SHARE: A Transparent Mechanism for Reliable Broadcast Delivery in CAN

Mohammad Ali Livani
mohammad@informatik.uni-ulm.de

University of Ulm
Department of Computer Structures
Technical Report No. 98-14

Abstract. Dependability is one of the most important design dimensions for a wide area of real-time cotrol systems. When moving from the centralized designs to cooperative distributed systems, reliable coordination of distributed tasks becomes a critical issue. Atomic broadcast is a mature technology to support distributed coordinated actions.

Field busses exhibit a highly reliable and predictable behavior, as it is required by embedded real-time control applications. One of the most important field busses is the Controller Area Network (CAN), which is widely used in automotive and industrial applications. Although CAN realizes a consistent view of every bit among all peer CAN controllers in fault-free situations, the bit-consistency may be lost due to a variety of faults, leading to the inconsistent delivery of broadcast messages. The paper discusses the possible causes of this inconsistency in CAN and some approaches to achieve consistent delivery of CAN messages in presence of faults.

Keywords. Real-time communication, fault-tolerance, reliable broadcast, CAN.

1 Introduction

Real-time control systems must provide timely and reliable computing service to the real-world environment. Distributed systems which inherently provide immunity against single failures, are an adequate architecture to achieve high availability of the computing system. Moreover, the availability of inexpensive, powerful micro controllers promotes distributed solutions. By replicating the processes on different sites the system is able to provide the service despite the failure of some processes or sites. The need for consistent replication of the process state information results in the demand for a communication service which provides reliable and consistent delivery of information to groups of processes. Such a communication service is termed atomic broadcast [5]. A real-time atomic broadcast protocol must terminate within a well-known worst-case message delivery delay. Some examples of real-time broadcast protocols are the Δ-protocol [5],[6] and the time-triggered protocol [11].

In the area of industrial automation and automotive applications, field busses are used to disseminate time critical messages. Field busses exhibit bounded message length, reliability of transfer, and efficiency of the protocol. Among field busses, the CAN bus [17] provides advanced features, which make it suitable for a wide range of real-time applications. Some of these features are high robustness against electromagnetical interference, priority-based multiparty bus access control, variable but bounded message data ($\leq$ 8 bytes), efficient implementation of acknowledgement and error indication, and automatic fail-silence enforcement.

There are application level protocols available for CAN which raise the network interface and shield the programmer from low level details. The most popular ones are CAL [4], CAN Kingdom [9], and Device Net [7]. However, these protocols do not provide one or more of the following features, which are required by safety-critical applications:

- consistent delivery of broadcast messages to all operational receivers,
- consistent ordering of broadcast messages at all operational receiver sites,
- guaranteed timely delivery of periodic, hard real-time messages in presence of faults,

This paper will first introduce the CAN protocol in section 2. It is shown that a consistent delivery of broadcast messages in presence of faults is not provided by the CAN protocol itself. Section 3 reviews some approaches to achieve fault-tolerant group communication in broadcast networks. Section 4 introduces an approach to provide consistent delivery of broadcasts in CAN transparently. A summary concludes the paper.

2 The CAN Protocol

2.1 Physical Layer

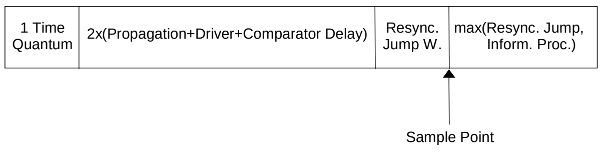
1 Time Quantum
2x(Propagation+Driver+Comparator Delay)
Resync. Jump W.
max(Resync. Jump, Inform. Proc.)
Sample Point

Fig. 1: Partitions of a bit-time in CAN

The key to the understanding of the CAN-Bus is the fact that all nodes (including the sender) scan the value of every bit while it is transmitted. Hence, all correct nodes have a consistent view of every bit. In order to achieve this consistency, the transmission time of a single bit (called bit-time) is guaranteed to contain twice the end-to-end signal propagation time on the bus line, twice the input comparator and the output driver delay, the maximum of an *Information Processing Time* (less or equal to 2 time quanta) and the *Resynchronization Jump Width* (between 1 and 4 time quanta) plus an additional *Resynchronization Jump Width* and an additional time quantum. A time quantum is typically about one tenth of a bit-time. The bus signal is sampled at a 'safe' time, as illustrated in Fig. 1. For instance, at a nominal bus speed of 1Mbit/s, the nominal bit-time is 1µs. For the required processing, propagation, and resynchronization times to be contained in one bit-time, the maximum bus length must be as short as 40 meters. The sample point is recommended to be about 750 to 800ns after the beginning of each bit-time in this case.

Since the physical connection of the CAN bus – based on a twisted pair of wires and voltage difference (Fig. 2-A) – is extremely robust against electro-magnetical noise, the signal level sensed at the sample point, reflects reliably the currently valid bit-value to be observed by all nodes.

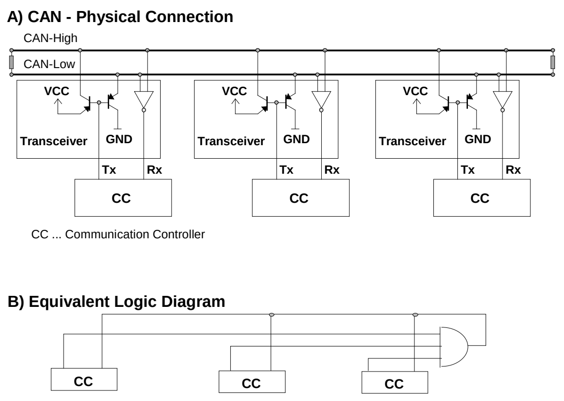
A) CAN - Physical Connection
CAN-High
CAN-Low
VCC
Transceiver
GND
Tx
Rx
CC
CC ... Communication Controller
B) Equivalent Logic Diagram
CC

Fig. 2: CAN bus connection

The nodes of the CAN-Bus are connected logically via a wired-AND or, alternatively, a wired-OR function. In a wired-AND connection – as implemented in commercial CAN controllers – a ‘1’ is a recessive bit and a ’0’ is a dominant bit (Fig. 2-B). Whenever different bit-values are put on the bus by different nodes simultaneously, the logical AND function of them is observed by all nodes. In the rest of the paper, the wired-AND connection is assumed.

In the CAN bus, the non-return to zero (NRZ) coding scheme is used to transmit a bit-stream. Since there is no falling or rising edge between consecutive equal bits in the NRZ scheme, resynchronization of the CAN nodes is ensured in the physical layer of CAN by stuffing a complimentary bit into the bit-stream after 5 consecutive equal bits by the sender during the transmission of the arbitration, data, and CRC field. At the receiver sites, the stuffed bits are removed from the bit-stream according to the same scheme.

2.2 Data Link Layer

Arbitration Mechanism.

The consistent view of every bit among all CAN controllers is exploited for a priority-based arbitration mechanism (called carrier sense multiple access with collision avoidance – CSMA/CA). According to this mechanism, as soon as the bus is idle, each node competing for the bus begins to send the arbitration field of its message, which mainly consists of an 11 or 29 bits identifier (depending on the „standard“ or „extended“ message format – Fig. 3).

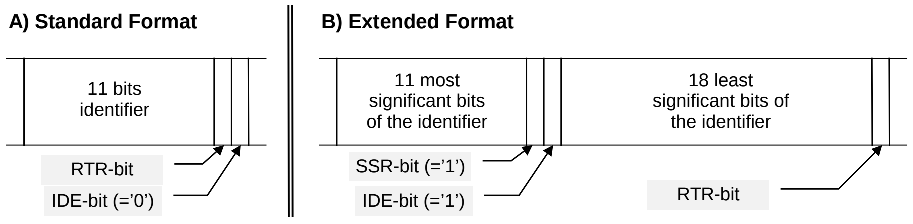

Fig. 3: The CAN arbitration field

If at this time a node sends a ‘1’ and senses a ‘0’, it becomes aware of a collision, stops transmitting, and switches to the receiver mode. At the end of the arbitration field, only the node which is sending the message with the lowest identifier value, will be transmitting. This priority-based arbitration mechanism requires that different CAN nodes never start simultaneously sending messages with equal identifiers. This requirement must be satisfied by higher-level protocols.

Due to the CSMA/CA arbitration technique, the identifier of a CAN-message is interpreted as its priority, with the lowest identifier value being the highest priority. The bus itself serves as a priority-based dispatcher. Since popular implementations of application layer protocols in CAN bus use a fixed identifier for each message, they implicitly implement a fixed priority scheduling of the bus resource. A detailed analysis of the schedulability and the worst-case transmission delay of real-time messages using the fixed priority scheduling approach, is given in [24]. In [26] a combination of fixed and dynamic priority scheduling is approached. But this approach fails to schedule messages in a bus with 3 or more sender nodes due to a too short *time horizon* [12]. For scheduling hard and soft real-time messages in a dynamic application system, a combination of TDMA, deadline scheduling, and fixed priorities is presented in [14].

CAN frames.

In the CAN bus, there are four different types of frames: the *data frame*, the *remote transmission request frame (RTR-frame)*, the *error frame*, and the *overload frame*.

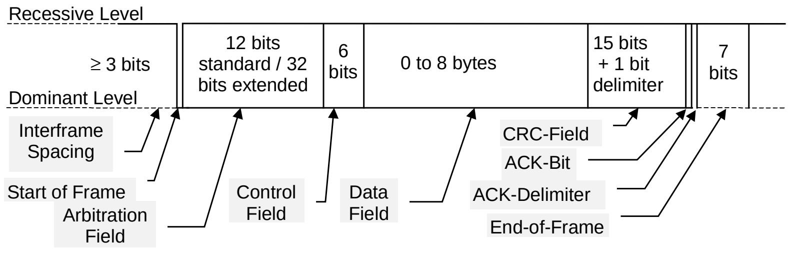

Fig. 4: The CAN Data Frame in Standard/Extended Format

The CAN Data Frame (Fig. 4) has its RTR-bit set to ‘0’ and contains up to 8 bytes of data. The RTR-Frame has its RTR-bit set to ‘1’, and contains no data. If a CAN controller receives an RTR frame, and it has a send buffer with the same identifier as the RTR-frame, the send buffer is activated, and the data is sent out immediately.

The Error Frame and the Overload Frame have identical structures. They both consist of six dominant bits. If a node detects an error frame (or overload frame), it transmits also an error frame (or overload frame), and then it waits until it detects a recessive bit (‘1’). When after an error frame (or overload frame) a sequence of 11 recessive bits has been observed, the CAN nodes may start again to transmit data or RTR frames.

Since the error frame and the overload frame have identical structures, they can be only distinguished by the time when they are transmitted. An error frame may be started only within a data or RTR frame, in order to interrupt the current transmission and signalize an error. An overload frame, however, may be started only immediately after the successful transmission of a data or RTR frame in order to indicate that a node is temporarily not able to accept further data frames due to an internal overload condition. While an error frame results in a faulty data frame to be discarded and retransmitted, an overload frame simply consumes the bus resource for 19 bit-times, just to allow the overloaded node to master the internal overload situation. It should be noted that modern CAN controllers (like Intel 82527, Motorola msCAN, Motorola TouCAN, Siemens SAE8192) do not initiate an overload frame, but they are able to correctly react on it.

Since an error frame may not be initiated after the end of a data frame (it would be then an overload frame), an error detected at the last bit of a message must be ignored by every receiver, according to the CAN specification [17].

Error detection mechanisms.

CAN applies several error detection techniques to achieve highly reliable data transfer. Table 1 gives an overview of the error detection mechanisms in CAN. Whenever a CAN node detects an error situation, it immediately switches to the error signaling mode, and discards the current frame.

Bit-monitoring means that a sender samples the bus while it transmits each bit, and compares the observed value with the currently transmitted value. If it detects a difference, it assumes an error situation. Note that sending a ‘1’ and observing a ‘0’ is not interpreted as an error during the arbitration phase.

Error detection by *CRC* is performed at the receiver sites. At the end of each frame, a Cyclic Redundancy Check field – containing 15 bits – is appended by the sender. All receivers compute the CRC while receiving the data, and compare it with the received CRC on the fly. If a receiver detects a difference between the computed and the received CRC, it assumes an error situation. The CRC in CAN is computed by an algorithm which is optimized for checking strings of up to 127 bits.

Error detection mechanism	Detection site
Bit-Monitoring	sender
Cyclic Redundancy Check	receiver
Bit Stuffing	receiver
Message Frame Check	receiver

Table 1: Error Detection Mechanisms of the CAN-Bus

Bit Stuffing errors are detected by a receiver whenever it observes more than 5 consecutive equal bit-values on the bus within the arbitration field, data field, or the CRC field. This violates the bit stuffing rule of the physical layer. In this case, the receiver assumes an error situation.

Message frame check means that a receiver verifies the value of any bit which must have a pre-defined, fixed value according to the CAN specification. If a receiver observes a violation of the frame format rules, it assumes an error situation.

There is an *acknowledgment* bit after the CRC field of each CAN frame, where every receiver transmits a ‘0’ to indicate the error-free reception of the frame. A sender transmits a ‘1’ and expect to observe a ‘0’ at this bit. If a sender does not observe a ‘0’ at the acknowledgment bit, it assumes that its message could not reach any receiver, and tries to retransmit the message.

The error detection mechanisms ensure the detection of the following errors [17]:

- all global errors
- all local errors at transmitters
- up to 5 randomly distributed errors in a frame
- burst errors of up to 15 bits in a frame
- errors of any odd number in a frame

The probability of a faulty frame to remain undetected, is less than $4.7 * 10^{-11}$.

Error signaling.

Whenever a CAN node detects an error, it switches to the error signaling mode, and starts to transmit an error frame, which consists of six dominant bits (‘0’) without bit stuffing. This violates the bit stuffing rule, and ensures that all other CAN nodes detect an error, too. After sending an error frame, a CAN node waits for all other nodes to finish their error frames (i.e. until it detects a recessive bit). A new transmission may start 11 bit-times after the bus value becomes recessive.

The combination of the error detection and error signaling mechanisms in CAN provides a highly reliable data transfer at the Data Link Layer.

2.3 Inconsistency in CAN

Although CAN applies several techniques at the physical and data link layer to ensure reliable and consistent data transfer, inconsistencies among different CAN nodes are still possible.

The lowest level of inconsistency that is observable in a CAN bus, is the inconsistent view of a bit. Depending on when and why an inconsistency at the bit level happens, it is either repaired by error detection and error signaling mechanisms, or it grows to an inconsistency at the frame level, i.e. an inconsistent view of the frame status among different CAN nodes.

In this section the possible causes and effects of these inconsistencies are analyzed.

Receiver Error

A bit-level inconsistency is caused whenever a receiver receives an erroneous bit-value because of its internal faults. Thus, the faulty receiver has a different view of the bit-value than all other (non-faulty) nodes. Possible causes of this error are e.g. unreliable (loose) plug connection to the bus, faulty comparator in the transceiver chip, or faulty CAN controller. If the faulty receiver does not crash due to its internal faults, it will detect with a high probability the error by means of bit stuffing/CRC/frame checking. Following cases can be distinguished depending on the time of the error detection.

1. If the error is detected at least two bits before the end of the frame, the receiver discards the frame, and initiates an error frame at the next bit-time after the error detection. As a result, all other (non-faulty) nodes of the bus detect the error, and discard the frame, too. The sender then attempts to retransmit the frame. In this case, the bit-level inconsistency is recovered by the error handling mechanisms and has no effect on higher levels.
2. If the error is detected in the last but one bit of the frame, the receiver discards the frame, and initiates an error frame at the next bit-time after the error detection. The sender discards the frame and attempts to retransmit. However, all non-faulty receivers detect the error at the last bit of the frame, ignore the error, and accept the frame. In this case, the bit-level inconsistency grows to a frame-level inconsistency, where the faulty receiver and the sender have another view of the frame status than all non-faulty receivers. This inconsistency may be easy or hard to recover depending on when the frame is retransmitted:

   2-a) If the frame is retransmitted immediately, the receivers can recover from the frame-level inconsistency by just detecting and discarding duplicate frames. This is a well-known technique applied in common high-level CAN protocols.

   2-b) if a higher-priority frame is transmitted between the first and the second transmission of the faulty frame, the simple detection and discarding of duplicates results in an inconsistent ordering of the frames. This means that the sender and the faulty receiver deliver the higher-priority frame to the application before the lower-priority frame, and all non-faulty receivers establish the opposite order. However, consistent delivery is ensured by the retransmission mechanism of the CAN protocol. Consistent message ordering among all nodes can be established by additional efforts, e.g. as described in [20].

   2-c) If the sender crashes before a successful retransmission, the non-faulty receivers will deliver the frame to the application, where the faulty receiver cannot deliver it. The communication system needs a mechanism to ensure the retransmission of such frames even when the original sender has crashed. A detailed description of this case can be found in [20].
3. If the error is detected in the last bit of the frame or not at all, the receiver ignores the error, and accepts the frame. Thus, the bit-level inconsistency has no effect on higher levels.

Sender Error

Bit-level inconsistency is caused by a sender error in following cases.

1. A sender transmits a bit-value correctly, but monitors a wrong bit-value. In this case, the sender has a different view of the bit-value than all (non-faulty) receivers. Possible causes of this error are e.g. unreliable (loose) plug connection to the bus, faulty comparator in the transceiver chip, or faulty CAN controller. The sender will detect the error by the monitoring mechanism. It will then discard the frame, initiate an error frame at the next bit-time after the error detection, and attempt to retransmit it as soon as the bus is idle.

   1-a) If the error is detected at least two bits before the end of the frame, the receivers discard the frame too, and no inconsistency at the frame-level occurs.

1-b) If the error is detected by sender at the one but the last bit of the frame, the receivers detect the error at the last bit of the frame, ignore the error, and accept the frame. Also, if the error is detected by sender at the last bit of the frame, the receivers do not detect the error, and accept the frame.
When the receivers accept the frame, and the sender discards and retransmits it, a frame-level inconsistency between the sender and the receivers is caused. In case of an immediate retransmission, the inconsistency is recovered by simply discarding duplicate frames. If the sender crashes before the successful retransmission, the inconsistency disappears, and all remaining nodes of the system will have a consistent view of the frame. However, if a higher-priority frame is transmitted before the retransmission of the inconsistent frame, an inconsistent ordering is caused between the sender and the receivers, which is similar to the case 2-b of receiver error. The communication system needs an additional mechanism to establish a consistent ordering at both the sender and the receivers, while consistent delivery is ensured by CAN.

2. A sender transmits a faulty bit such that the receivers observe a wrong bit-value. In this case, the sender as well as the receivers detect the error with a high probability by means of bit-monitoring and bit stuffing/CRC/frame checking.
   2-a) If the error is detected by receivers before the last bit of the frame, they all discard the frame, initiate an error frame, and the sender retransmits the frame as soon as the bus is idle. Thus, no frame-level inconsistency will occur.
   2-b) If the error is detected by the sender and receivers at the last bit of the frame, it will be accepted by the receivers, and also retransmitted by the sender. Again, an inconsistent ordering is caused, which is identical to the case 1-b of sender error.

Channel Error

Although the physical layer of CAN has been designed to be extremely robust to electro-magnetical interference, transient noise on the bus line must be considered in the reliability analysis. Also transient short circuits between the CAN-high and CAN-low or between a bus line and the ground (car body) due to a damaged wire coat and vibrations are possible.

A channel error is either detected by all CAN nodes at the same bit, or some nodes detect it at a bit-time, while other nodes detect it one bit later. In both cases, inconsistencies in the frame-level may be caused, as described below.

1. When all nodes detect a channel error at the last bit of the frame, an inconsistent view between the sender and the receivers is caused, which is identical to the case 1-b of sender error.
2. If some of the receivers detect the channel error at the last bit of the frame, and other receivers detect it one bit earlier, the first group will accept the frame, and the second group will discard it. Regardless of when the sender detects the error, it will discard the frame, and attempt to retransmit it. The frame-level inconsistency caused here, is equivalent to the case 2 of receiver error, when the faulty receiver is substituted by the receivers which detect the channel error earlier.

2.4 Anonymous Error Signaling Constituting a Weak-Point

The analysis in the previous section shows clearly, that a frame-level inconsistency is caused whenever an error is detected at the end of a frame, and the CAN nodes are not able to achieve consensus on whether to discard the frame, or to accept it.

The reason of this lack of consensus is that it is not possible to indicate an error of the last bit of a message – the error indication would seem to be an overload frame. Hence, receivers are forced to ignore an error of the last bit of a frame.

However, if some but not all of the receivers detect an error in the last but one bit of a frame, they will discard the frame, and start an error signal immediately after the error detection, i.e. at the last bit of the frame. Every other receiver will then detect an error in the last bit. Hence, those receivers will accept the frame, and inconsistency at the frame level occurs. In order to recover from this inconsistency, the CAN protocol specifies that a sender retransmits the frame upon the detection of an error in the last bit of a frame.

If the immediate retransmission of a faulty frame could be guaranteed, the frame-level inconsistency would be repaired by detecting and discarding duplicate frames. But an intermission of a higher-priority frame can cause an inconsistent ordering of the frames at different sites, as explained in the previous section. Also, a crash failure of the sender before the successful retransmission is possible, so the frame is not retransmitted at all [20].

An interesting question is: Is the identical format of error frames and overload frames the reason for the lack of consensus on accepting or discarding a frame?

The answer is “No”. Even if an overload frame could be distinguished from an error frame starting after the end of a frame, the CAN nodes would not be able to *always* achieve consensus on the message status. This handicap is because of the following facts:

1. For each frame, there must be a point of time where the frame must be accepted if it is not corrupted until then, and the frame must be discarded, if an error was detected earlier. Let’s call it the validation point. In CAN, the validation point is after the last but one bit of the frame.
2. When a node initiates an error frame, other nodes observe the error frame not before the following bit-time.

Due to the above facts, if a subset of nodes detect an error at the last bit before the validation point, they must discard the frame. Although they immediately initiate an error frame, this error frame will be observed by other nodes after the validation point, thus they will accept the frame. Of course, those nodes will observe an *error frame* immediately after the validation point, but they may not discard the frame in this case. This is because those nodes cannot be sure that they are observing an *error frame* and not, for instance, a burst of dominant bits caused by physical faults.

If a node discards a frame due to the observation of an ‘error frame’ immediately after the validation point, and this ‘error frame’ is actually a burst error, it is possible that some other nodes in the system will observe this burst one bit later. Hence, those nodes would accept the frame and the result is again a frame-level inconsistency.

The inconsistency noticed above, is a fundamental problem whenever diffusing simple signals (e.g. 6 dominant bits) without authentication to broadcast information (e.g. detection of an error). Since such signals are not authentic and can be produced by a faulty component (or a noisy channel), they cannot reliably lead to a consensus among all observers. In the following section, some approaches are introduced, which help deal with the frame-level inconsistency in the CAN bus.

3 Consistent Broadcast Delivery by High Level Protocols

Consistent delivery of broadcast messages means that every message is either delivered correctly to all operational receivers at the protocol termination time, or to none of the operational receivers at all. In order to decide on the delivery of a message at the protocol termination time, either all operational receivers must have received the message, or global agreement must be achieved, that at least one receiver has not received it. Reliable message transmission to all operational receivers can be ensured under anticipated fault conditions by appropriate retransmission mechanisms and resource adequacy [5] [6] [11] [16] [20]. A global agreement on the fact that at least one receiver failed to receive a message, however, needs at least a phase of individual acknowledgments and a phase of commitment, resulting in a two-phased commit protocol [1] [2] [3] [18] [25].

In broadcast networks, also the negative acknowledgement (NAK) approach can be applied in order to avoid the high protocol overhead due to multiple acknowledgements for each message. Erramili and Singh [8] have presented a reliable and efficient broadcast protocol for broadcast networks, using NAK and gap detection by sequential packet numbering. This method is especially suitable for applications with long messages, which consist of streams of packets. Typical cases for the NAK approach are distributed database applications and file transmission. In real-time control applications, however, the transmission and delivery deadline of a packet usually expires before the transmission of its successor. In this case, packet numbering and gap detection is an inappropriate mechanism for error detection.

The Data Link Layer of the CAN protocol ensures that a message is either accepted by the receiving CAN controller, or an error flag is sent by the receiver until the end of the message transmission. The error flag can be seen as an 'immediate NAK'. Upon detecting a NAK, a sending CAN controller automatically attempts to retransmit the message [17].

The reliable message transmission using the NAK/retransmission approach in CAN requires the provision of time redundancy which only depends on the anticipated number of subsequent transmission errors on the network. In contrast, the two-phased commit protocol – based on positive acknowledgements (ACK) and coordinator commitment – requires time redundancy that grows with the number of broadcast receivers in addition to the number of consecutive transmission errors. Thus in CAN the NAK approach is more efficient than the ACK approach. Another advantage of the NAK approach is that the sender does not have to know the number of receivers, and anonymous receivers are possible. Receiver anonymity is necessary e.g. in data fields in ADS [15] or in the Publisher/Subscriber model [10] [22].

In the following sections, two approaches are described, which aim at achieving consistent broadcast delivery in the CAN bus.

3.1 The EDCAN and RELCAN protocols

The EDCAN (CAN Eager-Diffusion) protocol [20] is based on a multiple transmission policy similar to [6]. It has been developed to cope with the frame-level inconsistency of CAN and immediate sender crash. According to the EDCAN protocol, every node attached to the CAN bus keeps track of the bus traffic, and attempts to retransmit every frame that it receives. The protocol recognizes duplicates of the frames, and if a node observes a certain number of duplicates of a frame (Rufino et al named this number the *inconsistent omission degree*), it does not retransmit the frame any more.

The EDCAN protocol ensures the consistent delivery at the cost of a considerable bus bandwidth. E.g. in the simplest scenario with an 'inconsistent omission degree' = 1, every receiver of a frame attempts to retransmit the frame until it receives a duplicate of it. Thus, every frame is retransmitted at least by one receiver. However, after receiving the duplicate at a receiver, the protocol takes more than a few µ-seconds to decide on canceling the retransmission in the CAN controller. Since a transmission request can be cancelled only within 3 bit-times from the beginning of the bus-idle period, at high bus rates the frames are often retransmitted twice, resulting in a communication overhead of 200%.

Another problem related to this approach is that the messages are retransmitted by ordinary nodes running application software. Thus, a faulty application software on a node may manipulate the contents of messages before they are retransmitted by that node. This results in the Byzantine agreement problem. Here in order to obtain consistent data at the receiver sites, several retransmissions by different nodes may be necessary, e.g. as described in [1]. Alternatively, the communication system of the nodes can be implemented as an independent hardware/software module with a well-defined data interface to the application node, e.g. a FIFO dual-ported memory in each direction [19]. Such an interface prevents the application node from manipulating the status of the communication subsystem.

Rufino et al. have also proposed a more efficient protocol called RELCAN [20], which uses the EDCAN protocol only when the receivers assume that the original sender has crashed. According to this protocol the sender first transmits the data frame, and immediately thereafter it transmits a short confirm frame without data. If a receiver receives the confirm message within a given time period, it delivers the data to the application. Otherwise, it starts the eager diffusion of the data frame. Due to this "lazy diffusion" policy, the protocol needs less than 100% overhead in fault-free situations under moderate bus loads. At high bus loads or upon the crash failure of the original sender, however, the protocol behaves like EDCAN, because the receivers do not receive the confirm frame until time-out.

3.2 Fault-tolerant Broadcasts Using Shadow Retransmitters

As already discussed in section 2.3, CAN ensures reliable message transmission (sometimes with replicas) unless a sender crashes after a frame was received only by a subset of the receivers, and before the sender retransmits it successfully. In this case the sender cannot retransmit the frame, and a permanent frame-level inconsistency is caused.

The approach presented in this paper, solves the problem of the crashed sender by dedicated nodes, which act as ‘Shadow Retransmitters’ (SHARE’s). These nodes detect the situations where an inconsistent frame transmission is possible, and retransmit the frame *simultaneously* with the original sender. Since the sender and all SHARE’s retransmit *identical* frames simultaneously, these frames are transmitted as a single physical frame. Hence the bus nodes will detect no conflicts, and the redundancy is not visible to the network components. This results in a transparent reliability improvement in the sense that the SHARE mechanism can be applied in an application system to achieve consistent broadcasting, without having to change the existing system components.

The detection of a possible frame-level inconsistency is based on a simple fact: a frame-level inconsistency is possible if at least one non-faulty node observes an error in the last bit of a frame. In this case, the sender is expected to retransmit the frame immediately, thus SHARE’s should also immediately retransmit the frame. In order to reliably detect an error in the last bit, the SHARE’s are designed to detect a ‘0’ at the last bit of a frame during the whole bit-time, and not only at the sampling point. Moreover, several SHARE’s can be added to the bus for high reliability.

Effectiveness of the SHARE Message Retransmission Approach

The effectiveness of the SHARE approach is based on the fact that a CAN message is retransmitted by SHARE's if and only if the retransmission is necessary. Since SHARE's retransmit CAN messages whenever they detect a dominant bit at the interval [*EOF-1∗Δbt* .. *EOF*] (*EOF*: the end of the last bit-time of the End-of-Frame field of a message), and the original sender would also retransmit the message upon this condition, the retransmission is performed only if necessary. It only must be shown that SHARE's perform the retransmission whenever it is necessary.

The retransmission of a CAN message is necessary, whenever at least one node detects an error before the last bit of the message, i.e. before *EOF-1∗Δbt.* Following is a discussion on the possible scenarios of error detection:

1. At least one node detects an error before *EOF-2∗Δbt*: in this case, those nodes will start an error flag at *EOF-2∗Δbt,* which is observed by all non-faulty nodes. Thus, all nodes will discard the message, and the message delivery status is consistent. Note that this consistent status will remain in the system even if the sender crashes immediately.
2. No node detects an error before *EOF-2∗Δbt* but at least one node detects an error in [*EOF-2∗Δbt* .. *EOF-1∗Δbt*]: in this case, those nodes will start an error flag at *EOF-1∗Δbt,* which is observed by all non-faulty nodes including SHARE's. The first group will discard the message, and the second group will accept it. If after this situation the sender of the message crashes before successfully retransmitting it, then the message must be retransmitted by a SHARE. If the SHARE's belong to the group of those nodes, which detect the error in [*EOF-2∗Δbt* .. *EOF-1∗Δbt*], then the retransmission won't be performed by them. For this reason, the GAL component of the SHARE locks the bus input on the recessive level during [*EOF-2∗Δbt* .. *EOF-1∗Δbt*] (see the signals RxD1, locked-RxD, and bus-in in Fig. 5). Thus, if a SHARE does not detect an error before *EOF-2∗Δbt* (see case 1) then it will accept the message and be able to retransmit it upon error detection during [*EOF-1∗Δbt* .. *EOF*]. This means, that in this scenario, all non-faulty SHARE's will retransmit the message.
3. No node detects an error before *EOF-1∗Δbt* but at least one node detects an error in [*EOF-1∗Δbt* .. *EOF*]: in this case, all receivers will accept the message. If any SHARE detects the message in [*EOF-1∗Δbt* .. *EOF*], then it will retransmit it. This retransmission, however, could be performed by the original sender in the same situation. Thus, the retransmission may be considered as necessary.

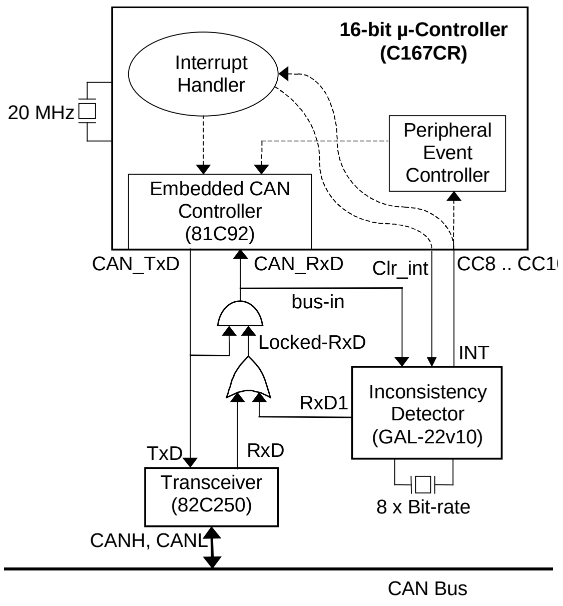

Fig. 5: The structure of a SHARE

The actions of SHARE’s are time critical. In order to start the frame retransmission simultaneously with the sender, a SHARE must be ready to retransmit a frame immediately after receiving it. To be more precise, after detecting an error at the last bit of a frame, a SHARE must start its retransmission within 3 bit-times. At the maximum bit-rate of 1Mbit/s, a SHARE has no more than 3 µ-seconds to start the re-transmission of the last received frame. Currently, SHARE is implemented as a hardware solution (Fig. 5).

The main component of a SHARE is a 16-bit micro-controller, which has an embedded CAN controller and a Peripheral Event Controller (PEC) for fast interrupt handling. PEC is a feature that enables transferring a word or byte from a source address to a destination address upon an interrupt. SHARE uses 3 PEC transfer channels to:

1) convert the CAN buffer of the last received frame to an output buffer,
2) initiate its transmission, and
3) prepare the next free buffer for receiving another frame.

Since a PEC transfer takes 2 processor clock cycles, and the processor interrupt response delay (due to the pipeline) takes no more than 7 processor bus access cycles [21], the whole procedure is completed in less than 2 µs at a processor clock rate of 20MHz, so the SHARE’s are fast enough for operating at the highest bit-rate of 1Mbit/s.

The error detector is realized by a 22v10 Gate-Array Logic (GAL) which detects a sequence of a Zero, 7 One’s, and a Zero. This sequence appears on a CAN bus only when the last bit of a frame is faulty. The error detector raises immediately an interrupt ‘INT’, which activates the PEC transfer channels, and a software interrupt handler for less time-critical actions. A listing of the GAL program can be found in Appendix 1.

The main advantage of dedicated shadow retransmitters is their transparency. Existing CAN solutions may add an appropriate number of SHARE’s to the bus in order to ensure consistent frame delivery even in the case of an inconsistent frame transmission followed by immediate sender crash. This transparency is based on the fact that the retransmissions performed by SHARE’s do not affect the schedulability of the bus resource. A message is retransmitted by a SHARE only in situations where a retransmission by the original sender may be required, too. Thus, adequate bus resources must be provided for such retransmissions even when no SHARE’s are used. Unlike the RELCAN protocol, the SHARE approach causes no additional communication overhead in fault-free situations.

4 Conclusion

The paper discussed the fault scenarios where a frame-level inconsistency can arise in CAN. A transparent solution (the SHARE approach) which enforces the frame-level consistency in presence of faults, has been proposed. In fault-free situations, SHARE's are passive components, and upon detection of a situation where a frame-level inconsistency is possible, SHARE's retransmit the frame simultaneously with the original sender in the same physical frame. Since SHARE components do not transmit any frames that are not expected to be transmitted by the original senders, adding SHARE components to a CAN bus system does not affect the schedulability of the existing system.

The SHARE approach relies on the properties of the CAN protocol and the provision of adequate resources (bandwidth) in case of communication errors. Resource adequacy must be provided by an appropriate bus scheduling mechanism, e.g. as proposed in [14] or [23].

The paper does not discuss the problem of consistent ordering of the messages. An approach to achieve total ordering of CAN messages, which relies on the reliable delivery, is proposed in [13]. Other approaches to achieve consistent ordering of messages in CAN are found in [20] and [27].

References

[1] Ö. Babaoglu and R. Drummond: "Streets of Byzantium: Network Architectures for Fast Reliable Broadcasts", *IEEE Tr. Software Eng. 11(6)546-554.*

[2] K.P. Birman and T.A. Joseph: "Reliable Communication in the Presence of Failures", *ACM Tr. Computer Systems, 5(1):47-76, Feb. 1987.*

[3] J.M. Chang and N.F. Maxemchuk: „Reliable broadcast protocols", *ACM Trans. on Computer Systems, 2(3), Aug. 1984, pp. 251-273.*

[4] CiA Draft Standards 201..207: „CAN Application Layer (CAL) for Industrial Applications", *may 1993.*

[5] F. Cristian et. al.: „Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement", IEEE *15th Int'l Symposium on Fault-Tolerant Computing Systems, Ann Arbor, Michigan, 1985.*

[6] F. Cristian: „Synchronous Atomic Broadcast for Redundant Broadcast Channels", *The Journal of Real-Time Systems, Vol. 2, pp. 195-212, 1990.*

[7] DeviceNet Specification 2.0 Vol. 1, *Published by ODVA, 8222 Wiles Road - Suite 287 - Coral Springs, FL 33067 USA.*

[8] A. Erramilli and R.P. Singh: "A Reliable and Efficient Broadcast Protocol for Broadband Broadcast Networks", *ACM Computer Communication Review 17(5)343-352.*

[9] L.B. Fredriksson: „A CAN Kingdom (Rev. 3.01)", *Published by KVASER AB, Box 4076, S-51104 Kinnahult, Sweden, 1996.*

[10] J. Kaiser, M. Mock : "Implementing the Real-Time Publisher/Subscriber Model on the Controller Area Network (CAN)", *2nd Int'l Symposium on Object-Oriented Distributed Real-Time Computing Systems, San Malo, May 1999.*

[11] H. Kopetz and G. Grünsteidl: „TTP - A Time-Triggered Protocol for Fault-Tolerant Real-Time Systems", *Res. Report 12/92, Inst. f. Techn. Informatik, Tech. Univ. of Vienna, 1992.*

[12] M.A. Livani and J. Kaiser: „Evaluation of a Hybrid Real-time Bus Scheduling Mechanism for CAN", *7th Int'l Workshop on Parallel and Distributed Real-Time Systems (WPDRTS'99), San Juan, Puerto Rico, Apr. 1999.*

[13] M.A. Livani and J. Kaiser: "A Total Ordering Scheme for Real-Time Multicasts in CAN", *Submitted for Publication.*

[14] M.A. Livani, J. Kaiser, W. Jia: „Scheduling Hard and Soft Real-Time Communication in the Controller Area Network (CAN)“, *23rd IFAC/IFIP Workshop on Real Time Programming, Shantou, China, June 1998.*

[15] K. Mori: „Autonomous Decentralized Systems: Concept, Data Field Architecture, and Future Trends“, *Proc. International Symposium on Autonomous Decentralized Systems (ISADS 93), Mar. 1993.*

[16] P. Ramanathan and K.G. Shin: "Delivery of Time-Critical Messages Using a Multiple Copy Approach", *ACM Tr. Computer Systems, 10(2):144-166, May 1992.*

[17] ROBERT BOSCH GmbH: „CAN Specification Version 2.0“, *Sep. 1991.*

[18] L. Rodrigues and P. Veríssimo: „*x*AMP: a Multi-primitive Group Communication Service“, *IEEE Proc. 11th Symposium on Reliable Distributed Systems, Houston TX, Oct. 1992.*

[19] J. Rufino, N. Pedrosa, J. Monteiro, P. Veríssimo, G. Arroz: "Hardware Support for CAN Fault-Tolerant Communication", *Proceedings of the IEEE 5th Int'l Conference on Electronics, Circuits and Systems, Lisbon, Portugal, Sep. 1998.*

[20] J. Rufino, P. Veríssimo, C. Almeida, L. Rodrigues: „Fault-Tolerant Broadcasts in CAN“, *Proc. FTCS-28, Munich, Germany, June 1998.*

[21] Siemens AG: „C167 User's Manual 03.96“, *Published by Siemens AG, Bereich Halbleiter, Marketing-Kommunikation, 1996.*

[22] G. Starovic, V. Cahill, B. Tangney: "An event based object model for distributed programming" *In OOIS (Object-Oriented Information Systems), Springer-Verlag, London, Dec. 1995.*

[23] K. Tindell and A. Burns: „Guaranteed Message Latencies for Distributed Safety-Critical Hard Real-Time Control Networks“, *Report YCS229, Department of Computer Science, University of York, May 1994.*

[24] K. Tindell, A. Burns, A. Wellings: „Calculating Controller Area Network (CAN) Message Response Times“, *Control Engineering Practice 3(8):1163-1169, 1995.*

[25] P.J. Veríssimo, L. Rodrigues, M. Baptista: "AMp: A Highly Parallel Atomic Broadcast Protocol", *SIGCOMM'89 Symposium on Communications Architectures & Protocols, Austin, Texas, Sep. 1989..*

[26] K. M. Zuberi and K. G. Shin, „Non-Preemptive Scheduling of messages on Controller Area Network for Real-Time Control Applications“, *Technical Report, University of Michigan, 1995.*

[27] K. M. Zuberi and K. G. Shin: „A Causal Message Ordering Scheme for Distributed Embedded Real-Time Systems“, *Proc. Symp. on Reliable and Distributed Systems, Oct 1996.*

Appendix 1: GAL Program for Inconsistency Detection

```
|gal22v10  clock: clk,
|            2..3: (bus_in, clr_int),
|          14..23: (RxD1, delay, q0, q1, q2, x3, x2, x1, x0, int)

           DETECTS A FAULT AT THE LAST BIT OF A CAN MESSAGE: this is used as a component
           of the Shadow Retransmitter (SHARE).

|  conditioning: clk // RxD1, delay, q0, q1, q2, x3, x2, x1, x0

           One stage delay of bus-input --- to detect edges
| delay = bus_in

           A three-bit-counter to divide a bit-time into 8 sections, so we can sample at
           sampling point = 6/8th of a bit-time;
           Resynchronization: the counter is reset when a falling edge is detected on
           the bus:
| q0 = (delay & bus_in')' &                        q0'
| q1 = (delay & bus_in')' & ((q1'&   q0) # (q1&    q0'))
| q2 = (delay & bus_in')' & ((q2'&q1&q0) # (q2&(q1&q0)'))

           A state machine to detect a sequence (zero, 7*one, zero) on the bus;
           On detection, an alarm signal (INT) is activated;
           When clr_int is activated from outside, INT is cleared and the detection
           starts over
| x0 = ((q2&q1&q0') & (
|       (x3'&x2'&x1'&x0  & bus_in)
|     # (x3'&x2'&x1 &x0' & bus_in)
|     # (x3'&x2 &x1'&x0' & bus_in)
|     # (x3'&x2 &x1 &x0' & bus_in)
|     # (x3 &x2'&x1'&x0')
|     # (x3 &x2'&x1'&x0  & clr_int')
|     # (x3 &x2'&x1'&x0  & clr_int & bus_in)
|       ))
           If after detecting 7 Ones (S8) a Zero is observed during the next bit-time
           (not only at the sampling point) switch to S9 and raise alarm!
|     # (x3 &x2'&x1'&x0' & bus_in')
|     # ((q2&q1&q0')' & x0)
| x1 = ((q2&q1&q0') & (
|       (x3'&x2'&x1'&x0' & bus_in)
|     # (x3'&x2'&x1 &x0' & bus_in)
|     # (x3'&x2 &x1'&x0  & bus_in)
|     # (x3'&x2 &x1 &x0' & bus_in)
|       ))
|     # ((q2&q1&q0')' & x1)
| x2 = ((q2&q1&q0') & (
|       (x3'&x2 &x1 &x0' & bus_in)
|     # (x3'&x2 &x1'&x0  & bus_in)
|     # (x3'&x2 &x1'&x0' & bus_in)
|     # (x3'&x2'&x1 &x0  & bus_in)
|       ))
|     # ((q2&q1&q0')' & x2)
| x3 = ((q2&q1&q0') & (
|       (x3 &x2'&x1'&x0  & clr_int')
|     # (x3'&x2 &x1 &x0  & bus_in)
|       ))
           If after detecting 7 Ones (S8) a Zero is observed during the next bit-time
           (not only at the sampling point) switch to S9 and raise alarm!
|     # (x3 &x2'&x1'&x0' & bus_in')
|     # ((q2&q1&q0')' & x3)

           Alarm signal: raised in S9 or after observing a Zero in S8
| int = (x3 &x2'&x1'&x0 ) # (x3 &x2'&x1'&x0' & bus_in')

           RxD recessive lock: locks RxD (bus_in and CAN_RxD) on recessive at the last
           but one bit of the message, so the SHARE accepts the message anyway.
| RxD1 = (x3'&x2 &x1 &x0 &q2'&q1'&q0') # (RxD1 & (q2 # q1 # q0))
```