A Consensus Protocol for CAN-Based Systems

G. M. A. Lima*
Laboratório de Sistemas Distribuídos (LaSiD)
Departamento de Ciência da Computação
Universidade Federal da Bahia, Ondina
Salvador, Bahia, 40.170-110, Brazil
gmlima@ufba.br

A. Burns
Real-Time Systems Research Group
Department of Computer Science
University of York, Heslington
York, YO1 5DD, UK
burns@cs.york.ac.uk

Abstract

Consensus is known to be a fundamental problem in fault-tolerant distributed systems. Solving this problem provides the means for distributed processes to agree on a single value. This, however, requires extra communication efforts. For some real-time communication networks such efforts may have undesirable performance implications due to their limited bandwidth. This is certainly the case with the Controller Area Network (CAN), which is widely used to support real-time systems. This paper shows how some underlying properties of CAN can be used to solve the consensus problem. The proposed consensus protocol tolerates the maximum number of process crashes, is efficient and flexible. The described solution is proved correct, its complexity is analysed and its performance is evaluated by simulation.

1. Introduction

Hard real-time systems are usually associated with critical activities, where failures may have catastrophic consequences. This requirement has motivated the use of distributed systems technology for implementing fault-tolerant hard real-time systems [12, 13, 14, 19, 20]. Indeed, having the computation of a system distributed onto several autonomous nodes (linked by a communication network) may make the system more resilient to faults.

Despite the advantages of using distributed systems for fault tolerance purposes, the costs of retaining consistency in distributed computation cannot be neglected. These costs are usually due to the need to exchange information among the system nodes in order to establish some sort of agreement on their computation. For example, suppose that a node fails at a given moment. In some cases, carrying out agreement on a node identifier, in order to choose which node will take over, may be necessary. If the service executed at the faulty node is replicated, agreement on the replicated results may have to be provided. In any case, extra communication effort, by means of some protocol, is necessary.

Sometimes, this extra communication cost implies significant performance degradation. In particular this is true for some real-time communication networks with limited bandwidth. An example of such a network is CAN [1, 10] (Controller Area Network), whose maximum transmission rate is 1 Mbps. CAN is a widely used network for supporting several types of hard real-time systems, although it was originally developed for automotive systems. Thus, dealing with extra communication costs due to the needs for agreement in CAN has practical relevance for the design of hard real-time systems.

In this paper the problem of providing agreement in CAN-based systems is addressed. In particular, consensus, a classical agreement problem, is solved. The consensus problem can be informally specified as follows: a group of distributed processes propose their initial (possibly different) values and all correct processes have to agree, despite faults, on a single value. The special interest in this problem comes from the fact that consensus is known to be fundamental in fault-tolerant distributed systems. Indeed, it has been shown that solutions to the consensus problem can be used as a basic building block to solve other agreement problems [2, 8, 9, 25].

The proposed protocol to solve the consensus problem is particularly interesting because it takes advantage of some properties of CAN for the sake of *fault tolerance*, *efficiency* and *flexibility*. The protocol is tolerant to the maximum number of process crashes and can deal with some inconsistent scenarios that may be present in CAN [22, 24]. Its efficiency is due to the low number of communication steps and messages that are needed to achieve consensus. Its flexibility comes from the lack of timing assumptions on the

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

system. For example, it will be shown that consensus is achieved regardless of: the time processes take to execute their actions; the time they start executing the protocol; or the time their messages take to be successfully delivered. In other words, the proposed protocol compares favourably with other approaches used to support real-time systems (e.g. [13, 16, 18, 23]) which rely on the assumption of synchronism in distributed processes and network communication.

The proposed protocol is also adjustable in terms of two parameters, Δ and θ. These parameters regulate, respectively, the time processes may wait for messages and how often each process broadcasts messages. The importance of these regulating parameters is that one can adjust the protocol to specific applications so that a trade-off between waiting time and number of messages can be achieved. Simulation results show how the protocol behaves for several values of Δ and θ.

The remainder of this paper is organised as follows. The next section presents the model of computation assumed in this paper. In section 3 the consensus problem is more precisely defined. This section also describes the limitations of solving consensus using CAN. The proposed consensus protocol is explained in section 4 while its correctness and complexity are given in section 5. The protocol is evaluated by simulation in section 6. Finally, section 7 gives the concluding comments.

2. Computational Model

The assumed system consists of a set of distributed nodes linked to each other by means of a CAN-based communication network. The computation in a node is carried out by processes, which are statically allocated to the nodes. Processes exchange information between each other only by broadcasting/receiving messages across the network.[1] It is assumed that a node may crash at any time during the lifetime of the system. If a node crashes, all processes allocated to it also crash. Processes only fail by crashing. If a process crashes at time t, it stops sending messages indefinitely from t. See Broster and Burns [3] for a discussion of how to achieve this property.

As the proposed consensus protocol takes advantages of some properties of CAN, it is worth highlighting its main characteristics. However, only relevant points, necessary for the description of the consensus protocol, are given. Interested readers can refer to specialised publications [1, 10] for more details.

1 Clearly, processes in the same node may exchange information by sharing resources. However, this paper in only concerned with communication that involves distributed processes and so internal communication is not considered.

The bus arbitration scheme used in CAN follows a policy called *carrier sense multi-access with deterministic collision resolution* (CSMA/DCR). In short, the bus arbitration in CAN is a bit-wise protocol. Either a *dominant* or a *recessive* bit can be transmitted at a time on the CAN bus. If two nodes simultaneously transmit two different bits, the resulting bus value will be the dominant one. While a message is being transmitted, both the sender and receivers monitor the bus. Also, each message has a unique identifier (priority). While transmitting the message identifier, for every bit, if the transmitted bit is recessive and a dominant value is monitored, the sender stops transmitting and starts receiving incoming data. Should a sender stop transmitting a message due to either this arbitration scheme or the detection of a transmission error, its message is automatically scheduled for retransmission.

As far as error detection and recovery are concerned, CAN provides an elaborated mechanism to handle transmission errors. If no errors are detected up to the last bit of a transmitted message, receivers can accept it. Otherwise, the message is rejected. Errors can be detected by the transmitter or by some receiver node. In case of faulty transmission, the message is automatically scheduled for retransmission according to its priority. Both the bus-arbitration policy and the error-handling mechanisms used in CAN make it a very reliable and predictable network. Based on this observation, we assume that valid messages cannot be either arbitrarily created or corrupted by the network.

Unfortunately, there are some specific fault scenarios that lead to inconsistencies in CAN. 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 the message while others reject it [22, 24]. In this situation the following may happen:

IS1 Inconsistent message omission due to crash failures. If the sender node crashes after the detection of the error and before the retransmission, its transmitted message will be inconsistently omitted at some nodes.

IS2 Inconsistent message duplication. If the sender does not crash, it retransmits the message and so some receivers may receive the message more than once.

IS3 Inconsistent message omission due to undetected transmission error. This third scenario has the same effect as IS1 and happens if the sender does not crash but it does not detect the faulty transmission due to another error at the sender node.

Notice that the inconsistency caused by IS1 is associated with process crashes. Thus, both events have to happen, the error in the last but one bit and the crashing of the sender node. On the other hand, the errors that lead to IS2 or IS3 are related to the transmission of the message.

The probabilities of occurrence for these inconsistent scenarios have been calculated [22, 24]. In one hour of transmission with 90% of workload these probabilities vary between 8.80×10^{-3} and 3.96×10^{-8}. It is important to emphasise that IS3 was first noticed by Proenza *et al.* [22], who have demonstrated that IS3 is 10 to 1000 times more likely to take place when compared with IS1. Although unlikely, all these three kinds of scenario have to be considered when dealing with critical applications. Indeed, as has been pointed out [21, 24], the aerospace industry recommends values not higher than 10^{-9} incidents per hour, a level of resilience also adopted by the automotive industry.

Despite the possibility of inconsistent scenarios, the communication network is assumed to be non-partionable. This means that messages may be arbitrarily delayed or even dropped due to inconsistent scenarios, but non-crashed processes cannot be permanently disconnected from each other. For example, a lower priority message may suffer delay due to the retransmissions of higher priority messages. However, if there is no inconsistent scenario, this lower priority message is guaranteed to arrive at its destinations within a finite time. This finite time includes retransmissions of messages in case of errors. Note that this assumption is in line with the characteristics of CAN since messages are automatically rescheduled for retransmission and errors are consistently detected apart from the three inconsistent scenarios.

In cases where no inconsistent scenario takes place, it is correct to say that CAN provides atomic broadcast. In general, atomic broadcast states that [8]: all messages transmitted by correct processes are eventually delivered at all correct processes (eventual delivery); if a message is delivered by a correct process, then the message is delivered to all correct processes (reliable broadcast); all correct processes deliver messages in the same order (total order); and every delivered messages was sent by some processes (integrity). It is not difficult to see that CAN provides eventual delivery and integrity but does not guarantee reliable broadcast or atomicity in the presence of the inconsistent scenarios [22, 24].

3. The Consensus Problem

The consensus problem is usually specified in terms of the following properties:

- *Termination*: Every correct process eventually decides some value.
- *Validity*: If a process decides v, then v was proposed by some process.
- *Agreement*: No two processes decide different values.

Actually, this specification of consensus is known as *uniform* consensus [17], a stronger version of the consensus problem since no process (correct or not) can violate the agreement property. Note that the set of correct processes may propose different values for the consensus due to inevitable non-determinism. The result of the consensus is that they all agree on a single value.

Real-time systems require that the maximum termination time for correct processes be known. Incorporating the termination time into the specification of the consensus problem is possible but not desirable. Such a specification would require some knowledge about the system synchronism (*e.g.* maximum message transmission time, starting time of the processes). The given specification is more generic since it does not take into account this kind of specific characteristic of the target system. Indeed, once one has a consensus protocol that complies with the above specification, process termination time can be derived by analysing the system's worst-case scenario, taking into consideration several factors such as transmitted messages, possible errors and the processing of the system tasks. Support for this kind of analysis in CAN-based systems can be found elsewhere [26, 27] and will not be covered in this paper.

If there were no inconsistent scenarios, solving the consensus problem stated above would be straightforward. Each correct process could propose its initial value by broadcasting it. Then, all correct processes could choose the first delivered message, say. Since all messages would be atomically delivered at all non-crashed nodes in the same order, by following this simple protocol, processes would be able to decide on the same value. Due to the possibility of inconsistent scenarios, this simple consensus protocol would fail, though. This is because inconsistent message omission/duplication may break the ordered delivery of messages and so processes would be unable to pick up a common message. This paper presents a more elaborated protocol that tolerates the occurrence of inconsistent scenarios.

3.1. Related Work

Due to its relevance for fault-tolerant distributed systems, consensus has been studied for decades and has generated a large number of relevant research results, *e.g.* [2, 4, 5, 6, 7, 23]. Space considerations mean that a comprehensive comparison and evaluation is not possible within this paper. However, in general, solutions for the consensus problem made simple assumptions about the properties of the underlying communication network. Here we build upon the real properties of CAN: although we do not assume that CAN delivers atomic broadcast, we do make use of the properties it does deliver. As a result, the proposed protocol is more efficient than the general schemes and more useful for the community of real-time systems. For example, as will be seen, the proposed protocol does not

rely on explicit timing assumptions, which gives more flexibility for application designers. Also, it needs a low number of communication steps and messages when compared with classical approaches [16].

Although consensus has not been studied in the context of CAN, there is some research work on the atomic broadcast problem. As mentioned before, solution for atomic broadcast can be used to solve consensus. However, these proposals are either based on modifying the basic CAN protocol at the hardware level [11, 22], or do not tolerate all the three inconsistent scenarios [24], or assume a restrictive computational model at the application level [18].

4. The Consensus Protocol

This section introduces a protocol that can be used by any set of processes that want to agree on a single value. Let $\Pi = \{p_1, \ldots, p_n\}$ be such a set. The protocol is tolerant to $n-1$ process crashes and f inconsistent scenarios that may occur in CAN. The value of f is a parameter of the protocol and must be chosen by system designers so that the system complies with the desired level of the fault resilience. Since the probability of inconsistent scenarios in CAN is low, several occurrences of inconsistent scenarios during the execution of the consensus protocol are rare. Thus, $f = 1$ may suit most applications and $f > 2$ is unlikely ever to be necessary. The algorithm that describes the protocol is given in figure 1.

The main idea behind the protocol can be explained in the following way. Each process $p_i \in \Pi$ counts the received messages which brings up to date information and keeps track of the order in which they are received. By the assumed computation model, if p_i receives messages m and then m', any process $p_j \in \Pi$ that receives both messages will receive them in the same order provided that there is no inconsistency on the communication network. Thus, in order to agree on a common value, p_i and p_j need to choose the same message, the first one, say. However, due to the occurrence of inconsistent scenarios, one of the processes may receive m' before m, which makes them disagree. As there may be at most f inconsistent scenarios, processes have to count for at least $f+1$ message receiving events to ensure that in one of these events all correct processes choose the same message. The processes do not know which is the correct one, though. In this case, the protocol guarantees that after picking up a common message, all correct processes can agree on the value such a message carries.

An overall description of the protocol is given in section 4.1. Then, section 4.2 enriches this description by presenting some illustrations of possible execution scenarios. Some comments on how the protocol characteristics can be used for improving the performance of applications are also given.

```
procedure consensus(v)
(1)  est_i ← v; r ← 1
(2)  k_i ← 0; id ← i mod θ
(3)  while k_i < f + 1 do
(4)      if id = r mod θ then
(5)          broadcast (k_i, est_i)
(6)          wait for [Msg_i(k_i) ≠ ∅]
(7)      else
(8)          SetTimeout(Δ)
(9)          wait for [ExpTimeout() ∨ Msg_i(k_i) ≠ ∅]
(10)     endif
(11)     if Msg_i(k_i) ≠ ∅ then
(12)         Let (k_j, est_j) be the first message in Msg_i(k_i)
(13)         est_i ← est_j
(14)         k_i ← k_j + 1
(15)     endif
(16)     r ← r + 1
(17) endwhile
return(est_i)
```

Figure 1. The consensus protocol.

4.1. Protocol Description

During the execution of the protocol, any process $p_i \in \Pi$ keeps its estimated value (est_i), which is initially set to its proposed value (line 1). At the end of the protocol, the value of est_i is the consensus value. Although not explicitly expressed in figure 1, it is assumed that each node is able to receive messages before the first node starts broadcasting.

The protocol is made up of several rounds, represented in the algorithm by variable r. Variable k_i is used to count in how many of these rounds messages that were expected were actually received by p_i. Expected messages are those whose counter value is greater than or equal to the current value of k_i. Whenever a process k_i equals value k, p_i is said to reach the protocol stage k.

The current values of k_i and est_i are part of every message broadcast by p_i. The distinction between r and k_i is necessary because there may be rounds in which processes do not receive any expected message. The identification of p_i, namely id, is a function of a parameter θ ($1 \leq \theta \leq n$), defined in line 2. The value of id and the current value of r are used by p_i to take control of its role in the protocol.

There are two roles processes can play in a given round r. Depending on its role, each correct process p_i either may or may not broadcast a message in r. It does whenever $\mathsf{id} = \mathsf{r} \mod \theta$. In this case p_i is known to be a *speaker* in r. Otherwise, p_i is a *listener* and skips the broadcast operation. The parameter θ is used to determine how often processes change their roles from speakers to listeners and vice versa.

If $\theta = 1$, every correct process is a speaker in every round.

The parameter $\Delta \geq 0$ defines the maximum round time (set for each round in line 8) for listeners. This is necessary to avoid situations where they get blocked waiting for messages indefinitely. There is no maximum pre-defined round size for speakers, though. Instead, they broadcast their messages. Both listeners (line 9) and speakers (line 6) wait to collect broadcast messages. The collected messages are represented by the set $\mathsf{Msg}_i(k)$. This set represents the messages received by p_i that contain counters at least as high as k. More formally,

$$\mathsf{Msg}_i(k) = \{(\mathsf{k}_j, \mathsf{est}_j) \text{ received by } p_i | \mathsf{k}_j \geq k\}$$

Whenever $\mathsf{Msg}_i(\mathsf{k}_i) \neq \emptyset$, p_i (whether it is a listener or a speaker) moves on to update both its counter (k_i) and its estimated value (est_i). Among the collected messages, only the one that is received first is selected in the update operations. Note that if there is no inconsistent scenario, all correct processes will receive the collected messages in the same order and so they will select the same message.

There may be rounds where the timeout Δ expires and some listener p_i does not receive any message. This situation may occur: (a) when the messages p_i are waiting for are late or inconsistently omitted at p_i; or (b) when no correct speaker process has broadcast such messages yet (due to either crash failures or to asynchronism). If (a) or (b) takes place, $\mathsf{Msg}_i(\mathsf{k}_i) = \emptyset$ at the end of r. Then, p_i moves on to the next round without updating est_i or k_i, *i.e.* p_i is kept in the same stage of the protocol.

Note that inconsistent omissions occur when some receiver rejects the transmitted message but such a message is not retransmitted, recall section 2. Transmitters, however, already have the message regardless of whether or not it is inconsistently omitted at some other process. In other words, the **wait for** operation for speakers (line 6) can be implemented such that the confirmation of message transmission from the CAN layer is taken into consideration. Therefore, if p_i is a speaker and it does not crash in r, it will receive at least its own message. This means that eventually $\mathsf{Msg}_i(\mathsf{k}_i) \neq \emptyset$ and so p_i will never be blocked indefinitely in line 6.

As may have been noticed, the protocol can be thought of as a game between speakers and listeners, where the objective of the former is to *impose* their estimated values on all correct processes in Π.

4.2. Protocol Illustration

In order to have an intuitive idea of how the protocol works, consider a group of $n = 3$ processes, $\Pi = \{p_1, p_2, p_3\}$. The values proposed by these processes are a, b and c, respectively. Also, assume that $\theta = 3$ and $f = 1$ (*i.e.* no more than 1 message may suffer inconsistent omission/duplication). The illustration given in this section consists of three different scenarios, which are illustrated in figure 2. Diagonal and horizontal lines represent, respectively, broadcast messages and the passage of time. The pair 'number, letter' below the time line of each process p_i indicates the values of k_i and est_i, respectively. The circles on the time line show the messages that each process selects. The dashed boxes represent the waiting time of processes when they are listeners. Inside each box is the value of the corresponding round number. In all three scenarios the processes start executing the protocol approximately at the same (real-) time. This assumption is only for illustration purposes. The protocol works regardless of the time processes start executing.

In scenario 2 (a), the first broadcast message suffers an inconsistent omission. As can be seen, this message, broadcast by p_1 (the speaker when $\mathsf{r} = 1$), arrives at p_1 but is omitted at both p_2 and p_3. After receiving its own message, p_1 updates the values of est_1 and k_1, as indicated in the figure. Then, p_1 moves to the next round, where it changes its role to be a listener. The other two processes wait Δ time units for the message from p_1. When this timeout expires, they update their round numbers to $\mathsf{r} = 2$, a round in which p_2 plays a speaker while p_3 is still a listener. Since p_2 does not receive any message from p_1, the message it broadcasts in the second round contains $\mathsf{k}_2 = 0$. Because of this, the message from p_2 is not selected by p_1, which has already set $\mathsf{k}_1 = 1$. However, both p_2 and p_3 select the message from p_2 and set their estimated value to b. After updating est_3 and k_3, p_3 moves on to the next round, where it is a speaker. As can be seen, all three processes select the message from p_3 and finish the protocol with consensus value b.

In scenario 2 (b), p_1 has its broadcast message inconsistently duplicated. Recall that this inconsistent scenario happens due to the built-in error-recovery mechanism of CAN. Thus, the second broadcast corresponds to the automatic message retransmission at the CAN level. In this example, after receiving the message from p_1, p_2 becomes a speaker and broadcasts its message with the values of both k_2 and est_2 already updated. Note that the retransmission is delayed by the bus arbitration of CAN, as illustrated in the figure. Thus, p_1 only receives its broadcast message after being informed by the CAN layer about the successful transmission. As can be seen, the retransmitted message is received by the processes after they have finished the execution of the protocol. In other words, such a message (in the example) is irrelevant to the processes. However, even if the retransmitted message was received before the end of the protocol (*e.g.* in cases where $f > 1$), the processes would ignore it. This is because the retransmitted message would contain out of date information for the processes that had re-

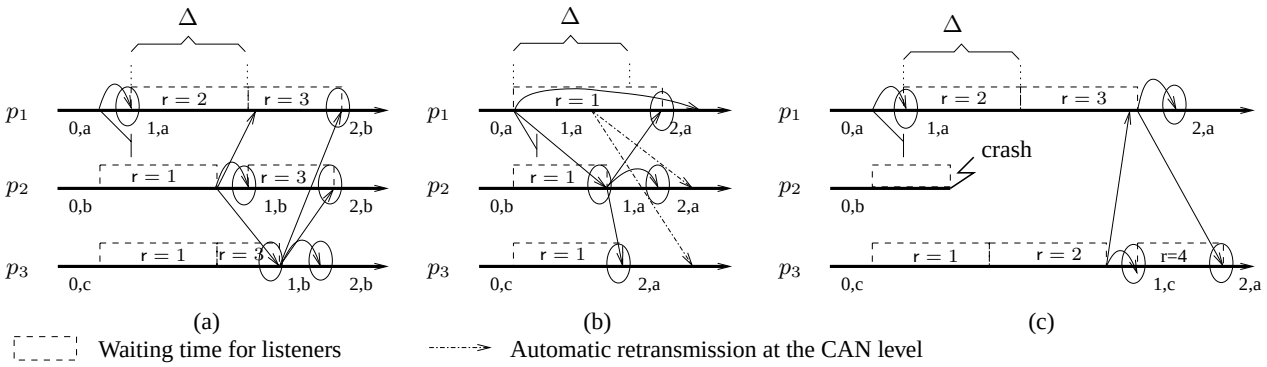

Figure 2. The behaviours of speakers and listeners under fault scenarios, where $f = 1$.

ceived the message from p_2 (*i.e.* the value of k_1 in the message would be too old).

Both process crash and inconsistent message omission are illustrated in scenario 2 (c). The process that crashes, p_2, is responsible for the second broadcast. As it crashes before the broadcast, the other processes wait two rounds for its message. In these two rounds both p_1 and p_3 are playing listeners. At the beginning of the third round, p_3 becomes a speaker and broadcasts its message. Note that this message is selected by p_3 but ignored by p_1 because by the time the message is received $\mathsf{k}_1 > \mathsf{k}_3$. Finally, when p_1 becomes a speaker again and broadcasts its message in its fourth round, which is received by p_3, the correct processes agree on a common value at the end of the round.

Making use of the described scenarios, it is worth emphasising some points regarding the protocol. Firstly, observe that, depending on the scenario, a correct process that starts as a listener may never become a speaker. For example, p_3 reaches agreement in scenario (b) just by accepting the message from p_2. This characteristic may make the protocol more efficient in terms of the number of broadcast messages. Secondly, not only does the number of messages vary but so does the number of rounds, depending on the execution scenario. For example, in scenario (b), p_3 achieves consensus after the first round while p_1 and p_2 take two rounds each. In scenario (c), on the other hand, both p_1 and p_3 take four rounds each to finish the protocol. Thirdly, the values of Δ or θ also have implications for the performance of the protocol since they may determine the maximum waiting time of listeners (Δ) and the number of broadcast messages (Δ and/or θ). Since CAN is a low bandwidth communication network, the values of Δ and θ may be chosen to suit the target system. This issue is discussed in section 6.

It is worth observing that the proposed protocol offers an interesting characteristics that can be used for performance purposes. If no inconsistent scenario regarding the first message broadcast during the protocol takes place, such a message will be atomically delivered at all correct processes. Then, the processes will update their estimated value to the value carried by this received message. After this point in time, no other value for consensus is possible, although the processes do not have such certain knowledge (they do not know whether an inconsistent scenario has taken place or not). Since the probability of an inconsistent scenario is low, such a kind of situation is more likely to take place. Based on the observations above, an optimistic approach can be carried out at the application level. Indeed, instead of waiting until the end of the protocol for the consensus value, the application can use a non-definitive value, produced at the beginning of the protocol execution. In cases where the non-definitive value is different from the consensus value, a cancel-and-recover operation can be released. More details of this optimistic approach, which is described by Lima [15], are not presented here due to space limitations.

5. Correctness and Complexity

5.1. Proof of Correctness

The following lemmas show that the protocol described in figure 1 satisfies termination (lemma 5.1), validity (lemma 5.2) and agreement (lemma 5.3).

Lemma 5.1 (Termination). *Every correct process in* Π *eventually decides some value.*

Proof. A correct process p_i does not decide some value if (a) it is a listener indefinitely blocked in line 9; (b) it is a speaker indefinitely blocked in line 6; or (c) if k_i never equals $f+1$. Statement (a) does not hold because processes do not wait more that Δ in each round where they play listeners. Also, since every correct speaker eventually receives and selects some (at least its own) message, (b) is not true either. Finally, if p_i never updates k_i as a listener, it will become a speaker at most $\theta - 1$ rounds after each time it starts

playing a listener. Consequently, as (b) does not hold, p_i increases k_i at least by one each time p_i plays speaker (line 14). Therefore, $\mathsf{k}_i = f + 1$ at most after p_i plays speaker $f + 1$ times, which means that the lemma follows. □

Lemma 5.2 (Validity). *If a process in* Π *decides* v*, then* v *was proposed by some process in* Π*.*

Proof. By the protocol, the only possibility for a process p_i to alter its estimate est_i is in line 13, where est_i is set to the value carried by the first received message in $\mathsf{Msg}_i(\mathsf{k}_j)$. As by assumption, messages are neither arbitrarily created nor corrupted, est_i is either proposed by p_i or by another process in Π. Therefore, any decided value must have been proposed by some process in Π. □

Lemma 5.3 (Agreement). *If there are no more than* f *inconsistent scenarios, then no two processes in* Π *decide on a different value.*

Proof. If there is no more than one process that decides some value, the lemma trivially holds. Assume that more than one process in Π decides some value.

First, some notation will be introduced. Define the set $\Pi(k) = \{p_i \in \Pi | \, \mathsf{k}_i \geq k\}$, where $0 \leq k \leq f + 1$. In other words, $\Pi(k)$ is the set of processes that eventually had their message receiving counters updated to at least k. For the sake of completeness, consider that processes that crashed before executing line 2 also belong to $\Pi(0)$. It is clear that $\Pi(f + 1) \subseteq \Pi(f) \subseteq \ldots \subseteq \Pi(0) = \Pi$ since processes fail by crashing and crashed processes do not recover.

Let M_i^k ($0 \leq k \leq f$) be the ordered set of every message $(\mathsf{k}_j, \mathsf{est}_j)$ received by $p_i \in \Pi(k+1)$ so that $\mathsf{k}_j = k$ and messages in M_i^k are sorted by receiving order. If $M_i^k = M_j^k$, then M_i^k and M_j^k contain the same messages and they appear in both ordered sets in the same order. As processes that decide some value belong to $\Pi(f + 1)$ and the decided values are their estimated values, it is necessary to show that any two processes in $\Pi(f + 1)$ have a common estimated value.

Now, consider the following claim: there is at least a value of k ($0 \leq k \leq f$) such that $M_i^k = M_j^k$ for all processes p_i and p_j in $\Pi(k + 1)$.

The proof of the claim is by contradiction. In other words, suppose that such a k does not exist. This means that: (a) there is some message in M_i^k that is not in M_j^k or vice-versa; or (b) the order of some messages in M_i^k and M_j^k is not the same. Both situations can only happen in the presence of inconsistent scenarios (by assumption). As a result, there must have been more than f inconsistent scenarios, which is a contradiction.

Now, using the result of the claim, assume that $M_i^k = M_j^k$ for some k ($0 \leq k \leq f$). By the protocol, any process in $p_i \in \Pi(k + 1)$ sets est_i to est_l, where $\mathsf{m} = (k, \mathsf{est}_l)$ is the first message in M_i^k. This makes processes in $\Pi(k + 1)$ have a common estimated value, est_l. If $k = f$ the lemma holds.

It is not difficult to see that the lemma holds for $k < f$ since: (a) no process p_i selects messages whose counter is inferior to k (lines 6 and 9); and so (b) once processes in $\Pi(k + 1)$ have a common estimated value, no other estimated value is broadcast (recall that by assumption the communication network neither arbitrarily corrupts nor creates valid messages). Therefore, by an easy induction on k' ($k < k' < f$), one can show that the lemma follows. □

From the lemmas 5.1, 5.3 and 5.2 the following theorem can be stated:

Theorem 5.1. *The protocol of figure 1 solves consensus for a group of* n *processes despite* $n - 1$ *process crashes and up to* f *inconsistent scenarios.*

5.2. Complexity Analysis

In this section the protocol is analysed in terms of message, time and priority complexity. The message complexity is expressed by the number and the size of the messages. Time complexity is given by the maximum number of rounds processes may execute before achieving consensus. Once the number of rounds is bounded, one can analyse the protocol timeliness for specific applications by carrying out other analysis approaches [26, 27], which are not covered here.

As for the number of priorities necessary for executing the protocol, from figure 1, it is clear that there must be n priority levels available. This requirement is due more to CAN characteristics than to the protocol itself. Indeed, different messages cannot be concurrently broadcast with the same priority in CAN.

The analysis presented in this section takes into account only aspects related to the proposed consensus protocol. Automatic message retransmission due to the bus-arbitration or error-recovery is not considered. Also, since CAN is a broadcast-oriented network, a broadcast operation accounts for one message.

In the worst-case scenario, all correct processes broadcast synchronously (see figure 3). This means that for each correct process p_i, one message is broadcast in each round and the value of k_i increases by one per round. As there may be at most n correct processes and $f+1$ broadcasts per process, the maximum number of messages transmitted during the execution is expressed by $(f + 1)n$.

It is important to note that on average the number of messages can be much less than $(f + 1)n$. For example, as can be seen from figure 2, some processes may broadcast no message. Nonetheless, at least $f + 1$ messages need to be broadcast. This is because a process only terminates when it reaches the $(f + 1)^{th}$ stage of the protocol.

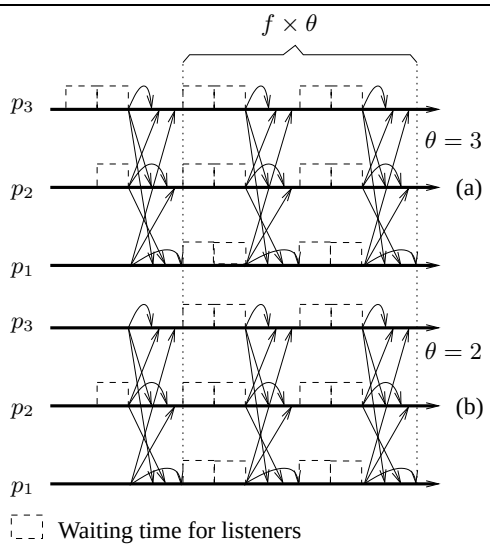

Figure 3. Maximum number of rounds.

In terms of message size, the proposed protocol is not costly. As each transmitted message by p_i contains $\mathsf{k}_i \leq f$ and est_i, one needs $\log_2(f) + \log_2(\mathsf{est})$ bits for representing the message content, where est stands for the maximum value of est_i.

In order to analyse the worst-case number of rounds, consider first that $\theta = n$. The worst-case for each process p_i is when it executes $i - 1$ rounds as listener, before receiving some message. This situation is illustrated in figure 3 (a), where $\Pi = \{p_1, p_2, p_3\}$, $f = 2$ and $\theta = 3$. In the figure, p_3 starts executing the protocol first, playing listener for two rounds before becoming a speaker. Process p_2 starts executing Δ time units after p_3, waits one round and then becomes a speaker. Two rounds after p_3 and one round after p_2, p_1 starts its execution. After this point all processes are synchronised, broadcasting messages concurrently. Observe that the maximum number of rounds executed by each correct process p_i is given by $i + f\theta$ rounds.

If $\theta < n$, it is not difficult to see that the worst-case scenario can be represented by a situation where all processes p_i and p_j such that $(i \mod \theta) = (j \mod \theta)$ execute their rounds synchronously. For example, making $\theta = 2$, as illustrated in figure 3 (b), p_1 and p_3 have the same behaviour: both are speakers in the first round. In other words, both p_1 and p_3 will execute at most $1 + f\theta$ rounds while the maximum number of rounds executed by p_2 is given by $2 + f\theta$, as can be seen from the figure. Following these observations, one can derive the general expression that gives the maximum number of rounds executed by each process p_i as

$$1 + (i - 1) \mod \theta + f\theta \quad (1)$$

In practice, the number of rounds and messages may be much less than the values given by the worst-case scenarios. For example, in figure 2 (b), p_3 finishes the execution of the protocol after the first round. Although this case may occur, it is important to emphasise that the sum of executed rounds by all correct processes cannot be less than $f + 1$. Other factors such as asynchronism between processes and faults may affect the behaviour of the protocol in terms of number of rounds and number of messages. These and other factors are considered in the next section, where the message and time complexity are analysed by simulation.

6. Assessment by Simulation

The main goal of this section is to evaluate the effects that the values of θ and Δ have on the performance of the protocol as measured in terms of number of messages, rounds and time spent executing the protocol. This evaluation, carried out by simulation, takes into consideration non-deterministic aspects of the environment/application, such as asynchronism among the processes or the errors that may occur.

6.1. Simulation Specification

The evaluation was carried out by simulating the execution of the consensus protocol described in figure 1 for a set of $n = 6$ processes, where several values of θ $(1, 2, \ldots, 6)$ and Δ (0,2, 5, 7, 10, 12, 15, 17, 20) were considered. In order to evaluate the behaviour of the protocol in fault scenarios, the simulation was configured so that there were up to $f = 2$ inconsistent scenarios and up to 2 process crashes during the execution of the protocol. In order to isolate the behaviour due to the protocol from the aspects of CAN, message retransmission at the CAN level was not considered. Thus, inconsistent scenarios were simulated by taking into account only message omissions. Since the effect of both kinds of inconsistent scenario is the same (break the atomicity/order of the CAN protocol), considering only omission does not restrict the results of the simulation.

All simulation was carried out in a uniprocessor machine, and time was measured as non-negative integers. The simulation was specified as follows. Each process p_i was set up to start proposing their values at time t_i^s, where t_i^s was defined according to a normal distribution with mean t_0 and standard deviation $t_0/2$. The time t_0 was chosen at random in the interval $[1, 250]$. Then, two processes were chosen, at random, to crash. Given that p_i was a chosen process, its crashing time t_i^c was determined by a uniform distribution in the interval $[t_0/2, 1.5t_0]$. If $t_i^c \leq t_i^s$, then p_i will never execute the protocol. Otherwise, p_i may crash during its execution in cases where the crash takes place before p_i can return the consensus value. As for inconsistent omission, f different integer numbers were generated according to a uniform distribution in the interval $[1, n(f + 1)]$. These

numbers are to identify which message will be omitted since there may be from $(f+1)$ to up to $n(f+1)$ broadcast messages during the execution of the protocol. Thus, if message k is chosen to be inconsistently omitted, then the k^{th} message that is broadcast during the simulation will fail to arrive at some processes, whichever its sender is.

Given the above specification, the simulation was implemented by a loop from time $\min_{\forall p_i \in \Pi}(t_i^s)$ until every process either returns the consensus value or crashes. At each instant of time t, procedures that simulate both the processes and the network are performed.

The simulation procedure described was performed $1,000$ times. The results collected from these executions were averaged. These results are given in terms of the total number of broadcast messages and the total number of executed rounds per process. The collected results are summarised in figures 4 and 5 and are discussed in the next section.

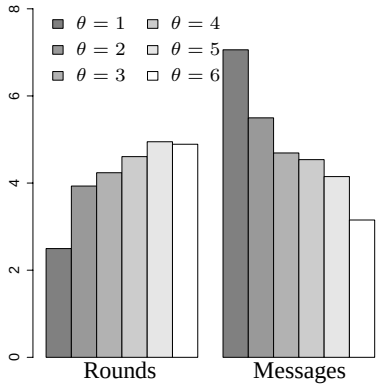

Figure 4. Effects of varying θ ($\Delta = 20$).

6.2. Simulation Results

It is clear that the value of θ, which determines how often listeners become speakers, affects both the number of rounds and messages needed for the execution of the protocol. This effect is illustrated in figure 4, which shows the number of rounds per correct process and the total number of messages. The value of Δ was considered fixed and equal to 20 time units. As can be seen, the higher the value of θ, the more rounds are executed per process. However, the number of messages grows with lower values of θ. These behaviours were expected (by the analysis of section 5.2) and they highlight some good characteristics of the protocol. For example, for values of θ greater than 1, no more than 6 messages (on average) were necessary to achieve

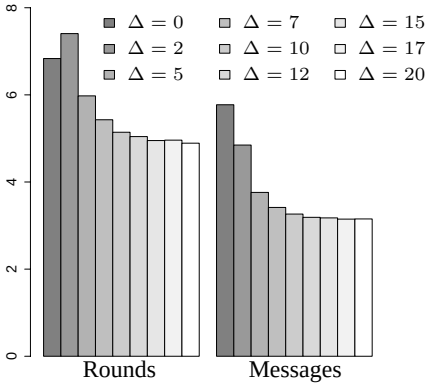

Figure 5. Effects of varying Δ ($\theta = 6$).

consensus. Recall that the number of messages lies between 3 (*i.e.* $f+1$) and 18 (*i.e.* $(f+1)n$). Therefore, the simulation indicates that the actual number of messages is, on average, much less than in the worst case. As CAN is a low bandwidth network, these results have significant practical relevance. Also, observe that each correct process (on average) finishes the execution of the protocol within no more than 5 rounds, which compares favourably with the worst-case values given by equation (1). For example, consider $\theta = 3$. This equation gives $7, 8, 9, 7, 8$ and 9 for each process $p_1, p_2, \ldots, p_6$, respectively. This corresponds to 8 rounds per process, which is much higher than 4.26, found in the simulation.

Figure 5 presents the obtained results in terms of numbers of rounds and messages for several values of Δ, where θ was kept fixed throughout the simulation. As can be seen, the numbers of both rounds and messages decrease when the value of Δ increases. As Δ defines the maximum time listeners will wait for messages, its value affects the speed with which listeners will increase their round numbers and so it also affects the time within which they become speakers.

An interesting effect shown in figure 5 is that the number of both rounds and messages converge to a certain value when Δ is increased. This effect is due to the fact that if listeners wait longer, it is likely that they will receive the expected messages before they become speakers, reducing the number of messages and rounds.

Overall the simulation shows that varying the values of θ and Δ gives designers considerable freedom over the performance of the protocol.

7. Conclusion

A consensus protocol for CAN-based systems has been presented. The proposed protocol relies on the fact that most

of the time transmitted messages are atomically delivered in the same order to all correct processes. The advantages of using this property instead of relying on message transmission delays is that the processes can achieve consensus regardless of the actual level of timing synchronism present in the system.

The consensus protocol was specified in terms of two parameters, Δ and θ, which can be adjusted in order better to suit the protocol to the needs of specific applications and/or environments. The effects of these parameters have been assessed by simulation.

A number of other agreement problems, such as atomic broadcast and membership control can be specified in terms of consensus: consensus on message order and consensus on membership list, respectively. Due to their relevance for fault-tolerant real-time systems and the effectiveness of the approach described in this paper, it is possible to build a complete set of services to support highest integrity applications. This issue will be part of future research.

References

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

[2] G. Bracha and S. Toueg. "Asynchronous Consensus and Broadcast Protocols Systems". *Journal of ACM*, 32(4):824–840, 1985.

[3] I. Broster and A. Burns. "An Analysable Bus-Guardian for Event-Triggered Communication". In *Proc. of the 24th Real-time Systems Symposium (RTSS)*. IEEE Computer Society Press, 2003.

[4] T. Chandra, V. Hadzilacos, and S. Toueg. "The Weakest Failure Detector for Solving Consensus". *Journal of ACM*, 43(4):685–722, 1996.

[5] D. Dolev, C. Dwork, and L. Stockmeyer. "On the Minimal Synchronism Needed for Distributed Consensus". *Journal of ACM*, 34(1):77–97, 1987.

[6] C. Fetzer and F. Cristian. "On the Possibility of Consensus in Asynchronous Systems". In *Pacific Rim Int'l Symposium on Fault-Tolerant Systems*, pages 86–91, 1995.

[7] M. J. Fischer, N. A. Lynch, and M. S. Peterson. "Impossibility of Distributed Consensus with One Faulty Process". *Journal of ACM*, 32(2):374–382, 1985.

[8] V. Hadziacos and S. Toueg. "Fault-Tolerant Broadcast and Related Problems". In S. Mullender, editor, *Distributed Systems*, chapter 5. Addison-Wesley, 2nd edition, 1993.

[9] V. Hadzilacos. "On the Relationship between the Atomic Commitment and Consensus Problems". In *Proc. of the Workshop on Fault-Tolerant Distributed Computing. Lecture Notes in Computer Science Vol. 448 (B. Simons and A. Spector, eds.)*, volume 448, pages 201–208. Springer-Verlag, 1990.

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

[11] J. Kaiser and M. A. Livani. "Achieving Fault-Tolerant Ordered Broadcasts in CAN". In *Proc. of the 3rd European Dependable Computing Conference*, 1999.

[12] R. M. Kieckhafer, C. J. Walter, A. M. Finn, and P. M. Thambidurai. "The MAFT Architecture for Distributed Fault Tolerance". *IEEE Transactions on Computers*, 37(4):398–405, April 1988.

[13] H. Kopetz. *Real-Time Systems Design for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.

[14] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger. "Distributed Fault-Tolerant Real-Time Systems: The MARS approach". *IEEE Micro*, 9(1):25–40, 1989.

[15] G. M. A. Lima. *"Fault Tolerance in Fixed-Priority Hard Real-Time Distributed Systems"*. PhD thesis, Deptartment of Computer Science, University of York, 2003.

[16] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publisher, 1996.

[17] G. Neiger and S. Toueg. "Automatically Increasing the Fault-Tolerance of Distributed Systems". *Journal of Algorithms*, 11(2):374–419, 1990.

[18] L. M. Pinho and F. Vasques. "Timing Analysis of Reliable Real-Time Communication in CAN Networks". In *Proc. of the 13th Euromicro Conference on Real-Time Systems*, pages 103–112. IEEE Computer Society Press, 2001.

[19] S. Poledna. *Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism*. Kluwer Academic Publishers, 1996.

[20] D. Powell. *Delta-4: A Generic Architecture for Dependable Distributed Computing*, volume 1 of *Research Reports ESPRIT*. Springer-Verlag, Berlin, Germany, 1991.

[21] D. Powell. "Failure Mode Assumptions and Assumption Coverage". In *Proc. of the 22nd Fault-Tolerant Computing Symposium (FTCS)*, pages 386–395. IEEE Computer Society Press, 1992.

[22] J. Proenza and J. Miro-Julia. "MajorCAN: A Modification to the Controller Area Network Protocol to Achieve Atomic Broadcast". In *Proc. of the of the IEEE Int'l Workshop on Group Communication and Computations (IWGCC)*, pages C72–C79, 2000.

[23] M. Raynal. Consensus in Synchronous Systems: A Concise Guided Tour. Technical Report PI-1467, IRISA research reports, July 2002.

[24] J. Rufino, P. Veríssimo, G. Arroz, C. Almeida, and L. Rodrigues. "Fault-Tolerant Broadcasts in CAN". In *Proc. of the 28th Fault-Tolerant Computing Symposium (FTCS)*, pages 150–159, 1998.

[25] L. S. Sabel and K. Marzullo. "Election Vs. Consensus in Asynchronous Systems". Technical Report TR95-1488, Cornel University, Feb. 1995.

[26] K. Tindell, A. Burns, and A. J. Wellings. "Analysis of Hard Real-Time Communications". *Real-Time Systems*, 9(2):147–171, 1995.

[27] K. Tindell, A. Burns, and A. J. Wellings. "Calculating Controller Area Network (CAN) Message Response Times". *Control Engineering Practice*, 3(8):1163–1169, 1995.