Timely use of the CAN Protocol in Critical Hard Real-time Systems with Faults

Ian Broster and Alan Burns
Real-Time Systems Research Group,
Department of Computer Science, University of York, UK
{ianb,burns}@cs.york.ac.uk

Abstract

The presence of network errors such as electrical interference affects the timing properties of a CAN (Controller Area Network) bus. In hard real-time systems it is often better to not receive a message than to receive it too late. Aborting late messages is a form of real-time error confinement which prevents late messages affecting the timeliness of other messages and processes. This can be used to help guarantee hard real-time performance in a distributed system using CAN in the presence of unbounded network errors.

1. Introduction

CAN [3, 6] is a bus protocol with many desirable properties for embedded and real-time systems. It is a priority based protocol where collisions are avoided by using priorities for bus arbitration. Simple worst case timing analysis is straightforward, based on process scheduling. However, in the presence of unpredictable network errors, no timing guarantees can be made.

The context of this work is high integrity, static, hard real-time systems such aircraft engine controllers. Currently, CAN is not considered reliable or predictable enough for certification of a system using it, current work is looking at how CAN may be used in such a system. This paper considers the hard real-time properties of CAN in the presence of unpredictable and unbounded network errors.

A CAN based protocol will be developed which *under fault conditions* sacrifices reliability (how many messages get through) for timeliness (any messages that arrive will arrive on time). We justify this approach by considering that a late message has no value; in a hard real-time system if a message does not arrive on time then it is considered an error and error recovery operations are performed.

2. Controller Area Network Bus

Worst case analysis of CAN with no network errors is straightforward. If the messages on the bus are scheduled using fixed priorities (as originally intended) using deadline monotonic priority assignment, then response time analysis in the style of non-preemptive process scheduling can be used to determine the worst case transmission time of each message and hence show whether or not messages are schedulable [16].

An accurate value for the worst case transmission time *in error free conditions* is given by equations (2) and (3)[1].

$$R_i = J_i + t_i \tag{2}$$

$$\begin{aligned} t_i \quad = \quad & \max_{\forall k \in lp(i)} (C_k) + S + C_i + \\ & \sum_{j \in hp(i)} \left\lceil \frac{t_i - C_i + J_j + \Delta}{T_j} \right\rceil (C_j + S) \end{aligned} \tag{3}$$

where
R_i is the worst case transmission time for a message i
J_i is a jitter component inherited from the response time of the sender process on the host CPU
C_i is the worst case time that message frame i will occupy the bus
$lp(i)$ is the set of messages of lower priority than i
S is the time to transmit an inter-frame space, 3 bit times
T_j is the period of message j
$hp(i)$ is the set of messages with a higher priority than i
Δ is one bit time

Equation (3) can be solved iteratively by forming a recurrence relation with $t_i^0 = C_i$ which terminates when $t_i^{n+1} = t_i^n$ or fails when $t_i^{n+1} > D_i$ where D_i is the deadline of message i and $D_i \leq T_i$.

[1] This analysis is slightly different in form to the analysis by Tindell but can be shown to be exactly equivalent to it by using the substitution

$$t_i = w_i + C_i \tag{1}$$

and noting that Tindell includes the inter-frame space, S, in his C.

The CAN protocol takes care of detection of errors. In this paper, it is assumed that the protocol detects all errors that occur. It is possible (but unlikely) that an error can go undetected by the CRC for example, but that only occurs in an extreme case which we consider to be outside the fault model.

CAN also attempts to provide atomic broadcast (either all nodes receive a given message or none do), however, analysis has showed that this is not necessarily the case in the presence of faults. This problem and solutions are considered elsewhere [11, 15] and for the purposes of this paper, it is assumed that CAN does provide atomic broadcast.

2.1. Unpredictability of CAN

In this paper, only network-born errors are considered; it is assumed that all *nodes* are well behaved. A network error occurs when one or more nodes read a bit on the bus as a different value to the value that the transmitter node transmitted (except where a node transmits a dominant bit to overwrite a recessive bit in accordance with the CAN protocol). Network errors may be caused by electrical interference or physical faults such as a poor connection. It *is* possible for different nodes on the bus to simultaneously read different values from the bus.

Errors on the bus can affect the timeliness of messages of the bus in many ways, consider the following scenarios:

- An error during a *data frame* will cause an *error frame* to be transmitted which aborts the data frame (which is queued to be retransmitted later). An error frame normally lasts for a time E. After the error frame, an inter-frame space occurs and normal bus arbitration continues. Therefore the error may introduce an additional delay to higher priority messages on the bus of up to E and to the message itself and lower priority messages on the bus of up to $C + E + S$.

  However, if a further error occurs when the message frame is being *retransmitted* then messages may be delayed by up to a further $C + E + S$. This situation, where the response time of messages is dependent on the number of errors, is not acceptable in many hard real-time systems.

- An error *during an error frame* or overload frame may cause the error frame to be retransmitted, introducing a delay on the bus of up to $E + S$, *i.e. further errors may lead to the extension of the error frame*. The only *bound* to the delay on the bus is limited by the nodes turning off automatically if they reach their maximum error counts.

- An error during idle time will cause all nodes to begin to listen for a message. When none arrives (since the start of frame signal bit was erroneous) then an error frame is generated, leading to a delay of at least $E + S$.

- A continuous block of severe interference prevents any node from successfully writing to the bus for the duration of the interference, Z. All messages are therefore delayed for up to $Z + C + E + S$.

A more complete taxonomy of CAN errors is given by [14] where they provide worst case scenarios for different types and numbers of errors. They note that it is possible for each single bit error to introduce a delay of 155 bit times, or (in extreme cases) 195 bit times. The only bound on the delays due to errors is when the CAN error confinement mechanisms automatically disconnect a node from the bus after a certain number of errors.

Early worst case response time analysis was based on the assumption that all the messages and the bus are 'well behaved', i.e. messages appear on the bus no more than once, and that no spurious messages are created[16]. The presence of such messages manifests itself as interference to other messages which can be included in the analysis if it can be bounded and then bounded low enough to be still practical. For example, we could bound the number of retransmissions, or even disallow retransmissions altogether[4]. However it is hard to bound the extension of error frames and errors in the inter-frame space, these errors are almost unavoidable. Clearly, these errors violate assumptions in any simple fixed priority timing analysis that is done on the system. In particular, extra frames are introduced to the bus which the timing analysis may not have included.

A simple analysis of CAN in the presence of faults is also provided by Tindell *et al*[16]. They consider a statistical error model for network behaviour and calculate worst case transmission times based on the predicted errors. The model may be a good enough guide for soft real-time systems, but it is fairly simplistic and unlikely to accurately describe errors. Furthermore, the research assumes that all errors occur in the data frames, whereas it is possible (and even likely) that errors will affect error frames, inter-frame spaces and bus idle times.

Punnekkat *et al*[12] extend Tindell's analysis to provide a more general model of network interference based on multiple sources of interference and considering errors which can occur outside of data frames. Pinho *et al*[9] also model the CAN bus under errors and provide a clear example of worst case analysis in the presence of network errors.

These error models may be used to cope with errors, but have two major drawbacks.

- Firstly, there is always the risk that the errors occurring at run time will exceed the errors predicted by the analysis. Accurate prediction of network errors is a difficult task. Work such as [12] and [9] is useful for modelling errors, but can never guarantee that the system will function correctly because we can never guarantee that at run-time, such errors will be within the error model. In unpredictable environments (such as an aeroplane, which may travel close to violent electrical storms) the error assumptions may be broken and we are left with an unpredictable system.

- Secondly, the more network errors that we set out to tolerate based on a predicted fault model, the less bandwidth is available for the 'normal' conditions where faults are rare. Usually, it is not possible to reserve sufficient bandwidth for the worst case scenario because the worst case scenario may be so bad that there would be no bandwidth left for any messages. Pinho[9] using the SAE benchmark[8, 16] notes that even with 1 network error per 100ms, it is not possible to guarantee the deadlines of all messages without increasing the network data rate.

Hence accurate analysis and modelling network errors is a step towards reliability, but it cannot justify the use of CAN in high integrity systems.

In this paper, errors are considered to be unpredictable and we will not rely on the distribution and nature of errors.

3. Unreliable Delivery of Late Messages

In a critical system, it is usually necessary to consider the consequences of not receiving a message (for example if the sending process fails, or if complete bus communication is lost). It is usual to write code to sensibly handle losing a message. A lost message may be considered fatal or not. This paper proposes that in the general case, losing a message should not be considered fatal and the system should continue to function as normally as possible without the message. For example, control laws are generally able to stabilise easily if a few data values are lost. This is used to develop a timely protocol as follows.

As no assumptions are made on the distribution and nature of network errors, no guarantees can be made that any messages will ever arrive. Therefore we must relax the requirement for messages to arrive (under error conditions). Since messages need not arrive, they need not be transmitted, so we can sacrifice a message which *would arrive late due to errors* in order to provide bandwidth to soak up the delay. A lost message can be considered to be a form of *real-time error confinement*.

Figure 1 shows an example of how the protocol can ensure the deadlines of other messages on the bus after a continuous block of interference. Figure 1a shows a typical scenario where messages are scheduled in priority order. In 1b, a burst of network interference prevents any message from appearing correctly on the bus for some time, all messages are delayed and arrive after their predicted worst case response time. Figure 1c shows how the protocol suggested in the paper deals with errors: messages 1 and 2 cannot possibly arrive before **D**, therefore they are abandoned, allowing all other messages to arrive in time.

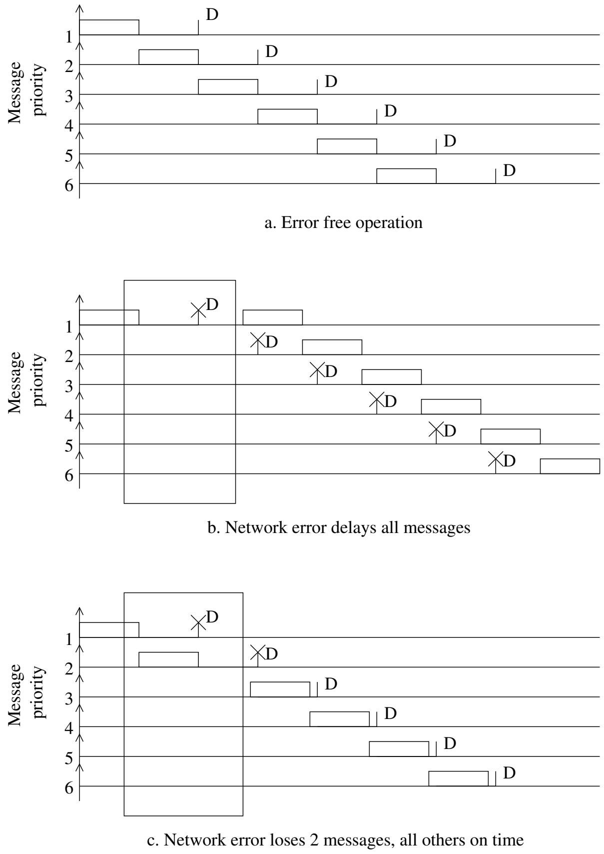

Figure 1. Example of losing messages in order to maintain other deadlines

For purely periodic messages, we know *a priori*, the worst case time that each message is created by the application and the worst case transmission time. Therefore the absolute time that each message should arrive by is *common knowledge* [10] and is known to all nodes *a priori*.

If we define the deadline of a message to be its calculated worst case delivery time then *if a message cannot arrive by its deadline then it should not be sent* and receivers only need to listen until the deadline.

In other words, we can do worst case response time analysis to determine the latest time that we expect each message to arrive (based on some fault model): the k^{th} message of type i begins transmission no later than $L_{i,k}$. Receivers only listen for the start of a message until time $L_{i,k}$. The sender is then responsible for never attempting to send a message if it cannot arrive on time.

The latest time that a message can begin transmission, $L_{i,k}$, is

$$L_{i,k} = O_i + k.T_i + R_i - C_i \quad (4)$$

where

where O_i is the initial offset of the first message in the stream.

Until this time, the message transmission protocol continues exactly as for normal CAN, including re-sending messages when errors occur.

This of course assumes, idealised clocks/global time and periodic messages, properties that are unlikely to exist in practical systems. These issues are addressed later.

4. Properties

Never attempting to send a message if it cannot get there in time will provide a service where

- messages do not arrive late;
- a late message cannot adversely affect the timing of any further messages;
- both the transmitter and the receivers know whether or not any given message has been sent;
- the point at which all nodes know whether or not any given message will arrive is *common knowledge*, $L_i + C_i$ (or possibly earlier, depending on implementation);
- timeliness is independent of errors;
- reliability of delivery is dependent on the nature of the errors and the amount of spare bandwidth reserved for errors.

This requires the application designer to carefully consider the failure actions of a message not being delivered, but this is a good thing and is helped by the fact that there is always a known point in time where we know that a message is definitely not going to arrive.

5. Deadline Calculation

Message deadlines are defined to be $O_i + k.T_i + R_i$ where R_i is the worst case transmission time of the message as determined by worst case response time analysis. It is useful to use a response time analysis based on some predicted error model, such as described by [12] so that R_i includes some slacktime for errors.

The general form for worst case response time including errors, based on equation (3) and work by [16], is:

$$\begin{aligned} t_i \;=\;& \max_{\forall k \in lp(i)} (C_k) + S + C_i + \\ & \sum_{j \in hp(i)} \left\lceil \frac{t_i - C_i + J_i + \Delta}{T_j} \right\rceil (C_j + S) + E(t_i) \quad (5) \end{aligned}$$

where

E(t) is a function which provides the expected upper bound on the overheads due to error recovery that could occur in the interval t.

Statistical error models were discussed in section 2.1 and can be used to guarantee 'normal' performance by using time redundancy to tolerate a specified level of errors. In this case, the latest send time acts as a *run-time guard to protect against the errors exceeding the error model.*

5.1. Flexible Deadlines

A certain amount of flexibility is available in deciding L_i. It is possible to assign more time for certain messages to get through where a message is more important. Consider a periodic sensor reading, if a process misses one reading then it may be appropriate to assume the previous value until a message actually arrives; control laws are generally able to cope easily if a data value is lost. A mode change message however should not normally be lost since it is expressing a *change*.

The deadlines can be manually adjusted to give more time for more important messages. In the case of a mode change message, the worst case send time could be set to infinity so that the message will eventually get through if it can, even if it means losing other messages.

It should be noted that this is a form of error confinement when network errors exceed the predicted maximum level. Under normal conditions, all messages are guaranteed to arrive and be timely. If all messages have their deadlines set to infinity then this is equivalent to the normal CAN protocol.

This gives two general classes of messages, which can run side by side on the same network.

A unreliable delivery, but guaranteed worst case delivery time;

B (almost) reliable delivery, but may arrive late (normal CAN).

Note however that the use of class **B** messages at higher priorities than class **A** messages may lead to more class **A** messages being lost (in the presence of extreme errors).

6. Implementation

Implementation of the protocol does require modification of the CAN protocol, but changes are minimal and can be seen as extensions, rather than deviations. In practice, it may require custom implementation of a CAN controller, (or at least the software component), but in critical systems this may well be done as a matter of course. The availability of dedicated single chip computers and FPGA hardware makes implementing a custom controller feasible. The protocol might also be implemented using certain controller ICs, for example the Intel 8257[5]. However the Motorola TouCAN[7] controller for example, is not capable of supporting the protocol because of its internal buffer arrangement.

Each message passed from the application to the CAN controller for transmission also has a *latest send time* parameter. Messages are queued and the highest priority message begins transmission when the bus is idle as for normal CAN, except that the *latest send times* are compared to the local clock; frames are rejected if they are too late.

A transmitter implementation must ensure that it can test and reject all messages in its buffers as soon as they expire in order to ensure that late messages are not erroneously transmitted and that there is no possibility of buffer overflow. A method must also be provided to inform the application that a message has failed to be sent in time.

Each receiving channel must specify the associated timeout parameter for each message. After the time-out, it is important that the application is aware whether or not a message was received.

7. Clock Synchronisation

If this sort of *a priori* knowledge about message send times is to be used then there must be a defined synchronisation of the clocks on different processors.

It is the responsibility of the sender to never send a message after the send deadline has expired, however if a receiver's clock is faster than the sender's then the transmitter may send the message too late for the receiver to receive it. Therefore the receiver will reject the message while others with slower clocks may receive it as in figure 2. This may or may not be a problem for a particular system, but it does break the atomic broadcast property of CAN.

Various algorithms and protocols exist to synchronise clocks on a distributed system, for example [13] provide a fault tolerant clock synchronisation protocol specifically for CAN. The bandwidth overhead limits the precision of the synchronisation. By performing the synchronisation every 45s, a precision of $\epsilon = 100\mu$s can typically be obtained (*i.e.* no two clocks may differ by more than 100μs). This is not accurate enough for synchronisation of worst case send times (consider that a message takes approximately 150μs). However, if we assume approximately synchronised clocks (bounded to within $\epsilon = 100\mu$s) then there are three ways to maintain atomic broadcast.

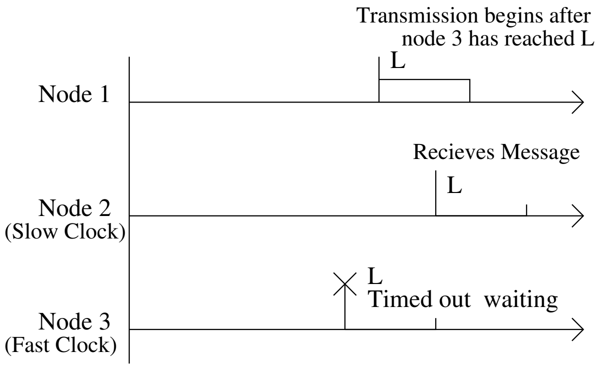

Figure 2. Effect of Clock Skew

One way is to consider that the aim of the latest send time protocol is to ensure that a message never appears on the bus after real time L. Therefore a sender must not begin to transmit a frame after time $L_{local} - \epsilon$, so a receiver should listen for the start of the message until time L_{local}. However, since ϵ is large (100μs is of a similar magnitude to the length of a message on the bus), assuming the worst case skew like this could easily lead to messages with short response times never being transmitted even under normal conditions!

Alternatively, the *receivers* could make the adjustment for bounded clock skew: all receivers accept messages until time $L_{local} + \epsilon$. Unfortunately, this can impose more traffic on the bus than was intended, however the extra traffic is small and may only occur in already faulty conditions. Applications are not otherwise affected. This is a practical way of dealing with clock skew.

The third method involves a further change to the protocol and provides a more robust and less pessimistic way of dealing with clock skew. If a receiver receives a message after the latest send time for the message has passed (according to its local clock) then it is an error. Errors can be signalled to all other nodes by transmitting a CAN error frame immediately after the complete arbitration field has been sent. Therefore all other nodes will reject the frame, as in figure 3.

The advantage of this mechanism is that the protocol still provides atomic broadcast even if the clock synchronisation fails to stay within the assumed precision. This is especially important since clock synchronisation relies on communication between the nodes, which is not guaranteed in the presence of faults. Some level of synchronisation is still required though as extreme differences in clock synchronisation will result in all messages being aborted.

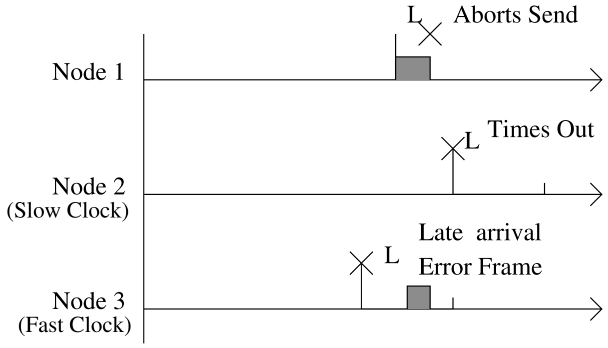

Figure 3. A Late Message Generates an Error so no nodes receive the message

8. Sporadic Messages

When dealing with periodic messages, a receiver always knows how long to wait for a periodic message because for a periodic message, the latest send time is common knowledge—it is known to all nodes *a priori*.

Sporadic messages do not have a known latest send time hence the receivers cannot know when to stop listening for a given message. There are at least two ways of dealing with sporadics. One way is to consider a sporadic message exactly like periodic message, (since there is already space on the bus reserved for it). *i.e.* the receiver listens periodically and when the message does not arrive within the deadline, the receiver concludes that the sporadic has not been sent. This solution is not satisfactory since there are problems with aborted messages if there is clock skew.

A second way of dealing with sporadic messages is to consider the nature of sporadic activity. Sporadics tend to occur infrequently and tend to contain data which indicates a *change*. For these messages, simply setting the deadline to infinity (class B message) may be more appropriate so that a sporadic message will never be lost due to timeout.

9. Simulation

A simulator for the protocol was constructed using the STRESS[1] simulator tool. The Society of Automotive Engineers (SAE) produced a benchmark[8] of messages that might be transmitted in a vehicle. Tindell's simplification[16] of the benchmark was used as the message set for the simulation using the (slow) data rate of 125kbit/s. At this speed, the bus utilisation is about 80%, which is fairly high and therefore provides a good basis for fault injection above any assumed error model.

The messages used for the simulation are the same as Tindell's except that a small jitter component is also used. There are 17 messages, with periods ranging from 5ms to 1s. Worst case response times (R_i) were calculated from equation (2) using the assumption that there are no errors on the bus. The messages are shown in table 1. Single bit network errors were injected into the system at random times (uniform distribution) during the simulations.

Table 1. CAN messages used for simulation

Pri	Bytes	C_i (ms)	T_i (ms)	J_i (ms)	D_i (ms)	R_i (ms)
17	1	0.480	1000	0.100	5	1.368
16	2	0.560	5	0.100	5	1.952
15	1	0.480	5	0.100	5	2.456
14	2	0.560	5	0.100	5	3.040
13	1	0.480	5	0.100	5	3.544
12	2	0.560	5	0.100	5	4.128
11	6	0.864	10	0.200	10	7.776
10	1	0.480	10	0.200	10	8.128
9	2	0.560	10	0.200	10	8.712
8	2	0.560	10	0.143	10	9.296
7	1	0.480	100	0.200	100	17.880
6	4	0.712	100	0.300	100	18.616
5	1	0.480	100	0.300	100	19.040
4	1	0.480	100	0.200	100	27.624
3	3	0.632	1000	0.400	1000	28.280
2	1	0.480	1000	0.300	1000	28.632
1	1	0.480	1000	0.300	1000	29.136

For occasional, single bit errors, the simulation shows that the pessimism introduced by the analysis is nearly always sufficient to prevent the errors delaying the messages beyond their calculated worst case transmission times.

For high error rates (using single bit errors), results are in table 2. The 'Lates' column shows the number of messages that arrive late due to the bit errors in normal CAN. The 'Aborts' column shows the number of messages that are not sent according to the protocol in section 3. The bit errors occurred at exactly the same times for both the normal CAN protocol and the Latest Send Time protocol simulations.

Table 2. Simulation results

Errors (per second)	Normal CAN **Late messages** (per second) with no aborts	LST Protocol **Aborts** (per second) none late
10	0.05	0.03
20	0.94	0.39
30	1.50	0.68
40	2.68	1.18

Messages queued per second = 1444

These results show that for high error rates, by not sending late messages, less than half of the late messages need to be aborted in order to make all the remaining messages arrive on time. This is a considerable increase in quality of service on CAN under extreme error conditions.

The messages that were aborted were not always the same priority: there was a wide distribution of aborted priorities. The worst affected messages were 5 and 9, however, the distribution varied for different error rates.

It is perhaps surprising that the number of aborted/late messages is so low: only 0.08% of messages aborted at the massive error rate of 40 errors per second. This is entirely due to the message set. The response time of all the messages includes a possible execution of the highest priority message. However, this message only has a period of 1000ms. Consider the second highest priority message, the highest priority message will only interfere for 1 transmission in 200 of this message. Coupled with the blocking and jitter components, there is considerable slack available before the latest send time. The same applies for the lower priority tasks. The only time that errors can make a message late is if there is sufficient blocking/interference that the response time is near the worst case response time. As Bernat[2] shows, the actual response times are rarely equal to the worst case.

It is worth noting that including the highest priority message in the analysis is almost the same as including an error term. If the highest priority message is removed from the simulation (and the response times lowered accordingly) then 40 errors per second gives averages of 23.6 late messages or 9.9 aborted messages per second. Again, to ensure the timeliness of all the remaining messages, less than half of the late messages needed to be aborted.

The authors acknowledge that the simulation results so far are not likely to be representative of a true system. In particular, only one message set has been simulated, the uniform distribution of errors may not be appropriate and no account is taken of clock drift or offsets. However, the results indicate that the protocol provides significant advantages over normal CAN. More detailed simulation results will be presented in a future work.

10. Conclusion

The protocol presented here is a slight extension of CAN which prevents network errors affecting the timeliness of messages by sacrificing reliability of delivery. In critical systems such as avionics systems, it can be used to systematically provide fail operational status with degraded performance in the presence of unbounded network errors.

While network-born errors are within the fault model assumed for worst case response time analysis, the protocol behaves exactly as normal CAN. However if errors exceed the fault model (introducing unanticipated delays) then late messages are not sent on the bus as a form of real-time error confinement. This allows the bus to quickly recover from errors. Simulation shows that the protocol is effective, results with the SAE benchmark indicate that by using the latest send time, less that half of the number of messages are aborted than would arrive late in normal CAN.

It is the sender's responsibility to never send a message if it cannot arrive on time; a *latest send time* is associated with each message, based on the predicted worst case response time under a given fault model. For periodic messages, this parameter is *common knowledge* to all nodes.

Messages queued for transmission may not necessarily arrive, but the sender and receiver always know whether or not a message has successfully been sent. They know this by a point in time which is known *a priori*. Therefore applications must be able to cope with lost messages. It is generally easier to reason about lost messages than it is about messages which may arrive at some unknown time later.

Normal CAN messages which do not have a timeout can also be transmitted simultaneously on the same bus. They will be reliably delivered but the timing behaviour of such messages in the presence of errors is as unpredictable as for normal CAN, (however, some speed-up might be expected if there are higher priority messages which do have a latest send time).

Further simulation of this protocol is planned, to measure its effectiveness. Ongoing and future work includes applying this protocol to different computational models and integration with various statistical fault models.

11. Acknowledgements

This work has been funded by EPSRC and Rolls-Royce PLC. The authors would also like to thank Luís Migel Pinho, Guillem Bernat and George Lima.

References

[1] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. STRESS: A simulator for hard real-time systems. *Software, Practice and Experience*, 24(6):543–564, June 1994.

[2] G. Bernat. *Specification and Analysis of Weakly Hard Real-Time Systems*. PhD thesis, Universitat de les Illes Balears, January 1998.

[3] Bosch, Postfach 50, D-700 Stuttgart 1. *CAN Specification*, version 2.0 edition, 1991.

[4] T. Fuhrer, B. Muller, W. Dieterle, F. Hartwich, R. Hugel, and M. Walther. Time triggered communication on CAN. techreport, Robert Bosch GmbH, http://www.can.bosch.com/, 2000.

[5] Intel. *Automotive Products*. Number 281792-007 in Automotive. Intel Literature, 1994.

[6] International Standards Organisation. *ISO 11898. Road Vehicles – Interchange of digital information – Controller area network (CAN) for high speed communication*, 1993.

[7] Motorola. *MC68336/376 Users Manual*. Motorola, 1996.

[8] S. of Automotive Engineers. Class C application requirement considerations. Technical Report J2056/1, SAE, 1993.

[9] L. M. Pinho, F. Vasques, and E. Tovar. Integrating inaccessibility in response time analysis of CAN networks. In *Workshop On Factory Communication Systems*, 2000.

[10] S. Poledna, A. Burns, A. Wellings, and P. Barrett. Replica determinism and flexible scheduling in hard real-time dependable systems. *IEEE Transactions on Computers*, 49(2):100–111, 2000.

[11] J. Proenza and J. Miro-Julia. Fixing the Atomic Broadcast impairments in the Controller Area Network (CAN) field-bus: the MajorCAN protocol. Technical Report 1-99, Universitat de les Illes Balears, Departament de Ciéncies Matemátiques i Informática, April 1999.

[12] S. Punnekkat, H. Hansson, and C. Norstrom. Response time analysis under errors for CAN. In *Proc. RTAS 2000*, pages 258–265, Washington DC, 2000. IEEE.

[13] L. Rodrigues, M. Guimarães, and J. Rufino. Fault-tolerant clock syncronization in CAN. In *Proceedings of the 19th Real-Time Systems Symposium*, pages 420–429. IEEE, December 1998.

[14] 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.

[15] J. Rufino, P. Veríssimo, G. Arroz, C. Almeida, and L. Rodrigues. Fault-tolerant broadcasts in CAN. In *Digest of Papers, The 28th International Symposium on Fault-Tolerant Computing Systems*, pages 150–159, Munich, Germany, June 1998. IEEE.

[16] K. Tindell, A. Burns, and A. J. Wellings. Calculating controller area network (CAN) message response times. *Control Engineering Practice*, 3(8):1163–1169, 1995.