A CAN-BASED APPLICATION LEVEL ERROR DETECTION AND FAULT CONTAINMENT PROTOCOL

Juan Pimentel, John Kaniarz

Department of Electrical and Computer Engineering
Flint, Michigan USA

Abstract: A new distributed protocol that provides an efficient and accurate method of switching between a primary node that fails and a number of backup or redundant nodes in an FTU (fault tolerant unit) has been developed. Upon node failure, the protocol accurately determines the new primary node in the system. This causes failed components to have a fail silent behaviour. With the protocol engaged, there is a seamless execution of distributed applications in systems with multiple nodes under the presence of failures. The protocol has been implemented, tested, and evaluated as part of a distributed safety-critical architecture.

Keywords: CAN, time-triggered, fault-containment, protocol, safety-critical, safety, steer-by-wire, X-by-wire, fail-silent, redundancy.

1. INTRODUCTION

Error detection and fault containment for safety-critical systems has been extensively studied at the communication architecture level (Kopetz, 2003). Examples of well known safety-critical architectures include TTA (time triggered architecture) (Kopetz,1994; Kopetz,1997) and FlexRay (FlexRay Consortium). Critical failure modes at the communication architecture level is another area that has been extensively studied. Kopetz (2003) summarizes the way the TTA architecture handles various critical modes.

It is well known that some communication services for safety-critical applications can be offered at layers 1 and 2 (i.e., the communication controller) or above, including the application (i.e., the host computer). For example the TTP/C protocols offers several safety-critical oriented services (e.g., group multicast) at layers 1 and 2 while leaving the responsibility of other services for the host computer. Unfortunately much less is known about offering services for safety-critical applications at the application level. The area of application level safety-critical protocols needs considerable more development and experience particularly solutions that meet the following requirements: scalable, flexible, simple, and cost effective. The contribution of this paper is an application level error detection and management protocol for safety-critical applications that meets these requirements. In addition, the proposed protocol is designed on top of CAN, a highly successful and widely deployed protocol for real-time applications. Application level protocols can be improved by taking into account the nature of actual applications. For our protocol, we have chosen a steer-by-wire system as our target application. A steer-by-wire system is considered a safety-critical application because component, communication, or control failures may lead to injury or loss of equipment and/or life (Bretz, 2001; Amberkar et al 2002).

The specific objective of this paper is to present an application level, CAN-based error detection and fault containment protocol for safety critical applications and to implement and evaluate the protocol in the context of a simulated steer-by-wire system (See Fig. 1). The protocol is part of a ultra-dependable distributed architecture developed at Kettering University. Details of the architecture will be provided in other publications.

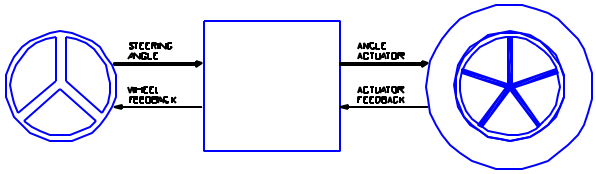

Figure 1. Components of a steer-by-wire system: hand-wheel (left), controller (center) and road wheel (right).

1.1 Motivation.

There are several trends that provided motivation to seek simple and cost effective solutions to the development of architectures and protocols for safety-critical applications. First, there are already some manufacturers that provide multiple CAN communication controllers for some of their microcontrollers. For example the S12 microcontrollers offers 5 independent CAN channels and it is expected that this trend will continue. Second, there are already some simple and inexpensive sensor and actuators analog interfaces with incorporated CAN interfaces (e.g., Microchip MCP250XX). Again, this trend is expected to continue. Third, recent advances in mechatronics indicate that there will be important developments in distributed electromechanical components (e.g., several replicated steering-wheel actuators performing the function of a single larger actuator). Last but not least, it is expected that the current trend on miniaturization of digital systems and microcontrollers will continue yielding more functionality at a lower cost.

1.1 Main Features of the Safety-Critical Protocol

The main features of the of the safety-critical protocol include the following:

1 Massive replication of components
2 CAN2.0B-based (29 bit)
3 Scaleable, transparent (compatible with normal CAN applications)
4 Simple, flexible, cost-effective

1.2 Other Relevant Work.

The idea of using a dependable time-triggered paradigm driven by the application is new in the proposed architecture and protocol described in this paper. However the idea of using a dependable time-triggered paradigm driven by the communication controller is well known (Poledna, 1996; Kopetz, 1997; Pimentel and Sacristan, 2001). Two examples of time-triggered architectures at the communication level is TTA (time triggered architecture) (Kopetz, 1997) and Flexray (FlexRay Consortium). Whereas the former is a pure time-triggered communication protocol, the latter is a combination of time-triggered and event triggered. There are also architectures that have taken an event-triggered communication protocol (e.g., CAN) and made it appear as a time-triggered communication protocol. Two architectures in this category are TTCAN (time triggered CAN) (Muller et al, 2002) and FTT-CAN (flexible time-triggered CAN) (Ferreira et al, 2002). The major difference in prior approaches from the one considered in this paper is that prior architectures still focus on the communication controller (i.e., layers 1 and 2 in the ISO reference model) whereas our architecture uses an event-triggered communication protocol (i.e., CAN) and a time-triggered synchronization protocol at the application layer.

2. PROCESS MODEL

As noted, the architecture and protocol are specific to certain safety-critical applications that could be event based or time-triggered based. Example of event based applications include body bus automotive functions such as opening a window or door. The actual time of opening or closing a window is completely random and is best modeled as an event based system. Other functions such as engine control, and steer-by-wire systems are continuous in nature and are best modeled as a time-triggered system.

Whether the application is event based or time-triggered based, it can be modeled by the system in Fig. 2 involving the system to be controlled, sensors, actuators, and controllers interconnected by a communication network. Node 1 collect measurements from sensors, send the measured values to a controller (Node 2) which in turn send control values to actuators through Node 3. Fig. 3 depicts the events and parameters of the communication process associated with nodes 1, 2, and 3. In particular we note the following significant communication events:

1. E1: sensor node queues message for transmission (beginning of cycle)
2. E2: sensor message is sent on the bus
3. E3: sensor message is received by controller node
4. E4: controller node finishes control algorithm and queues control message for transmission
5. E5: control message is sent on the bus
6. E6: control message is received by actuator node (end of cycle)

2.1 Assumptions

Regarding the process model and its communications shown in Figs. 2 and 3 we assume the following:

1. A source (i.e., sensor) node will send messages with a minimum inter-departure time. If periodic, the source node will send messages with a minimum period. This assumption is needed in order to guarantee a deterministic behaviour of the underlying CAN protocol.
2. A controller node (Node 2) will receive one or more messages corresponding to sensor values and generate controlled signals configured as one or more output messages.
3. Source nodes will generate time-triggered cycles (source nodes dictate the beginning of each cycle via events E3 in Fig. 3) that are specific to an application. Time triggered cycles are defined from the viewpoint of an application rather than from the viewpoint of communications. From a timing perspective then, source nodes are master nodes and all remaining nodes are slave nodes.
4. The communication system can support multiple different applications each with their own communication cycles (or equivalently sample periods Tn). That is the system is multi-rate.
5. A multiple application system assumes that each application has a sensor, controller (optional), and actuator.
6. Deterministic behaviour can be guaranteed (using CAN schedulability analysis) since the traffic on the network is carefully controlled thanks to assumption 1 above.
7. Messages of one cycle are not inter-mingled with messages from a different cycle. This is possible if schedulability analysis is used to ensure that all possible messages in one cycle will be sent before the next cycle begins.

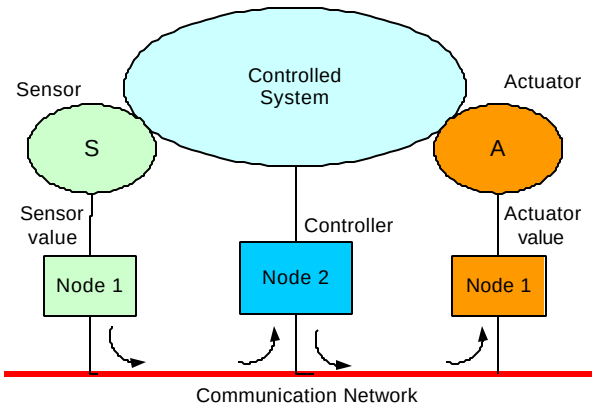

Figure 2. Model for a typical control system

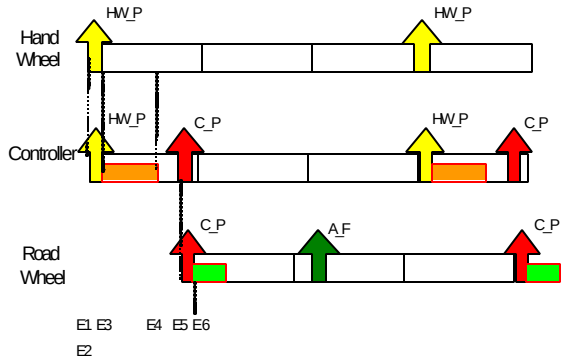

Figure 3. Events and parameters of network communications for a typical control system.

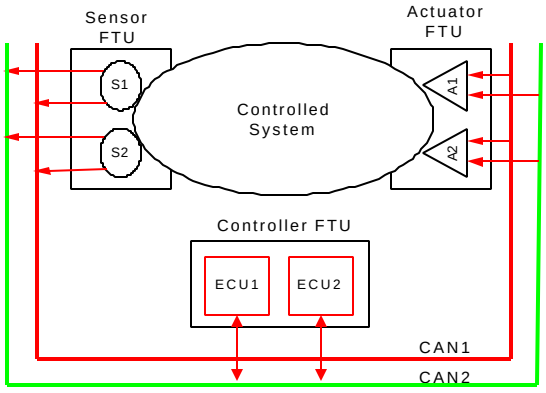

Fig. 4. A safety-critical system with replicated components.

3. FAULT MANAGEMENT PROTOCOL

To deal with component, communication, and control failures the architecture and protocols makes use of massive replication of sensors, controllers, and actuators. Component failures are detected by the absence of messages generated by the component. Communication failures are detected by absence of messages on the communication line. Detecting control failures depends on details of the architecture, other protocols, and application software which are outside the scope of this paper. Each communication node (i.e., sensor, controller, or actuator) has one or more replicated components and configured as an FTU (fault tolerant unit). Fault tolerant units are well known mechanisms to deal with fault containment of safety-critical systems. The regions protected by fault containment mechanisms are referred to as fault containment regions (FCR). Fig. 4 depicts the system of Fig. 2 where each node is replaced by an FTU consisting of two internal nodes.

If each replicated component sends an independent message corresponding to sensor signals and each message is treated independently from one another (even though they correspond to the same sensor signal) then there could be up to 32 different messages arriving at the actuator FTU. Obviously, the system would be extremely complex if it were to handle all possible messages. To reduce complexity, the main idea behind the protocol is to manage all components in an FTU in such a way that it would appear as just one node to the other FTU's. Inside each FTU, replicated components would communicate with one another to elect the so-called primary node with the remaining nodes acting as secondary, tertiary, etc. For Fig. 4, the number of independent messages arriving at the actuator FTU is reduced from 32 to 4 thus simplifying the complexity of the system.

Although the main idea behind the protocol is the same, some details are different for sensor FTU's, controller FTU's, and actuator FTU's. This is because sensor FTU's are producers of information, controller FTU's are both consumers and potentially producers and actuator FTU's are consumers. The protocol will be detailed for controller FTU's. The details of the protocols when applied to sensor and actuator FTU's vary slightly.

3.1 Fault Management Protocol for Controller FTU's.

For some application domains (e.g., automotive) controller nodes are also known as electronic control modules (ECU's) thus we will use these terms interchangeably. As noted, the fault management protocol is only executed only by members (primary and all its replicas) of a certain FTU, that is, it is an intra-group protocol. It is envision that there could be multiple applications on the same system thus the application (App) type is encoded in the protocol header of Fig. 5. For efficiency reasons, this header is encoded using some of the 29 bits of the message ID as specified in the CAN2.0B standard.

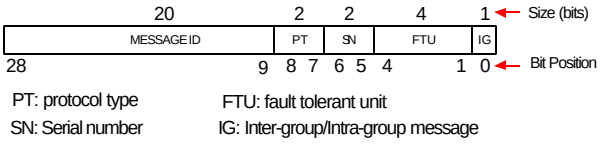

Figure 5. Protocol fields encapsulated in the 29-bit identifier of CAN 2.0B.

The main assumption made by the fault management protocol is that there is always a node acting as the Primary node. If this condition is not true, the entire system is considered to have failed and appropriate messages are generated to indicate a human user of this event. There are several protocol mechanisms that help the system converge to a state involving a Primary node and several active backup nodes. At initialization, the primary node is specified. Thereafter, each node (other than the primary) within an FTU executes the fault management protocol to determine the secondary, and additional backup rankings of all nodes in the FTU. For example, in a system with three controllers, both backup controllers will negotiate which one will act as the secondary node, and which one will act as the tertiary node.

At startup, all controllers belonging to the same FTU request that a ranking of all replicated components be started by sending the *Rank* message. The ranking procedure is controlled by a timer and ends with a ranked set of replicated components. In case more than one Rank message is received by a node before the associated timer expires, the Rank message is ignored. All other nodes in the same FTU respond to Rank messages by sending their serial numbers on the bus via the *Serial* message. As a result of all controllers sending their serial numbers, all components in the particular ECU know of all the other components on the same ECU. The backup nodes are ranked based on their serial numbers into secondary, tertiary, etc. If the primary node component fails the other controllers detect this by noting the absence of the missing message that corresponds to the Primary. Upon failure of the primary component, the secondary component will take over immediately, transmit the message and declare itself primary via a message called *Active*. If a component returns after a failure, a ranking procedure is invoked via the *Rank* message to determine a new ranked set of backup nodes (i.e., secondary, tertiary, etc.) If a new component joins the network, the same ranking procedure is performed. Thus, the primary node is not disturbed when failed nodes are repaired or when new nodes are added to the network.

3.2 Main Protocol Rules

The main protocol rules are summarised as follows:

1. A primary node is designated when the system is initialised
2. If the primary node fails, the secondary node becomes primary automatically
3. If the secondary node fails, a ranking procedure is invoked immediately to find a new secondary node.
4. If the primary node goes into a CAN error passive mode (as a result of noise on the bus) it forces a reset thus acting as it has failed to let other backup nodes to take over.
5. If a node acting as primary detects that there is another primary, the first node will issue an Identify message requesting the identity of the Primary node. If a valid response is obtained, the inquiring node ceases to be Primary and forces a ranking immediately.

The following are the main functions of the primary and secondary nodes.

Primary Node

The primary node is responsible for sending inter-group data messages. For the hand wheel node, this involves sending the hand wheel position to the ECU FTU.

Secondary Node

The secondary node is responsible for monitoring the primary node. The secondary node will check that the primary node sends its inter-group messages on time. This is achieved by monitoring event E4. If the secondary node detects that the primary node has missed sending a message (i.e., event E4 did not occur), the secondary node will immediately send the late message, becomes primary and force a ranking of all similar nodes.

Features of the Ranking Mechanism

- Ranking occurs on powerup, and when a failure is detected
- Ranking does not stop or pause a currently functioning system (A primary node does not stop sending inter-group messages during ranking. It will stop only when a new primary node takes over and is ready to send inter-group messages)
- Allows for a node to be deactivated, then return to service once fixed at a later time without disruption
- Allows for nodes to be added or removed without disruption
- When the primary node fails, messages are never completely missed, but are only delayed (for example if primary message failure timer is 100ms, the secondary will wait 110ms, the tertiary will wait 120ms (assuming that there is no secondary), etc, to send the message instead and take over as primary)

3.3 Protocol Messages

Communication on a CAN network is based on a set of unique messages each with a specific identification known as message ID. Since the fault management protocol is based on CAN, it involves a number of messages listed below.

Active Message

This message declares that the sender is the Primary, Secondary, or Subsequent node. The type of node is one parameter data of the message. The Active message indicating Primary is automatically sent by the secondary node upon detecting that the primary node has failed. The Active message is also sent by a node as a response to an Identify message indicating that the node is Primary, Secondary, or Subsequent node according to its current ranking. Nodes receiving this message will disable sending inter-group messages, and should have already been ranked by the fault management algorithm. This action also provides additional checks in case any nodes may have missed the ranking algorithm and to prevent two nodes attempting to simultaneously assume the role of Primary node.

Rank Message

This message requests that all nodes in a certain FTU (other than the primary node) start their rank procedure. If a node is already executing its ranking procedure this message is ignored. The *Rank* message is sent whenever the following events happen:

1. On power-up initialization

2. When a failure has been detected on any backup node (i.e., other than the Primary node)
3. When a new backup node enters the system
4. When a failed node re-enters the system after it has been re-initialized or repaired

When the Rank message is received by a node, the receiving node will send its serial number on the bus. During the ranking procedure, all nodes will also listen on the bus a specific amount of time and compare the all serial numbers received with that of its own. When that time period expires, the node will decide whether it is a primary, secondary, tertiary, etc. node based on the relative value of its serial number relative to the serial numbers of all remaining nodes in the FTU (other than the Primary node). Nodes with lowest serial number is declared secondary, the next lowest is tertiary, etc.

Serial Message

The *Serial* message is used during the ranking procedure and is sent immediately after a Rank message is received. As the name implies, the Serial message contains the serial number of the sending node.

Identify Message

The identify message is used whenever there is a perceived inconsistency on what nodes are primary, secondary, tertiary, etc. For example, a primary node receiving a message with the same ID as that just sent in the current cycle will conclude that there is another primary node and will issue the identify message to resolve the inconsistency.

4. REPLICATED MESSAGES ON MULTIPLE CAN BUSES

To deal with communication failures, every node in the system listens on all replicated CAN buses, and will also send messages over all replicated CAN buses. This creates a problem due to the need to replicate messages on each bus. To illustrate the problem consider the situation in Fig. 4 involving two sensors, two controllers, and two buses. When a controller node receives a message; it will receive two versions of the same message denoted $m_1(k)$ and $m_2(k)$ each on a separate bus corresponding to the k^{th} cycle. The controller will perform its control functions and generates messages $C(m_1(k))$ and $C(m_2(k))$. The controller then will proceed to send these two messages on each CAN bus thus generating a total of four messages $C_1(m_1(k))$, $C_2(m_1(k))$, $C_1(m_2(k))$, and $C_2(m_2(k))$ where $C_i(m_j(k))$ stands for controller message on the i^{th} bus corresponding to message $m_j(k)$. This mechanism creates redundant traffic, and a potential cause for congestion. In order to alleviate the problem, the protocol has a *message discarding* mechanism. Since all messages are sent in duplicate, once on each bus, when a message is received by the controller, the message with the same message ID is ignored on the other CAN bus within the same cycle. For example, if a message is received on CAN bus 1 with message ID 0x00001000, the node will ignore the next message with ID 0x00001000 on CAN bus 2. Effectively, this mechanism will eliminate the problems posed by the replicated messages created by sending messages on two CAN buses.

Sequence Numbers

The message discarding protocol is implemented using sequence numbers. Each message contains a sequence number that is sent as the first data byte. This number is used to encode the order in which the producer node put the same message on the various buses. The consumer will only keep the first message received with the same message ID; the other messages with the same ID on the other buses will be ignored. This guarantees that no old data will be passed along to another node group. Sequence numbers also increase the likelihood of ignoring any errant message that may show up. Sequence numbers add to the robustness of the protocol.

CAN Message ID

CAN Message IDs are used to identify the sender of the message as well as the type of message being sent. The bit assignments for message identification is shown in Fig. 5.

Intra-group Message

Messages for nodes within the same FTU are referred to as intra-group messages. For example, all hand wheels nodes (primary and backups) send messages to each other during arbitration. These messages are considered intra-group messages since the intended targets are also hand wheels.

Inter-group Message:

Messages for nodes in different FTU's are referred to as inter-group messages. For example, the hand wheel FTU will send a wheel position message to the controller FTU. Since the message is being sent from one type of node to another, it is an inter-group message.

5. IMPLEMENTATION, TESTING, AND VERIFICATION

We have implemented the proposed fault management algorithm in CANoe using CAN2.0A (11 bit) for the steer-by-wire application described in section I. We have extensively tested and verified the protocol behaviour as well as the steer-by-wire behaviour. Since CANoe is also a simulator, we simulated the detailed operation of the hand-wheel as well as road wheels. We tested failures in the simulation by simply turning off the corresponding node (using a simulated switch in CANoe). Because the entire system was simulated using CANoe, we could not extensively test failures of the CAN bus (unless the entire bus was disconnected). The CAN2.0A implementation was our first and we are now in the process of simulating the protocol on CANoe using CAN2.0B (29 bit) and implementing an actual steer-by-wire system on a golf cart using Motorola S12 microcontrollers as target ECU's and

Microchip MPC 250XX devices as sensor and actuator nodes.

Failure Modes

Failure modes can be dealt with at the architectural level, at the protocol level, and at the application level. In this paper we only deal with failure modes at the protocol level. In the following, we describe the behaviour of the protocol under single-point and multiple-point failures.

Single-Point Failure

We tested that the system recovered from the following single point failures and verified their fault-silent behaviour.

- One of the Hand Wheel node fails.
- One of the Road Wheel node fails.
- One of the ECU controllers fails.
- One of the CAN buses fails.

Multiple-Point Failure

Any combination of the above single-point failures (taken two at a time) was tested and verified their fault-silent behaviour.

We identified and detected several multiple point failures and dealt with them in an appropriate manner. For each of the multiple-point failures listed below, we straightened the road wheels and applied the brakes.

- Both of the Hand Wheel nodes fail.
- Both of the Road Wheel nodes fail.
- Both of the ECU controllers fail.
- Both of the CAN buses fail.

6. SUMMARY AND CONCLUSIONS

This paper has presented a CAN-based application level error detection and management protocol for safety-critical applications. The time-triggered paradigm is important for safety-critical applications. From the viewpoint of the OSI reference model, there are two main approaches for incorporating the time-triggered paradigm in safety-critical applications: bottom-up (i.e., starting at layers 1 and 2 and moving to higher layers) or top-down (i.e., starting at the application layer and moving to lower layers). Most approaches use the former, this paper describe one example of the latter. In addition, application level protocols add additional degrees of freedom for the design of ultra-dependable systems by taking into account the nature of actual applications. For our protocol, we have chosen a steer-by-wire system as our target application. The protocol has been simulated and implemented in CANoe and initial test and evaluation results are very encouraging.

REFERENCES

Bretz, E.A., (2001), By-Wire Cars Turn the Corner, *IEEE Spectrum*, Vol. 38, No. 4, pp.68-73.

Poledna S. (1996), *Fault-Tolerant Real-Time Systems*, Kluwer Academic Publishers.

L. Lawrenz (1997), *CAN System Engineering: From Theory to Practical Applications*, Springer.

Kopetz, H.(2003), Fault Containment and Error Detection in the Time-Triggered Architecture. In *Proc. IEEE Int. Symp. On Autonomous Descentralized Systems, ISADS 2003*, pp. 139-148.

H. Kopetz, G. Grunsteidl, (1994), TTP – A protocol for Fault-Tolerant Real-Time Systems. *IEEE Computer*, pages 14-23.

Kopetz H. (1997), *Real-Time Systems, Design Principles for Distributed Embedded Applications*, Kluwer Academic Publishers.

FlexRay Consortium, http://www.flexray-group.com

Pimentel, J.R, and Sacristan, T. (2001), A Fault Management Protocol for TTP/C. In Proc. *IECON'01 (Int. Conf. On Industrial Electronics)*, pp. 1800-1805, Denver, CO.

Amberkar, S., D'Ambrosio, J.G., Murray, B.T., Wysocki, J., and Czerny, B.J. (2002). A System-Safety Process for By-Wire Automotive Systems, *SAE Congress paper*.

Muller, B., Fuhrer, T., Hartwick, F., Hugel, R., and Weiler, H. (2002). Fault Tolerant TTCAN Networks. In *Proc. 8th Int. CAN Conference*, Las Vegas.

Ferreira, J., Pedreiras, P., Almeida, L., and Fonseca, J. (2002). Achieving Fault Tolerance in FTT-CAN. In *Proc. 2002 Int. Workshop on Factory Communication Systems (WFCS2002)*, pp. 125-132, Vasteras, Sweden.