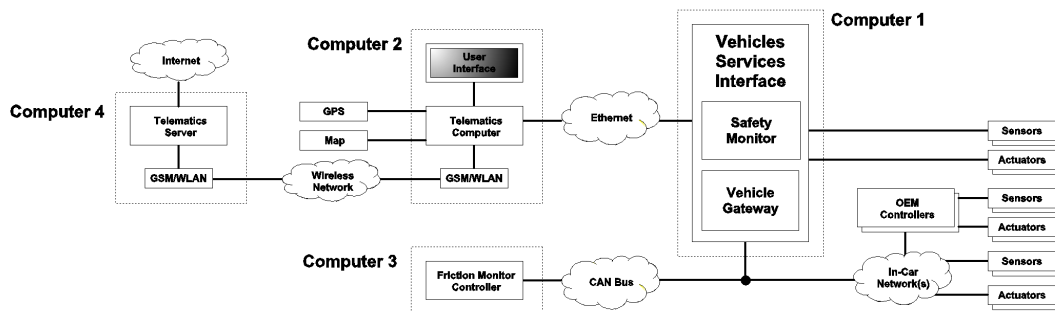


Figure 11: CoDrive hardware architecture for initial demonstrator

Computer 1 is the *vehicle services interface*. It functions as an access point to the sensors, actuators and controllers within the car and it provides a safety monitor observing the proper status of these vehicle components. The vehicle services interface software will run as a real-time process under the RT-Linux operating system on an embedded PC. The vehicle gateway communicates with the friction monitor controller over a standard CAN-bus while communication with the telematics computer is done via Ethernet.

Computer 2 is the *telematics computer*. It provides a run-time environment for the services and a touch-screen based graphical user-interface to the driver. The services will run within the JADE agent framework on a standard Java Virtual Machine from SUN while the underlying operating system is Microsoft Windows 2000 Professional. The telematics computer communicates with the vehicle gateway via an Ethernet network, while the wireless communication towards the telematics server is done via WLAN or GSM/GPRS/UMTS.

Computer 3 is the *friction monitor controller*. Currently it is a DSPACE based system executing all the algorithms associated with determining the 'friction number' between tires and road surface. The algorithms are developed on standard PC's using Matlab/Simulink, while the Real Time Workshop package is used to automatically generate the code necessary to run these algorithms on the DSPACE

computer system. The friction monitor controller communicates with the vehicle gateway over a standard CAN-bus.

Computer 4 is the *telematics server*. It is a standard computer system (or a set of computers) located in a building somewhere in the neighborhood or maybe even at a far location. This computer system communicates with the in-vehicle computers via a wireless network connection. For short-range communication a standard WLAN network might be used, but for normal operation a GSM/GPRS/UMTS telecommunication connection is preferred.

CONCLUSIONS

Several implementations of systems based upon the proposed open automotive development platform have been realized. We now have a wide set of open system components (based on commercially available and/or proprietary products) with which it is now much easier and quicker to realize other new systems.

The concept of using gateways proved to be an efficient way of interfacing existing proprietary products to this platform. Not only is it easier to guarantee system safety, it is also easier to upgrade the system with more advanced (or cheaper) proprietary products, since only the gateway application has to be expanded to match the new capabilities of the proprietary product, while all remaining applications/services can stay the same.

The choice of both CAN and Ethernet as physical communication networks made it possible to guarantee an undisturbed (real-time) behavior. Current state-of-the-art Ethernet network components and technologies provide a level of performance and capabilities which were un-thought of only a couple of years before.

The concept of openness and distribution by itself makes it very easy to support both hardware-in-the-loop (HIL) and software-in-the-loop (SIL) approaches, since to the system it is irrelevant if a hardware component is replaced by a software simulation/emulation or vice versa.

FUTURE ACTIVITIES

First of all we plan to continue extending the development platform, for example by implementing some more advanced concepts and adding new gateways.

Until now components in the development platform were often chosen in such a way that they favored the way of working as is common in a developmental phase (hardware overkill, lots of additional I/O capabilities, etc). However our focus will now also be on the mapping of existing sub-systems onto more integrated components to make the overall system also commercially more attractive (cost price, size, etc).

REFERENCES

- [1] Zoltan Papp et al: "A Runtime Framework for System Safety", *Proceedings IEEE Intelligent Vehicle Symposium (IV'2003), Columbus, OH, USA, June 9 – 11, 2003*
- [2] Zoltan Papp et al: "Multi-Agent Based HIL Simulator with High Fidelity Virtual Sensors", *Proceedings IEEE Intelligent Vehicle Symposium (IV'2003), Columbus, OH, USA, June 9 – 11, 2003*
- [3] Dirk J. Verburg et al: "VEHIL Developing and Testing Intelligent Vehicles", *Proceedings IEEE Intelligent Vehicle Symposium (IV'2002), Versailles, France, June 18 – 20, 2002*
- [4] Peter Morsink et al: "CarTALK2000: Development of a Co-operative ADAS based on Vehicle-to-Vehicle Communication", *Proceedings Intelligent Transport Systems and Services 2003*
- [5] Allard Zoutendijk et al: "Implementing Multiple Intelligent Services in an Intelligent Vehicle with a Safe Workload Aware HMI", *Proceedings Intelligent Transport Systems and Services 2003*
- [6] Tanja Vonk et al: "Co-operative Driving in an Intelligent Vehicle Environment (CO-DRIVE)", *Proceedings Intelligent Transport Systems and Services 2002*
- [7] Automotive Multimedia Interface Collaboration, www.ami-c.org
- [8] Open Service Gateway Initiative Alliance, www.osgi.org