Centro de Sistemas Telemáticos e Computacionais

Instituto Superior Técnico

NavIST Group
Fault-Tolerant Real-Time Distributed
Systems and Industrial Automation

Design of Fault-Tolerant Broadcast Protocols for CAN

CSTC Technical Report RT-97-06

José Rufino, *et al.*

December 1997

Centro de Sistemas Telemáticos e Computacionais - NavIST Group
Fault-Tolerant Real-Time Distributed Systems and Industrial Automation

Design of Fault-Tolerant Broadcast Protocols for CAN

A condensed version of this report has been published in the *Digest of Papers of the 28th Fault-Tolerant Computing Symposium*, Munich, Germany, June 1998.

Technical Report: CSTC RT-97-06
Authors: J. Rufino, P. Veríssimo, G. Arroz, C. Almeida and L. Rodrigues
Date: December 1997 (Revised in June 1998)

Design of Fault-Tolerant Broadcast Protocols for 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 work, 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 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.

[*] 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.

However, the migration of fault-tolerant communication systems to the realm of field-buses presents non-negligible problems, that we address in this work, 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 based on the assumption that CAN supports a (totally ordered) atomic broadcast service [14, 15, 6]. 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.

This work starts 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 [9, 17], for details about the CAN protocol.

2 Controller Area Network

The Controller Area Network (CAN) is a bus with a multi-master architecture [9, 17]. 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.

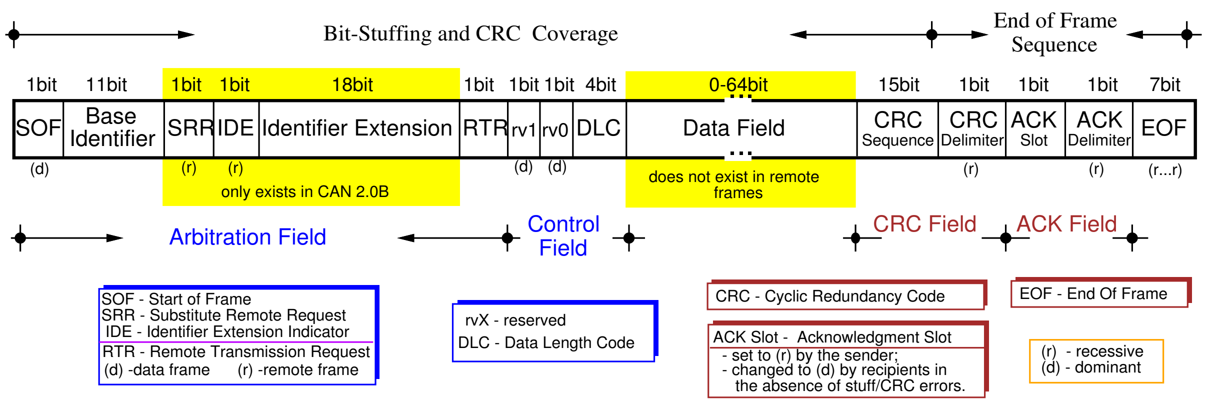


Figure 1: CAN frame structure

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 [9, 17] 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. We do not discuss all of them, but the interested reader is referred to [9, 17, 2, 20, 26] 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.

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 [9, 17] 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.

[1] Provided that the DLC field (Figure 1) is equal for all nodes. Otherwise, an un-resolvable 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.

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. [8]) are able to issue a warning signal, to be used for that purpose, if any error counter exceeds a given threshold [17]. 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.

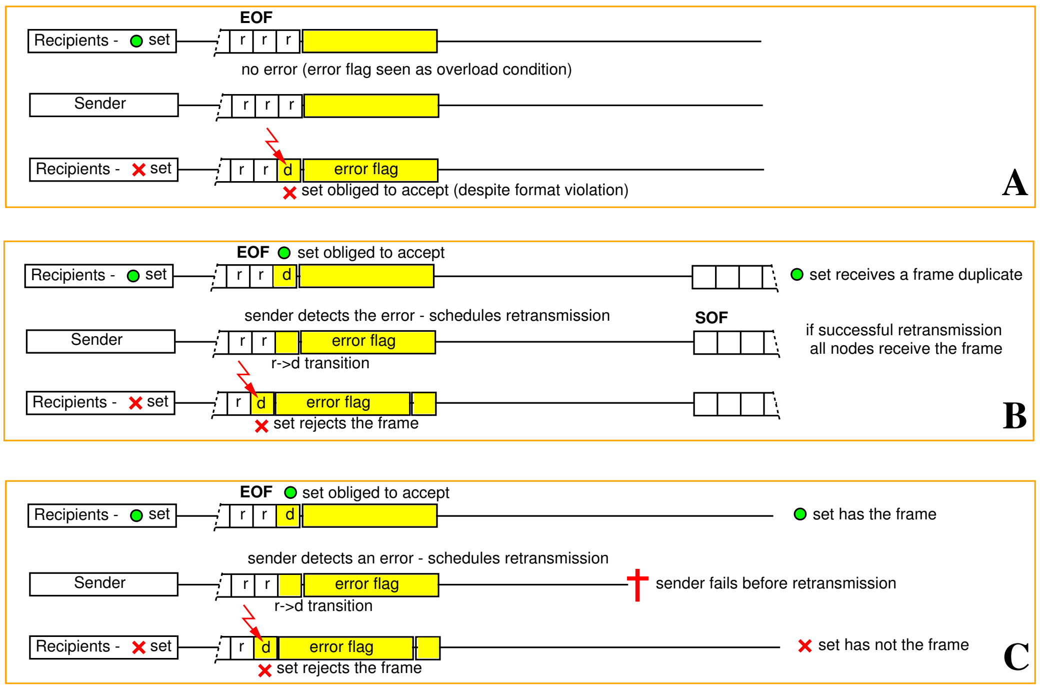

EOF
Recipients - ● set
r r r
no error (error flag seen as overload condition)
Sender
r r r
Recipients - × set
r r d
error flag
× set obliged to accept (despite format violation)
A
EOF ● set obliged to accept
Recipients - ● set
r r d
● set receives a frame duplicate
sender detects the error - schedules retransmission
SOF
Sender
r r
error flag
r->d transition
if successful retransmission
all nodes receive the frame
Recipients - × set
r d
error flag
× set rejects the frame
B
EOF ● set obliged to accept
Recipients - ● set
r r d
● set has the frame
sender detects an error - schedules retransmission
Sender
r r
error flag
r->d transition
sender fails before retransmission
Recipients - × set
r d
error flag
× set rejects the frame
× set has not the frame
C

Figure 2: Sources of inconsistency in CAN error handling

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-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 $\bullet$ 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 $\times$ set of recipients (Figure 2-B); signaling of the error begins at the bit following the corrupted one; no node in the

[2]This subset may have only one element.
[3]Examples of causes for inconsistent detection are: electromagnetic interference or deficient receiver circuitry.

$\times$ 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 $\bullet$ 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 $\bullet$ 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 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].

Node crash in time interval $[t, t+\Delta t]$	$p_{fail} = 1 - \exp^{-\lambda.\Delta t}$
Inconsistent frame omissions	$p_i = (1 - ber)^{\mathcal{T}_{data}-2} \, . \, ber$
Inconsistent Message Duplicates	$p_i \, . \, (1 - p_{fail})$
Inconsistent Message Omissions	$p_i \, . \, p_{fail}$

Figure 3: Probabilities of inconsistent errors

The results of our evaluation are summarized in Figure 3. 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* occurs. Otherwise, the sender retransmits the message, but this recovery action generates *inconsistent message duplicates*.

To finalize, we estimate the error probabilities in failures per hour, for several scenarios, in the reference period of one hour. 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. 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 (cf. Table 1) both for benign and aggressive environments, such as noisy industries and automotive. Node crash failure

Bit Error Rate (ber)	Node failures per hour (λ)	Inconsistent Message Duplicates per hour		Inconsistent Message Omissions per hour	
		$\Delta t = 5ms$	$\Delta t = 20ms$	$\Delta t = 5ms$	$\Delta t = 20ms$
10^{-4}	10^{-3}	2.84×10^{3}	2.84×10^{3}	3.94×10^{-6}	1.58×10^{-5}
	10^{-4}	2.84×10^{3}	2.84×10^{3}	3.94×10^{-7}	1.58×10^{-6}
10^{-5}	10^{-3}	2.86×10^{2}	2.86×10^{2}	3.98×10^{-7}	1.59×10^{-6}
	10^{-4}	2.86×10^{2}	2.86×10^{2}	3.98×10^{-8}	1.59×10^{-7}
10^{-6}	10^{-3}	2.87×10^{1}	2.87×10^{1}	3.98×10^{-8}	1.59×10^{-7}
	10^{-4}	2.87×10^{1}	2.87×10^{1}	3.98×10^{-9}	1.59×10^{-8}

Table 1: CAN inconsistent errors per hour

rates are compliant with the values in [25, 11]. Two different latencies (5 and 20 ms) are used as Δt, with the former 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 1, should be compared with the reference value of 10^{-9} incidents per hour, the well-known safety number from the aerospace industry [16], which is today also a goal for automotive applications [10].

The results of Table 1 are a clear indication that the influence of inconsistent errors on system correctness cannot be neglected, at least for highly fault-tolerant applications of CAN. A solution to this problem is required.

3 System Model

In this section, we explain our fault assumptions, and discuss the CAN properties that underpin 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).

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 [24] for a description of abstract properties of a LAN. Figure 4 enumerates the set of MAC-level CAN properties relevant for this work. 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 4: 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 [5, 19]:

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.

[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.

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 5 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. The *total order* property (AB5) is not even ensured (LCAN5). 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*.

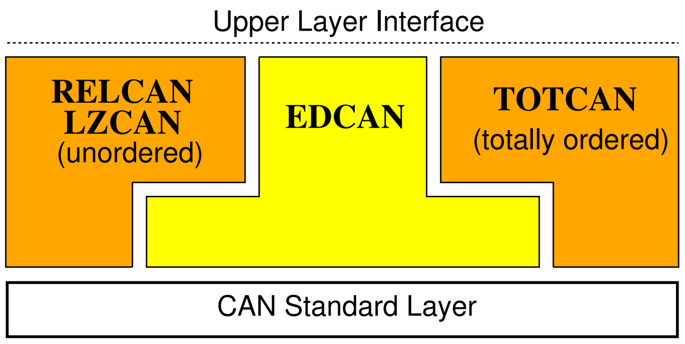
LCAN6 - Bounded Inconsistent Omission Degree: in a known time interval T_{rd}, inconsistent omission failures may occur in at most j transmissions.

Figure 5: CAN LLC-level properties

In consequence, the objective of this work 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 protocol, 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; a lazy (unordered) reliable broadcast primitive, called LZCAN; a totally ordered atomic broadcast primitive, called TOTCAN. The protocol suite, which is illustrated in Figure 6, 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	Protocol			
Type	EDCAN	RELCAN	LZCAN	TOTCAN
Request	edcan.req	relcan.req	lzcan.req	totcan.req
Confirm	edcan.cnf	relcan.cnf	lzcan.cnf	totcan.cnf
Indication	edcan.ind	relcan.ind	lzcan.ind	totcan.ind

Figure 6: CAN fault-tolerant broadcast protocol suite

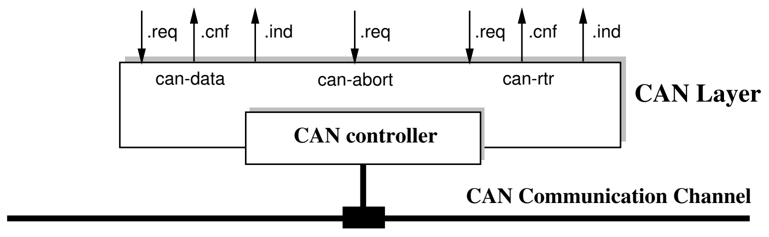
None of the protocols is based on the exchange of acknowledgments [12, 19]: such approach is not an interesting solution in CAN, because it consumes too much bandwidth (a scarce resource in CAN) and makes no use of the built-in error detection properties.

4.1 CAN Layer

The CAN layer is made from a CAN controller (e.g. [8]) and the corresponding software driver, that includes a set of standard primitives for: *request* the transmission (.req) of data or control messages[5]; *confirm* to the user a successful message transmission (.cnf); *indication* of a message arrival (.ind). The semantics of each particular primitive is summarized in Figure 7. The information encapsulated in data/remote frames always include a message control field. Data frames may also include message data. Most of the attributes are defined in the standard document [9] 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 [8];
ii) reception of own transmissions is not assured in all controllers [8], 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].

[5]Control messages are encapsulated in remote frames.
[6]This kind of omission failures have not been included in our model.



Primitives		Semantics
Data	Remote	
can-data.req		Arbitration of requests by urgency level, on local and global basis. Only a node is allowed to transmit, at a time.
	can-rtr.req	Arbitration of requests by urgency level, on local and global basis. Several nodes may simultaneously transmit the same remote frame. Remote frames do not have a data field.
can-data.cnf	can-rtr.cnf	Signals the successful transmission of a data/remote frame. Provides the guarantee that property LCAN1 is secured.
can-data.ind	can-rtr.ind	Signals the arrival of a data/remote frame, including own transmissions. The local execution environment has to guarantee that no frame is lost due to CAN controller receive buffer overrun.
can-data.nty		*Extension to standard:* signals the arrival of a data frame, without delivering the message data.
can-abort.req		Aborts a frame transmission request. Has effect only on pending requests. In-progress transmissions cannot be aborted.

Figure 7: CAN layer structure and interface

The set of primitives specified in Figure 7 includes an extension to the standard interface: a *notification* primitive (.nty), signaling the arrival of a data frame (own transmissions included). However, the message data field is not delivered. Only the message control information is included in the notification.

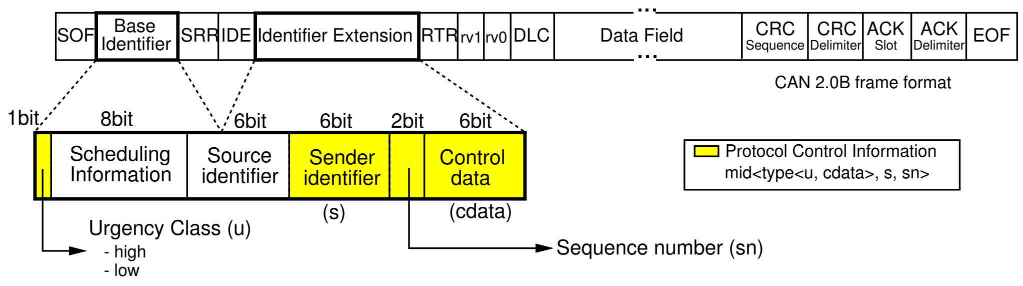


Figure 8: Protocol control information in CAN frame identifiers

The protocols above the CAN layer use the message format illustrated in Figure 8. The fields relevant for protocol operation include: a *type* reference, the *sender identifier* (*s*) and a *sequence number* (*sn*). The type reference merges *urgency class* (*u*) and *control data* (*cdata*) information. The remaining fields only matter 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 [23, 27].

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 9. 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 (*type*), sender identifier (*s*), and sequence number (*sn*).

Eager Diffusion-based Protocol (EDCAN)

```
Initialization
i00  ndup(mid) := 0;                                      // number of duplicates, kept for each message

Sender
s00  when edcan.req(mid⟨type,s,sn⟩, mess) invoked at p do  // mid, message identifier
s01      if mess = NULL then
s02          can-rtr.req (mid);
s03      else
s04          can-data.req (mid, mess);
s05      fi;
s06  od;
s07  when can-rtr.cnf(mid, mess:=NULL) confirmed at p or can-data.cnf(mid, mess) confirmed at p do
s08      edcan.cnf (mid, mess);
s09  od;

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

Figure 9: Eager diffusion-based protocol

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 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.

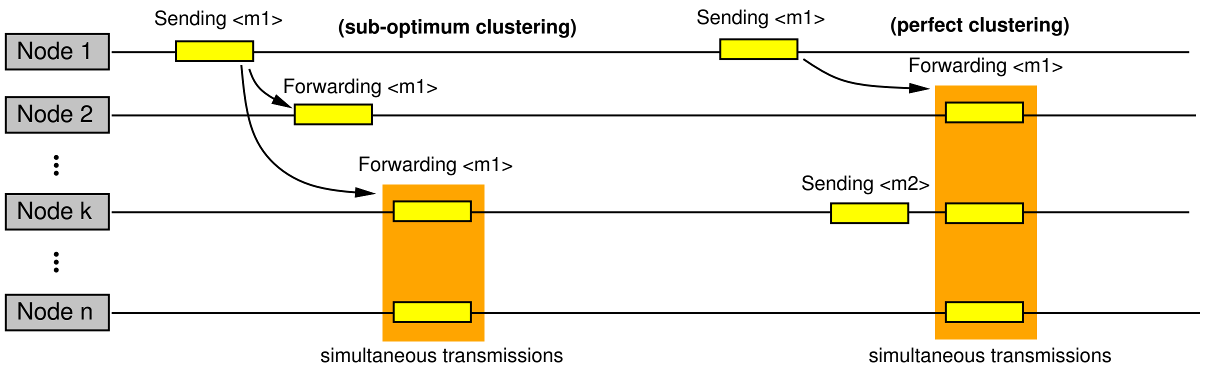


Figure 10: CAN remote frame clustering

In a lightly loaded network, one may expect the fastest node to start remote frame retransmission in advance, as shown in Figure 10. 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.

Note that a protocol variant where message dissemination continues even after the j transmissions, which corresponds to removing lines *r09-r10* in Figure 9, makes EDCAN resilient to a possible lack of coverage on the value assumed for the inconsistent omission degree bound, j (LCAN6). Naturally, this is achieved through a higher utilization of the network bandwidth.

[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 Unordered Reliable 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 11, 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.

Unordered Reliable Message Diffusion Protocol (RELCAN)

```
Initialization
i00 rel_sn := 0;                          // local sequence number
i01 ndup(mid) := 0;                       // number of duplicates, kept for each message
i02 data(mid) := NULL;                    // buffer to store the data part of the message

send-and-confirm (auxiliary function)
a00 send-and-confirm(mid⟨R-DATA,s,sn⟩, mess) do
a01     can-data.req (mid⟨R-DATA,s,sn⟩, mess);
a02     when can-data.cnf(mid⟨R-DATA,s,sn⟩, mess) confirmed do
a03         can-rtr.req (mid⟨CONFIRM,s,sn⟩);
a04     od;
a05 od;

Sender
s00 when relcan.req(mess) invoked at p do
s01     rel_sn := rel_sn + 1;
s02     send-and-confirm (mid⟨R-DATA,s:=p,rel_sn⟩, mess);
s03     relcan.cnf (mess);
s04 od;

Recipient
r00 when can-data.ind(mid⟨R-DATA,s,sn⟩, mess) received at q do
r01     ndup(mid) := ndup(mid) + 1;
r02     alarm_start.req ($T_{RELCAN}$,mid);       // $T_{RELCAN}$, is the protocol timeout value
r03     if ndup(mid) = 1 then                     // new message
r04         data(mid) := mess;                    // stores the received message
r05         relcan.ind (mess);
r06     fi;
r07 od;
r08 when can-rtr.ind(mid⟨CONFIRM,s,sn⟩) received at q do
r09     data(mid) := NULL;                        // clears the stored message
r10     alarm_cancel.req (mid);
r11 od;
r12 when alarm.nty(mid) received at q do          // timer expires
r13     edcan.req (mid, data(mid));
r14 od;
r15 when edcan.ind(mid⟨R-DATA,s,sn⟩, 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 11: Unordered reliable broadcast protocol

The protocol works as follows. The sender assigns a unique identifier to the data message based on the node unique identifier (*s*) and on a local sequence number (*rel_sn*). The control information is carried within the message identifier (type is set to R-DATA). Then, the sender calls an auxiliary "send-and-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 Lazy Message Diffusion

The utilization of the CAN network bandwidth by the RELCAN protocol is optimized in the unordered reliable broadcast protocol, to be discussed next. The protocol, illustrated in Figure 12, was called LZCAN and performs the "lazy" reliable diffusion of a message, trading a potentially high protocol maximum termination time in the presence of sender failure with a low utilization of the network bandwidth, should no failures occur.

The LZCAN protocol does not make use of any control message and works as follows. The sender assigns a unique identifier to the data message based on the node unique identifier (*s*) and on a local sequence number (*lz_sn*). The control information is carried within the message identifier (the type field is set to L-DATA). The message is sent simply using the CAN standard layer primitive, can-data.req.

The recipients store[8] and deliver the first copy of the message. In the absence of sender failure, the CAN properties *validity* (LCAN1) and *best-effort agreement* (LCAN2) guarantee message delivery to all correct nodes.

Message retransmission is initiated only upon the notification of sender failure, as provided by a companion membership service, such one of those described in [18]. The design of the LZCAN protocol assumes that no node is reintegrated before the notification of its failure, by the membership service. Upon node failure notification and for each node in the failed set, the protocol checks if some message is stored in the retransmission buffer (lines *r08-r12*, of Figure 12). Should this condition hold, the EDCAN protocol is invoked (edcan.req) for the reliable dissemination of each one of those messages.

The extension to the CAN standard layer interface, specified in Figure 7 and signaling the reception of a given data message, is used by the LZCAN protocol as an implicit confirmation that the previous data message has been successfully disseminated[9].

A message is removed from the LZCAN retransmission buffer when: its dissemination, by the sender, has been implicitly confirmed (line *r17*, of Figure 12); the message is being disseminated by the EDCAN protocol (line *r23*).

[8]Each received message is temporary kept in the protocol retransmission buffer.

[9]This optimization scheme assumes that within a node message transmissions are ordered according to their urgency and scheduling information (cf. Figure 8), which implies that message dissemination is confirmed only upon the reception of a message with an overall urgency level lower than the one being confirmed (line *r16*, of Figure 12).

The LZCAN protocol makes properties LCAN2 and LCAN3 equivalent to properties AB2 and AB3, thus ensuring message reliable broadcast. The advantage of the LZCAN protocol is that in the absence of sender failure, it does not incur in the control message overheads of the RELCAN protocol.

Lazy Reliable Message Diffusion Protocol (LZCAN)

```
Initialization
i00 lz_sn := 0;                  // local sequence number
i01 ndup(mid) := 0;              // number of duplicates, kept for each message
i02 data(mid) := NULL;           // message identifier and data, kept for each message

Sender
s00 when lzcan.req(mess) invoked at p do
s01     lz_sn := lz_sn + 1;
s02     can-data.req(mid⟨L-DATA,s:=p,lz_sn⟩, mess);
s03 od;
s04 when can-data.cnf(mid⟨L-DATA,p,lz_sn⟩, mess) confirmed at p do
s05     lzcan.cnf (mess);
s06 od;

Recipient
r00 when can-data.ind(mid⟨L-DATA,s,n⟩, mess) received at q do
r01     ndup(mid) := ndup(mid) + 1;
r02     if ndup(mid)= 1 then                                   // new message
r03         data(mid) := mess;
r04         lzcan.ind (mess);
r05     fi;
r06 od;
r07 when msh-can.nty(f_nodes) received at q do                 // node failure notification
r08     for each s ∈ f_nodes do
r09         for each d_mid⟨s⟩ and data(d_mid⟨s⟩) ≠ NULL do
r10             edcan.req (d_mid⟨s⟩, data(d_mid⟨s⟩));
r11         od;
r12     od;
r13 od;
r14 when can-data.nty(mid⟨s⟩) received at q do                 // data message notification
r15     for each d_mid⟨s⟩ and data(d_mid⟨s⟩) ≠ NULL do
r16         if u_rank(d_mid⟨s⟩) > u_rank(mid⟨s⟩) then          // u_rank accounts for message overall urgency
r17             data(d_mid⟨s⟩) := NULL;
r18         fi;
r19     od;
r20 od;
r21 when edcan.ind(mid⟨L-DATA,s,n⟩, mess) received at q do
r22     ndup(mid) := ndup(mid) + 1;
r23     data(mid) := NULL;
r24     if ndup(mid)= 1 then                                   // new message
r25         lzcan.ind (mess);
r26     fi;
r27 od;
```

Figure 12: Lazy reliable broadcast protocol

4.5 Totally Ordered Protocol

The previous protocols make 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).

Totally Ordered Message Diffusion Protocol (TOTCAN)

```
Initialization
i00  tot_sn := 0;                              // local sequence number
i01  tot_queue := empty                        // queue of received messages
i02  // enqueue(tot_queue,mid,mess)            inserts a message at the end of the queue as UNSTABLE
i03  // mess := dequeue(tot_queue, mid)        removes a message from the queue
i04  // discard(tot_queue, mid)                removes a message from the queue and releases the message buffer
i05  // stable(tot_queue,mid)                  marks a message as STABLE

deliver-in-order (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 (tot_queue, mid);
a03          totcan.ind (mess);
a04      od;
a05  od;

Sender
s00  when totcan.req(mess) invoked at p do
s01      tot_sn := tot_sn + 1;
s02      can-data.req (mid⟨T-DATA,s:=p,tot_sn⟩, mess);
s03  od;
s04  when can-data.cnf(mid⟨T-DATA,p,tot_sn⟩, mess) confirmed at p do
s05      edcan.req (mid⟨ACCEPT,p,tot_sn⟩, NULL);
s06  od;
s06  when edcan.cnf(mid⟨ACCEPT,p,tot_sn⟩,NULL) confirmed at p do
s07      totcan.cnf (mess);
s08  od;

Recipient
r00  when can-data.ind(mid⟨T-DATA,s,sn⟩, mess) received at q do
r01      discard (tot_queue, mid);                 // preserves network order, by: discarding the queued message;
r02      enqueue (tot_queue, mid, mess);           // queuing the last received message duplicate.
r03      alarm_start.req ($T_{TOTCAN}$,mid);       // $T_{TOTCAN}$, is the protocol timeout value
r04  od;
r05  when edcan.ind(mid⟨ACCEPT,s,sn⟩, NULL) received at q do
r06      alarm_cancel.req (mid);
r07      stable (tot_queue, mid);
r08      deliver-in-order (tot_queue);
r09  od;
r10  when alarm.nty(mid) received at q do         // timer expires
r11      discard (tot_queue, mid);                 // discards the queued message
r12      deliver-in-order (tot_queue);
r13  od;
```

Figure 13: Totally ordered protocol

The protocol is illustrated in Figure 13. 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 (s) and a sequence number (*tot_sn*). 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, if remote frame clustering is perfect. When remote frame clustering is sub-optimum, an extra ACCEPT message is transmitted (Figure 10).

At this point, we also have secured property LCAN5 (equivalent to AB5), finally reaching our original goal of ensuring that CAN satisfies atomic broadcast.

4.6 Bounded Sequence Numbers

For sake of clarity, we describe the protocols using unbounded sequence numbers. The synchronous properties of the system allow to bound the sequence numbers: just two bits in the CAN message identifier are required to ensure correct protocol operation. In case of sender failure, the recipients are able to establish the message order and, if required, retransmit the last message. The use of a sequence number based on a single bit will not allow message ordering. This problem was identified and thoroughly analyzed in [4].

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 [12]. In our system, the existence of bounded and known message transmission delays is assumed, as in other *synchronous* communication models [1, 3, 19]. Matching the application area of distributed control, a synchronous communication protocol is described in [11] 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 [17, 9]. A set of CAN high layer protocols (SDS [7], J1939, OSEK [13]) 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 [14, 15, 6].

6 Bandwidth Utilization

In this section, we analyze the utilization of CAN bandwidth by each protocol in our suite of fault-tolerant broadcast protocols.

Unless stated otherwise, we assume the dissemination of a data message of maximum length, which is encapsulated in a CAN data frame of nominal duration $\mathcal{T}_{Mdata}$. We use the superscripts ^{bc} and ^{wc} to signal the best and worst-case durations, of a given frame. Control messages are encapsulated in remote frames, whose nominal duration is given by $\mathcal{T}_{m-rtr}$.

The numeric values for these parameters, drawn from [22], are summarized in Table 2. The values in Table 2 include the nominal duration of the intermission period, i.e. the nominal three bit bus idle period that usually precedes the transmission of any data or remote frame [9, 17].

Frame	Symbol	Nominal data field length (*bits*)	Duration (*bit-times*)			
			CAN 2.0A		**CAN 2.0B**	
			min. (^{bc})	*max.* (^{wc})	*min.* (^{bc})	*max.* (^{wc})
Data frame	$\mathcal{T}_{mdata}$	0	47	55	67	80
	$\mathcal{T}_{Mdata}$	64	111	135	131	160
Remote frame	$\mathcal{T}_{m-rtr}$	0	47	55	67	80

Table 2: Normalized duration of CAN data and remote frames

6.1 EDCAN protocol

Let us start our analysis discussing the CAN bandwidth used by the EDCAN protocol in the reliable dissemination of a data message.

Dissemination of Data Messages

In the absence of omission failures, the utilization of network bandwidth, U, expressed in bit-times, is simple given by[10]:

$$U_{EDCAN}^{bc\leftarrow nf}(data) = (j + h + 1) \cdot \mathcal{T}_{Mdata}^{bc} \tag{1}$$

$$U_{EDCAN}^{wc\leftarrow nf}(data) = (j + h + 1) \cdot \mathcal{T}_{Mdata}^{wc} \tag{2}$$

which accounts for: the best and worst-case durations of the $(j+1)$ messages required to terminate protocol execution; a number h of message transmit requests, that the EDCAN protocol is unable to timely abort.

In the event protocol execution is disturbed by the occurrence of j inconsistent frame omissions, the CAN bandwidth consumed in these frame transmission attempts[11] has to be accounted for, in addition to the network bandwidth utilization in the absence of failures.

[10]The superscripts ^{bc} and ^{wc} are used to signal, respectively, the best and worst-case utilization of CAN bandwidth. The superscript ^{nf} is used to signal a *no failure* scenario.

[11]A scenario signaled through the use of superscript ^{ifo}.

$$
\begin{aligned}
U^{wc\leftarrow ifo}_{EDCAN}(data) &= j\,.\,\mathcal{T}^{wc}_{Mdata} + U^{wc\leftarrow nf}_{EDCAN}(data) \\
&= j\,.\,\mathcal{T}^{wc}_{Mdata} + (j+h+1)\,.\,\mathcal{T}^{wc}_{Mdata} \\
&= (2\,.\,j+h+1)\,.\,\mathcal{T}^{wc}_{Mdata}
\end{aligned}
\tag{3}
$$

DISSEMINATION OF CONTROL MESSAGES

We now account for the CAN bandwidth used by the EDCAN protocol in the dissemination of a control message. Several situations are considered. In the first case, we assume that no omission failures occur and the perfect clustering (cf. Figure 10) of control message retransmissions.

$$U^{bc\leftarrow nf}_{EDCAN}(control) = 2\,.\,\mathcal{T}^{bc}_{m-rtr} \tag{4}$$

Thus, only two contributions accounted for in equation (4): the original dissemination of the control message; the (clustered) retransmission of this message by each recipient. Should clustering be sub-optimum (cf. Figure 10), an extra dissemination of a control message needs to be accounted for, as described by equation:

$$U^{wc\leftarrow nf}_{EDCAN}(control) = 3\,.\,\mathcal{T}^{wc}_{m-rtr} \tag{5}$$

In a worst-case scenario, we assume that the dissemination of the control message is disturbed by the occurrence of j inconsistent frame omissions. Thus:

$$U^{wc\leftarrow ifo}_{EDCAN}(control) = j\,.\,\mathcal{T}^{wc}_{m-rtr} + 3\,.\,\mathcal{T}^{wc}_{m-rtr} \tag{6}$$

6.2 RELCAN protocol

In the analysis of the RELCAN protocol, we assume again the dissemination of a data message of maximum length. In the absence of failures, we simply account for the transmission of the data message and of the corresponding CONFIRM message, encapsulated in a CAN remote frame, of duration $\mathcal{T}_{c-rtr} = \mathcal{T}_{m-rtr}$.

$$U^{bc\leftarrow nf}_{RELCAN} = \mathcal{T}^{bc}_{Mdata} + \mathcal{T}^{bc}_{c-rtr} \tag{7}$$

$$U^{wc\leftarrow nf}_{RELCAN} = \mathcal{T}^{wc}_{Mdata} + \mathcal{T}^{wc}_{c-rtr} \tag{8}$$

Should the sender fail before the issuing of CONFIRM signal, the EDCAN protocol is invoked. Under the assumption that the operation of the EDCAN protocol is disturbed by the occurrence of j inconsistent frame omissions, the utilization of CAN bandwidth is given by:

$$
\begin{aligned}
U^{wc\leftarrow ifo}_{RELCAN} &= \mathcal{T}^{wc}_{Mdata} + U^{wc\leftarrow ifo}_{EDCAN}(data) \\
&= \mathcal{T}^{wc}_{Mdata} + (2\,.\,j+h+1)\,.\,\mathcal{T}^{wc}_{Mdata} \\
&= (2\,.\,j+h+2)\,.\,\mathcal{T}^{wc}_{Mdata}
\end{aligned}
\tag{9}
$$

6.3 LZCAN protocol

Similar results can be drawn for the LZCAN protocol. However, since the LZCAN does not make use of control messages, the corresponding utilization of CAN bandwidth in the studied cases, is given by:

$$U_{LZCAN}^{bc \leftarrow nf} = \mathcal{T}_{Mdata}^{bc} \tag{10}$$

$$U_{LZCAN}^{wc \leftarrow nf} = \mathcal{T}_{Mdata}^{wc} \tag{11}$$

$$U_{LZCAN}^{wc \leftarrow ifo} = (2 \,.\, j + h + 2) \,.\, \mathcal{T}_{Mdata}^{wc} \tag{12}$$

6.4 TOTCAN protocol

Finally, we account for the utilization of CAN bandwidth by the TOTCAN protocol. In the absence of failures, one should take into account that the dissemination of the ACCEPT message, encapsulated in a CAN remote frame of duration $\mathcal{T}_{a-rtr} = \mathcal{T}_{m-rtr}$, may cluster in a perfect or sub-optimum ways. Thus:

$$U_{TOTCAN}^{bc \leftarrow nf} = \mathcal{T}_{Mdata}^{bc} + 2 \,.\, \mathcal{T}_{a-rtr}^{bc} \tag{13}$$

$$U_{TOTCAN}^{wc \leftarrow nf} = \mathcal{T}_{Mdata}^{wc} + 3 \,.\, \mathcal{T}_{a-rtr}^{wc} \tag{14}$$

In addition, should the dissemination of data messages be disturbed by the occurrence of j inconsistent frame omissions, then:

$$U_{TOTCAN}^{wc \leftarrow ifo} = (j + 1) \,.\, \mathcal{T}_{Mdata}^{wc} + 3 \,.\, \mathcal{T}_{a-rtr}^{wc} \tag{15}$$

6.5 Analytic results

The results from our evaluation are represented in Figure 14. The use of a 32 node CAN field-bus at 1 Mbps is assumed. The other relevant network parameters are: $j = 1$; $h = 1$.

The individual contributions of each protocol component to the overall utilization of CAN bandwidth are specifically identified in Figure 14. Two relevant contributions, which cannot be avoided, do concern: the original transmission of a message, by the CAN controller; message retransmission, by the CAN controller, in the event of inconsistent frame omissions (IFO). The other contributions, which are not intrinsically handled by the CAN controller, concern the specific overheads of each protocol: the issuing of simple control messages (e.g. the CONFIRM message, in the RELCAN protocol); the reliable broadcast of a control message, by the EDCAN protocol (e.g. the ACCEPT message of the TOTCAN protocol); the reliable dissemination of a data message, by the EDCAN protocol, in the event of sender failure.

Though protocol overheads may represent a significant part of the overall bandwidth used by each particular protocol, these messages are fundamental to secure protocol correctness in the presence of sender failure.

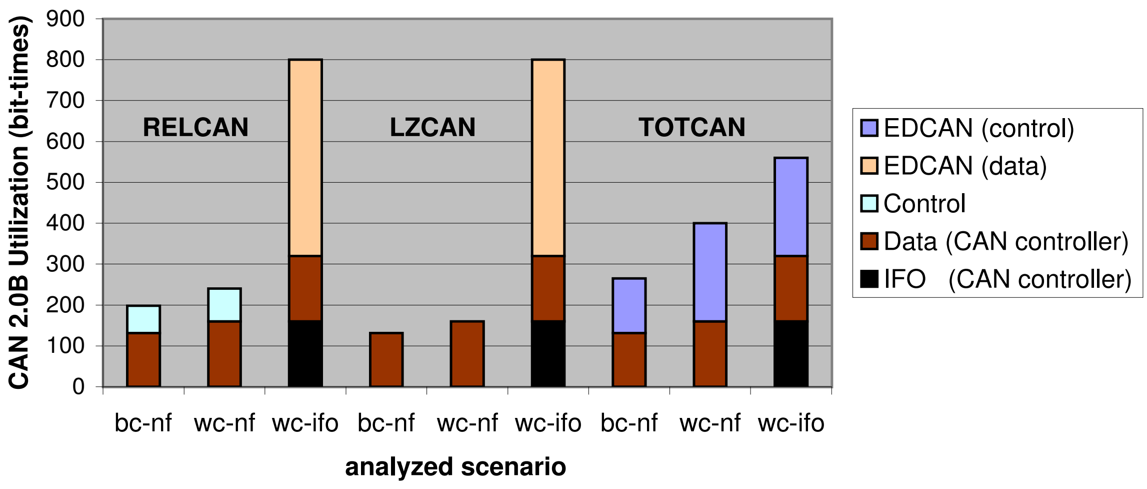


Figure 14: Utilization of CAN bandwidth by fault-tolerant broadcast protocols

7 Dimensioning of Protocol Timers

In this section, we discuss some specific guidelines for the dimensioning of the protocol timers to be used in the RELCAN and TOTCAN protocols. The dimensioning of protocol timeout values heavily depend on the scheduling policy of control traffic in regard to data messages. Under the assumption that EDCAN retransmissions and protocol control traffic have precedence over data messages, one use the following expression to estimate protocol timeout values:

$$T_{out} = T_{cdly} + \left\lceil \frac{T_{cdly}}{\mathcal{T}^{bc}_{mdata} \, . \, t_{bit}} \right\rceil . \, U^{wc \leftarrow nf}_{EDCAN}(control) \, . \, t_{bit} + f_r \, . \, U^{wc \leftarrow nf}_{EDCAN}(data) \, . \, t_{bit} + T_{td-prot} \qquad (16)$$

where $\lceil \ \rceil$ represents the *ceiling* function[12]. The different contributions in equation (16) account for:

- the protocol processing delay upper bound, in the issuing of a control message, T_{cdly};
- the second term depends on the maximum number of message transmit requests[13] which could be issued during the protocol processing period. The term accounts for the period required to the reliable broadcast of the corresponding control messages;
- the third term accounts for the time required to the transmission of f_r data messages, by the EDCAN protocol. This term results from the assumption that f_r nodes executing the RELCAN protocol may fail after transmitting the data message, but before the issuing of the corresponding CONFIRM message. Thus, recovery by the EDCAN protocol is needed;
- finally, $T_{td-prot}$ is the maximum transmission delay introduced by the traffic of other protocol components, such as the node failure detection and membership protocols discussed in [18].

[12]The *ceiling* function $\lceil x \rceil$ is defined as the smallest integer not smaller than x.
[13]The corresponding normalized frame duration is given by, $\mathcal{T}^{bc}_{mdata}$.

Note that the occurrence of network errors is not included in equation (16), being treated under a general inaccessibility model, as explained in [21].

Given the parameters: $f_r = 2$, $T_{cdly} = 80\mu s$ and $T_{td-prot} = 0\mu s$, one obtain a typical timeout value of $T_{out} = 1520\mu s$.

8 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. In addition, we have analyzed the performance of our protocol suite in terms of CAN bandwidth utilization and provided some guidelines for the dimensioning of protocol timeout values.

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 RJ 7203, IBM Research Division, San Jose, California, April 1990.

[4] J. Feio and J. Lageira. CANAMp - Controller Area Network Atomic Multicast protocol. IST Graduation Project Report, Advisors: J. Rufino and L. Rodrigues, Instituto Superior Técnico, Lisboa, Portugal, February 1998. (in portuguese).

[5] 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.

[6] H. Hilmer, H. Kocks, and E. Dittmar. A fault-tolerant architecture for large-scale distributed control systems. In *Proceedings of the 4th IFAC Workshop on Distributed Computer Control Systems*, pages 43–48, Seoul, Korea, July 1997. IFAC.

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

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

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

[10] 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.

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

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

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

[18] A. Rodrigues and J. Conceição. A CAN-based membership protocol. IST Graduation Project Report, Advisors: J. Rufino and L. Rodrigues, Instituto Superior Técnico, Lisboa, Portugal, September 1997. (in portuguese).

[19] 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. (also as INESC AR/66-92).

[20] J. Rufino and P. Veríssimo. A study on the inaccessibility characteristics of the Controller Area Network. In *Proceedings of the 2nd International CAN Conference*, pages 7.12–7.21, London, England, October 1995. CiA.

[21] J. Rufino and P. Veríssimo. Hard real-time operation of CAN. Technical Report CSTC RT-97-02, Centro de Sistemas Telemáticos e Computacionais do Instituto Superior Técnico, Lisboa, Portugal, January 1997.

[22] José Rufino. An overview of the Controller Area Network. In *Proceedings of the CiA Forum CAN for Newcomers*, Braga, Portugal, January 1997. CiA.

[23] 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.

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

[25] P. Veríssimo and H. Kopetz. Design of real-time systems. In S. Mullender, editor, *Distributed Systems*, ACM-Press, chapter 19, pages 491–536. Addison-Wesley, 2nd edition, 1993.

[26] 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.

[27] 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.