Timing-Independent Safety on Top of CAN

George Lima* Alan Burns

Real-Time Systems Research Group
Department of Computer Science, University of York,
Heslington, York, England, YO1 5DD
{gmlima,burns}@cs.york.ac.uk

Abstract

We describe an approach to designing CAN-based distributed real-time systems so that safety is preserved regardless of timeliness. Our approach offers gains with respect to both fault tolerance and flexibility aspects and so it is attractive to support those systems that have critical tasks (e.g. control systems) and at the same time are connected to non-predictable networks (e.g. the Internet).

1. Introduction

The correctness of real-time systems is specified in terms of both safety and timeliness. The safety requirement has led to the use of distributed platforms to implement fault tolerance mechanisms. Making a system distributed consists of spreading the processes that carry out its computation across different machines linked to each other by means of a communication network. In turn, the timeliness requirement has made the *synchronous* model of computation a natural choice for implementing distributed real-time systems. According to this model *all* sent messages arrive within a known interval of time (communication is synchronous) and *all* computation is finished within a bounded time (processing is synchronous).

Assuming synchronous processing in real-time systems is reasonable since bounds on processing times can be derived by carrying out appropriate schedulability analysis. However, assuming synchronous communication may be a restriction. Indeed, the communication network is a critical component of distributed systems since it is a shared resource and is most subject to transient faults and overload conditions. As systems that are based on the synchronous model use the knowledge about the assumed bounds to guarantee safety, we say that they are *timing-dependent safe*. For example, if a message that is supposed to be received within a given interval of time does not arrive, the receiver process may conclude that the sender is faulty. However, if the message is just late, the result of the computation by the receiver may be inconsistent (i.e. unsafe).

In this work we demonstrate that it is possible to build timing-independent safe hard real-time systems on top of the Controller Area Network (CAN) [2]. CAN is a broadcast network that is widely used in the implementation of distributed hard real-time systems. The idea is to make co-operating processes *agree* on their computations by exchanging messages. As we will see, due to the message scheduling and error-recovery mechanisms of CAN this can be done straightforwardly. The benefits of our approach can be verified by considering a *semi-synchronous* model of computation based on CAN properties. This model relaxes the communication synchronism until a point beyond which the system's timeliness would be compromised. Moreover, the described approach is particularly interesting due to its flexibility since by using it systems may tolerate unpredictable behaviour caused by overload or faulty scenarios in some nodes of the system. These characteristics make the approach attractive, mainly for supporting those systems that have critical tasks (e.g. control systems) and at the same time are connected to non-predictable networks (e.g. the Internet).

2. Model of Computation

In this section we define a semi-synchronous model of computation having CAN as the communication network. As our main goal is to show that one can design timing-independent safe protocols using CAN, this

*Supported by CAPES/Brazil under grant: BEX1438/98-0.

model allows message timing/omission faults to take place.

2.1. Processing Model

We consider systems made of geographically distributed nodes, which are fully connected to each other by means of a CAN-based communication network. Each process is allocated to a node. Processes communicate to each other only by exchanging messages across the network. Processes may only fail by crashing. Correct processes are those that never crash. If a process crashes at time t, it stops both sending and receiving messages indefinitely from time t (i.e. crashed processes do not recover).[1]

Processes may perform local and non-local tasks. Local tasks are those that do not depend on the communication network (i.e. message-send or message-receive events). We assume synchronous processing, by which we mean that the worst-case response times of local tasks are known. This can be guaranteed in practice by applying real-time scheduling techniques [1].

2.2. Communication Model

The assumed communication network is typified by the Controller Area Network (CAN) [2]. Due to its deterministic collision resolution based on priorities and the built-in error-recovery schemes, CAN is widely used for supporting hard real-time systems. Indeed, CAN provides a very resilient error-detection and recovery mechanism that can handle most failures consistently. Hence, we assume that messages cannot be either arbitrarily created or corrupted by the network.

Errors on CAN are detected by the transmitter or receiver nodes while monitoring the transmission of messages on a bit-by-bit base. If a message is detected corrupted, it is scheduled for re-transmission according to its priority. Although this error recovery and the message scheduling schemes used in CAN provide a high degree of reliability and predictability, they may lead to some inconsistency is some specific cases. In fact, it has been shown that in some scenarios (involving the last but one bit of the transmitted message) a set of receivers can accept a given transmitted message while others reject it [4, 3]. In this situation three inconsistent scenarios may take place: (a) if the transmitter crashes after the detection of the error and before the re-transmission, its transmitted message will be inconsistently omitted at some nodes; (b) if the transmitter does not crash, it re-transmits the message and so some receivers will receive the message more than once; and (c) this scenario has the same effect as (a) and happens if the transmitter does not crash but it does not detect the faulty transmission [3]. Notice that scenario (a) is associated to process crashes while (b) and (c) are due to the way error-recovery is carried out in CAN. According to some simulations [4, 3], the probability of occurrence for these inconsistent scenarios varies between 8.80×10^{-3} and 3.96×10^{-8} per hour. Although these scenarios are unlikely, they have to be considered when dealing with critical applications.

Based on the characteristics of CAN described above, we assume that messages may be dropped by the network (due to the inconsistent scenarios) or arbitrarily delayed by the network (due to CAN message scheduling mechanism). However, in the absence of the inconsistent scenarios CAN provides what is called *atomic broadcast* [4, 3]: transmitted messages are totally ordered and received either by all correct processes or by none. As we will see in the next section, this property can be used to design timing-independent safe systems.

[1]A protocol for re-introducing recovered processes could be added but this is beyond the scope of this work.

3. Timing-Independent Safety

Ideally, systems must be safe regardless of the present level of synchronism. This means that processes must only take decisions during their computation based on their view about the whole system, instead of on the time. It is clear that such an approach does not work in general. The following example illustrates this.

Example 3.1. *Two processes, p and q say, are cooperating throughout their execution. Suppose a moment during the execution of p when it is waiting for a message from q in order to take a decision in accordance with q's computation. As process p eventually has to make progress (i.e. it has to meet deadlines), it cannot wait forever (neither can q). Hence, there may be a moment at which p has to make progress regardless of q's message. If some fault prevents q's message from being delivered at p, p may violate safety. If q is crashed, though, p is free to take its own decision.*

The example above illustrates a dilemma between safety and timeliness: favouring one may compromise the other. Notice that the reason behind this dilemma is that it is impossible for processes to have reliable information about failures of remote processes if the synchronous model is not assumed. A tradeoff between safety and timeliness, though, can be achieved by considering other kinds of synchronism. For instance, if the system provides atomic broadcast, it is possible to implement the system so that no inconsistent decision can be taken. Making use of this atomic broadcast primitive, our illustrative example can have the following solution. After waiting for the message from q, p atomically broadcasts a message to pass on the decision on its computation. After receiving its own message, p knows that if q is not faulty, it will also receive the same message and in the same order so that q will also make progress according to p's message. Therefore, both processes will be safe regardless of the time messages take to be delivered.

As we have seen, however, CAN does not provide perfect atomic broadcast due to some inconsistent scenarios. Hence some extra effort has to be made. Indeed, our approach to building timing-independent safe systems on top of CAN requires that co-operating processes execute an agreement phase during their computation to ensure safety. During this phase processes exchange messages in order to reach the same view about the system despite scenarios (a), (b) and (c). Notice that by assumption (section 2.2) if a transmitted message is received by a process and scenarios (a), (b) and (c) do not take place, then this message is received by all correct processes. In order to take these scenarios into account we assume that no more than f inconsistent scenarios may take place during the agreement phase. As we have seen, the probability of these scenarios taking place, although not negligible, is not significantly high. Thus, one can choose a value for f that is suitable for the targeted system. In the next sections we discuss the safety, timeliness and flexibility aspects.

3.1. Ensuring Safety

Assume that there is up to f inconsistent scenarios. If any message is received by some process and scenarios (a), (b) and (c) do not take place, the message is also received by all correct processes (by the CAN properties). As a process that receives a message does not know whether or not other processes also received this message, it has to re-transmit the message f times. After the reception of the last re-transmission the process knows that all correct processes also received the message (at least once). Hence, up to $f + 1$ transmissions by each process are necessary to guarantee the reception of the message by all correct processes. This is the basic idea of the agreement phase and is described in the algorithm of figure 1. In other words, lines 1-7 of the algorithm can be inserted into the normal code of any critical task of processes that co-operate.

The agreement phase consists of up to $f+1$ rounds of

```
        /* ... normal computation ... */
        /* m contains the result of the computation */
(1)     m.k ← 0
(2)     **while** m.k < $f + 1$ **do**
(3)         **broadcast**(m)
(4)         **wait for** [ receive m′ such that m′.k ≥ m.k ]
(5)         get the first received m′ such that m′.k ≥ m.k
(6)         m ← m′; m.k ← m′.k + 1
(7)     **endwhile** /* ... processes agree on m ... */
```

Figure 1. The agreement phase algorithm

message exchanges. Any exchanged message is tagged with an integer counter that is used to keep track of the number of rounds seen by the processes. At the end of this phase, all correct processes agree on the same message so that they can make progress based on the same view of the system. Notice that there is no reference to time in the agreement phase. Safety is ensured just by message exchanging. For the sake of illustration, consider example 3.1 and suppose that $f = 1$. Two possible executions of p and q are shown in figure 2. The numbers along the time line represent the values of m.k at each process. In A, process p does not receive any message from q. This may be due to a crash fault or asynchrony between the executions of p and q, say. Then, p sends its message twice during its agreement phase. If q is not crashed, it receives at least one transmission consistently. During its execution q eventually picks up the message sent by p and takes the decision on its computation accordingly in order not to violate safety. In execution B, an inconsistent scenario takes place. The message from q is not received by p but is received by q. After the second transmission, though, both p and q choose the same message (q's message).

It is important to emphasise that this simple agreement protocol does not guarantee atomic broadcast. Messages are still being delivered out of order. This agreement phase is enough, however, to ensure safety regardless of time. Indeed, in the example B p gives up its own message to accept q's.

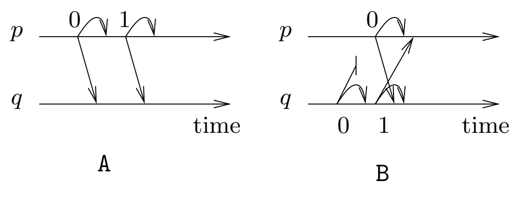

Figure 2. Illustration of the agreement phase

3.2. Ensuring Timeliness

A real-time system is made of several services, which may have different priorities. These priorities are usually assigned according to the urgency of execution. Thus, the higher the priority of the service, the higher the priority of the messages sent by its processes. A system characterised in such a way makes the analysis of its feasibility straightforward. For example, one can carry out well known schedulability analysis for this purpose (e.g. [1, 5]). Indeed, the approach discussed in this work does not require any special mechanism to check the system timeliness.

For example, consider a distributed system made of three services, `H` (highest priority), `M` (medium priority) and `L` (lowest priority). The worst-case scenario in each node is determined (as usual) when all tasks of the node are released at the same time. In this situation, messages sent by service `M` and `L` may be transmitted only after messages sent by `H`. Now, in order to illustrate the strength of our approach assume that in the absence of the inconsistent scenarios just the highest priority message arrives at all its destinations within a known bounded delay. This assumption may represent a given worst-case scenario due to possible transient faults in the network, say. Even with this low level of synchronism in the communication network one is assured that safety is not violated. Yet, to derive timeliness, the message scheduling provided by CAN guarantees that after `H` finishes sending messages, `M` can make progress and so on. If all three services meet their deadlines, we say that timeliness is ensured.

Clearly, some considerations regarding the application has to be taken into account when analysing the system timeliness. For example, if it is known that processes p and q, say, of a given service start executing their tasks approximately at the same time in different nodes, some tightness guarantee between their computation can be derived. However, it is important to emphasise that the guarantee of safety does not need any reference to time whatsoever. In other words, the dynamics of the system dictate the time spent by its computation and can be derived by analysing the system after knowing that its safety is not violated.

3.3 The Flexibility Aspect

It is important to note that considering safety and timeliness independently brings flexibility for systems with respect to both the communication synchronism and the processing synchronism. Consider a typical application which has hard real-time tasks distributed across a set of nodes, say. It would be useful if information about the system could be remotely monitored. Doing this using fieldbus networks (such as CAN) may not be viable due to their low bandwidth. Hence, the monitoring tasks (well modelled as soft tasks) might use non-predictable communication networks (such as the Internet), which in turn might overload the nodes in which such tasks run (since TCP connections may introduce unpredictable delays). If the system is designed in line with the timing-independent safety approach, one can avoid the unpredictable behaviour of the monitoring system (soft tasks) interfering in the critical tasks. For instance, even if one of the processes in figure 2 is subject to these overload conditions (since it may be running in the same node as the monitoring system), the monitoring system will never present inconsistent information.

4. Conclusion

The problem of designing timing-independent safe real-time systems has been addressed. As we have seen, CAN offers powerful properties that can be used to achieve such an objective. In general, the approach discussed in this work is very attractive due to its simplicity and can be used to enhance both the fault tolerance and the flexibility of real-time systems.

References

[1] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages*. Addison-Wesley, 3nd edition, 2001.

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

[3] J. Proenza and J. Miro-Julia. MajorCAN: A Modification to the Controller Area Network Protocol to Achieve Atomic Broadcast. In *IEEE Int'l Workshop on Group Communication and Computations (IWGCC 2000). Taipei, Taiwan*, Apr. 2000.

[4] J. Rufino, P. Veríssimo, G. Arroz, C. Almeida, and L. Rodrigues. Fault-tolerant broadcasts in CAN. In *Symposium on Fault-Tolerant Computing*, pages 150–159, 1998.

[5] K. Tindell, A. Burns, and A. Wellings. “Analysis of Hard Real-Time Communications”. *Real-Time Systems*, 9(2), Sept. 1995.