Fault-Tolerant Broadcasts in CAN

José Rufino
ruf@digitais.ist.utl.pt
IST-UTL*

Paulo Veríssimo
pjv@di.fc.ul.pt
FC/UL†

Guilherme Arroz
pcegsa@alfa.ist.utl.pt
IST-UTL

Carlos Almeida
cra@digitais.ist.utl.pt
IST-UTL

Luís Rodrigues
ler@di.fc.ul.pt
FC/UL

Abstract

Fault-tolerant distributed systems based on field-buses may take advantage from reliable and atomic broadcast. There is a current belief that CAN native mechanisms provide atomic broadcast. In this paper, we dismiss this misconception, explaining how network errors may lead to: inconsistent message delivery; generation of message duplicates. These errors may occur when faults hit the last two bits of the end of frame delimiter. Although rare, its influence cannot be ignored, for highly fault-tolerant systems. Finally, we give a protocol suite that handles the problem effectively.

1 Introduction

Fault-tolerant distributed systems are nowadays a mature technology, used in a variety of applications and settings, from information repositories to computer control. The latter field is an extremely challenging one, since it must normally combine distribution and fault-tolerance with real-time, and given

*Instituto Superior Técnico - Universidade Técnica de Lisboa, Avenida Rovisco Pais - 1096 Lisboa Codex - Portugal. Tel: +351-1-8418397 - Fax: +351-1-8417499. NavIST Group CAN WWW Page - http://pandora.ist.utl.pt/CAN.

†Faculdade de Ciências da Universidade de Lisboa, Portugal. Navigators Home Page: http://www.navigators.di.fc.ul.pt.

the decentralized nature of many of its problems, it is a natural application for distributed systems. Furthermore, distributed computer control systems have increasingly been based on field-bus networks. While there is a reasonable body of research on LAN-based distributed fault-tolerant systems, we have not seen a great deal of such systems based on standard field-buses, such as Profibus, FIP or CAN.

One reason may be because the efficient implementation of distributed fault-tolerance techniques relies on well-known paradigms like state machines and replication management protocols, and these are hard to implement in the simple field-bus environment. Given the multi-participant nature of the interactions between replicated entities, the system may benefit to a great extent from the availability of reliable communication services, such as those provided by group communication, membership and failure detection. In fact, these services may be extremely relevant for the design of distributed computer control systems, based on field-buses: not only do they give replicas a uniform treatment, but they easily handle constructs specifically intended for real-world interfacing, such as functional groups of sensors and/or actuators.

However, the migration of fault-tolerant communication systems to the realm of field-buses presents non-negligible problems, that we address in this paper, in the context of CAN, the Controller Area Network. CAN is a multi-master field-bus that has assumed increasing importance and widespread acceptance in control application areas as diverse as shop-floor or automotive.

Perhaps influenced by a certain lack of accuracy in the standard CAN documentation, there have been published works that assume CAN supports a (totally ordered) atomic broadcast service [12, 13]. The coverage of this assumption is only acceptable under modest requirements on system reliability, and would lead to the implementation of fault-tolerant systems that would function incorrectly, with unpredictable consequences for the controlled systems.

In this paper, we start by dismissing that misconception, explaining how network errors may lead to: inconsistent data frame transfers; generation of data frame duplicates. Given their probability of occurrence, that we also estimate, the influence of those errors cannot be ignored, for fault-tolerant systems and applications.

Secondly, since the need remains for fault-tolerant group communication on field-buses, we address the problem in a comprehensive way, reasoning

about the reliability of CAN communications and their weaknesses, integrating CAN own properties into a systemic model and showing how a fault-tolerant broadcast primitive can be efficiently supported by a simple software layer built on top of an exposed CAN controller interface.

The following discussion assumes the reader to be fairly familiar with CAN operation. In any case, we forward the reader to the relevant standard documents [7, 15], for details about the CAN protocol.

2 Controller Area Network

The Controller Area Network (CAN) is a bus with a multi-master architecture [7, 15]. The transmission medium is usually a twisted pair cable and the network maximum length depends on the data rate. Typical values are: 40m @ 1 Mbps; 1000m @ 50 kbps. Bus signaling takes one out of two values: *recessive*, otherwise the state of an idle bus, occurs when all competing nodes send recessive bits; *dominant*, which only needs to be sent by one node to stand on the bus. This behavior comes from the wired-and nature of the CAN physical layer.

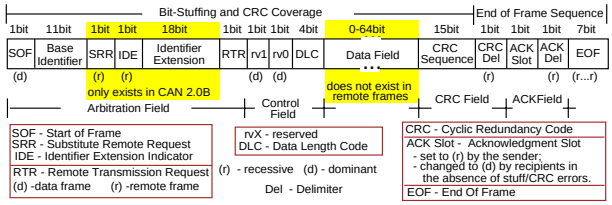


Figure 1: CAN frame structure

Frame identifiers are unique, and this feature, together with the wired-and behavior, is exploited to resolve conflicts in the access to the shared bus, whose access policy is *carrier sense multi-access with deterministic collision resolution* (CSMA/DCR) scheme: several nodes may jump on the bus at the same time, but while transmitting the frame identifier each node monitors the bus; for every bit, if the transmitted bit is recessive and a dominant value is monitored, the node gives up transmitting and starts to receive incoming data; the node transmitting the frame with the lowest identifier goes through and gets the bus. Automatic scheduling of a frame for retransmission is provided after a loss in an arbitration process.

The terminology we will use is explained below. A *message* is a user-level piece of information. A *frame* is a piece of encapsulated information that travels on the network. It may contain a *message*: in CAN, a *data frame* is used for that purpose. However, it may consist of control information only, such as a *remote frame*, which may be used in CAN to request the transmission of a data frame from one or more remote nodes. We will use *remote frames* in support of our protocols, as will be explained in Section 4.

Some details about CAN operation: the same identifier is used for data and remote frames, the distinction being made through the *remote transmission request* (RTR) bit (Figure 1); no data field is included in a remote frame; several nodes may simultaneously transmit the same remote frame[1]. Finally, we assume the utilization of the CAN 2.0B extended format: the identifier extension (Figure 1) is used to carry protocol control information, leaving the data field free to hold pure data.

2.1 Impairments to dependability

Let us now discuss the impairments of the CAN protocol [7, 15] with regard the provision of highly-dependable communication services. Those include shortcomings in fault-confinement and error detection/signaling mechanisms. CAN has a comprehensive set of such mechanisms, that make it very resilient. Due to lack of space, we do not discuss all of them, but the interested reader is referred to [7, 15, 2, 20] for details. Most failures are handled consistently by all nodes.

However, we have identified failure scenarios that can lead to undesirable symptoms such as inconsistent omission failures and duplicate message reception. These scenarios occur when faults hit the last two bits of the seven-bit end of frame delimiter (see Figure 1). However infrequent it may be, we also show ahead that the probability of occurrence of this scenario is high enough to be taken into account, at least for highly fault-tolerant applications of CAN. In fact, a naive atomic multicast protocol based on CAN properties alone, would fail under such a scenario. So, in this section we start by discussing the fault confinement mechanisms, then we discuss inconsistent failures, and finally equate the probability of such failures occurring.

[1]Provided that the DLC field (Figure 1) is equal for all nodes. Otherwise, an unresolvable collision would prevail. The CAN specification allows any value within the admissible range $[0, 8]$, to be used in the DLC field of remote frames.

Fault confinement aims at restricting the influence of defective nodes in bus operation. It is based on two different counters recording, at each node, transmit and receive errors, that is, *omission errors* causing frames not to be received at their destinations. A fully-integrated node is in the *error-active* state, the normal operating condition, where it is able to transmit/receive frames and fully participates in error detection/signaling actions. In the presence of errors, the error counters are updated, according to rules [7, 15] that make faulty nodes experience, with a very high probability, the highest error counter increase. When any error counter exceeds 127, the node enters an *error-passive* state where it is still able to transmit and receive frames, but after transmitting a data or remote frame is obliged to an extra eight-bit wait period, before it is allowed to start a new transmission. Furthermore, an error-passive node can only signal errors while transmitting. After behaving well again for a certain time, a node is allowed to re-assume the error-active status.

The erratic behavior of error-passive nodes represents a source of inconsistency that cannot go uncontrolled. A possible solution is that prior to a node reaching the error-passive state, it will have given a pre-specified number of omission errors, after which it will be shut-down, by forcing it to enter what is called the *bus-off* state. Most of existing CAN controllers (e.g. [6]) are able to issue a warning signal, to be used for that purpose, if any error counter exceeds a given threshold [15]. A node in the bus-off state does not participate in any bus activity, being unable to send or receive frames.

In consequence, the first problem, concerning the control of omission failures, is easily solvable, but the failure assumptions must be quantified and the protocols must take those assumptions into account (see Section 3 ahead). In absence of failures other than consistent omissions and node failures, the CAN protocol would assure what is called atomic multicast: a totally ordered message delivery either to all nodes or to none. For example, amongst the several error recovery mechanisms, the sender automatically submits the same message for retransmission, upon the occurrence of an error. Unfortunately, inconsistency scenarios may occur, that we discuss next.

If the sender detects no error up to the last bit of the end of frame delimiter, it considers that transmission as successful and no retransmission is due. However, should a subset of recipients[2], tagged $\times$ set in Figure 2-

[2]This subset may have only one element.

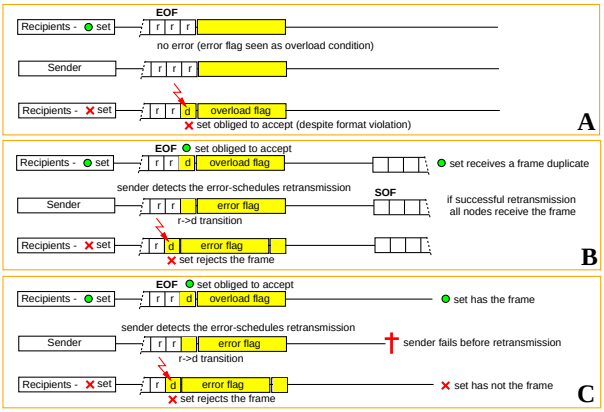


Figure 2: Inconsistency in CAN error handling

A, detect an incorrect dominant value in the last bit of the end of frame delimiter[3], the protocol specifies that they must accept the frame in order to preserve consistency with the complementary set of recipients, tagged • set in Figure 2-A, where a correct recessive value was detected.

This opens room for inconsistent frame omissions, that occur in the following case: a disturbance corrupts the last but one bit of the end of frame delimiter in the × set of recipients (Figure 2-B); signaling of the error begins at the bit following the corrupted one; no node in the × set accepts the frame. The sender also detects an error and schedules the frame for retransmission, after having performed its own error signaling actions. On the other hand, as explained in the previous paragraph, the recipients in the • set must accept the frame because the error is only signaled in the last bit of the end of frame delimiter.

At this point, we have a problem: an exact duplicate of the message will be accepted by the recipients in the • set of Figure 2-B, once retransmission is accomplished. This happens because the CAN protocol automatic message retransmission does not modify any frame field.

The problem gets worse if the sender fails after the first transmission and before the retransmission. This last scenario is depicted in Figure 2-C, which shows that inconsistent message omissions take place, affecting only

[3]Examples of causes for inconsistent detection are: electromagnetic interference or deficient receiver circuitry.

the $\times$ set.

2.2 Probability of inconsistent errors

In order to establish the importance of inconsistent error scenarios we have evaluated the probability of their occurrence. Other types of errors are not addressed: consistent errors are correctly processed by the CAN controllers; the residual probability of errors undetected by built-in CAN error-detection is negligible[2].

Inconsistent frame omissions
$p_{ifo} = (1 - ber)^{\mathcal{T}_{data}-2} \,.\, ber$
Node crash failures
$p_{fail} = 1 - \exp^{-\lambda.\Delta t}$
IMD - Inconsistent Message Duplicates
$p_{ifo} \,.\, (1 - p_{fail})$
IMO - Inconsistent Message Omissions
$p_{ifo} \cdot p_{fail}$

Table 1: Probabilities of inconsistent errors

The results of our evaluation are summarized in Table 1. The CAN inconsistent error probabilities are established as a function of a fundamental communication channel parameter - the *bit error rate* (ber). The model further considers an exponential distribution for node crashes (λ is the failure rate) and those events are regarded as independent from frame omissions. The probability of having an error in a particular bit of a frame obeys a geometric distribution, because the sender stops transmitting after the signaling of the first error. In addition, it is assumed that the probability for the same bit error being perceived simultaneously by all the nodes in the system is much lower than having it perceived only by a subset of the nodes. Thus, in this slightly simplified model the probability of inconsistent frame omissions only accounts for a temporal distribution of errors, occuring in the last but one bit of a frame with an overall length of $\mathcal{T}_{data}$ bits. Given a Δt period,

corresponding to the interval between the end of a transmission and the end of the last retransmission, if the sender crashes within Δt after the first error, with probability $(1-\exp^{-\lambda.\Delta t})$, an *inconsistent message omission* (IMO) occurs. Otherwise, the sender retransmits the message, but this recovery action generates *inconsistent message duplicates* (IMD).

Bit Error Rate (ber)	Node failures per hour (λ)	IMD/hour	IMO/hour
		$\Delta t = 5ms$	
10^{-4}	10^{-3}	2.84×10^{3}	3.94×10^{-6}
	10^{-4}	2.84×10^{3}	3.94×10^{-7}
10^{-5}	10^{-3}	2.86×10^{2}	3.98×10^{-7}
	10^{-4}	2.86×10^{2}	3.98×10^{-8}
10^{-6}	10^{-3}	2.87×10^{1}	3.98×10^{-8}
	10^{-4}	2.87×10^{1}	3.98×10^{-9}

Table 2: CAN inconsistent errors per hour

To finalize, we estimate the error probabilities in failures per hour, for several scenarios, in the reference period of one hour, for a 32 node CAN field-bus at 1 Mbps. A network overall load of 90% and an average frame length of $\mathcal{T}_{data} = 110$ bits are assumed. Bit error rates are presented both for benign and aggressive environments, such as noisy industries and automotive. Node crash failure rates are compliant with the values in [19, 9]. A latency of 5 ms is used as Δt, a time interval roughly corresponding to the time required for the transmission of one frame from each node in the network. The results from this evaluation, presented in Table 2, should be compared with the reference value of 10^{-9} incidents per hour, the well-known safety number from the aerospace industry [14], which is today also a goal for automotive applications [8]. The number of inconsistency incidents per hour goes down proportionally with a decrement in the network data rate, overall offered load or number of nodes.

3 System Model

In this section, we explain our fault assumptions, and discuss the CAN properties that underpins our system model.

Assumptions

We enumerate our assumptions for the system, formalizing the discussion made in Section 2.1. The model addresses a set of communicating processes sitting on a message passing subsystem implemented by CAN. Each process is attached to the network through a CAN controller. Together, they form a node. We assume that the processes are fail-silent and blame all temporary failures on the CAN network components. However, when a process crashes, the whole node crashes. In consequence, we may refer to *process* and *node* interchangeably.

We introduce the following definition: a component is **weak-fail-silent** if it behaves correctly or crashes if it does more than a given number of omission failures in an interval of reference, called the component's *omission degree*. This assumption can be enforced by the error confinement mechanisms discussed in Section 2.1, and is important to parameterize our protocols.

The **CAN bus** is a single-channel broadcast local network with the following failure semantics for the network components (anything between two processes, including network adapters and medium):

- individual components are **weak-fail-silent** with *omission degree* f_o;
- failure bursts never affect more than f_o transmissions in an interval of reference [4];
- omission failures may be inconsistent (i.e., not observed by all recipients);
- there is no permanent failure of shared network components (e.g. medium partition).

[4] For instance the duration of a broadcast round. Note that this assumption is concerned with the total number of failures of possibly different components.

CAN MAC-level properties

We can look at CAN as having a basic medium access control (MAC) sub-layer, that behaves basically like a LAN MAC sub-layer— as do most other field-buses— and as such, exhibits the same kind of properties that have been identified in previous works on LANs. See for example [18] for a description of abstract properties of a LAN. Figure 3 enumerates the set of MAC-level CAN properties relevant for this paper. MCAN4 maps the failure semantics introduced earlier onto the operational assumptions of CAN, being $k \geq f_o$.

MCAN1 - Broadcast: correct nodes receiving an uncorrupted frame transmission, receive the same frame.

MCAN2 - Error Detection: correct nodes detect any corruption done by the network in a locally received frame.

MCAN3 - Network Order: any two frames received at any two correct nodes, are received in the same order at both nodes.

MCAN4 - Bounded Omission Degree: in a known time interval T_{rd}, omission failures may occur in at most k transmissions.

Figure 3: CAN MAC-level properties

CAN LLC-level properties

However, CAN has error-recovery mechanisms on top of this basic functionality, that yield interesting message properties. Again, this has the flavor of the logical link control (LLC) sub-layer in LANs. Such properties have substantiated the claim that CAN exhibits atomic broadcast capability. Let us start by analyzing the definition of such a broadcast, in order that we may

understand why this is not so under all circumstances. We use an adaptation of the definition of atomic broadcast used by several authors [4, 16]:

AB1 - Validity: if a correct node broadcasts a message, then the message is eventually delivered to a correct node.

AB2 - Agreement: if a message is delivered to a correct node, then the message is eventually delivered to all correct nodes.

AB3 - At-most-once Delivery: any message delivered to a correct node is delivered at most once.

AB4 - Non-triviality: any message delivered to a correct node was broadcast by a node.

AB5 - Total Order: any two messages delivered to any two correct nodes, are delivered in the same order to both nodes.

However, the failure modes that we have identified cause the message-level properties of CAN to be somewhat different. Namely, while the omission failures specified by MCAN4 are masked in general at the LLC level by the retry mechanism of CAN, the existence of inconsistent omissions as discussed in Section 2.1 postulates two things:

- that there may be message duplicates when they are recovered;
- that some j of the k omissions will show at the LLC interface as inconsistent omissions.

Figure 4 enumerates the LLC-level properties of CAN. LCAN6 specifies the probability of inconsistent omission failures j, where j is normally several orders of magnitude smaller than k (cf.§2.1). The other five properties explain why CAN does not ensure atomic broadcast alone. LCAN1 and LCAN4 are in conformity with the AB specification. However, LCAN2 is conditioned to the sender not failing, and LCAN3 postulates that a message can be delivered in duplicate. LCAN5 is not even ensured. This clearly violates the atomic broadcast specification. In fact, it does not even guarantee reliable broadcast, since a reliable broadcast specification is equivalent to properties AB1 to AB4.

LCAN1 - Validity: if a correct node broadcasts a message, then the message is eventually delivered to a correct node.

LCAN2 - Best-effort Agreement: if a message is delivered to a correct node, then the message is eventually delivered to all correct nodes, if the sender remains correct.

LCAN3 - At-least-once Delivery: any message delivered to a correct node is delivered at least once.

LCAN4 - Non-triviality: any message delivered to a correct node was broadcast by a node.

LCAN5 - Total Order: *not ensured.*

LCAN6 - Bounded Inconsistent Omission Degree: in a known time interval T_{rd}, inconsistent omission failures may occur in at most j transmissions.

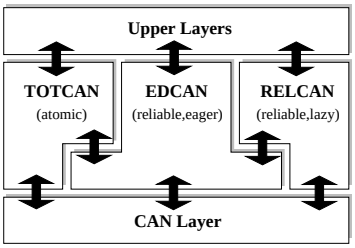
Figure 4: Basic CAN LLC-level properties

In consequence, the objective of the paper is to devise a set of mechanisms to be inserted between the exposed interface provided by the CAN layer and the user processes, in order to transform the LCAN properties provided by the former, into the AB properties expected by the latter. This will be addressed in the next section.

4 Fault-Tolerant Broadcasts in CAN

We now present a set of fault-tolerant broadcast protocols that make use of the unique CAN properties. We depart from an eager diffusion-based pro-

tocol, called EDCAN. This protocol exploits the properties of CAN *remote* frames to optimize the diffusion of messages with an empty data field. Useful for the dissemination of control information, EDCAN is less efficient in disseminating messages with a non-empty data field. So, we have improved the basic protocol to provide an unordered reliable broadcast primitive, called RELCAN, and a totally ordered primitive, called TOTCAN. The protocol suite, which is illustrated in Figure 5, executes on top of the CAN layer. Each protocol provides a request primitive (used to invoke the protocol), a confirm primitive (used to inform the sender of protocol local completion), and an indication primitive (used to deliver the message to the upper layer).



Primitive Type	Protocol		
	EDCAN	RELCAN	TOTCAN
Request	edcan.req	relcan.req	totcan.req
Confirm	edcan.cnf	relcan.cnf	totcan.cnf
Indication	edcan.ind	relcan.ind	totcan.ind

Figure 5: CAN fault-tolerant broadcast protocol suite

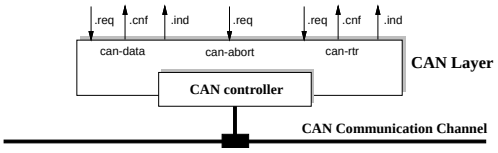
None of the protocols is based on the exchange of acknowledgments [10, 16]: such an approach is not an interesting solution in CAN, because it consumes too much bandwidth (a scarce resource in CAN) and makes no use of the built-in error detection properties.

4.1 CAN layer

The CAN layer is made from a CAN controller (e.g. [6]) and the corresponding software driver, that includes primitives for: *request* the transmission (.req) of data or control messages[5], supporting arbitration of requests by ur-

[5]Control messages are encapsulated in remote frames.

gency level on both local and global basis; *confirm* to the user a successful message transmission (.cnf), guaranteeing that property LCAN1 is secured; *indication* of a message arrival (.ind). The semantics of each particular primitive is summarized in Figure 6. Most of the attributes are defined in the standard document [7] and have an appropriate support from the CAN controller. However, a few exceptions exist: i) local arbitration by urgency level may require specific management actions [6]; ii) reception of own transmissions is not assured in all controllers [6], so low-level engineering may be required; iii) the local execution environment must process frame arrivals with a latency low enough to guarantee that no receive buffer overrun incidents will ever occur[6].



Primitives		Semantics summary
Data	Remote	
can-data.req		Only a node is allowed to transmit, at a time.
	can-rtr.req	Several nodes may simultaneously transmit the same remote frame.
can-data.cnf	can-rtr.cnf	Signals the successful transmission of a frame.
can-data.ind	can-rtr.ind	Signals the arrival of a frame, including own transmissions.
can-abort.req		Aborts a frame transmission request. Has effect only on pending requests.

Figure 6: CAN layer structure and interface

The protocols above the CAN layer use the message format illustrated in Figure 7. The fields relevant for protocol operation include: a *type* reference, the *sender identifier* and a *sequence number*. The type reference merges *urgency class* and *control data* information. The remaining fields only matter

[6]This kind of omission failures have not been included in our model.

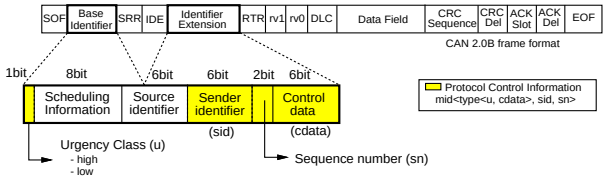


Figure 7: Information in CAN frame identifiers

to communication channel access arbitration. In data frames, the *source identifier* references the node actually sending the frame; in remote (control) frames it is identical to the *sender identifier*. The *scheduling information* specifies message urgency, given traffic patterns, latency classes and overall offered load [17, 21].

4.2 Message diffusion

The first protocol that we discuss is a diffusion-based protocol [3, 1] with some optimizations to save channel bandwidth. In this protocol, the recipients are responsible for retransmitting the message. Retransmissions are issued as soon as the original message is received; thus we have called this protocol "Eager Diffusion", or simply EDCAN. If enough nodes retransmit the message, one of these nodes will be a non-faulty sender and CAN properties will ensure the reliability of message delivery. The protocol is sketched in Figure 8. The protocol is invoked by the upper layer providing two parameters: a unique message identifier and an optional data field. As discussed in Section 4.1, the control information in the message identifier includes a message type, source identifier, and sequence number.

The protocol works as follows. The sender requests the transmission of the message to the CAN layer. For messages with data field the can-data primitive is used. For messages with an empty data field, remote frames (can-rtr) are used. If the sender does not fail the original message is delivered. To tolerate the failure of the original sender, recipients deliver the first copy of the message and eagerly retransmit it.

For messages with a data field, retransmissions flow on the channel one at a time. This may be too costly in terms of network load. The bounded inconsistent omission degree property (LCAN6) is exploited to optimize network bandwidth consumption: as soon as a node receives $(j + 1)$ copies of

Eager Diffusion-based Protocol (EDCAN)

```
Initialization
i01  ndup(mid) := 0; // number of duplicates, kept for each message

Sender
s10  when edcan.req(mid⟨type,p,n⟩, mess) invoked at p do
s11       if mess = NULL then
s12            can-rtr.req(mid);
s13       else
s14            can-data.req(mid, mess);
s15  od;
s16  when can-rtr.cnf(mid)
s17  or can-data.cnf(mid, mess) confirmed do
s18       deliver edcan.cnf (mid,mess);
s19  od;

Recipient
r00  when can-data.ind(mid, mess) received at q
r01  or can-rtr.ind(mid, mess=NULL) received at q do
r02       ndup(mid) := ndup(mid) + 1;
r03       if ndup(mid)= 1 then // new message
r04            edcan.ind (mid, mess);
r05            if mess = NULL then
r06                 can-rtr.req(mid); // clustered
r07            else
r08                 can-data.req(mid, mess);
r09            fi;
r10       elif ndup(mid) > j then
r11            can-abort.req(mid);
r12       fi;
r13  od;
```

Figure 8: Eager diffusion-based protocol

the same message it tries to abort the corresponding send request. However: only pending requests can be aborted (cf.§ 4.1); protocol execution delays may prevent a non-negligible number of requests to be timely aborted. As a result, a number of transmissions greater than $(j + 1)$ should be expected. Although we do not advocate the straight utilization of EDCAN to broadcast messages with a data field, it may be useful to other protocols. For example, ahead we will use EDCAN for error recovering upon sender failure, in a reliable broadcast protocol.

A more efficient optimization of network bandwidth utilization can be implemented when EDCAN is requested to broadcast a message with no data

field. It exploits an interesting property exhibited by remote frames: if two or more nodes transmit simultaneously identical remote frames, these transmissions can be "clustered" in a single physical frame, due to the wired-and nature of the physical layer. For the same reason, all recipients receive the original message at approximately the same time. However, slight variations on the corresponding processing delays prevent the different retransmission requests to be issued "exactly" at the same time.

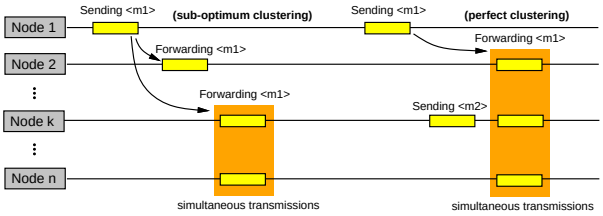


Figure 9: CAN remote frame clustering

In a lightly loaded network, one may expect the fastest node to start remote frame retransmission in advance, as shown in Figure 9. However, for acceptably short processing delay variances, other nodes will "cluster" their remote frame retransmissions, in a bounded number of physical channel packets[7]. Conversely, for a heavy loaded network it is reasonable to expect pending transmissions to have started in the meantime. The delays in network access, introduced by these transmissions, balance processing delays variance and thus it is reasonable to expect all retransmissions following the original dissemination of a remote frame to be clustered in a single physical layer transmission. In any case, for a network with a moderate number of nodes, this allows significant savings in network bandwidth. The upper layer should use remote frame features as much as possible, relying on control frames that do not require a data field. We will later present an (unordered) reliable protocol and an atomic broadcast protocol that use this approach.

[7]For example, in a system with a processing delay variance lower than $64\mu s$ (the duration of a 2.0B remote frame at 1 Mbps), these remaining transmissions will cluster in a single frame.

4.3 Lazy message diffusion

Despite the optimization we have introduced, the "Eager Diffusion" approach is not cost-effective for broadcast of data messages due its high bandwidth consumption. We now present a protocol that exploits CAN *validity* (LCAN1) and *best-effort agreement* (LCAN2) properties. The protocol, illustrated in Figure 10, was called RELCAN as it provides an unordered reliable broadcast service for data messages. Message retransmission by the protocol is only due in the event of sender failure.

The protocol works as follows. The sender assigns a unique identifier to the data message based on the node unique identifier and on a local sequence number. The control information is carried within the message identifier (type is set to R-DATA). Then, the sender calls an auxiliary "send-an-confirm" function, that initiates a two-phase protocol.

In the first phase, send-and-confirm requests message transmission and awaits the corresponding confirmation from the CAN controller. When this confirmation is obtained, the sender is sure that the message has been received by all correct recipients and initiates the second phase, disseminating a CONFIRM message. The reception of the CONFIRM message indicates to all recipients that the associated data message has been received and that no retransmission is required. Recipients deliver the first copy of the message and prepare themselves to retransmit the message. However, and in opposition to the eager protocol, retransmissions are not initiated immediately. Instead, recipients wait first for the CONFIRM message. Only in the case the CONFIRM message is not received, receivers retransmit the message by invoking the EDCAN protocol.

In the best case, the RELCAN protocol sends once the data message and once the CONFIRM control message. In the event of sender failure, the performance of RELCAN approaches the one observed in the EDCAN protocol. At this stage, we have succeeded in making properties LCAN2 and LCAN3 equivalent to properties AB2 and AB3.

4.4 Totally ordered protocol

The previous protocol makes no effort to enforce a total order on message delivery. In this section we propose a new protocol, called TOTCAN, that uses the CAN network order property (MCAN3) to provide a totally ordered

reliable broadcast service. The basic idea of the protocol is to have the messages delivered in the same order by which the encapsulating frames cross the communication channel. If due to omissions, the same message is forced to cross the channel more than once, only the order of the last retransmission (the successful one) is considered (previous duplicates are discarded).

The protocol is illustrated in Figure 11. As RELCAN, the protocol is also a two-phase protocol. In the first phase, called the *dissemination phase*, the sender tags the data message with its identification and a sequence number. As before, control information is carried in the identifier field (type is set to T-DATA). Then, the sender broadcasts the message using the bare CAN interface. When the message is received, instead of being immediately delivered to the application, it is held in a receive queue marked as UNSTABLE. In the presence of inconsistent omissions, the same message can be received more than once. To preserve network order, an UNSTABLE message is moved to the tail of the queue each time a message duplicate is received. The data message is never retransmitted by the recipients; should the sender fail before the message becomes stable, it is simply discarded by all recipients.

The second phase is initiated as soon as the sender receives, from the local CAN controller, a confirmation of success in the broadcast of the data message. At this point, the sender can be sure that all correct recipients have received the message. To make this information available to all recipients, the sender transmits an ACCEPT message. Because the ACCEPT message must be reliably broadcast to all recipients, the EDCAN protocol is used. Since the control field is able to hold all the information required, the ACCEPT message has no data field. When the ACCEPT is received, the associated message is marked as STABLE and can be delivered as soon as it reaches the head of the queue. The use of EDCAN in the second phase ensures that all recipients receive ACCEPT (or none does). In the case of sender failure before it is able to issue the ACCEPT to at least one correct destination, deadlock is prevented by timeout. This approach is possible due to the synchronous nature of the system.

In the best-case, TOTCAN requires the transmission of the data message plus the bandwidth corresponding to a pair of *remote* frames, required by the EDCAN protocol, in the reliable broadcast of the ACCEPT message. At this point, we also have secured property LCAN5 (equivalent to AB5), finally reaching our original goal of ensuring that CAN satisfies atomic broadcast.

4.5 Bounded sequence numbers

For sake of clarity, we describe the protocols using unbounded sequence numbers. The synchronous properties of the system allows to bound the sequence numbers: just two bits in the CAN message identifier are required to ensure correct protocol operation.

Due to space limitations CAN timeliness and synchronism properties were not included in the system model of Section 3. All these aspects will be addressed in a future paper.

5 Related Work

A number of authors have studied the problem of implementing fault-tolerant broadcasts. Some authors consider an *asynchronous* communication model, where no known bound is explicitly placed on message transaction delays [10]. In our system, the existence of bounded and known message transmission delays is assumed, as in other *synchronous* communication models [1, 3, 16]. Matching the application area of distributed control, a synchronous communication protocol is described in [9] that integrates a comprehensive set of services relevant for the implementation of fault-tolerant systems (e.g. group communication, membership and clock synchronization).

The use of group communications is not very common, in the so-called field-bus arena where most standards rely on OSI-like point-to-point communications. One of the few exceptions is the Controller Area Network [7, 15]. A set of CAN high layer protocols (SDS [5], J1939, OSEK [11]) specify the use of group communications, but lack to provide a clear definition of the corresponding system fault-model. An accurate definition of the system fault-model is essential to evaluate whether or not CAN weakness with regard fault-tolerant broadcast have been taken into account. Perhaps mislead by some lack of accuracy in CAN standards, some researchers neglect those aspects and claim that CAN supports (totally ordered) atomic broadcasts [12, 13].

6 Conclusions

There is a growing importance of fault-tolerant distributed systems based on field-buses. Given the utility of reliable and atomic broadcast for implementing applications on those systems, we studied the reliability of these protocols as provided by CAN native mechanisms. We discovered that under infrequent but plausible fault scenarios, CAN provides neither reliable nor atomic broadcast. Fault-tolerant systems using those primitives would function incorrectly, with unpredictable consequences for the controlled systems. In consequence, we formalized the properties actually secured by CAN, and we gave a suite of protocols that complement CAN's functionality in order to achieve reliable and atomic broadcast. As future work, we plan on doing a thorough study of the performance of our protocols.

References

[1] O. Babaoğlu and R. Drummond. Streets of Byzantium: Network Architectures for Fast Reliable Broadcasts. *IEEE Transactions on Software Engineering*, SE-11(6), June 1985.

[2] J. Charzinski. Performance of the error detection mechanisms in CAN. In *Proceedings of the 1st International CAN Conference*, pages 1.20–1.29, Mainz, Germany, September 1994. CiA.

[3] F. Cristian. Synchronous atomic broadcast for redundant broadcast channels. Technical report, IBM Almaden Research Center, San Jose, California, USA, 1990.

[4] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S.J. Mullender, editor, *Distributed Systems*, ACM-Press, chapter 5, pages 97–145. Addison-Wesley, 2nd edition, 1993.

[5] Honeywell Inc - MICRO SWITCH Division, Freeport, IL, USA. *Smart Distributed System - Application Layer Protocol (version 2.0)*, November 1996.

[6] Intel. *82527 - Serial Communications CAN Protocol Controller*, December 1995.

[7] ISO. *ISO International Standard 11898 - Road vehicles - Interchange of digital information - Controller Area Network (CAN) for high-speed communication*, November 1993.

[8] H. Kopetz. Automotive electronics - present state and future prospects. In *Digest of Papers of the 25th International Symposium on Fault-Tolerant Computing Systems - Special Issue*, pages 66–75, Pasadena, California-USA, June 1995. IEEE.

[9] H. Kopetz and G. Grunsteidl. TTP - a protocol for fault-tolerant real-time systems. *IEEE Computer*, 27(1):14–23, January 1994.

[10] P.M. Melliar-Smith and L.E. Moser. Fault-Tolerant Distributed Systems Based on Broadcast Communication. In *Proceedings of the 9th Internacional Conference on Distributed Computing systems*, pages 129–133. IEEE, June 1989.

[11] OSEK/VDX Working Group. *OSEK/VDX Communications - Open Systems and the corresponding interfaces for automotive electronics (version 2.0A)*, October 1997.

[12] M. Peraldi and J. Decotignie. Combining real-time features of local area networks FIP and CAN. In *Proceedings of the 2nd International CAN Conference*, pages 8.11–8.21, London, England, October 1995. CiA.

[13] S. Poledna. Fault tolerance in safety critical automotive applications: Cost of agreement as a limiting factor. In *Digest of Papers of the 25th International Symposium on Fault-Tolerant Computing Systems*, pages 73–82, Pasadena, California-USA, June 1995. IEEE.

[14] D. Powell. Failure mode assumptions and assumption coverage. In *Digest of Papers, The 22nd International Symposium on Fault-Tolerant Computing Systems*, pages 386–395, Boston, Massachusetts-USA, July 1992. IEEE.

[15] Robert Bosch GmbH. *CAN Specification Version 2.0*, September 1991.

[16] L. Rodrigues and P. Veríssimo. xAMp: a Multi-primitive Group Communications Service. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 112–121, Houston, Texas, October 1992. IEEE.

[17] K. Tindell and A. Burns. Guaranteeing message latencies on Controler Area Network. In *Proceedings of the 1st International CAN Conference*, pages 1.2–1.11, Mainz, Germany, September 1994. CiA.

[18] P. Veríssimo. Real-time Communication. In S.J. Mullender, editor, *Distributed Systems*, ACM-Press, chapter 17, pages 447–490. Addison-Wesley, 2nd edition, 1993.

[19] P. Veríssimo and H. Kopetz. Design of distributed real-time systems. In S.J. Mullender, editor, *Distributed Systems*, ACM-Press, chapter 19, pages 511–530. Addison-Wesley, 2nd edition, 1993.

[20] P. Veríssimo, J. Rufino, and L. Ming. How hard is hard real-time communication on field-buses? In *Digest of Papers, The 27th International Symposium on Fault-Tolerant Computing Systems*, Washington - USA, June 1997. IEEE.

[21] K. Zuberi and K. Shin. Non-preemptive scheduling of messages on Controller Area Networks for real-time control applications. In *Proceedings of the IEEE Real-Time Technology and Application Symposium*, Chicago, Illinois-USA, May 1995. IEEE.

Lazy Diffusion-based Protocol (RELCAN)

```
i01  rel_sn := 0; // local sequence number
i02  ndup(mid) := 0; // number of duplicates, kept for each message
i03  data(mid) := NULL; // data part of the message

send-and-confirm (auxiliary function)
a01  when send-and-confirm(mid⟨R-DATA,s,n⟩, mess) invoked at p do
a02       can-data.req(mid⟨R-DATA,s,n⟩, mess);
a03  od;
a04  when can-data.cnf(mid⟨R-DATA,s,n⟩, mess) received do
a05       can-rtr.req (mid⟨CONFIRM,s,n⟩);
a06  od;

Sender
s01  when relcan.req(mess) invoked at p do
s02       rel_sn := rel_sn + 1;
s03       send-and-confirm (mid⟨R-DATA,p,rel_sn⟩, mess);
s04       relcan.cnf (mess);
s05  od;

Recipient
r00  when can-data.ind(mid⟨R-DATA,p,n⟩, mess) received at q do
r01       ndup(mid) := ndup(mid) + 1;
r02       data(mid) := mess;
r03       start alarm (mid);
r04       if ndup(mid)= 1 then // new message
r05            relcan.ind (mess);
r06       fi;
r07  od;
r08  when can-rtr.ind(mid⟨CONFIRM,s,n⟩) received at q do
r09       data(mid) := NULL;
r10       cancel alarm(mid);
r11  od;
r12  when alarm(mid) expires at q do
r13            edcan.req (mid, data(mid));
r14  od;
r15  when edcan.ind(mid⟨R-DATA,p,n⟩, mess) received at q do
r16       ndup(mid) := ndup(mid) + 1;
r17       if ndup(mid)= 1 then // new message
r18            relcan.ind (mess);
r19       fi;
r20  od;
```

Figure 10: Reliable broadcast protocol

Totally Ordered Protocol (TOTCAN)

```
i00  tot_sn := 0; // local sequence number
i01  tot_queue := empty // queue of received messages
i02  // enqueue(tot_queue,mid,mess)
          inserts the message at the end of the queue as UNSTABLE
i03  // mess := dequeue(tot_queue, mid)
          removes a message from the queue
i04  // stable(tot_queue,mid) marks a message as STABLE

deliver-in-order(tot_queue) // auxiliary function
a00  deliver-in-order(tot_queue) do
a01       while message mid at the head of tot_queue is STABLE do
a02            mess = dequeue (mid);
a03            totcan.ind (mess);
a04       od;
a05  od;

Sender
s10  when totcan.req(mess) invoked at p do
s11       tot_sn := tot_sn +1;
s12       can-data.req(mid⟨ T-DATA, p, tot_sn ⟩, mess );
s13  od;
s14  when can-data.req(mid⟨ T-DATA, p, tot_sn ⟩, mess) confirmed do
s15       edcan.req(mid⟨ACCEPT, p, tot_sn ⟩, NULL);
s16  od;
s16  when edcan.conf(mid⟨ACCEPT, p, tot_sn ⟩, NULL) received do
s17       totcan.cnf(mess);
s18  od;

Recipient
r00  when can-data.ind(mid⟨T-DATA,p,tot_sn ⟩,mess) received at q do
r01       // preserve network order
r01       dequeue(tot_queue, mid);
r02       enqueue(tot_queue, mid, mess);
r03       start alarm (mid);
r04  od;
r05  when edcan.ind (mid⟨ ACCEPT, p, tot_sn⟩, NULL) received do
r06       stable(tot_queue, mid);
r07       deliver-in-order (tot_queue);
r08  od;
r09  when alarm (mid) expires do
r10       dequeue(tot_queue, mid); // discard the message
r11       deliver-in-order (tot_queue);
r12  od;
```

Figure 11: Totally ordered protocol