Fault-Tolerant Clock Synchronization in CAN

Luís Rodrigues
FCUL[*]
ler@di.fc.ul.pt

Mário Guimarães[†]
IST-UTL
Mario.Guimaraes@inesc.pt

José Rufino
IST-UTL[‡]
ruf@digitais.ist.utl.pt

Abstract

This paper presents a new fault-tolerant clock synchronization algorithm designed for the Controller Area Network (CAN). The algorithm provides all correct processes of the system with a global timebase, despite the occurrence of faults in the network or in a minority of processes. Such global time-frame is a requirement of many distributed real-time control systems.

Designing protocols for CAN is justified by the increasing use of this network in industrial automation applications. CAN owns a number of unique properties that can be used to improve the precision and performance of a clock synchronization algorithm. Unfortunately, some of its features also make the implementation of a fault-tolerant clock synchronization service a non-trivial task. Our algorithm addresses both the positive and the negative aspects of CAN.

1. Introduction

The availability of a global timebase in all correct processes, despite the occurrence of faults in a minority of processes or in the network itself, is a requirement of many distributed real-time control systems. For instance, synchronized clocks can be used for the synchronization of external actions, distributed trace of events, measurement of actions that spawn multiple processes and the development of (higher level) fault-tolerant distributed algorithms.

A common solution for the global time-base problem consists in using the node hardware clock to create a virtual clock at each process, which is locally read. All virtual clocks are internally synchronized by a *clock synchronization algorithm*. Surveys of existing clock synchronization algorithms can be found in [17, 13]. Clock synchronization algorithms differ on issues such as the precision they achieve (i.e., how far clocks can be from each other), number and type of tolerated faults, number and size of messages exchanged, etc. Naturally, the solution for clock synchronization deeply depends on the properties of the underlying network.

The Controller Area Network (CAN) [14, 8] is a communication bus for message transaction in small-scale distributed environments. Originally designed to reduce cabling complexity and saving wiring costs in automotive applications, CAN gathers nowadays an increasing acceptance in other areas, like control and automation. In the design and implementation of real-time distributed control systems, CAN represents a very cost-effective *field-bus* solution for real-time sensing and actuating in harsh environments with strict timeliness and reliability requirements.

This paper presents a new clock synchronization algorithm designed for CAN. The paper discusses the CAN properties that can be used to improve the precision of clock synchronization and the properties that make the implementation of a fault-tolerant version of such service a non-trivial task. The algorithm can be implemented exclusively in software, tolerates process and network faults, and provides precision and accuracy preservation in the order of a few microseconds. The algorithm is inspired of the generic *a posteriori agreement* algorithm for broadcast networks [22, 23] and of a non fault-tolerant algorithm specially designed for CAN [4], but differs significantly from these algorithms. We have named our new algorithm "phase-decoupled" *a posteriori* agreement.

The paper is organized in three major parts. The first part provides the background: Section 2 provides a brief description of CAN operation, discussing its relevant properties; Section 3 introduces the clock synchronization problem; related work is surveyed in Section 4. The second part describes our work: the design approach is sketched in Section 5; a straightforward implementation of the *a posteriori* agreement on CAN is described Section 6; the new "phase-decoupled" *a posteriori* agreement algorithm is presented in Section 7. The last part is concerned with improvements and performance issues: use of CAN message priorities is discussed in Section 8 and the performance is analyzed in

[*] Faculdade de Ciências da Universidade de Lisboa
[†] Currently at Inesc, Portugal.
[‡] Istituto Superior Técnico da Universidade Técnica de Lisboa

CAN1 - Validity: if a correct node broadcasts a message, then the message is eventually delivered to a correct node.

CAN2 - Best-effort Agreement: if a message is delivered to a correct node, then the message is eventually delivered to all correct nodes, if the sender remains correct.

CAN3 - At-least-once Delivery: any message delivered to a correct node is delivered at least once (a message is automatically retransmitted if an error occurs).

CAN4 - Non-triviality: any message delivered to a correct node was broadcast by a node.

CAN5 - Bounded Transmission Delay : the time elapsed between the request of a broadcast and the corresponding message delivery at any correct node is bounded by two known values $\Gamma^{min} < \Gamma^{max}$.

CAN6 - Tightness: if the sender remains correct, the last retransmission of the same message is delivered to any two correct nodes at real time values that differ, at most, by a known interval $\Delta\Gamma_{tight}$.

Figure 1. CAN Properties

Section 9. Section 10 concludes the paper.

2. Controller Area Network

CAN is a bus with a multi-master architecture [14, 8]. The transmission medium is usually a twisted pair cable. The network maximum length depends on data rate; typical values are: 40m @ 1 Mbps; 1000m @ 50 kbps. Bus state takes one out of two values: *recessive*, which only appears on the bus when all the nodes send recessive bits; *dominant*, which only needs to be sent by one node to stand on the bus.

Any message sent by a CAN node must be tagged with a network-wide unique identifier. Access control to the shared bus uses a *carrier sense multi-access with deterministic collision resolution* (CSMA/DCR) scheme. Bus access conflicts are resolved through the bitwise comparison of message unique identifiers: if the transmitted identifier bit is recessive and a dominant bit is monitored, the node gives up from transmitting and starts to receive incoming data; the node transmitting the message with the lowest identifier goes through and gets the bus. If the arbitration process is lost, a new attempt to send the message is made when the bus is released.

The CAN network can be modeled by the set of properties summarized in Figure 1. We note that only the properties relevant for clock synchronization are listed: a more precise model can be found in [16]. Properties CAN1 to CAN4 are a consequence of the comprehensive set of error detection, error signaling and error recovery features of the CAN network. Messages corrupted by errors are discarded at correct receivers and automatically submitted for retransmission by a correct sender. This procedure secures property CAN1. Unfortunately, it also allows the same message to be received by a correct node more than once [16] (property CAN3). When no CAN protocol violation is detected until the last but one bit of a message, any correct receiver will always locally accept that message, even if the following bit gets corrupted. Conversely, a correct sender will consider such corruption an error and it will retransmit the message.

Properties CAN5 and CAN6 describe system behavior in the time domain. Ensuring property CAN5 depends on multiple factors: traffic patterns, latency classes and offered load bounds, as well as their relation with CAN message identifiers[20, 24]; error patterns and maximum error recovery latency [15]. Property CAN6 is crucial for achieving a high precision on synchronized clocks. The upper bound of message reception real time variance ($\Delta\Gamma_{tight}$) has two different contributions [10]: the maximum variance on the network physical propagation delay $\Delta\Gamma_{prp}$; the maximum variation of message processing time at any correct receiver, $\Delta\Gamma_{rec}$. By correct design, $\Delta\Gamma_{rec}$ can be bounded by values in the order of a few microseconds (some controllers offer dedicated support to minimize this bound [21]). On the other hand, CAN is particularly advantageous with regard to variation of $\Delta\Gamma_{prp}$: the bus transmission line is operated in a *quasi-stationary* mode, giving enough time for bit signal stabilization along the bus before performing sampling. The exact value of $\Delta\Gamma_{prp}$ depends on the bus lenght and on network configuration parameters, but it is always a small fraction of the network bit time (10%-30%).

3. Clock synchronization

The goal of clock synchronization is to establish a global timebase in a distributed system composed of a set of processes $\mathcal{P}$ which can interact exclusively by message passing. Processes can only observe time through a *clock*. One commonly used solution to achieve this goal is to provide each process p ($p \in \mathcal{P}$) in the distributed system with an imperfect physical clock pc_p (notation closely follows that of [17]). The clock at a correct process p can then be viewed as implementing, in hardware, an increasing continuous[1] function pc_p that maps (non-observable) real time[2] t to a clock time $pc_p(t)$. Through a clock synchronization algorithm it is possible to derive, from the physical clock at each process p, a virtual clock vc_p satisfying the *precision* (VC1), *rate* (VC2), and *accuracy* (VC3) properties, presented in

[1] It is known that digital clocks have a finite granularity and increase by ticks [10]. However, for sake of clarity, we chose to simplify our expressions in this matter.

[2] In an assumed Newtonian time frame.

PC - Physical clocks; VC - Virtual clocks
(for some positive constants μ and ρ, $\forall_{k,l} \in \mathcal{P}$)

PC1- Initial value,
$pc_k(0)$: $0 \leq pc_k(0) \leq \mu$

PC2- Rate,
$\exists \rho$: $0 \leq 1 - \rho \leq \frac{pc_k(t_2)-pc_k(t_1)}{t_2-t_1} \leq 1 + \rho$ for $0 \leq t_1 < t_2$

VC1 - Precision,
$\exists \delta_v$: $|vc_k(t) - vc_l(t)| \leq \delta_v$ for $0 \leq t$

VC2 -Rate,
$\exists \rho_v$: $1 - \rho_v \leq \frac{vc_k(t_2)-vc_k(t_1)}{t_2-t_1} \leq 1 + \rho_v$ for $0 \leq t_1 < t_2$

VC3 - Accuracy,
$\exists \alpha_v$: $|vc_k(t) - t| \leq \alpha_v$ for $0 \leq t$

Figure 2. Summary of Clock Properties.

Figure 2.

Precision δ_v characterizes how closely virtual clocks are synchronized to each other, ρ_v is the drift rate of virtual clocks. Accuracy α_v characterizes how closely virtual clocks are synchronized to real time. Due to the nonzero drift rate of physical clocks, accuracy cannot be ensured without some external source of real time. However, a good algorithm should maintain clocks as close as possible to the best real time source available, which may be one of the correct clocks in the system. In that sense, minimizing[3] ρ_v, should *preserve* accuracy, and that term will be used in this paper when informally discussing these attributes.

Since physical hardware clocks can be permanently drifting from each other, virtual clocks must be resynchronized from time to time. A clock synchronization algorithm should then be able to: (i) generate a periodic resynchronization event. The time interval between successive synchronizations is called the resynchronization interval, denoted T; (ii) provide each correct process with a value to adjust the virtual clocks in such a way that *precision* and *rate* hold. The clock adjustment can be applied instantaneously or spread over a time interval. In both techniques, for the sake of convenience, the adjustment is usually modeled by the start of a new virtual clock upon each resynchronization event [5, 17].

The worst-case clock precision δ_v is obtained by adding to the precision δ_{vi} achieved with the synchronization the drift between clocks during the resynchronization interval T, that is $\delta_v = \delta_{vi} + 2\rho T$. The physical clock drift ρ is typically of the order of 10^{-4} to 10^{-6} and the resynchronization interval T can be selected such that the desired precision is guaranteed. If the algorithm exhibits a good precision enhancement property a longer resynchronization interval can be chosen.

4. Related work

Software based clock synchronization protocols can be fully generic or tailored for certain classes of networks. In this paper we are looking for solutions that can exploit the CAN properties mentioned in Section 2, namely the ability to generate a "simultaneous" event through the broadcast of a message. Thus we will concentrate our attention on two algorithms that are targeted for networks with these properties.

4.1. A posteriori agreement

The *a posteriori agreement* for clock synchronization [22, 23] is a technique that uses tightness property of some networks (see CAN6) to avoid the influence of the network access delay variability on the precision of virtual clocks. An aim of the *a posteriori agreement* technique is to improve precision by making the clock synchronization algorithm depend on $\Delta\Gamma_{tight}$, instead of depending on the variance of message delivery ($\Delta\Gamma = \Gamma^{max} - \Gamma^{min}$, according to property CAN5) or on the worst-case message delivery (Γ^{max}). The improvement on clock precision is high because $\Delta\Gamma_{tight} << \Delta\Gamma$ (note that $\Delta\Gamma$ also includes the network access delay variance).

Synchronization starts with each process disseminating a *start* message at a pre-agreed instant on its clock. Reception of *start* messages trigger the start of a new virtual clock. Note that, due to process or network faults, not all broadcasts will be received by all correct processes. Thus, clocks triggered by a *start* message must be kept merely as candidates for synchronization until an agreement is obtained on a broadcast yielding high precision. This agreement can be used to select an adjustment to the absolute value of the elected clock, in order to yield the best accuracy preservation possible. Since the agreement is executed after the candidate virtual clocks have been started, the algorithm was called *a posteriori agreement*. As a consequence of this approach, the resulting precision is mainly limited by $\Delta\Gamma_{tight}$ and marginally by the time required to reach agreement (Γ^{max}_{agreem}). The precision achieved by the *a posteriori* agreement algorithm was proven [22, 23] to be limited by:

$$\delta_{vi} \geq (1 + \rho)\Delta\Gamma_{tight} + 2\rho\Gamma^{max}_{agreem}$$

The general algorithm is communication and agreement protocol independent, i.e., the choice of different communication infrastructures and agreement protocols would lead

[3] In any case, limited to ρ [18].

to different implementations of the algorithm. An implementation of the *a posteriori* technique for local area networks such as Ethernet and Token-Bus has been proposed [23]. However, the bandwidth and message size required by such implementation is not supported by the maximum data field size (8 bytes) allowed in CAN messages. Our work departs from designing a specialization of the original *a posteriori* protocol that defines an agreement protocol tailored to the CAN network.

4.2. CAN oriented algorithms

Gergeleit and Streich have proposed a clock synchronization algorithm for CAN based on a master-slave configuration [4]. The algorithm can be seen as a non fault-tolerant implementation of the *a posteriori* agreement approach. The master periodically emits a *start* message that triggers the start of a new virtual clock in all the slaves. CAN properties guarantee that, if the master survives, these virtual clocks are precise. Accuracy is achieved by calculating the adjustment *a posteriori*. Since the algorithm is not fault-tolerant, no agreement protocol needs to be executed. The master sends an absolute clock value based on its own measurement of the delay incurred for the dissemination of the *start* message (typically, the master will be connected to an external source of time) and all slaves adjust their clocks accordingly. To reduce traffic, the master reference value required for the adjustment is piggybacked in the next *start* message.

A positive aspect of this algorithm is its low bandwidth consumption. A single message at every synchronization round is enough to keep the clocks synchronized. The major drawback is its complete lack of tolerance with regard to the failure of the master process. To overcome this drawback it was suggested to use multiple cooperating masters using a token-based approach; in each synchronization round a different master would be responsible for ensuring synchronization [1]. Unfortunately, since CAN does not guarantee reliable delivery when the sender fails [16], it is possible for a failed master to leave the system in an inconsistent state.

4.3. Other approaches

A major limitation of all known software clock synchronization algorithms designed for arbitrary networks, is that precision is limited either by the variance of the message delivery delay [12], or worse, by its upper bound [18]. This problem may be minimized with hardware support, either by implementing clock synchronization exclusively by hardware [7, 11] or by using hybrid schemes [13] which attempt at reducing that variance, for instance, using clock synchronization units that are able to timestamp messages [9] and receive GPS signaling. Although designing specifically for CAN, our goal is to allow the use of "off-the-shelf" components. Statistical techniques can also be used to minimize the effect of the network variance [2]. The work of [3] provides an interesting integration of internal and external clock synchronization but, being based on remote clock reading, it is not clear how it can be adapted to exploit CAN tightness.

5 Design overview

The CAN owns a number of characteristics that offer the potential for achieving highly precise clock synchronization, in particular it exhibits a network tightness in the order of a few microseconds and built-in error handling facilities. On the other hand, it has a low bandwidth (compared with today's LANs) and supports only small messages which favors simple protocols. Also, only best-effort agreement is provided (i.e., in case of sender failure the message may be received by just a subset of the nodes) which difficults agreement on clock values.

As seen in the previous sections, some clock synchronization algorithms have been designed specifically for CAN. However, these protocols exhibit limited or no fault-tolerance features, having thus limited applicability for dependable applications. On the other hand, most of the generic algorithms described in the literature cannot make explicit use of the unique (positive) features of CAN.

The *a posteriori* agreement approach seems suitable for CAN since the precision achieved is in the order of the network tightness. Our work is based on the idea of applying the *a posteriori* technique to CAN. However, limitations of a straightforward implementation of this technique, lead us to develop a new algorithm, particularly suited for the CAN network.

6. What's missing for *a posteriori* agreement on CAN

To motivate the need for a new algorithm, we describe first a straightforward implementation of the *a posteriori agreement technique* for CAN. The algorithm is obtained by enhaning the generic *a posteriori* algorithm described in [22, 23] with a CAN-specific agreement protocol. The proposal of an agreement protocol suited to CAN is also a contribution of this work.

The resulting algorithm offers excellent precision and requires only two phases of message exchanges. On the other hand, it requires a large number of messages in the first phase and does not provide good accuracy. A run of this basic algorithm requires at least $n(n+1)$ messages; in the next section, we will describe an algorithm that lowers the number of messages required down to $3n$.

The algorithm is fully decentralized. In order to tolerate f faults, at least $2f + 1$ processes must try to generate a simultaneous event (for clarity, we will simply assume that all processes try to do so). No matter how many processes trigger the synchronization algorithm, all correct processes need participate in the agreement to select one of the simultaneous events as the source of the clock for the next synchronization interval. We assume that there is a total order of the processes identifiers; this order is used to rank votes on the agreement phase of the algorithm.

A pseudo-code description of the algorithm is given in Figure 3. Let T denote the resynchronization interval. Each period is initiated by a process p when its virtual clock reaches iT, the time for synchronization round i, by broadcasting a $\langle$*start*, i$\rangle$ message on the network (l. 107). If the sender does not fail, CAN guarantees that this message will be eventually received by all correct processes at approximately the same time (properties CAN2 and CAN6). Note however, that the occurrence of faults may lead to message retransmission by the CAN protocol "cast in silicon". Thus, duplicates of the same $\langle$*start*$\rangle$ message can be received (property CAN3). Each time a *start* message is received a new candidate clock is started (duplicates restart this clock). Typically, several processes will send a *start* message at approximately the same time. Only tight events may be eligible as candidates. In CAN, only the sender can detect reliably when a message is delivered to all correct processes. Thus, only the sender can safely propose its own message (and associated clock) as a valid candidate.

When a new candidate clock is started, it is started with some dummy pre-defined value (l. 113). In fact, candidate clocks may be precise but are inaccurate because there is a variable and unpredictable delay in the dissemination of the *start* message. At the end of the agreement, the selected clock is adjusted to a value that best preserves the accuracy. In this basic algorithm, this adjustment is computed by the sender of the associated *start* message, based on the local measurements, at each process, of the virtual time at which the corresponding message was locally received. Let $rt^{i,p}$ denote the reception time of the $\langle$ *start*, $i\rangle$ message from p, according to vc_q^{i-1}. In order to make this information available at the sender, every $\langle$*start*, i$\rangle$ message is acknowledged directly to the sender p by every process q, with an $\langle$*ack*, i, p, $rt^{i,p}\rangle$ message (l. 119).

The protocol proceeds with a second phase of message exchange where the processes agree on which candidate clock should be used for the next round. This phase is initiated by a sender p when: it detects the succesfull transmission of its own *start* message; at least $f + 1$ *start* messages have been received; it has received all the associated *ack* messages or the correponding `AckTimer` has expired (l. 124). When these conditions are satisfied, the sender computes the adjustement for its own clock, by selecting the

```
100  // variable description
101  //    voted$^i$: voted candidate.
102  //    $RT^i$[]: reception times.
103  //    $\Delta^i$: adjustment for this candidate.
104  //    starts$^i$, acks$^i$, and votes$^i$: counters
105  starts$^i$ := 0; acks$^i$ := 0;
106  voted$^i$ := NONE;

107  when $vc^{i-1}(NOW) = iT$
108    and voted$^i$ $=$ NONE do
109        broadcast ($\langle$start, i$\rangle$);

110  when received S=$\langle$start, i$\rangle$ from $q$ do
111        $rt^{i,q}$ := $vc^{i-1}$ (NOW);
112        // start new candidate clock
113        $cc^{i,q}$ (NOW) := 0;
114        if not-duplicate(S) then starts$^i$ := starts$^i$ $+$ 1;
115        if p = q then // my own start message
116           acks$^i$ := acks$^i$+1;
117           RT$^i$[$p$] :=$rt^{i,q}$
118        else
119           unicast ($\langle$ack, i, $rt^{i,q}\rangle$);
120        if notstarted(AckTimer) then start(AckTimer);

121  when received A=$\langle$ack, i, $rt^{i,p}\rangle$ from $q \neq p$ do
122        RT$^i$[$q$] :=$rt^{i,p}$
123        if not-duplicate(A) then acks$^i$ := acks$^i$ $+$ 1;

124  when transmission-confirmed ($\langle$start, i$\rangle$)
125    and starts$^i$ $\geq f + 1$
126    and (acks$^i$ = N or expired(AckTimer))
127    and voted$^i$ $=$ NONE do
128           N := acks$^i$;
129           voted$^i$ := p;
130           votes$^i$ := 0;
131           $\Delta^i$ := median($\forall_x RT^i[x] > 0$);
132           broadcast($\langle$vote, i, p, $\Delta^i\rangle$);

133  when received V=$\langle$vote, i, v, $\Delta^v$ $\rangle$ from $q \neq p$
134    and not-duplicate(V) do
135           if not-started(VoteTimer) then start(VoteTimer);
136           if rank(v) $>$ rank(voted$^i$) then do
137              voted$^i$ := v;
138              votes$^i$ := 1;
139              $\Delta^i$ := $\Delta^v$;
140              broadcast($\langle$vote, i, v, $\Delta^v\rangle$ );
141           if voted$^i$ $=$ v then votes$^i$ := votes$^i$ $+$1;

142  when expired(VoteTimer) or votes$^i$ = N
143    and voted$^i$ $\neq$ NONE do
144           $vc^i$ := $cc^{i,voted^i}$ $+ \Delta^i$;
145           N:= votes$^i$;
```

Figure 3. *A posteriori* **algorithm for CAN**

median value of the receive times returned in the *ack* messages and broadcasts a $\langle$*vote*, i, p, $\Delta^{adjust,p}\rangle$ message (l. 132). When another process q receives the *vote* message, it confirms this choice by sending a similar *vote* message (l. 140). In the best case, all processes vote on the same candidate clock and this phase ends as soon as the same vote is received from every correct process (l. 142). If two or more senders receive (approximately at the same time) the confirmation of the transmission of their own *start* messages, concurrent votes for different candidates will be issued. In this case, the vote with higher rank is preferred (this means that a process may change its vote during this phase). This voting protocol is similar to the election algorithm described in [19]. It should be noted that the adjustment computed by this algorithm may be inaccurate because there is no way to match the acknowledgement (that carry the values needed to compute the adjustment) with the appropriate retransmission of the start message (that triggered the clock being adjusted). Our new protocol adresses this aspect.

The protocol is further complicated due to the possibility of process failures. In such case, some *ack* or *vote* messages will be missing. To prevent deadlock, a simple timeout mechanism is used in both phases: if an *ack* (or *vote*) message is missing after a pre-defined time limit the faulty processes are marked as failed. Note that the CAN properties guarantee the reliable and timely delivery of messages when the sender is correct. Thus, the protocol embodies a minimal fault-detection functionality that can be provided as input for a complementary membership service.

Assume that all processes issue a *start* message. According to the *a posteriori* agreement algorithm, all processes must acknowledge these start messages. Finally, in the best case, all processes vote on the same candidate clock. Thus, the protocol requires at least n start messages, each requiring $n - 1$ acknowledgements and n additional vote messages, for a total of $n + n(n-1) + n = n(n+1)$ messages. The worst-case is much higher than this: *start* messages may need to be retransmitted and all retransmissions need an acknowledgement from every process; several processes may concurrently vote for their own candidate clock, resulting in a cascade of voting messages. In the next section, we will present a "phase-decoupled" algorithm that alleviates these problems.

7. The new "phase-decoupled" *a posteriori* algorithm

The "phase-decoupled" *a posteriori* algorithm addresses the drawbacks of the basic *a posteriori* algorithm in face of the properties of the CAN network, namely the large number of acknowledgement messages, the potentially large number of concurrent votes, and the inaccuracy of clock adjustments (due to automatic retransmissions). These problems are solved using different mechanisms.

In the original *a posteriori* agreement protocol, acknowledgement messages are used for two different purposes: to disseminate reception times (used to compute the adjustment) and to ensure (and detect) reliable delivery of the *start* message. In CAN, reliable delivery is guaranteed as long as the sender remains correct. Thus, acknowledgements are only needed to disseminate reception times. The proposed modification is based on the observation that only the selected clock needs to be adjusted and that the number of messages is strongly reduced if the reception times for the other clocks are not disseminated. This can be achieved by voting on the candidate clock before the acknowledgment phase. Since reception times are no longer available at voting time, the reduction on the number of messages is achieved at the cost of "decoupling" the start phase from the adjustment computation phase (which are overlapped in the simple protocol), thus the name of the new protocol.

Decoupling these two phases has another advantage in terms of accuracy of clock synchronization. Since acknowledgements are only produced when the *start* is stable (i.e., when it has already been successfully transmitted), all disseminated reception times refer to the last correct transmission. This allows the final adjustment to be computed based on accurate reception times.

The problem of concurrent votes is a consequence of the precision of clock synchronization. Since all clocks exhibit approximately the same time, all processes will reach iT at approximately the same real time, all processes will send a start message concurrently, and so on. Although the network will enforce a serialization of all these messages, the delays incurred by such serialization are not enough to prevent concurrent executions. It should be noted that, with most existing CAN controllers, it is difficult to cancel in due time a message submitted for transmission. In the "phase-decoupled" algorithm this problem is solved using a simple Time-Division Multiplexing (TDM) approach: each processes delays its own vote by a period that is inversely proportional to its rank. This artificially extends the agreement phase but, as seen in Section 4.1, this is not the major factor on the final precision.

A pseudo-code description of the "phase-decoupled" algorithm is given in Figure 4. As in the basic algorithm, processes transmit a *start* message when their virtual clock reaches the time to resynchronize (l. 210). Unlike the basic algorithm, *start* messages do not generate acknowledgements. Instead, when enough start messages have been observed, the voting phase is immediately started (l. 220).

The voting phase is similar to that of the basic algorithm with some minor changes. One of the differences its that, instead of the final adjustment, voting messages disseminate the reception time of the associated start message (l. 226). The other difference is that each process delays the vote on

```
200  // variable description
201  //   voted$^i$: voted candidate.
202  //   $RT^i$[]: reception times.
203  //   $\Delta^i$: adjustment for this candidate.
204  //   adjuster$^i$: proposer of the adjustment.
205  //   starts$^i$
206  //   votes$^i$, and adjusts$^i$: counters
207  starts$^i$ := 0;
208  voted$^i$ := NONE;
209  adjuster$^i$ := NONE;

210  when $vc^{i-1}(NOW) = iT$
211    and voted$^i$ $=$ NONE do
212      broadcast($\langle$start, i$\rangle$);

213  when received S=$\langle$start, i$\rangle$ from $q$ do
214      $rt^{i,q}$ := $vc^{i-1}$ (NOW);
215      // start new candidate clock
216      $cc^{i,q}$ (NOW) := 0;
217      if not-duplicate(S) then starts$^i$ := starts$^i$ $+ 1$;

218  when transmission-confirmed($\langle$start, i$\rangle$)do
219      start(VoteTDMTimer, rank(p));

220  when expired(VoteTDMTimer)
221    and starts$^i$ $\geq f + 1$
222    and voted$^i$ = NONE do
223        voted$^i$ := p;
224        votes$^i$ := 1;
225        RT$^i[p]$ := $rt^{i,p}$;
226        broadcast($\langle$ vote, i, p, $rt^{i,p}$ $\rangle$);

227  when received V = $\langle$ vote, i, v, $rt^{i,v}$ $\rangle$) from $q \neq p$
228    and not-duplicate(V) do
229        if not-started(VoteTimer) then start(VoteTimer);
230        RT$^i[q]$ := $rt^{i,v}$;
231        if rank(v) $>$ rank(voted$^i$)
232          or voted$^i$ $= NONE$ then
233            voted$^i$ := v;
234            votes$^i$ := 1;
235            broadcast($\langle$ vote, i, v, $rt^{i,v}$ $\rangle$);
236        else-if voted$^i$ = v
237          and $(i = 0$
238            or $(i > 0$ and $rt^{i,v} > 0))$ then
239              votes$^i$ := votes$^i$+1;

240  when expired(VoteTimer)
241    or votes$^i$ $= N$ do
242      start(AdjustTDMTimer, rank(p));

243  when expired(AdjustTDMTimer)
244    and adjuster$^i$ = NONE do
245        adjuster$^i$ := p;
246        adjusts$^i$ := 0;
247        N := votes$^i$;
248        $\Delta^i$ := median($\forall_x RT^i[x] > 0$);
249        broadcast($\langle$ adjust, i, p, $\Delta^i$ $\rangle$);

250  when received D=$\langle$adjust, i, a, $\Delta^a$ $\rangle$ from $q \neq p$
251    and not-duplicate(D) do
252        if not-started(AdjustTimer) then start(AdjustTimer);
253        if rank(a) $>$ rank(adjuster$^i$)
254          or adjuster$^i$ = NONE then
255            adjuster$^i$ := a;
256            adjusts$^i$ := 1;
257            $\Delta^i$ := $\Delta^a$;
258            broadcast($\langle$ adjust, i, a, $\Delta^a$ $\rangle$);
259        else-if adjuster$^i$ = a
260          and $(i = 0$
261            or $(i > 0$ and $rt^{i,v} > 0))$then
262              adjusts$^i$ := adjusts$^i$+1;

263  when expired(AdjustTimer)
264    or adjusts$^i$ = N do
265      $vc^i$ := $cc^{i,voted^i}$ $+ \Delta^i$;
266      N:= adjusts$^i$;
```

Figure 4. The "phase-decoupled" algorithm

its own clock by an amount of time that is dependent of its rank (l. 218). If correct, the process with higher rank will propose its own clock first and the other processes will confirm this vote. Only in the case of failure, the process with succeeding rank will issue a different vote message. As before, timeouts are used to terminate the voting phase in case of missing votes.

At the end of the voting phase, all correct clocks have agreed on the same candidate clock (l. 240). However, different processes can have different sets of votes. Note that if a process fails during the transmission of its own vote the CAN does not ensures the reliable delivery of this message. Thus, processes cannot apply a local function to compute the adjustment for the selected clock: an additional agreement phase needs to be performed. This second phase is quite similar to the voting phase. The process with higher rank will locally compute the adjustment and disseminate it using an *adjust* message (l. 249). This message needs to be confirmed by all correct processes (l. 258). Again, in case of failure of the process with higher rank, all other processes would, in turn, compute and propose an adjustment for the selected clock.

8. CAN message priorities

The CAN priority based arbitration scheme allows the assignment of a different priority to each protocol message. This section discusses how this feature can be exploited to promote faster protocol termination. Our proposal assumes that the message identifiers are constructed using three fields, namely *protocol priority*, *message priority* and *rank priority*.

The *protocol priority* field is mapped onto the high priority bits of the message identifier and reflects the relative priority of clock synchronization with regard to other activities in the system. A positive feature of our algorithm is that, as long as enough bandwidth is reserved to execute the protocol in due time, the use of the higher CAN priorities is not required in order to achieve good precision.

The *message priority* field reflects the relative priorities of protocol messages with regard to each other. Here, message urgency increases as the algorithm execution approaches its final phase (that is, the *adjust* messages have higher priority than the *vote* messages, which in turn have higher priority than *starts*). The rationale is that, as soon an a new protocol phase is started, messages regarding previous phases become obsolete and should be given a lower priority (the time-division multiplexing scheme minimizes the number of these messages).

Finally, the *rank priority* field ensures that messages of the same type are given a priority which reflects the rank of their senders. This means that the vote (or adjust) from the process of higher rank (which is bound to win the election) is given a higher priority than other votes.

The use of the CAN arbitration scheme complements the time-division multiplexing technique when, due to processing or network transient overloads, requests from different processes compete for network access. In the performance section, this CAN-based message ordering scheme was used in all simulations.

9. Performance

This section discusses the performance of the "phase-decoupled" *a posteriori* algorithm in terms of number of messages exchanged, precision and accuracy preservation.

9.1. Number of messages

The minimum number of messages generated by an execution of the algorithm is n starts, n votes and n adjusts, for a total of $N_{min} = 3n$ messages[4]. Worst-case values depend on the number of faults and on system configuration. If all nodes configured to generate a start message reach the synchronization point approximately at the same time, the first phase of the algorithm generates n messages. Nodes should then vote for electing a candidate clock. In the worst-case, each node begins voting on its own clock, changing afterwards the vote, successively, to higher rank clocks. This means each node generates a number of messages equal to its rank numbering; the sum of the messages generated by all the nodes represents the sum of the first n terms of an arithmetic sequence with ratio one. This model applies also to the adjustment phase. Thus:

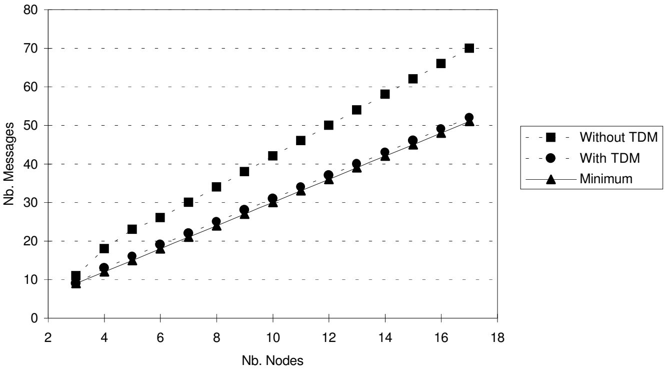

Figure 5. Variation of generated messages with number of nodes

$$N_{max} = n + \frac{n}{2}(1+n) + \frac{n}{2}(1+n) = n^2 + 2n$$

Naturally, the average number of messages exchanged on a typical execution environment is much less than N_{max}. The purpose of the time-division multiplexing scheme on the voting and adjust phases is to approximate the average number of messages exchanged to N_{min}. To evaluate the effectiveness of our approach, we have used the MIT LCS Advanced Network Architecture group's network simulator (NETSIM [6]). In this experiment, we have considered the CAN 2.0B @ 1 Mbps and we have set the time-division multiplexing timers for a value of $400\mu s$. This value is 2.5 times bigger than the time required to propagate a message ($160\mu s$) but is still small enough to have a minor impact on clock precision in case of process crashes (each timeout adds $400\mu s$ to the agreement phase, thus even two consecutive failures would affect the precision in less than $10^{-3}\mu s$). The results for a fault-free scenario are shown in Figure 5, where the use of the time-division multiplexing method is compared with a scenario that does not use such technique. It is clear that the number of messages generated in the former case closely approximates the minimum number of messages required by the algorithm.

9.2. Precision and accuracy preservation

The precision achieved by an algorithm based on the *a posteriori* agreement technique was proven [23] to be limited by:

$$\delta_{vi} \geq (1+\rho)\Delta\Gamma_{tight} + 2\rho\Gamma_{agreem}^{max}$$

Additionally, at each resynchronization there is a potential accuracy loss of, approximately, $(1+\rho)\Delta\Gamma_{tight}$ (see [23] for exact formulas).

[4] Actually, through a network management interface, it is possible to load a configuration where only $2f + 1$ processes are required to send a start message. However, to simplify the explanation, we have selected a configuration where all nodes run the same code.

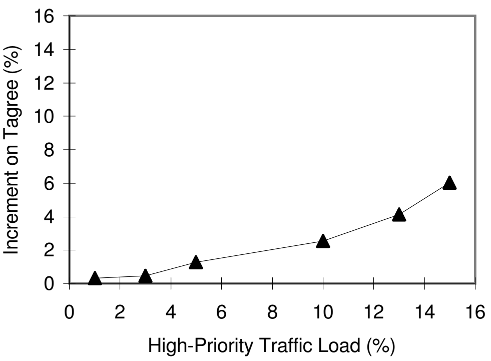

Figure 6. Average agreement time versus network load

Intended precision (μs)	T (s)	Worst-case accuracy loss ($\mu s/hour$)
50	20	3600 + 1800
100	45	3600 + 800
200	95	3600 + 370
300	145	3600 + 240

Table 1. Resynchronization interval

The parameters needed to compute results are: $\Delta\Gamma_{tight}$, which depends on maximum network propagation delay variance and on the maximum variance of timestamping processing overheads that can be observed at any correct receiver; ρ, that depends on the specifications and operational condition of the clock; and Γ^{max}_{agreem} which depends on the number of tolerated faults, resulting number of messages exchanged, configuration of the time-division multiplexing timers using for voting phases, and on background traffic of higher priority.

To evaluate the impact of the high-priority traffic load on the time required to reach agreement we have run a series of simulations of our protocol under different traffic loads. The results are depicted in Figure 6. It can be seen that even a traffic load of high-priority background traffic in the order of 15% has a small impact on the agreement time (which in turn has only a minor impact on clock precision).

Table 1 presents the resynchronization interval required for different values of worst-case precision. It also shows the maximum accuracy loss per hour of operation using such a resynchronization interval. We have considered a value of $10\mu s$ for $\Delta\Gamma_{tight}$ (a conservative value) and a value of $\rho = 10^{-6}$, common for crystal based clocks. The worst-case accuracy loss has two components, one that depends exclusively of the drift of physical clocks (without external synchronization, this is also the best accuracy preservation achievable [18]), and other that represents the protocol-induced accuracy loss. If required, the *a posteriori agreement* technique can be extended to perform external synchronization [23]. Nevertheless, it is important to exhibit a small accuracy loss even when external synchronization is used (this makes the system robust to transient faults of the external source).

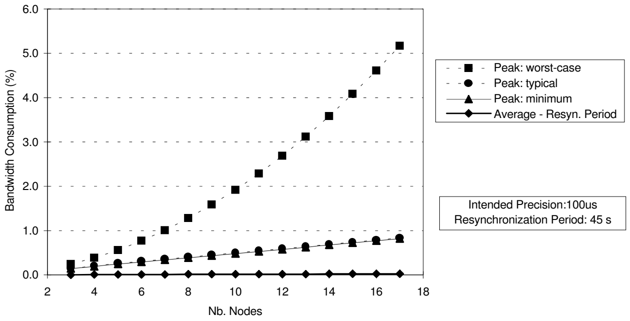

Figure 7. Bandwidth consumption (T=$45s$)

As it can be seen, to offer a precision in the order of $100\mu s$ (which is less than the average time required to disseminate a message in the CAN) clocks need to be synchronized only once every $45s$ and the protocol induced accuracy loss is much smaller than the accuracy loss due to the drift of the physical clocks. Figure 7 shows the CAN bandwidth consumption due to the protocol traffic in this scenario for different number of nodes. Since clock synchronization traffic exhibits a bursty behavior, the figure shows worst-case, minimum and typical values of bandwidth consumption during the execution of the protocol. Again, it should be noted that typical values are much lower than the theoretical worst-case value. The figure also shows the average bandwidth consumption over the entire synchronization period (lower line); naturally these values are very small.

10. Conclusions and future work

Designing clock synchronization protocols for CAN is justified by the increasing use of this network in industrial automation applications. Our work departures from a straightforward implementation of the *a posteriori* algorithm on CAN, which is obtained by enriching the generic algorithm described in [23] with a CAN-specific agreement protocol. This approach has several limitations, namely the large number of messages exchanged and the low accuracy of clock synchronization. It is interesting to observe that an optimization for local area networks (the use of acknowledgements for the dual purpose of reliability and clock value collection) actually degrades the performance on CAN. A new "phase-decoupled" *a posteriori* agreement algorithm that carefully addresses the limitations of CAN was presented. The algorithm offers a tight precision and good accuracy with a reasonable cost. For instance, to ensure a precision of $100\mu s$, clocks have to be synchronized only once every $45s$ and the accuracy loss is only in the order of $4.2ms$ per hour.

It was shown that the a posteriori agreement technique can be combined in an hierarchical manner with other synchronization algorithms to provide clock synchronization beyond the borders of a single broadcast segment [23]. A similar approach could be used here to synchronize several CAN buses. The integration of this technique with the approach suggested in [3], would also allow to support both internal and external synchronization.

References

[1] E. Christer, H. Thane, and M. Gustafsson. A communication protocol for hard and soft real-time computer systems. In *Proc. of the European Workshop on Real-Time Systems (EURWRTS)*, L'Aquila, Italy, Jun. 1996.

[2] F. Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3(3):146–148, 1989.

[3] C. Fetzer and F. Cristian. Integrating external and internal clock synchronization. *Journal of Real-Time Systems*, 12(2), 1997.

[4] M. Gergeleit and H. Streich. Implementing a distributed high-resolution real-time clock using the CAN-Bus. In *Proc. of the 1st International CAN-Conference*, Mainz, Germany, Sep. 1994.

[5] J. Halpern, B. Simons, R. Strong, and D. Dolev. Fault-tolerant clock synchronization. In *Proceedings of the 3Rd ACM Symp. on Principles of Distributed Computing*, pages 89–102, Vancouver Canada, Aug. 1984.

[6] A. Heybey. The network simulator version 2.1. Technical report, M.I.T., Sep. 1990.

[7] A. Hopkins, T. Smith, and J. Lala. FTMP - A highly reliable fault-tolerant multiprocessor for aircraft. *Proceedings of the IEEE*, 66(10):1221–1240, Oct. 1978.

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

[9] H. Kopetz and W. Ochsenreiter. Clock syncronization in distributed real-time systems. *IEEE Transactions on Computers*, C-36(8):933–940, Aug. 1987.

[10] H. Kopetz and W. Schwabl. Global time in distributed real-time systems. Technical Report 15/89, Technische Universitat Wien, Wien Austria, Oct. 1989.

[11] C. Krishna, K. Shin, and R. Butler. Ensuring fault tolerance of phase-locked clocks. *IEEE Transac. Computers*, C-43(8):752–756, Aug. 1985.

[12] J. Lundelius and N. Lynch. An upper and lower bound for clock synchronization. *Information and Control*, (62):190–204, 1984.

[13] P. Ramanathan, K. Shin, and R. Butler. Fault-tolerant clock synchronization in distributed systems. *IEEE, Computer*, 23(10):33–42, Oct. 1990.

[14] Robert Bosch GmbH. *CAN Specification Version 2.0*, Sep. 1991.

[15] J. Rufino and P. Veríssimo. A study on the inaccessibility characteristics of the Controller Area Network. In *Proc. of the 2nd International CAN Conference*, London, England, Oct. 1995.

[16] J. Rufino, P. Verissimo, G. Arroz, C. Almeida, and L. Rodrigues. Fault-tolerant broadcasts in CAN. In *Digest of Papers, The 28th IEEE International Symposium on Fault-Tolerant Computing*, Munich, Germany, Jun. 1998.

[17] F. Schneider. Understanding protocols for byzantine clock synchronization. Technical report, Cornell University, Ithaca, New York, Aug. 1987.

[18] T. Srikanth and S. Toueg. Optimal clock synchronization. *Journal of the Association for Computing Machinery*, 34(3):627–645, Jul. 1987.

[19] A. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 1992.

[20] K. Tindell and A. Burns. Guaranteeing message latencies on Controler Area Network. In *Proc. of the 1st International CAN Conference*, Mainz, Germany, Sep. 1994.

[21] K. Turski. A global time system for CAN networks. In *Proc. of the 1st International CAN Conference*, pages 3.2–3.7, Mainz, Germany, Sep. 1994.

[22] P. Veríssimo and L. Rodrigues. A posteriori agreement for fault-tolerant clock synchronization on broadcast networks. In *Digest of Papers, The 22nd International Symposium on Fault-Tolerant Computing*, Boston, USA, Jul. 1992. IEEE.

[23] P. Veríssimo, L. Rodrigues, and A. Casimiro. Cesiumspray: a precise and accurate global time service for large-scale systems. *Journal of Real-Time Systems*, 12(3):243–294, 1997.

[24] K. Zuberi and K. Shin. Non-preemptive scheduling of messages on Controller Area Networks for real-time control applications. In *Proc. of the IEEE Real-Time Technology and Application Symposium*, pages 240–249, Chicago, Illinois, USA, May 1995. IEEE.