Preprints of the 19th World Congress
The International Federation of Automatic Control
Cape Town, South Africa. August 24-29, 2014

Mixed-Criticality Systems based on a CAN Router with Support for Fault Isolation and Selective Fault-Tolerance

Roland Kammerer * **Roman Obermaisser** ** **Mino Sharkhawy** ***

* *Vienna University of Technology, Austria*
kammerer@vmars.tuwien.ac.at
** *University of Siegen*
roman.obermaisser@uni-siegen.de
*** *Vienna University of Technology, Austria*
mino.sharkhawy@student.tuwien.ac.at

Abstract: In many application domains there is an increasing trend for mixed-criticality systems with functions of different assurance levels on shared computing platforms. Today's CAN-based platforms do not support the requirements of mixed-criticality systems. A single CAN bus provides low cost, real-time support and flexibility for applications where the communication service is not safety-relevant. Fault-tolerance extensions for CAN impose incompatibility to legacy applications, high cost and overhead for the entire CAN communication.
This paper introduces a CAN infrastructure for fault isolation and selective fault-tolerance, which permits a balanced trade-off between cost and fault-tolerance for each subsystem of a mixed-criticality system. We introduce replicated CAN routers that perform fault isolation based on a priori knowledge of the permitted behavior of CAN nodes. Fault masking is supported selectively through the redundant transmission of messages from safety-critical subsystems. The CAN routers perform input agreement on pending messages for replica deterministic behavior, as well as output agreement on the delivery status of messages. Software layers hide the fault-tolerance mechanisms to establish compatibility to legacy software. The benefits of the proposed communication infrastructure are demonstrated in a simulation of an example system.

Keywords: CAN, CAN router, Mixed-Criticality, Fault Isolation, Selective Fault Tolerance, Redundancy

1. INTRODUCTION

The need to reduce the number of nodes and cables leads to mixed-criticality systems, where multiple functions with different importance and certification assurance levels are integrated using a shared computing platform. Mixed-criticality is the concept of allowing applications at different levels of criticality to seamlessly interact and co-exist on the same networked distributed computational platform.

Controller Area Network (CAN) (see ISO-11898 (1993)) is one of the most-widely used protocols in applications where the communication network is not safety-relevant. In the automotive domain, today CAN is deployed in every produced car although safety-relevant electronic functions employ either internal fault-tolerance within the CAN nodes or use other communication networks (e.g., FlexRay).

CAN lacks essential properties for systems that have substantial timeliness and dependability requirements. The CAN protocol does not support fault-tolerance by network redundancy and multiple bit-flips can result in inconsistent message disseminations (i.e., no atomic broadcast mechanism, see Kaiser and Livani (1999)). Furthermore, the mechanisms for achieving a faulty node's self-deactivation may cause substantial periods of inaccessibility (2.5ms at 1 Mbps, see Veríssimo et al. (1997)).

In previous work solutions for addressing fault-tolerance by active redundancy (see Rufino (1997)) and supporting a consistent atomic broadcast mechanism have been developed (refer to Rufino et al. (1998); Kaiser and Livani (1999); Livani (1999)). For example ReCANcentrate (Barranco et al. (2005)) and CANbids (Proenza et al. (2012)) provide fault-tolerance by using star couplers but do not tackle support for mixed-criticality systems.

This paper introduces fault-tolerance for CAN based systems based on redundant CAN routers. The presented solution focuses on mixed-criticality systems, since safety-relevant nodes can exploit media redundancy via two redundant routers and a redundancy-management layer at the nodes. Non safety-relevant nodes perceive a conventional CAN bus as the network interface and messages are exchanged via a single router. Redundant and non redundant communication shares the same network infrastructure.

A coordination protocol between the routers ensures a consistent state and a consistent redirection of CAN messages despite unavoidable timing differences at the communication links to the redundant routers. As a consequence safety-critical CAN nodes consistently perceive the same temporal order of messages despite an arbitrary failure of another CAN node.

The paper is organized as follows: In Section 2 we provide an overview about the system model and the services of the

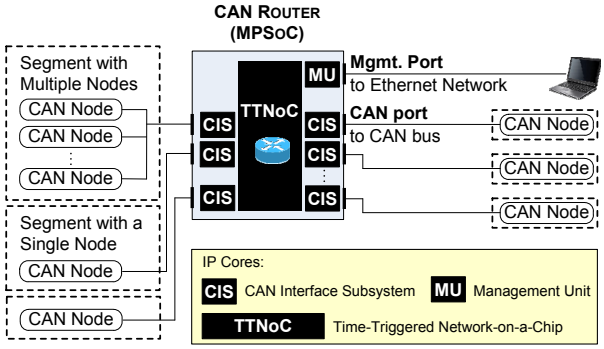

Fig. 1. System model of the CAN router

CAN router. We continue with the requirements for a redundant router in Section 3. Section 4 discusses the design of algorithms necessary to establish a consistent system state in a mixed-critical router-based system. In Section 5 we evaluate the proposed algorithms with the help of a model of the router, and discuss the results in Section 6. Finally, we conclude the paper in Section 7.

2. FAULT-TOLERANT CAN ROUTER

In previous work (see Obermaisser and Kammerer (2010); Kammerer et al. (2012)) an MPSoC based CAN router was introduced that supports fault containment and fault masking. Figure 1 gives an overview about the CAN router. It is a device that receives messages at its ports, processes these messages and selectively forwards them to one or more destination ports. Every router port is connected to a CAN bus that consists of one or more CAN nodes. Every router port features its local Central Processing Unit (CPU), memory, and software. This design eases the distribution of the overall workload compared to a single processor that has to deal with the whole workload and allows better scalability (i.e., adding additional router ports). We denote the combination of CAN controller, CPU, memory, and software as a CAN Interface Subsystem (CIS). A CIS can be in the role of a source CIS and/or destination CIS. It is in the role of a source CIS if it received a CAN message from its connected CAN segment and in the role of a destination CIS if it delivers messages to its CAN segment. A CIS samples the CAN bus and processes new messages periodically. We call these periods activity cycles which are triggered faster than the minimum interarrival time of CAN messages.

The router implements a star topology and is designed for fault detection and fault containment. For the sake of brevity we provide a short overview about the most important services, but the interested reader is referred to papers describing the services in more detail (see Obermaisser and Kammerer (2010); Kammerer et al. (2012)).

- Message rate control: Monitoring and enforcing minimum and maximum interarrival times of CAN messages. If a message violates the specified minimum interarrival time, the router blocks that message and logs the violation. With the help of that service a faulty node like a babbling idiot can only have a limited influence on other CAN segments.
- Message multicasting: The router allows for selective multicasting of messages instead of broadcasting every message to every other node. Selective multicasting forwards a message to nodes interested in that particular message. On one hand this allows to use the existing bandwidth more efficiently and on the other hand this separation increases composability (see Kammerer et al. (2013)).
- Identifier validation and translation: The router checks for every CAN message that arrives at a CIS, if its Identifier (ID) is in a (re)configurable set of allowed IDs. If this is not the case, the message gets blocked and the violation is reported. Identifier translation maps the ID of an incoming message to another ID before it is delivered to the destination CAN segment. This service is mainly used to ease legacy system integration.
- Message checks and content translation: If desired, the router can check the content of a message (e.g., range checks) and it can translate the content of messages (e.g., converting between different measuring units).
- Message scheduling: Because of the star topology and the resulting distinct CAN segments, it might happen that two distinct source CISes send a message to the same destination CIS in the same activity cycle. Therefore, the destination CIS also receives these two messages in the same activity cycle and has to decide which message it should send first. To resemble the priority driven arbitration of a traditional CAN bus, the destination CIS adds all the received messages to a priority queue sorted by CAN IDs and tries to send the first message in the queue. In the following activity cycles all new messages are again added to the priority queue. We overwrite old messages with the same CAN ID in the queue. On one hand this simplifies the implementation tremendously (i.e., we know the maximum size of the priority queue before runtime) and we consider newer messages always more important than old/outdated messages. This service is the only one that has to be slightly modified to support mixed-criticality. We realized these higher level services by implementing a store and forward behavior.

3. REDUNDANT CAN ROUTERS

A single CAN router already allows a basic level of mixed-criticality integration. By cleanly decoupling CAN segments with the help of a star topology and by selective multicasting, High-Critical CAN Nodes (HCNs) can be separated from Low-Critical CAN Nodes (LCNs). Even if LCNs send messages to higher-critical CAN segments, the influence of LCNs can be bound with the help of the message rate control service, especially by enforcing minimum interarrival times of messages.

With redundant CAN routers we achieve the two goals which are the key contributions of this paper:

(1) Increased dependability: While a single router provides extended fault detection and isolation mechanisms compared to traditional bus-based setups, a single router can be seen as a single point of failure. By means of redundancy we avoid this single point of failure.
(2) Finer grained mixed-criticality: Even in case LCNs influence the timing of HCNs at one CAN segment, HCNs feature a second connection to a copy of the router that is not influenced by LCNs.

3.1 System Model for Redundant CAN Routers

A fault-tolerant system is based on two replicated CAN routers that are interconnected by a so called interlink. Depending on

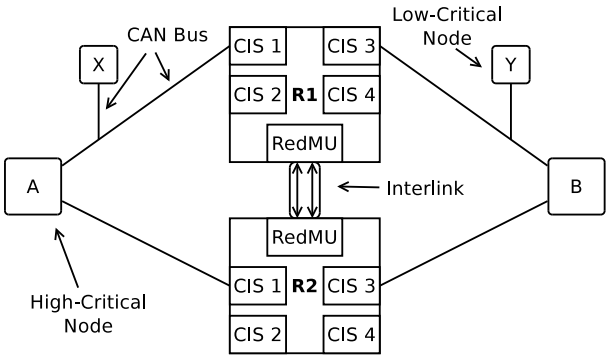

Fig. 2. Redundant router setup with two HCNs (A, B) and two LCNs (X, Y)

the criticality of a CAN node, it is connected via standard CAN buses to one or both routers. Figure 2 shows a setup consisting of redundant routers. In the following we describe the functionality of all the components:

- CAN Nodes: These nodes are the source and/or the destination of CAN messages. LCNs are connected to only one router and are not changed in any way compared to legacy CAN nodes (e.g., nodes X and Y in Figure 2). HCNs, like the nodes A and B, provide two CAN controllers and a redundancy-management software layer to interact with the replicated controllers. This layer handles the interaction between the application software executed on a CAN node and the underlying controller/hardware layer. The layered approach provides transparent fault-tolerance (see Bauer (2001)) and eases legacy software integration because the application software does not need to care about replication.

  The software layer is also used for deduplication of CAN messages at the destination CAN nodes (see Section 4).
- CAN Buses: CAN buses are non redundant and conform to the CAN standard. They interconnect CAN nodes and CAN routers.
- CAN Routers: Routers provide the services described in Section 2 and additionally handle the functionality that is required for replication.
- Router Interlink: Redundant routers have to exchange information about messages they received from CAN nodes and if none, one, or both routers were able to deliver a CAN message to the destination node(s). This information gets exchanged via a redundant interlink. In addition to the before mentioned agreement on messages, the interlink is also used to detect whether the opposite redundant router is still operational by continuously sending and receiving alive-messages.
- Redundancy Management Unit (RedMU): The RedMU is a new component of the router that is responsible for redundancy management. The advantages of using a dedicated component will be discussed in Section 4. The RedMU is similar to a CIS (i.e., a dedicated component with its own CPU and memory). It does not feature a CAN bus or CAN controllers itself. It acts as a proxy for high-critical CAN messages and is connected to the interlink.

3.2 Agreement on Consistent State

In order to guarantee a consistent system state, the redundant routers have to agree on the input they receive from connected

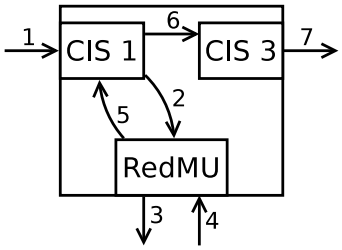

Fig. 3. High-critical message flow in a redundant router setup

CAN nodes as well as on the output the redundant routers sent to destination nodes. Without these agreements messages could be lost or messages could get duplicated. In the following we discuss these agreements in detail:

Input Agreement Different cable lengths and slightly different sending times of the replicated CAN controllers of a HCN make input agreement necessary. Even if the CAN controllers were perfectly synchronized and the cable lengths were exactly the same, routers still need to decide if a message was already processed or not, no matter if the routers operate time or event-triggered. The CAN routers operate in cycles of activity and the routers themselves are not perfectly synchronized. It is not even possible to synchronize the individual time bases of the routers in such a way that it would be guaranteed that the redundant CAN messages are processed in the same activity cycle (refer to Kopetz (1998)). CAN messages are event-triggered and can occur indeterminately.

In a redundant setup the routers need to agree if a message has already been processed or not. In order to allow message checking especially in the temporal domain, the routers also have to agree on timestamps when a message was received. If two routers would use timestamps using only their local time, it could lead to situations where one router discards a message because of a minimum interarrival time violation and the other one does not discard it. This could happen because the two routers cannot be perfectly synchronized and one router detects a violation of the minimum interarrival time whereas for the second router the message interarrival time is within the specified bound.

Output Agreement Assuming that the agreement at the input side of the router works as expected, there has to be a second agreement phase at the output of the CAN routers. Again, the routers are synchronized, but only with a certain precision. Even in a perfectly functional setup it might happen that two routers try to send the CAN message to their final destination and that one router successfully sends the message and the other one is not able to deliver its message. This might be the case if the destination HCN tries to send a message itself. Assume a setup like the one in Figure 2, where R_1, R_2, and B want to send CAN messages. R_1 and R_2 want to deliver a message from A to B and B simply wants to send a CAN message to the routers. Due to different cable lengths and because the routers are not perfectly synchronized, it might happen that R_1 wins the arbitration race against B, but R_2 loses it because the replicated CAN controller on B connected to R_2 started sending faster.

Message Flow in a Redundant Router Setup Figure 3 shows a typical message flow from a source HCN to a destination HCN. First, the application of a HCN forwards the message it intends to send to the software layer residing at the HCN. This software

layer takes care that this message gets delivered to at least one CAN router.

It is possible that the message is received by the CISes of redundant routers at different activity cycles. Source CISes first receive a CAN message from their CAN segment and simply forward it to the RedMU. The RedMUs then agree on the message and on the timestamp it was received. This agreed information is then sent back to the source CISes. Therefore, source CISes only receive agreed CAN messages and agreed timestamps from the RedMU. These agreed timestamps are the only source that is used for message checks in the temporal domain. There is no point where the local time of the routers is used for comparisons. Then, the CISes execute all the logic that is executed in a single CAN router setup (e.g., message checks, looking up the destination of CAN messages) and then forward the message as usual to their destination CIS(es), which eventually send the message.

At first it might seem strange that messages are sent from source CISes to the RedMU and then back to the source CIS again. While forwarding agreed messages from the RedMU directly to the destination CISes is possible from a technical point of view, it would have severe disadvantages. CISes already contain the logic for checking CAN messages in the time and value domains. Shifting this functionality to the RedMU would increase the workload of the RedMU. It would also render the RedMU as a single point of failure. With our design, where message checking is still executed by the CISes, we can fall back to the behavior of a single non-redundant CAN router in case the RedMU fails (e.g., in a never-give-up scenario). If we now assume that checking messages is better handled by CISes, then it is obvious why we need a back-channel from the RedMU to the source CISes.

We favored a dedicated RedMU component for several reasons. Shifting additional logic required for redundancy management to CISes would increase their complexity. Additionally, the overhead for executing code required for redundancy management would have a negative impact on the already stringent timing requirements of CISes. With our design the impact on CISes is as minimal as possible.

After the agreed CAN message is forwarded to the destination CIS, every router tries to send the next message pending in its queue. After the routers tried to send a message, they have to agree if a router was able to deliver a message. If exactly one router was successful, the redundant message gets removed from the redundant router.

As both routers try to send a redundant message in the same activity cycle, it is of course possible that both messages are received at the redundant CAN controllers of the destination HCN. The software layer at the destination node ensures the deduplication of redundant messages. After deduplication only one copy of the message is forwarded to the application executed at the destination node.

Summarizing, the rest of the paper provides solutions for these challenges:

- Sending HCNs: It might happen that a message is sent to one router but not to the other (e.g., the router wanted to deliver a message on the replicated CAN bus and was faster).
- Router Input Agreement: Redundant CAN messages sent from a HCN might be received by the routers at different activity cycles.
- Router Output Agreement: CAN messages sent to a destination CAN node might be delayed. One router wins the arbitration to the destination CAN node, the other one loses it.
- Receiving HCNs: A high-critical destination CAN node has to decide when it should forward a message to its application layer. How long has it to wait until it assumes that a message received at one redundant CAN controller will not arrive at the redundant copy? How do these nodes handle deduplication of messages received at both CAN controllers?
- Mixed-criticality: How do redundant routers process messages that only arrive at one router? Are the proposed agreement protocols sufficient to handle mixed-criticality setups? What are the limitations on the topology in which mixed-critical nodes are connected to the router?

After providing our fault hypothesis, we will elaborate on these challenges in Section 4.

3.3 Fault Hypothesis

The fault hypothesis defines the assumptions about the types of faults that a system based on the CAN router has to tolerate. This fault hypothesis is needed to design the fault-tolerance algorithms of the CAN router and to perform its validation. We assume that each CAN router is a fault-containment region. The assumed failure mode of the CAN router is fail-silence, which is justified by the internal error detection mechanisms and the fault-tolerant MPSoC architecture (see Paukovits (2008)) used for the CAN router. In addition, each node with its CAN buses is considered as a fault-containment region. A CAN node can exhibit an arbitrary failure mode in the value or time domain. We assume a single fault between fault containment regions. For example the overall system is operational if one CAN bus from a HCN connected to the router is faulty. In that case the redundant router receives the message on the redundant CAN bus. Extending this scenario where a HCN sends a message to another HCN via the router, the system tolerates a single bus failure on the receiving side of the router, a single fault on the router interlink, and a single fault on one of the buses that connects the router to the destination node. Possible influences in the time domain for mixed-criticality systems will be discussed in the following sections, especially in Section 6.

4. DESIGN AND ALGORITHMS USED FOR REDUNDANT MIXED-CRITICAL ROUTERS

In this section we describe how we intend to solve the challenges stated in Section 3.2.

4.1 Software Layer at High-Critical Sending CAN Nodes

Conceptionally, the software layer resides between the application executed on a CAN node and the underlying replicated CAN controllers. The software layer provides a simple Application Programming Interface (API) to ease legacy software integration. Whenever this layer receives a message from the application, it stores the message in the send buffers of the replicated CAN controllers and issues for both controllers a

single transmit request (i.e., single-shot behavior in CAN terminology). Additionally, the layer has to keep track if the last message was successfully sent to at least one CAN router. As long as one copy was received by one router, it clears both transmit buffers of the CAN controllers. Only if both send attempts failed, it issues a retransmit.

4.2 Router Input Agreement for High-Critical Messages

For high-critical traffic the RedMU might receive new data from its connected CISes in every activity cycle. It is now the purpose of the RedMU to communicate via the interlink to the other RedMU, agree on the input and to decide if a new message has to be sent back to the source CIS for further message processing. Algorithm 1 shows the input agreement that is executed in every activity cycle for every CIS. Assume that the RedMUs exchanged their local input (i.e., the variable `local`) and got back a message from the other RedMU (i.e., the variable `remote`). The variable `agreed` always holds the value of the last agreed message. The variable `agreed` is initialized to `nil` before the algorithm gets executed the first time. Further assume that if there is no new CAN message to agree, CISes send a `nil` message.

Require: RedMUs successfully exchanged their messages, local message is denoted as *L*, remote message as *R*.

```
1:  if R.data ≠ nil then
2:      if R.data ≠ agreed.data then
3:          if R.data = L.data then
4:              L.time ← min(L.time, R.time)
5:          else
6:              L.time ← R.time
7:          end if
8:          L.data ← R.data
9:      end if
10: else                                        ▷ R.data = nil
11:     if L.data = agreed.data then
12:         L.data ← nil
13:         L.time ← nil
14:     end if
15: end if
16: agreed.data ← L.data
17: agreed.time ← L.time
18: sendMsgToCIS(agreed)
```

Algorithm 1: Input agreement protocol

First, the algorithm checks if there is a new message from the remote side (Line 1). For now assume that the remote side did not send a new message and the algorithm continues at Line 10. The only thing the RedMU has to decide is if the new local message has already been sent as the agreed message in the last activity cycle. This can happen if the remote CIS received the message in activity cycle n, but the local one in cycle $n+1$ (see Section 3.2). Therefore, it compares its local message to the stored agreed one (Line 11) and if they are equal, the message was already processed and the local message gets discarded.

The other possibility is that we get a new message from the remote side. If that is the case (Line 1), we then check if this remote message is equal to the last agreed message. If this is not the case, therefore it is an unprocessed new message, the RedMU overwrites its local message. After executing this algorithm the `local` variables hold the new agreed value which is then stored to the variable `agreed` in Line 16 and Line 17.

Agreeing on a timestamp is very similar to agreeing on the data content. If we got new data from the remote side we have to check if we also got that message on the local side (Line 2). If this is the case we agree on the minimum timestamp of both messages in order to be on the safe side. If we did not receive a message locally, we agree on the timestamp received from the remote side.

4.3 Router Output Agreement

Algorithm 1 showed that routers agree on the same value for high-critical traffic. This value is then sent back to the source CISes which finally send the message to the destination CIS(es). For reasons explained in Section 3.2 it might happen that one router successfully sends the message to the destination CAN node and the other router does not. We solve this issue in the following way: Destination CISes try to send a message exactly once in one activity cycle (single-shot). Then, destination CISes have to wait before they can check if the message was successfully sent. Checking immediately would not be correct, because activity cycles are shorter than the time it takes to send a CAN message. After waiting and checking the status of the CAN controller, CISes send the current state to the local RedMU. These messages contain the CAN ID and boolean values if the message was successfully sent and if it was a high-critical message. The RedMUs then decide how to proceed and send a message back to the CISes. Algorithm 2 shows the proposed output agreement. In this section we explain the steps undertaken because of redundancy, the part of the algorithm that adds support for mixed-critical setups gets explained in Section 4.5.

The algorithm is executed by the RedMU for every CIS acting as a destination CIS. It follows simple, but important rules: CISes are synchronized within one activity cycle, so is their agreement. Both CISes send their status (e.g., successful, failed, or a `nil` value if they did try to send), then wait for the RedMU to decide, act accordingly in the next - and only the next - activity cycle, wait if a potential transmission was successful and then send their new status to the RedMU. The sequence of sending status, waiting for the RedMU, reacting, waiting for the CAN controller is synchronized and periodic.

The algorithm first checks if the messages are high-critical. For now assume a non mixed-critical setup where both routers always tried to send high-critical message that are input-agreed and therefore have the same CAN ID. In that case the algorithm continues at Line 13. If both routers failed to send the message, they have to retry. If at least one successfully sent the message, they continue with the next message in the priority queue.

4.4 Software Layer at High-Critical Receiving CAN Nodes

The software layer at HCNs is responsible for deduplication of redundant messages. The destination CISes of redundant CAN routers will try to send messages within the same activity cycle, but there will be an unavoidable offset when these redundant messages are received by a HCN. Factors for this offset include that routers can only be synchronized within a certain precision and that the cable length from a HCN to one router is probably longer than the cable to the other router. We assume that the routers are synchronized with a precision of one activity cycle.

Require: RedMUs successfully exchanged their messages, local message is denoted as *L*, remote message as *R*. Messages contain the CAN ID (*ID*, default: *nil*), the send status (*sent*, default: *false*), and if the message is critical (*crit*, default: *false*).

```
 1: if L.crit = true and R.crit = true then
 2:     msg ← AGREECRITICAL(L, R)
 3: else if L.crit = false and R.crit = false then
 4:     msg ← AGREENONCRITICAL(L)
 5: else
 6:     msg ← AGREEMIXED(L, R)
 7: end if
 8: if L.ID = nil then
 9:     msg ← continue
10: end if
11: sendMsgToCIS(msg)
12:
13: function AGREECRITICAL(L, R)
14:     if L.sent = false and R.sent = false then
15:         return retry
16:     else
17:         return continue
18:     end if
19: end function
20:
21: function AGREENONCRITICAL(L)
22:     if L.sent = true then
23:         return continue
24:     else
25:         return retry
26:     end if
27: end function
28:
29: function AGREEMIXED(L, R)
30:     if L.crit = true then
31:         msg ← AGREENONCRITICAL(L)
32:     else
33:         if L.sent = true then
34:             if R.sent = true then
35:                 msg ← (del_R.ID, continue)
36:             else
37:                 msg ← continue
38:             end if
39:         else
40:             if R.sent = true then
41:                 msg ← (del_R.ID, retry)
42:             else
43:                 msg ← retry
44:             end if
45:         end if
46:     end if
47:     return msg
48: end function
```

Algorithm 2: Output agreement protocol

If a HCN detects identical CAN messages at its redundant CAN controllers that arrive within the duration of one activity cycle, these messages are redundant and the software layer forwards one copy to its application layer. The HCN does not need to be synchronized to the routers, it just needs the ability to measure one activity cycle. An important fact in this context is that the time the routers take for their output agreement does not influence the receiving HCN. Routers start to send redundant messages within one activity cycle. Even if routers would need hours for the output agreement, it is guaranteed that they will not resend the message as long as one message was successfully sent. A problem would arise if routers would send an identical - but new - message within one activity cycle. This can never happen as activity cycles are shorter than the time it takes to send a CAN message. Routers have to wait until they are able to agree if a message was successfully sent or not.

4.5 Support for Mixed-Critical Traffic

In order to support mixed-critical traffic all CAN nodes interested in this low-critical (i.e., non redundant) traffic have to be connected to the same router, but without any limitations on the topology. For example, some of the LCNs can be connected to the same CAN segments, while others are connected to distinct CAN segments connected to distinct CISes. It is even possible to have multiple low critical subsystems, where some are connected to the first router and others are connected to the second one, as long as these subsystems do not need to receive messages from LCNs connected to the distinct router. Messages from LCNs do not cross router boundaries.

Input agreement modifications The router's configuration contains an additional boolean flag for every valid CAN ID that is set if this message is of high criticality. Only if a message is flagged as high-critical, it is sent to the RedMU. If it is low critical, it is directly processed by the CIS and if all checks succeed it is sent to the destination CIS(es). Without executing the input agreement stage and because high-critical messages are replicated to the second router, we can ensure that low-critical messages do not have a timing impact on high-critical messages. For example, a low-critical message that blocks a high-critical message on one CAN bus has no timing influence because the high-critical message gets also received on the redundant router (assuming a fault free setup). Routers have to agree on high-critical messages, but low critical messages do not block this agreement. In order to avoid blocking from LCNs, we suggest that all LCNs are connected to the same router. It might happen that the router has to forward two messages from a source CIS to destination CISes in one activity cycle. This happens if the router has to forward a low critical message from a LCN and a high critical message sent back from the RedMU. We deal with that situation by doubling the bandwidth in the Time-Triggered Network-on-a-Chip (TTNoC). Our TTNoC is triggered by the system frequency which is considerably faster than the activity cycles executed in the routers. Therefore, doubling the bandwidth does not have a practical impact.

Output agreement modifications In a mixed-critical setup there might be a situation that the priority queues of the redundant CISes contain different entries. In the simple case both CISes tried to send low-critical messages. In that case Algorithm 2 continues at Line 21 and the RedMUs only have to care about their local messages. If the local message was successfully sent, they continue with the next message in the priority queue, if not, they resend their local message.

Now assume a setup similar to that one shown in Figure 2. It might happen that the priority queue of a destination CIS at router R_1 contains a low-critical message at the head of the priority queue and a high-critical message after it. For the sake of simplicity, assume that the message with ID 3 is the head of the queue followed by ID 5. The router R_2 which only connects

Table 1. Output agreement for mixed-criticality

Router	Queue	Sent	Sent	Sent	Sent
R_1	5,3	T	T	F	F
R_2	5	T	F	T	F
R_1 gets		$del_{R.ID}$, cont	cont	$del_{R.ID}$, retry	retry
R_2 gets		cont	retry	cont	retry

HCNs lacks the entry with ID 3, but also contains the message with ID 5 (i.e., the redundant high-critical message). Again, the routers try to send the messages and send their status to the RedMU. We have to differentiate four cases which are shown in Table 1. In the following we discuss these four cases from left to right:

If both CISes successfully sent their messages, R_1 has to delete the entry for the redundant message which was successfully sent by the remote node from its priority queue. R_2 can continue with the next message. If the low critical message was successfully sent but the high critical one was not, the router that sent the low critical message can continue, the router that tried to send the high-critical one will retry. The third case is the one in which the high-critical message was sent, but sending of the low-critical failed. In that case the router that successfully sent the high-critical message can simply continue. The router that failed to send the low-critical message has to delete the successfully sent message from its queue as it was successfully sent by the redundant router and then it has to retry to send its own failed message. In case both routers failed to send a message, both have to retry. Starting from Line 29, Algorithm 2 executes the described steps.

While Algorithm 2 provides the base for a successful output agreement, there is one last step that deserves a closer look, namely the $del_{R.ID}$ function executed at the destination CISes. Assume the most left case in Table 1 where both CISes sent their messages successfully. Then, as discussed, the CIS at R_1 has to delete the message with the CAN ID 5 from its queue because it was successfully sent by R_2. But what if in the meantime new messages with ID 5 arrived at the destination CISes? In that case R_2 would have added the ID to its priority queue because its priority queue did not contain an entry with ID 5. Remember that R_2 consumed the entry with ID 5 from its priority queue when it first tried to send the message with that ID. New messages with ID 5 that arrived at R_1 in the time between the send attempt of the message with ID 3 and the output agreement overwrite the old message with ID 5 in the priority queue. When R_1 now deletes the message with ID 5 from its queue, it actually does not remove the message R_2 sent, but it removes a newer one. In a fault free setup this would not lead to message loss because R_2's CIS contains the new message in its queue and R_2 will eventually send it. But in case of a single fault like a cable break between R_2 and the destination node, R_2 is not able to send this new message and R_1 does not contain it. We solve this issue as follows: Whenever a destination CIS receives a new message from the TTNoC, it checks if the ID is already in the priority queue. If not, it inserts it and sets a boolean flag `updated` to `false`. If then new messages arrive and the ID is already in the queue, the content of the message gets overwritten, and the `updated` flag gets set to `true`. When a CIS gets the message from the RedMU that it has to delete an ID from the priority queue, it checks if the `updated` flag is set. If it is not set, therefore no new messages with this ID arrived, the CIS simply deletes the entry from its priority queue because the opposite router successfully sent it. If the `updated` flag is set, the message is not removed from the priority queue because it is in fact a new message. Again, assume the left most case in Table 1 and that new messages with the ID 5 arrived. For R_2 this means that the first arrival generated a new entry in the priority queue. In case of R_1 the old and unsent message with the ID 5 got overwritten and the `updated` flag was set. In that case R_1 does not remove the message from its priority queue. After the agreement is executed (and because of the `updated` flag), now both destination CISes contain an entry with the ID 5 in their priority queues.

Finally, we discuss the code starting at Line 8. This part of the algorithm is used to handle situations in which the local destination CIS did not try to send a message and therefore did not contribute to the agreement except by sending a `nil` message. Basically, if it did not contribute it can simply continue with sending messages (or being silent).

4.6 RedMU Failure Detection

Detecting a failure of a RedMU is straight forward. Even if there are no new CAN messages to agree upon, the input agreement is executed in every activity cycle and the CISes send `nil` messages that are distinguishable from silence on the router interlink. These messages therefore act as an alive-signal and given our fault hypothesis that the routers' failure mode is fail-silent, we can detect a failed router. The operation that is then performed is application dependent. It might be desirable to reach a safe system state or to continue with one router in a never-give-up scenario.

5. EVALUATION

For evaluation we modeled the CAN router with the help of TrueTime (see Henriksson et al. (2002)). TrueTime is a Matlab/Simulink-based simulator for real-time control systems. We started by modeling a single CAN router based setup (i.e., CISes, the TTNoC, CAN buses, CAN nodes) and reevaluated previous results (see Kammerer et al. (2012)) that were executed on the Field Programmable Gate Array (FPGA) based prototype. After evaluating that the model matches the FPGA based prototype in the time and value domains, we extended the model to a redundant, mixed-critical setup. The setup consists of the same components that are shown in Figure 2.

5.1 Test Cases and Results

During the development of the agreement algorithms we defined a set of test cases that was executed on every iteration of the protocols and compared to the expected outputs. Our set of test cases allows us to do statement and branch coverage, but due to the limited paper length we have to focus on a test case that covers the proposed techniques in a mixed-critical setup [1]. Figure 4 shows the temporal sequence of the test case as well as the results. Bold arrows drawn between the RedMUs symbolize the periodic output agreement. The dashed arrow between the RedMUs in activity cycle 4 symbolizes input agreement. Please note that the input agreement is executed in every activity cycle, but for the sake of clarity we showed it only in cycle 4, which is the only cycle where input agreement has an influence in this test case.

[1] Feel free to send an email to the first author to get access to the TrueTime model and the source code used for branch coverage analysis.

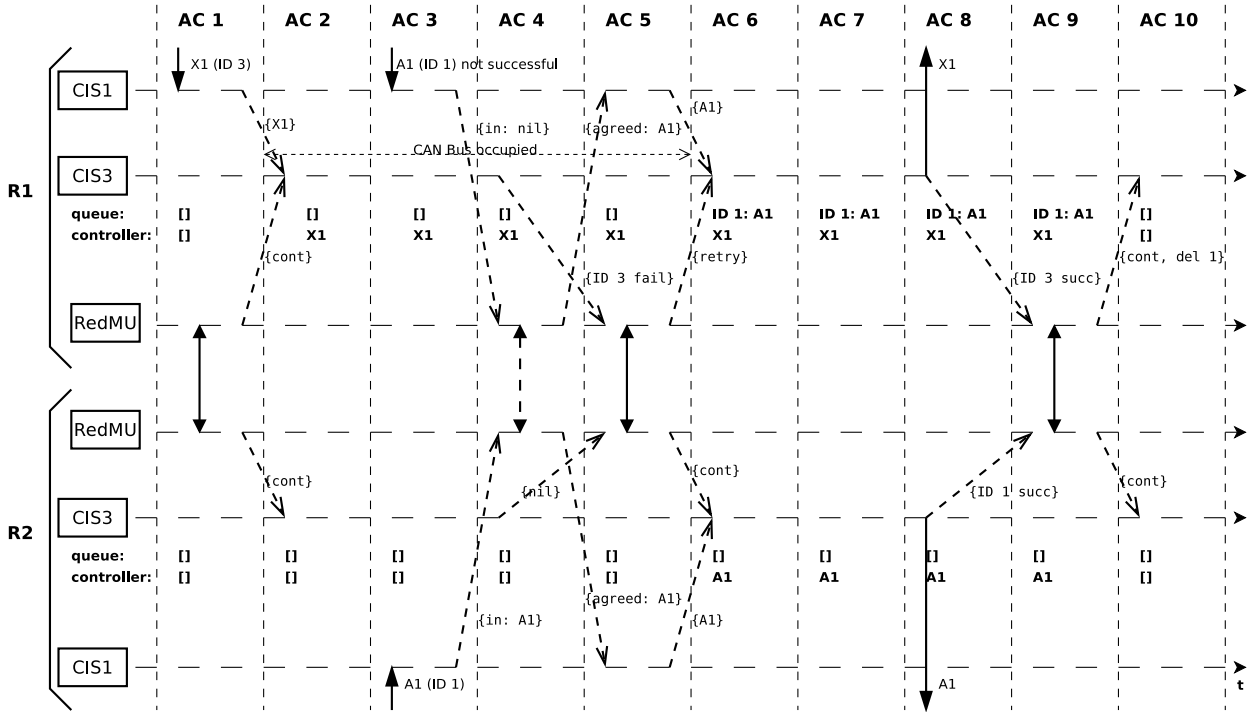

Fig. 4. Result showing input and output agreement in a mixed-criticality setup

In the selected case the LCN node X sends its message to router R_1. The message gets processed at CIS 1 of router R_1 and gets delivered to CIS 3. Starting from activity cycle 2, we assume that the CAN bus connected to CIS 3 is occupied. It could be occupied because other higher priority messages are sent by node Y. For CIS 3 at R_1 this leads to a situation that it is not able to send the CAN message to its destination node. In cycle 2, CIS 3 tries to send the message $X1$, but fails. In activity cycle 3 we assume that the HCN A wants to send a message with a higher criticality than the first message that was sent by node X. At the CAN bus connected to CIS 1 of router R_1 this attempt fails because the bus is faulty. At R_2 the message is successfully received. Due to the input agreement this high-critical message gets replicated to R_1 in activity cycle 4 and is available at the source CIS in cycle 5. The next output agreement is executed in cycle 5. The first attempt to send $X1$ was in cycle 2, it takes one cycle until the CIS knows if the transmission was successful, therefore the information that the message with ID 3 failed is available and sent in activity cycle 4. The outcome of the agreement leads for CIS 3 at R_1 to a retry request that is received in cycle 6. CIS 3 of R_2 did not send a message because it had no message to agree on and is allowed to continue. Both try to send their according messages in cycle 6. Note hat R_1 (re)tries to send the low-critical message because it was already in the queue, while R_2 tries to send the high-critical message. Both CISes have to wait one cycle (AC 7) before they send the status for the next output agreement in cycle 8. The agreement algorithm decides for the CIS at R_2 that it is simply allowed to continue. CIS 3 at R_1 has to delete the high-critical message from its queue and is otherwise allowed to continue with the next message.

6. DISCUSSION

From Figure 4 we can conclude that high-critical messages are not delayed by low-critical messages under our fault assumptions and our suggested system design (i.e., all non-critical nodes connected to the same router). As long as a high-critical message is successfully sent to one router, the message will be replicated to the redundant router (Figure 4, activity cycles 3 to 5). In the test case shown, the bus connecting A and R_1 was faulty, but R_2 successfully received the message, which was then replicated to R_1. If the system only consists of HCNs, a single fault between the fault containment regions does not have an influence on the message. If we consider a mixed-criticality setup, the situation changes. As long as the subsystem that connects only HCNs is fault free, the subsystem that also features LCN does not influence the high-critical system at all. In a mixed-critical system it might be the case that the CAN bus from router R_2 to node B is faulty, and the LCN Y blocks the bus between R_1 and node B. The influence can be damped by assigning high-priority CAN-IDs to high-critical messages.

A second interesting result is shown in the behavior of R_1. Although the second message has a higher-criticality, the message is blocked by the message $X1$ that was received by CIS 3 in a previous activity cycle but not delivered because the CAN bus to the destination node was occupied. This phenomenon is called head of line blocking (see Davis et al. (2011)). The interesting fact is that in a redundant router setup blocking of high-critical messages by low-critical messages does not have an influence if the CAN bus to the destination nodes that receive high-critical messages is operational and all LCNs are connected to a single router. In that case the redundant router that only processes high-critical messages will deliver the high-critical messages without any interference from the redundant router that might suffer from head of line blocking.

Starting from activity cycle 8 we see how the output agreement handles mixed-critical systems. It is obvious that CIS 3 at router R_2 is allowed to continue. In that case R_2 sent a high-critical message which is also available at the redundant CIS due to input agreement of high-critical messages at the source CIS.

Therefore, router R_1 has to delete the corresponding message from the queue before it is allowed to continue. By deleting this message we ensure that the message is only delivered once to the destination CAN node. In our scenario there was no new occurrence of a high-critical message with the same CAN-ID which allows us to simply delete the message from the queue of CIS 3. In case there would have been new occurrences, they would have been replicated due to the input agreement and the message would not have been deleted because of the `updated` flag discussed in Section 4.5.

7. CONCLUSION

In this paper we introduced a fault-tolerant CAN-based communication infrastructure for mixed-criticality systems. We provided a system model for two redundant CAN routers and motivated the necessity to agree on a consistent system state and how to achieve it. In the following we elaborated on the software layers used at high-critical CAN nodes and the algorithms for input and output agreement with a focus on mixed-criticality. For the evaluation we selected a conclusive scenario in which the proposed agreements are necessary to guarantee a consistent system state. We showed and discussed that realizing mixed-criticality systems based on the CAN router is feasible and how to design such systems to achieve fault detection an isolation in the temporal and value domain (e.g., connecting low-critical nodes to a single router).

REFERENCES

Barranco, M., Almeida, L., and Proenza, J. (2005). ReCANcentrate: A replicated star topology for CAN networks. In Proceedings of the 10th IEEE Conference on Emerging Technologies and Factory Automation (ETFA 2005), volume 2, 8 pp. –476. doi:10.1109/ETFA.2005.1612714.

Bauer, G. (2001). Transparent Fault Tolerance in a Time-Triggered Architecture. Ph.D. thesis, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria.

Davis, R.I., Kollmann, S., Pollex, V., and Slomka, F. (2011). Controller Area Network (CAN) Schedulability Analysis with FIFO Queues. In Real-Time Systems (ECRTS), 2011 23rd Euromicro Conference on, 45–56. IEEE.

Henriksson, D., Cervin, A., and Årzén, K.E. (2002). Truetime: Simulation of control loops under shared computer resources. In Proceedings of the 15th IFAC World Congress on Automatic Control. Barcelona, Spain.

ISO-11898 (1993). Road vehicles – Interchange of Digital Information – Controller Area Network (CAN) for High-Speed Communication. Int. Standardization Organisation, ISO 11898.

Kaiser, J. and Livani, M. (1999). Achieving fault-tolerant ordered broadcasts in CAN. In Proc. of European Dependable Computing Conference, 351–363. URL `citeseer.nj.nec.com/235561.html`.

Kammerer, R., Obermaisser, R., and Froemel, B. (2012). A Router for the Containment of Timing and Value Failures in CAN. EURASIP Journal on Embedded Systems.

Kammerer, R., Froemel, B., Obermaisser, R., and Milbredt, P. (2013). Composability and compositionality in can-based automotive systems based on bus and star topologies. INDIN 2013.

Kopetz, H. (1998). Real-Time Systems: Design Principles for Distributed Embedded Applications. Kluwer Academic Publishers.

Livani, M. (1999). SHARE: A transparent approach to fault-tolerant broadcast in CAN. In Proc. of 6th Int. CAN Conference (ICC6). Torino, Italy.

Obermaisser, R. and Kammerer, R. (2010). A router for improved fault isolation, scalability and diagnosis in can. INDIN 2010.

Paukovits, C. (2008). The Time-Triggered System-on-Chip Architecture. Ph.D. thesis, TU Vienna.

Proenza, J., Barranco, M., Rodriguez-Navas, G., Gessner, D., Guardiola, F., and Almeida, L. (2012). The design of the CANbids architecture. In Proceedings of the 17th IEEE Conference on Emerging Technologies and Factory Automation (ETFA 2012).

Rufino, J. (1997). Dual-media redundancy mechanisms for CAN. Technical Report CSTC RT-97-01, Centro de Sistemas Telemáticos e Computacionais do Instituto Superior Técnico, Lisboa, Portugal.

Rufino, J., Veríssimo, P., Arroz, G., Almeida, C., and Rodrigues, L. (1998). Fault-tolerant broadcasts in CAN. In Proceedings of the 28th International Symposium on Fault-Tolerant Computing Systems, 150–159. Munich, Germany.

Veríssimo, P., Rufino, J., and Ming, L. (1997). How hard is hard real-time communication on field-buses? In Proceedings of Symposium on Fault-Tolerant Computing, 112–121.