A Priority-Based Consensus Protocol

George Lima[1], **Alan Burns**[2]

[1]Distributed Systems Lab (LaSiD), Department of Computer Science
Federal University of Bahia, Brazil

[2]Real-Time Systems Group, Computer Science Department
The University of York, UK

gmlima@ufba.br, burns@cs.york.ac.uk

Abstract. *Consensus is a basic agreement problem whose solutions are fundamental for building fault-tolerant distributed systems. Consensus for real-time systems is usually designed under strong timing assumptions, which state that there are upper bounds on both processing and message transmission times. Since violating these bounds may compromise safety, such systems are usually implemented based on pessimistic bounds. In this paper the consensus problem is revisited taking into consideration that the system provides a priority-based communication network. It is shown that for those systems the message transmission time bound can be relaxed so that all but the highest priority messages may be arbitrarily delayed or even lost. The derived consensus protocol can also cope with process crashes, processes may start executing the protocol asynchronously, and consensus is reached within a known bounded time. These characteristics make the proposed solution very interesting to real-time systems. The protocol is proved correct and analyzed by simulation. Since there are a number of priority-based networks, the results presented here have both theoretical and practical implications.*

1. Introduction and Motivation

Consensus is a fundamental problem in distributed systems and, due to its importance, has been extensively studied for decades. Informally, the problem can be stated as follows. A set of distributed processes (interconnected by a communication network) have their own estimated values and have to agree on a single value. Despite this simple formulation, a solution for this problem is hardly simple and may even not exist for some systems. For example, it has been shown that in asynchronous systems, where there are no bounds on processing speeds and communication delays, it is impossible to solve consensus deterministically if processes are subject to faults [Fischer et al. 1985].

For real-time systems, a kind of system that requires that tasks are executed within a maximum time, solutions for the consensus problem have to be timely. This means that once a process starts executing a consensus protocol, it has to reach agreement by a given maximum time. For those systems, consensus protocols are usually based on the synchronous model of computation. This means that the consensus protocol is designed assuming that there are bounds on both processing speeds and communication

[*]Dr. Lima is supported by CNPq (grant 475851/2006-4) and FAPESB (grants 7630/2006 and 3381/2005).

delays [Lynch 1996]. In other words, in these cases transmitted messages are assumed to be delivered and processed within a known maximum time. Although such an assumption allows one to design consensus protocols that terminate in bounded time, such solutions also imply that if some message does not arrive on time during the execution of the protocol, say, the processes may agree on different values, violating safety. Because of this, the assumed bounds are usually derived pessimistically, which has performance implications. For example, consider a given communication network. In order to derive the message transmission bounds, one may be interested in taking into account possible message retransmissions due to communication errors. Since errors are rare events, the derived bounds tend to be too pessimistic for most execution scenarios.

Another relevant point regarding the assumed communication bounds is that usually a general communication network is considered when designing synchronous consensus protocols. Nonetheless, there are some networks that are priority-based, which means that higher priority messages are transmitted before lower priority ones. This is the case for Controller Area Network (CAN) [ISO-11898 1993], some real-time Ethernet buses [Decotignie 2001]. For those networks, even taking into account possible communication errors, higher priority messages take less than lower priority messages to be delivered. In such cases assuming the maximum message transmission time (for the lowest priority) may degrade the system performance even more. Also, for some real-time networks, which have low available bandwidth, such performance degradation may be not allowed.

Figure 1 represents scenarios where message retransmissions due to errors take place. These scenarios illustrate possible message exchanging over an interval of time during the execution of a given protocol $\mathcal{P}$ over a priority-based network, such as CAN. The vertical dotted line indicates the point in time at which the process cannot wait for income messages any more in order not to violate the timeliness requirement. In scenario (a), which is a fault-free scenario, the correctness of protocol $\mathcal{P}$ is not compromised because all messages are delivered on time. Thus, the processes can make progress in their computation before the dotted line. The same is not true for scenario (b), where an error caused the retransmission of a message, which caused the lowest priority message to be delayed beyond the assumed transmission delay bound. This scenario evidences the vulnerability of $\mathcal{P}$ regarding assumed communication synchronism (under timing faults). Indeed, $\mathcal{P}$, which was designed relying on the fact that *any* message is delivered within a known interval of time, may fail due to a late message.

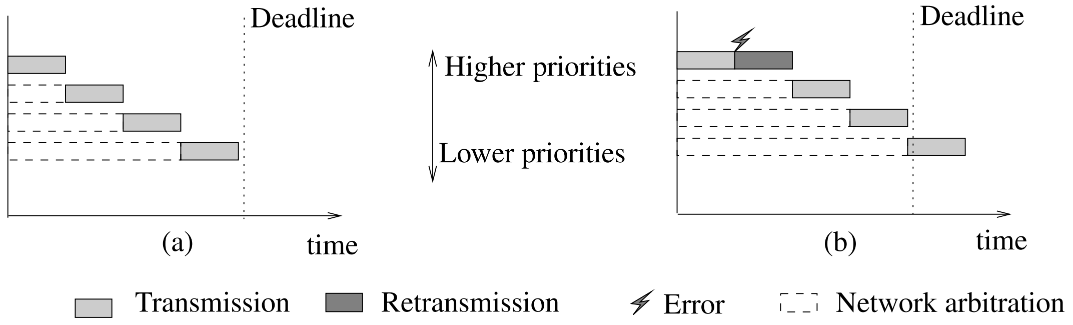


Figure 1. Message transmission over a priority-based network.

In this paper the consensus problem is addressed taking into consideration that the system provides a priority-based network. No specific network is assumed. Instead, we require only that the communication subsystem is able to schedule messages for transmission based on their priorities. Since several communication networks have this property, the solution proposed here can be instantiated to actual systems.

The proposed protocol solves the consensus problem within a maximum known time. This is very important to industry and real-time applications, the ones that usually make use of the kind of assumed communication model. They have higher reliability constraints and are specified in terms of time restrictions (deadlines). The protocol requires only that some messages (not all) are on time. As a consequence, the proposed protocol is more resilient to timing communication faults when compared to the protocols that consider the purely synchronous communication model. Good characteristics of the proposed protocol include its simplicity, its ability to cope with process crashes and its resilience to some late or omitted messages.

The remainder of the paper is organised as follows. The computation model is described in the next section. Section 3, after giving a more precise definition of the consensus problem, describes the proposed solution. Then, in section 4, its proof of correctness is presented. Performance analysis is discussed in section 5 and related work is sumarised in section 6. Finally, conclusions and some directions for future work are presented in section 7.

2. Computational Model

The assumed system consists of a set of distributed nodes linked to each other by means of a priority-based communication network. The computation in nodes is carried out by processes. As we are interested only in the set of processes that are involved in consensus protocol, we refer to the system as the set of these processes. We also assume that these processes are statically allocated to the nodes and exchange information between each other only by broadcasting/receiving messages across the network. Processes only fail by crashing. If a process crashes at time t, it stops sending messages indefinitely from t.

The assumed synchronism on communication is defined in relation to the priorities of the transmitted messages. A more precise description of the synchronism assumption makes use of some definitions. Firstly, let δ be the worst-case transmission time of the highest priority message in the system. The value of δ is a function of several factors such as electric propagation delay, buffer manipulation etc. Also, it must include the necessary time to retransmit messages due to transmission errors. Ways for deriving the value of δ can be found elsewhere. For example, for CAN, a priority-based network, the reader can refer to other sources [ISO-11898 1993] [Davis et al. 2007]. For the purposes of this paper, it is enough to see δ as an input value which expresses the communication delay in the system regarding the highest priority message.

If only one message is being transmitted at a time (sequential transmission), it is assumed that δ is the maximum transmission delay for any transmitted messages in the system. Indeed, such a message would be the highest priority one in this scenario. In other words, in this special case, the communication would be synchronous. However, general *concurrent* transmission of messages may be carried out. Any two messages m and m' are concurrently transmitted if $\mathsf{m} \neq \mathsf{m}'$ and their sender processes broadcast them within δ apart from each other.

From the definition above, δ actually represents a bound on the transmission of those messages whose priorities are higher than the priority of any other message *concurrently* transmitted. There may be a chain of concurrent transmitted messages. In this case, δ is assumed to hold only for the highest priority message from the chain. Hence, if (a) no other message is concurrently transmitted with some message m or (b) m is the highest priority concurrently transmitted message, then m is assumed to be delivered at all correct processes within δ unless a transmission error takes place. If either (a) or (b) is not true, no assumption is made regarding the transmission time of m. Thus we can assume that there is a known maximum transmission delay, δ, for any transmitted message provided: that it has the highest priority among all messages that are concurrently transmitted with it; and that it does not suffer transmission errors.

Note that transmission errors, due to possible temporally electromagnetic interferences, say, may prevent some messages from being delivered at some processes. We assume that no more than f such errors take place during the execution of the consensus protocol. Also, when a transmission error takes place, the communication subsystem may retransmit the message. Hence, not only may messages be omitted at some destinations but also there may be scenarios where some messages are received at some processes more than once. Both cases may lead to possible inconsistences. For example, a process that does not receive a given message delivered elsewhere may choose its decision value differently, violating system safety. As will be seen, the protocol is resilient to such inconsistent omissions and duplications. Furthermore, we do not assume partitioning in the network nor consider messages to be arbitrarily created by the network.

It is important to emphasize that the synchronism assumed here is much less strict than assuming a completely synchronous communication. In fact, instead of relying on the existence of a known bound on message transmission delays for *any* transmitted message, it just states that this bound holds for *some* messages. Even though, the assumed bound (for the highest priority message) may be violated (at most f times) in the presence of communication errors.

Each node in the system is assumed to be equipped with a local clock, which can be accessed by the processes. Also, it is assumed that the drift rates of these clocks are bounded by a known constant, denoted ρ. This assumption is in line with the characteristics of most hardware nowadays, where clock drift rates are very small. Typical values of ρ have shown to be in the order of 10^{-6}. Similar assumptions on local clocks are common [Kopetz 1998] [Ostrovsky and Patt-Shamir 1999].

We also assume that a processing speed bound exists and is known. This bound can be derived in real-time systems using scheduling analysis [Burns and Wellings 2001]. For the purposes of this paper it is sufficient to define α as the worst-case response time spent on the internal computation of correct (i.e. non-crashed) processes that execute the consensus protocol. The meaning of α will be clearer later on, when the protocol is described.

3. The Consensus Protocol

Before presenting the proposed protocol, a more precise definition of the problem is given. The consensus problem is usually specified in terms of the three properties, termination, validity and agreement. In this paper we are interested in a specification that ensures a

```
procedure consensus($v$)
    send $(v)$ at priority 0 to itself
    Let $\mathsf{m} = (\mathsf{est}_h)$ be the highest priority received message
    $\mathsf{est}_i \leftarrow \mathsf{est}_h$
    $\mathsf{r} \leftarrow \max(1, \lceil \frac{\mathsf{pr(m)}}{n} \rceil)$
    while $\mathsf{r} \leq f + 1$ do
        SetTimer($\Delta$)
        broadcast $(\mathsf{est}_i)$ at priority $n(\mathsf{r} - 1) + i$
        wait until $[\mathsf{Timer}() \vee (\forall p_j \in \Pi :$ received $\mathsf{m}$ from $p_j$ s.t. $\mathsf{pr(m)} > n(\mathsf{r} - 1))]$
        Let $\mathsf{m} = (\mathsf{est}_h)$ be the highest priority received message
        $\mathsf{est}_i \leftarrow \mathsf{est}_h$
        $\mathsf{r} \leftarrow \max(\mathsf{r} + 1, \lceil \frac{\mathsf{pr(m)}}{n} \rceil)$
    endwhile
    return($\mathsf{est}_i$)
```

Figure 2. The proposed consensus protocol.

bounded termination time, which we call the timed consensus problem:

- *Bounded termination*: Every correct process decides some value within a bounded known time.
- *Validity*: If a process decides v, then v was proposed by some process.
- *Agreement*: No two processes decide different values.

Now, let $\Pi = \{p_1, p_2, \ldots, p_n\}$ be the set of processes that want to agree on a common value with each other complying with the above definition. In order to perform the consensus protocol, any process $p_i \in \Pi$ calls the primitive consensus(v), where v is its proposed value. This primitive is given in figure 2, which is described in detail later.

The proposed protocol tolerates $n - 1$ process crashes and up to $f \geq 0$ message omissions. Also, it is tolerant to any number of inconsistent message duplication, as long as the assumed bound δ on communication delays holds. Consensus is achieved after executing $f + 1$ communication steps (rounds), during which correct processes exchange their estimated values. All messages are sent with distinct priorities.

3.1. Protocol Overview

The protocol is divided into three parts, defined by lines 1-4, 5-12 and 13, respectively. Part 3 is simply the finishing of the protocol, where the consensus value is returned. The general idea behind the other parts is to make the processes accept the values carried by the highest priority messages they receive. To do so, processes perform actions to send messages with their estimated values. Then the highest priority incoming message is selected to update the estimated value of the receiving processes. Thus, a process that broadcasts the highest priority message may 'impose' its estimated value on all correct processes in cases where such a message is not inconsistently omitted. By inconsistent omission we mean that a message may be received by only a subset of the processes.

The purpose of lines 1-4 is to select the initial estimated values of processes upon starting. These values are set either to their proposed values or to the values carried by the highest priority message they receive by the moment they start executing the protocol (processes are not required to start the protocol synchronously). In order to do so, processes send (locally) their proposed values to themselves. These messages are represented as if they had been transmitted at the lowest priority level (line 1). This operation is internal and so does not use communication resources. If there is no other received message, the selected estimated values will be their proposed values (lines 2-3). Otherwise, they accept the value carried out by the highest priority received message.

As can be seen in line 4, before moving on, processes update their round number. If some message from some round r' is received by the time a process p_i starts executing the protocol, p_i moves to round $\mathsf{r} = \mathsf{r}'$ skipping all rounds $1, \ldots, \mathsf{r}' - 1$. Note that $\lceil \frac{\mathsf{pr(m)}}{n} \rceil$ gives the round in which m was broadcast. This avoids the need for transmitting round numbers, which saves network bandwidth.

The idea of updating r in line 4 is to prevent p_i from executing unnecessary rounds. As messages are selected by their priorities and their priorities increase proportionally to round numbers, messages from rounds inferior to r' are irrelevant.

Lines 5-12 are the main part of the protocol, which consists of $f + 1$ rounds. Regarding each round r, each process p_i may either broadcast its message in r or skip r. It broadcasts a message in r if no message broadcast in some round $\mathsf{r}' > \mathsf{r}$ has been received by the time p_i starts r. Otherwise, p_i skips r without broadcasting any message in r.

A message broadcast by p_i in some round r is transmitted with priority $n(\mathsf{r} - 1) + i$ to all processes in Π (including itself). This priority function gives higher priorities to messages broadcast in higher rounds and ensures that: different processes do not broadcast messages with the same priorities; and processes that are ahead in their processing have more chances to get their messages through. Thus, this strategy allows processes that finish earlier to 'impose' their decision values on the others.

After broadcasting its message in r, p_i waits for incoming messages from all other processes that are processing rounds greater than or equal to r. Some of these messages may not arrive by the expected time. Others may be from processes that are still in earlier rounds or from those already crashed. To avoid waiting too long, p_i waits a maximum time, Δ. The function $\mathsf{SetTimer}(\Delta)$ (line 6) sets this timeout. When the timeout expires the function $\mathsf{Timer}()$ returns true. The value set to Δ must be big enough to avoid unsafe executions of the protocol and small enough to maximise its performance. A discussion on criteria to choose adequate values for Δ will be presented in section 3.2.

Upon receiving all the expected messages (from the other processes) or upon the expiration of the timeout, p_i selects the highest priority received message. Let r' and est_h be, respectively, the round the selected message was broadcast and the estimated value it contains. Then, p_i makes its estimated value equal to est_h and moves forward. If $\mathsf{r}' = f + 1$, p_i returns its estimated value. Otherwise, it goes to the next round.

Line 11 has a similar meaning to line 4. If p_i, which finishes round r, receives by then a message broadcast in some round $\mathsf{r}' > \mathsf{r}{+}1$, p_i skips all rounds $\mathsf{r}{+}1, \mathsf{r}{+}2, \ldots, \mathsf{r}'{-}1$. Otherwise, p_i moves to round $\mathsf{r} + 1$. Due to faults, however, there may be some round r where the timeout expires before p_i receives any message. In this case, to avoid being blocked in the same round, p_i updates its round number to $\mathsf{r} + 1$.

Given that the assumption on communication delay δ holds, the timeout is set

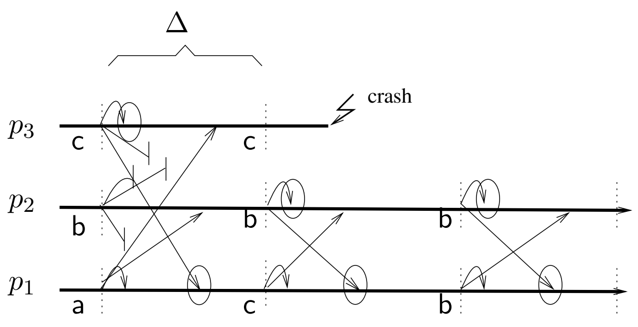


Figure 3. Illustration of the protocol.

appropriately and if there is no inconsistent omission concerning the process which sends the highest priority message in a given round, say p_i, all correct processes will timely receive the message sent by p_i. Then, these processes will update their estimated values to p_i's estimate within a bounded time after starting the protocol (by assumption, within α time units). However, the message sent by p_i may suffer inconsistent omission. This may lead to situations in which only a subset of Π receive p_i's message. As up to f messages may be inconsistently omitted, $f + 1$ rounds suffices to solve consensus.

As an illustration of the protocol, consider figure 3, where a scenario with three processes $\Pi = \{p_1, p_2, p_3\}$ which executes the protocol for $f = 2$. Suppose that their proposed values are a, b and c, respectively. The selection of the highest priority message in each round by each process is indicated by the circles. Rounds are delineated by vertical dotted segments on the time line. As can be seen, the message sent by p_3 is inconsistently omitted at p_2 in the first broadcast. Because of this, the highest priority message p_2 receives in the first round is its own message while p_1 receives p_3's message. Hence, in the second round the estimated values of processes p_1 and p_2 are c and b, respectively. At the end of the second round, though, they agree on a common estimated value and in the third round they can decide on it. This is the value sent in the highest priority message in the last two rounds. Note that inconsistent message duplication does not compromise the protocol behaviour. Indeed, provided that the last retransmission (at network level) of such a inconsistently duplicated message gets through within δ from the time the message was first broadcast, all correct processes will receive it by the end of the round.

The protocol also is able to cope with non-synchronous rounds. For illustration, modify the given example so that p_1 starts the execution of the protocol after p_2 and p_3 finish theirs. After part 1, the highest priority message received at p_1 is the last message sent by p_2, which contains the consensus value already. This message is selected by p_1, which updates its round number and its estimated value accordingly. Then, p_1 moves on to execute round $f + 1$, where it broadcasts b as its estimated value. At the end of this round, p_1 selects its own message and returns the decision value.

3.2. Determining the Round Duration

If the maximum round duration, determined by the value of Δ, is too short, processes may not have enough time to receive any broadcast message during the execution of the protocol. In this case all correct processes may individually decide on their own proposed values, which violates agreement. On the other hand, if the value of Δ is too big, the system may suffer performance degradation. Thus, Δ must be chosen so that correctness

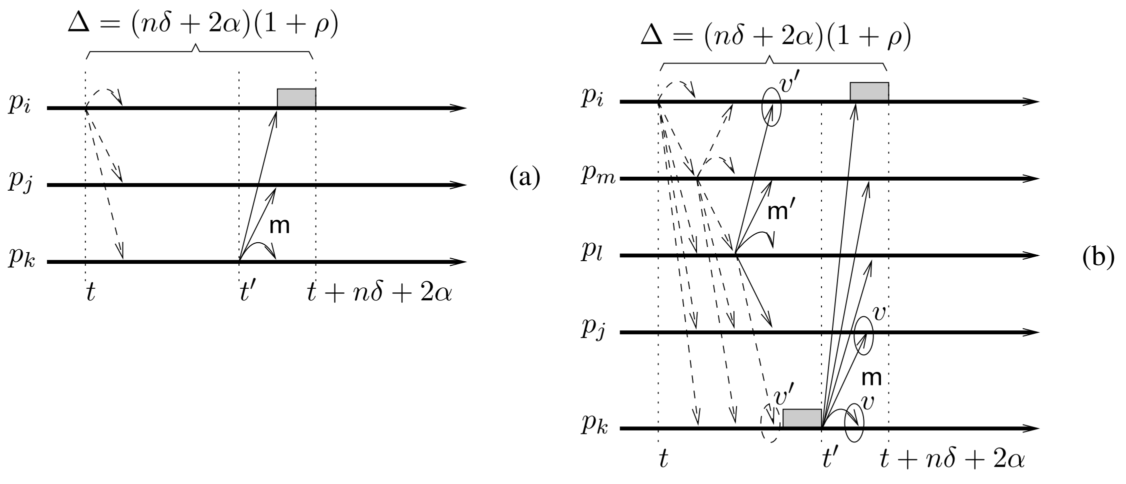


Figure 4. Illustration of agreement despite asynchronous execution.

is guaranteed without compromising performance. To do so, one has to account for the time necessary to deliver highest priority messages (δ), local processing time (α) and local clock drift rate (ρ).

Before deriving the value of Δ, some notation is needed. Let $V_i(\mathsf{r})$ denote the view (*i.e.* the estimated value) of a correct process p_i at the end of round r if $1 \leq \mathsf{r} \leq f+1$ and let $V_i(0)$ be the first estimated value set by p_i (in line 3) of the protocol. If $\mathsf{r} \geq 1$ is a skipped round, $V_i(\mathsf{r}) = V_i(\mathsf{r}-1)$.

One of the properties of the protocol is that if Δ is set adequately, then the set of correct processes can reach a *common view* on some estimated value. They reach a common view in some round r if each correct process has the same estimated value by the end of r. The round r in which p_i has a common view is called the common view round and its estimated value $V_i(\mathsf{r})$, the common view value.

The following lemma states the conditions under which processes reach a common view.

Lemma 3.1. *Let Π be a group of $n > 0$ processes that perform the consensus protocol described in figure 2. If* r *is a round of the protocol in which the highest priority message does not suffer omission and $\Delta \geq (n\delta + 2\alpha)(1+\rho)$, the set of correct process reaches a common view in* r.

Proof. If there is only one correct process that finishes r, then the proof is straightforward. Assume that there are at least two correct processes and let t and t' be the time they broadcast their estimated values in r, respectively. Also, without loss of generality, consider that no other process starts r before t or after t'. There are two cases to be considered: (a) $t' - t \leq (n-1)\delta + \alpha$ and (b) $t' - t > (n-1)\delta + \alpha$. These cases are represented in figure 4 (a) and (b), respectively.

Case (a). Let p_k be the process that broadcasts m, the highest priority message broadcast in r. As the interval between the first and the last broadcast is at most $(n-1)\delta + \alpha$, t' is the latest time that p_k can broadcast its message. From the assumed model, and from the definition of r, m arrives at all correct processes within δ. As processes do not spend more than α time units on local computation, at most at $t' + \delta + \alpha$ all correct processes that do not finish r before $t' + \delta + \alpha$ must have received m and set their estimated value to a common value at the end of r. Recall that processes do not receive a message different from those that are transmitted.

From the definition of t, no process finishes r before $t + n\delta + 2\alpha \geq t' + \delta + \alpha$ if it does not receive the messages from all other processes by then. This means that all correct processes receive m and set their estimated values at the end of round r to the value carried by m. Note that this follows even if p_k had broadcast m earlier. Therefore, a common view is reached, as required.

Case (b). Assume by contradiction that no common view is reached in r. This means that there are at least two processes that finish r with different estimated values. Thus, by the definition of r, there is some process that finishes r before receiving the same set of messages as other correct processes. More specifically, such a process misses, by the end of r, the highest priority message, m say, received by some other process, see figure 4 (b). Let v and v' be the estimated values of the processes that end up round r having and not having received m, respectively. In the figure, the estimated values are indicated by the respective letters above the circles.

Note that by the protocol, correct processes that do not receive m by the end of r wait at least $n\delta + 2\alpha$ for m. Also, from the definition of t no correct process that misses m by the end of r finishes r before $t + n\delta + 2\alpha$. Hence, if a process does not receive m before finishing r, then the processes that start r at t must also have missed it. Without loss of generality assume that p_i is one process that misses m in r (as illustrated in the figure). Thus, p_i must also have finished r with a different estimated value, v'.

From the definition of m and from the assumed model, m must have been broadcast not before $t + (n-1)\delta + \alpha$ by some process p_k. Thus, assume that m was broadcast at the latest possible time, t'. Also, it is clear that the highest priority message received by p_i, m' say, must have arrived at p_k after $t + (n-1)\delta$. Otherwise, p_k would have updated its estimated value to v' before broadcasting m. Such a scenario is illustrated in figure 4 (b), where both m' and its selections are represented by a dashed arrow and a dashed circle, respectively.

Let p_l be the process that broadcasts m' (see the figure for illustration). Similarly, the message from p_i could not have reached p_l nor p_k before they broadcast m' and m, respectively. If this was the case, both p_l and p_k would have set their estimated value (before they broadcast their messages) to the value carried by the message from p_i. Thus, a higher priority message must have been transmitted concurrently with the message from p_i. Let p_m be the process that broadcasts such a message. Recall that no other process broadcasts messages in r before p_i.

For the same reasons as with the message from p_i, the message from p_m could not have reached either p_k or p_l before they broadcast their messages in r. Thus, another process must have broadcast its message concurrently with a higher priority message from p_m. In order to keep on constructing this chain of concurrently transmitted messages in the interval $[t, t + (n-2)\delta]$, more than $(n-2)$ messages are necessary. Including m and m', this would mean that there have been more than n messages broadcast in r. This is a contradiction since there are at most n correct processes and by the protocol processes broadcast at most one message per round. This is indicated in the figure, where it can be seen that the message from p_l must have arrived before $t' - \alpha$. Therefore, the common view in r follows. □

Note that in lemma 3.1 there is no assumption about the time processes start their rounds. If one assumes a certain synchronism between round execution, a much lower value of Δ is needed.

4. Proof of Correctness

In order to prove the correctness of the proposed protocol, one has to show that it satisfies bounded termination, validity, and agreement properties. These properties are shown by the following lemmas.

Lemma 4.1 (Bounded termination). *Each correct process in* Π *decides some value within a known maximum period of time.*

Proof. By the protocol, it is clear that no correct process can be indefinitely blocked because: (a) for each time a process awaits messages in some round, the waiting time is bounded by Δ time units; (b) the update of the round number in line 11 guarantees that no round is executed more than once; and (c) there are $f + 1$ (skipped and not skipped) rounds for each process. Therefore, each correct process terminates the execution of the protocol at most within $\Delta(f + 1)$ units of time from the time it starts. □

Lemma 4.2 (Validity). *If a process in* Π *decides* v*, then* v *was proposed by some process in* Π.

Proof. By the algorithm and because of the fact that processes may fail only by crashing, a process p_i can only update est_i to either its proposed value (line 3) or to some value carried by some message received during the execution of the protocol (lines 3 or 11). As by assumption messages are neither arbitrarily created nor corrupted, est_i is either proposed by p_i or by some other process in Π. Therefore, any decided value must have been proposed by some process in Π. □

Lemma 4.3 (Agreement). *No two processes in* Π *decide on a different value.*

Proof. By the protocol, any process that decides some value reaches round $f + 1$ without crashing and returns its estimated value. Thus, there is a need to show that the estimated values of the correct processes at the end of round $f + 1$ are the same. Consider a round r in which no message is inconsistently omitted. This round exists because (a) there are $f + 1$ rounds; (b) in all of them at least one message is broadcast; and (c) there are at most f messages that can be inconsistently omitted at some processes.

If $\mathsf{r} = f + 1$, there is nothing to prove. Assume that $1 \leq \mathsf{r} < f + 1$. The proof is by induction on the round number. The base case is round $\mathsf{r} + 1$. Let v be the common view value in r (by lemma 3.1 such a property holds in r). Every process that broadcasts a message in $\mathsf{r} + 1$ has set up its estimated value in r, which by assumption is v. Thus, v is the only value broadcast in $\mathsf{r} + 1$. Since no received messages could be arbitrarily created or corrupted, all messages broadcast in $\mathsf{r} + 1$ contain v. Thus, any correct process that receives in $\mathsf{r} + 1$ messages broadcast in $\mathsf{r} + 1$ will update its estimated value to v.

Also, note that processes that do not receive any message broadcast in $\mathsf{r}+1$ will keep v as their estimated value. This is because the highest priority message such processes have received contains v since such a message belongs to r. This makes $\mathsf{r}+1$ a common view round. Using similar arguments as for the base case, it can be shown that the lemma holds for $\mathsf{r}' = \mathsf{r}+1, \ldots, f+1$. □

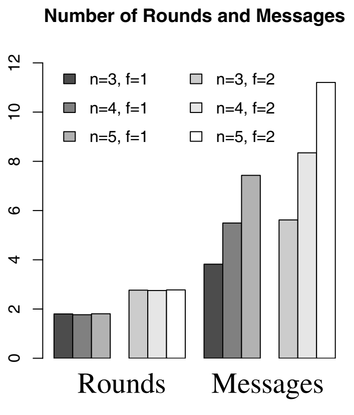
As a consequence of the presented lemmas, it follows that:

Theorem 4.1. *The protocol described in figure 2 solves the timed consensus problem in the assumed model of computation for a set of n processes despite fault scenarios, which involves inconsistent message duplication, up to f inconsistent message omission and up to $n-1$ process crashes.*

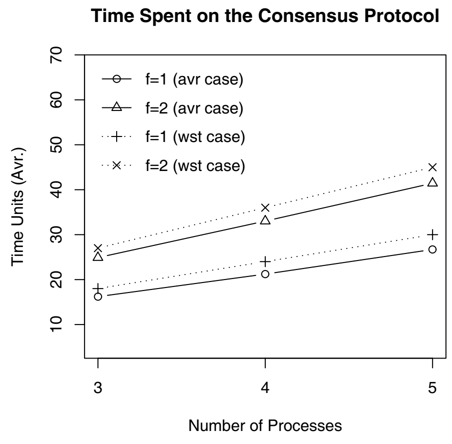
5. Performance Analysis

It is not difficult to see the behaviour of the protocol in the worst case. Each process executes $f+1$ rounds, each of which lasts Δ time units. Similarly, the protocol spends $n(f+1)$ messages in scenarios where no process crashes and there is no skipped rounds. Also, as no two processes can broadcast messages with the same priorities and messages broadcast in different rounds by the same process have different priorities, $n(f+1)$ priority levels are needed. For example, for a group of $n = 4$ processes and $f = 2$, three rounds are necessary, which gives 12 priority levels.

Despite this worst-case behaviour, it is important to emphasise that a process may skip all but the last round. Thus, although every round is executed by some process, from the point of view of an individual process, it may spend from 1 to $f+1$ rounds reducing the number of rounds and messages on average. The greater the value of f, the bigger this reduction is likely to be.



(a)



(b)

Figure 5. Average behaviour of the protocol.

In order to illustrate the protocol behaviour for average cases, some simulations were carried out. Figures 5(a) and 5(b) illustrate the obtained results. The simulation took into account 1,000 runs of the protocol. Groups of $n = 3, 4, 5$ processes were set up to perform the consensus protocol within a time window of 100 time units. Their starting times were randomly generated following a normal distribution with mean and standard deviation equal to 20 and 10, respectively. One of these processes was randomly chosen to crash during the execution of the protocol and $f = 1, 2$ inconsistent omission scenarios were taken into account. Both the crash time and the communication fault time were also randomly chosen. If the crash time happened to be after the finishing time of the process, no process was considered to be crashed. The values of $\delta = 3$ time units, $\alpha = 0$ time units and $\rho = 0$ determined the synchronism of the execution system. These values of α and ρ do not have major implications on the behaviour of the protocol and were chosen so for the sake of simplicity.

The first graph shows the average cost in terms of the number of rounds per process (not skipped) and messages necessary for the execution of the protocol. As can be seen from the figure, the number of rounds per process was kept almost constant (on average) for the different consensus group size and varies with the values of f. For example, taking the configurations for $f = 1$, correct processes participated of 1.7-1.8 rounds, compared to 2 rounds in the worst case. Similarly, for $f = 2$, 2.75-2.77 rounds per processes were executed while 3 rounds were expected in the worst-case.

The number of exchanged messages increases proportionally with both n and f. For $n = 5$ and $f = 2$, for instance, 11.2 messages were broadcast. Note that in the worst case a total of $(f + 1)n = 15$ messages (in scenarios with no crashes) or $(f + 1)n - (f + 1) = 10$ messages (with crashes before the execution of the protocol) are expected.

The average termination time per process is shown in figure 5(b). This time was measured as the difference between the finishing and the starting time of each correct processes. For comparison purpose, the graph also plots the expected behaviour in the worst case. Simulated and worst-case data are close because most of the time we are forcing crashes and omissions during the execution of the protocol, making each process wait Δ in each round. For the sake of illustration, consider a scenario for $n = 4$ and $f = 2$ where one process finishes the protocol before any other one starts its execution. In this case, the first process will take $(f + 1)\Delta = 36$ time units to choose a value and all the other processes will skip all but the last round. Hence, the skipping processes may take $\Delta = 12$ time units each to finish executing the protocol, leading the average finishing time to 18, which is half of the worst-case value. During simulation we did not take such good scenarios into consideration.

6. Related Work

Although very simply stated, the consensus problem has been shown to be impossible to solve deterministically in asynchronous systems subject to faults [Fischer et al. 1985]. This result has motivated extensive research (too extensive to be summarised here) in the field of partial synchronous distributed models. In general, in such models, it is considered that time bounds hold during a period of time when the processes are allowed to choose a common value. While waiting for such a time period, processes are not allowed to make progress. For systems where timeliness is considered, consensus is usually solved taking into account complete synchronism [Lynch 1996]. The solution presented here relaxes some synchronism restrictions present on classical synchronous protocols so that the system is made resilient to crash/omission/timing faults of some processes/messages and processes are not required to start the protocol synchronously.

For the best of our knowledge, message priorities have not been considered as a model parameter to tune a consensus protocol. Some work has solved the atomic broadcast problem considering priorities on system services [Wang et al. 2002]. Other researchers have considered the priority of messages as a criterion to atomically order them [Nakamura and Takizawa 1992]. Since atomic broadcast is one of the problems that can be reduced to consensus and vice versa [Hadziacos and Toueg 1993], these results are related to the one presented here.

In the context of the atomic broadcast problem, solutions have been given to CAN. Some approaches are based on hardware modifications [Proenza and Miro-Julia 2000] [Kaiser and Livani 1999] while others [Rufino et al. 1998] [Pinho and Vasques 2001] use the standard hardware. Also, a consensus solution for CAN has been presented recently by ourselves [Lima and Burns 2003]. However, none of these protocols take message priorities into consideration. The protocol described here had the first ideas discussed as a work-in-progress paper [Lima and Burns 2001]. In this preliminary version, however, processes have to start the protocol synchronously.

7. Conclusion

A simple but effective solution for the timed consensus problem has been presented. The protocol, which requires a communication network able to transmit messages according to their priorities, is attractive for supporting fault-tolerant real-time systems. Indeed, by making use of the message transmission priority ordering, the proposed solution works adequately even when the communication network only offers a very weak level of timing synchrony.

The main advantages of the proposed solution for the timed consensus problem against the standard synchronous protocols that are available are: its level of fault resilience; and the fact that protocol safety is guaranteed regardless of when processes start proposing their values (*i.e.* synchronised execution of the protocol is not needed).

Extensions of the protocol are possible and may be part of future work. For example, the proposed solution is based on the assumption that during execution of the protocol there is at least one message that is timely delivered at all destinations. This can represent extreme situations, although in normal execution more messages may be timely delivered. Instead of simply getting the estimated values of highest priority messages it would be interesting to have a more adaptable protocol where in normal execution a larger set of messages are considered. Another point to be considered is to make the protocol work when multiple consensus groups are concurrently executed. Also, it would be interesting to find out the minimum number of priority levels needed to achieve consensus. We believe that the results presented here and the new issues it opens up have both practical and theoretical implications in the field.

References

Burns, A. and Wellings, A. J. (2001). *Real-Time Systems and Programming Languages*. Addison-Wesley, 3rd edition.

Davis, R., Burns, A., Bril, R., and Lukkien, J. (2007). "Controller Area Network (CAN) schedulability analysis: Refuted, Revisited and Revised". *Real-Time Systems*, 35(3):239–272.

Decotignie, F.-D. (2001). A Perspective on Ethernet as a Fieldbus. In *FeT01, Proc. of the 4th International Conference on Filedbus Systems and their Applications*.

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

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

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

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

Kopetz, H. (1998). "The Time-Triggered Model of Computation". In *Proc. of the 19th Real-Time Systems Symposium (RTSS)*, pages 168–177. IEEE Computer Society Press.

Lima, G. M. A. and Burns, A. (2001). "A Timely Distributed Consensus Solution in a Crash/Omission-Fault Environment". In *Proc. of the Work-in-Progress Session of the 22nd RTSS*, pages 41–44. Available in the Tech. Report YCS337 of the University of York, England.

Lima, G. M. A. and Burns, A. (2003). "A Consensus Protocol for CAN-Based Systems". In *Proc. of the 24th Real-Time Systems Symposium (RTSS)*. IEEE Computer Society Press.

Lynch, N. A. (1996). *Distributed Algorithms*. Morgan Kaufmann Publisher.

Nakamura, A. and Takizawa, M. (1992). Priority-based total and semi-total ordering broadcast protocols. In *International Conference on Distributed Computing Systems*, pages 178–185.

Ostrovsky, R. and Patt-Shamir, B. (1999). "Optimal and Efficient Clock Synchronization under Drifting Clocks". In *Proc. of the Symposium on Principles of Distributed Computing (PODC)*, pages 3–12.

Pinho, L. M. and Vasques, F. (2001). "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.

Proenza, J. and Miro-Julia, J. (2000). "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.

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

Wang, Y., Anceaume, E., Brasileiro, F., Greve, F., and Hurfin, M. (2002). Solving the Group Priority Inversion Problem in a Timed Asynchronous System. *IEEE Transactions on Computers*, 51(8):900–915.