Performance Evaluation of a Consensus Algorithm with Petri Nets

Nicole Sergent
Département d'Informatique
Ecole Polytechnique Fédérale de Lausanne
CH-1015 Lausanne (Switzerland)
sergent@lse.epfl.ch

Abstract

This paper presents an application of Hierarchical Coloured Timed Petri Nets to the modeling, the evaluation and the improvement of the performance of a distributed consensus algorithm. The asynchronous system model is augmented with Failure Suspectors, in order to overcome the impossibility of reaching consensus in the presence of crash failures. We consider here two different implementations of the Failure Suspectors: a general implementation that shows poor performances, and an implementation optimized for the execution of the consensus. The termination time of the consensus algorithm is determined by the communications cost, i.e., by the timing characteristics of the network used for exchanging messages. In other words, the network latency is taken into account in the system model. By simulating the consensus Petri net model it is possible to provide probabilistic guarantees on the termination time of the consensus and to compare the performances of the different implementations of the Failure Suspectors.

1. Introduction

In asynchronous distributed systems message delays and relative processor speeds are unbounded. Since there is no shared memory nor global clock, the communication and the synchronisation between processes on different sites are done by message passing, through a network.

Usually, the performance of distributed algorithms is measured in terms of the number of messages exchanged during the execution, but evaluating the termination time is more significant. In practice, the main source of asynchrony is the unexpected load of the network or processors. It is impossible to evaluate the termination time of a distributed algorithm running in an unknown environment. The best we can do is to make assumptions about the external workloads and then study the behaviour of the algorithm. In this paper we consider that the consensus algorithm runs in a private network, the only source of asynchrony is then the nondeterministic traffic generated during the execution.

Providing fault-tolerance in asynchronous distributed systems is closely related to solving agreement problems. In [14] it was shown how all the agreement problems are related to the consensus problem, and how the agreement protocols can be built using a generic consensus service. The consensus algorithm allows processes to reach a common decision that depends on their initial inputs, even in the presence of failures. In asynchronous systems subject to failures, consensus cannot be solved deterministically [6]. The reason is that it is impossible, in such a system, to distinguish a slow process from one that has failed. To achieve fault-tolerance the system model has been extended by including the concept of Failure Suspectors [2]. Informally, a Failure Suspector is a module that gives processes some information about failures in the system. Each process has a local Failure Suspector module, which maintains a list with processes suspected to have crashed. The Failure Suspector of a process p will suspect another process q if during some time p does not receive any message from q. The list is permanently subject to changes; if a Failure Suspector module has erroneously added a process p among the suspected processes, it removes p from the list as soon as it receives some message from p.

In distributed systems, evaluating the termination time using experimental measurements is difficult and not reliable, since computation begins and ends at different processors and processes do not have access to a global time. Another major impediment is that results are usually biased by unpredictable network and station loads or by scheduling delays, unrelated to the studied algorithms. Analytical techniques are not suitable for a detailed study of multi-layered protocols, due to their complexity. A successful attempt is reported in [9], for an algorithm with a deterministic number of exchanged messages. This is not the case for the Chandra-Toueg consensus algorithm [2], considered in this paper, where the number of generated messages depends on the communication delays. Inversely, the traffic produced by the algorithm has a direct influence on the end-to-end delay experienced by a message in the underlying network. A previous attempt to study the termination time of the Chandra-Toueg consensus algorithm was made in [15], by using an analytical approach, but the solution assumes a maximum message delay. The model proposed in this paper can express both unbounded message delays and processor speeds.

Our approach is to use a formal description technique with sound mathematical basis, such as Petri nets. Bysimulating the model we obtain statistically significant results on the termination time of the consensus [12]. Simulation is also used to study the consensus behaviour under different input conditions and to choose optimum values for the parameters of the Failure Suspectors. These optimum values depend on the number of participating processes *n*, as the network traffic may considerably vary for different *n* values.

There are several possibilities to implement a Failure Suspector. One was proposed in [2], another one was considered for evaluating the termination time of the consensus in [12]. Both are independent of the execution of the consensus algorithm, which means that they could be implemented by processes independent of those running the consensus. In this paper we propose a different implementation for the Failure Suspectors, which improves the performance of the consensus by reducing the traffic generated on the network. By simulating the PN model, we evaluate the performance of the consensus for different Failure Suspectors implementations. The results of these simulations show that the termination time is considerably improved by using these Failure Suspectors.

The rest of the paper is organized as follows. A description of the modeled system is given in Section 2. The corresponding Petri net model is presented in Section 3. Section 4 contains the simulation results and some experimental results. Section 5 concludes the paper.

2. The system model

2.1. The asynchronous system model

The distributed system is composed of a finite set $P=\{p_1, p_2,..., p_n\}$, $n \in N$, of processes completely connected by a network. Processes are subject to crash failures and crashed processes never recover. A process $p_i \in P$ may (1) send a message to another process, (2) receive a message sent by another process, (3) perform some local computation or (4) crash. Communication is by message passing, asynchronous (the transmission delays are unbounded) and reliable. A reliable communication channel guarantees that a message sent by p_i to p_j is eventually received by p_j if both p_i and p_j are correct (i.e., do not crash).

Each process p_i has access to a local module called Failure Suspector FS_i which maintains a list of the processes that it currently suspects to be unreacheable. A Failure Suspector FS_i may make mistakes by incorrectly suspecting a correct process p_j. Suspicions are unstable, that is, if p_i is suspecting p_j and p_i receives a message from p_j, it will then remove p_j from its list of suspected processes. Disagreement among Failure Suspectors is tolerated: at the same time we may have p_i which suspects p_j and p_k which does not suspect p_j.

2.2. The consensus algorithm

In the consensus each participating process p_i, i=1,..., n, proposes an initial value v_i. At the end of the consensus each process p_i agrees to adopt one value v, chosen from the set of the proposed values $\{v_i\}$.

The problem specifications are as follows:

-*Termination*: Each correct process eventually decides.

-*Validity*: If a process decides v_i, then v_i was proposed by some process.

-*Agreement*: All correct processes decide on the same value v.

We denote by f the maximum number of faulty processes. The Chandra-Toueg algorithm, presented in [2], requires that $f < n/2$. The protocol is based on the *rotating coordinator* paradigm. The computation proceeds in asynchronous rounds, where in each round another process is the coordinator. The coordinator of round r is the process with rank $c=(r \bmod n) + 1$, where n is the number of processes involved in the consensus and the rank of the process p_i is i. Each coordinator tries to reach an agreement on a decision value and to decide. If the current coordinator is correct and is not suspected by any surviving process, then it will succeed and it will broadcast this decision value using the *Reliable Broadcast* algorithm [2]. If the agreement is not possible, the processes pass to the next round, with a different coordinator.

The *Reliable Broadcast* communication primitive guarantees that all correct processes will receive the decision. This primitive is implemented with the following algorithm: whenever a process receives a message for the first time, it sends the message to all the other processes.

Each round of the consensus algorithm is divided into 4 asynchronous phases. In *Phase 1* each participant sends its current estimate, timestamped with the round number when this estimate was adopted, to the current coordinator c. In *Phase 2* the coordinator c waits until it receives n-f estimates from the participants, chooses the one with the largest timestamp and sends it to all the processes as the new estimate proposed by c, $estimate_c$. In *Phase 3* a process p waits for $estimate_c$ or for a suspicion from its FS that c has crashed. If p receives $estimate_c$, it replies with an acknowledgement *ack* to c indicating he adopted $estimate_c$ as its own estimate. If p suspects c, it replies with a negative acknowledgement *nack* to c. In *Phase 4* c gathers n-f votes (*acks* or *nacks*). If all the votes are *ack*, then c broadcasts a request to decide $estimate_c$. Any time a process delivers a decision request, it *reliably broadcasts* the decision and then it decides. A possible execution of the consensus algorithm, when the decision is taken in the first round, is presented in Figure 1 (for readability reasons, in Phase 2 and Phase 4 the coordinator gathers n-1 votes, thus the messages are clearly separated within the phases).

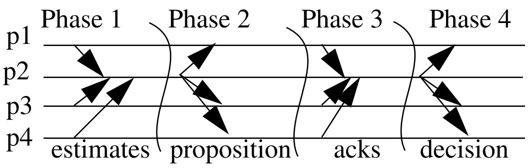

Figure 1 *The consensus algorithm*

The *termination time* of the consensus is defined as the time elapsed since the first process sends a message in Phase 1 of the first round until all correct processes have decided.

2.3. Implementation of Failure Suspectors

One possible implementation of the Failure Suspectors was proposed in [2]. In this paper we consider three other different implementations. The first one is as follows: each FS_i sends periodically, with an interrogation delay Δ_i, an interrogation message to all FS_j, $i \neq j$. Whenever FS_i receives an interrogation message from FS_j, it replies with an answer message to FS_j. If the processor p_i has not received any message from p_j, after a time-out delay Δ_{to}, FS_i adds p_j to the suspected processes set (see Figure 2).

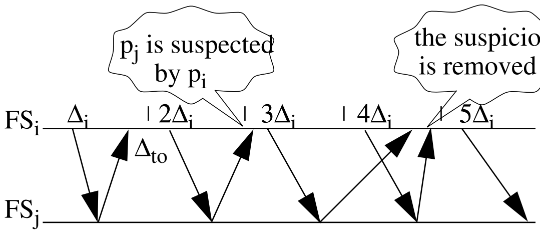

Figure 2 *The first implementation of the Failure Suspectors*

The two other possible implementations of the Failure Suspectors are strongly related to the execution of the consensus but have the advantage to generate less network traffic. Since the network traffic and the end-to-end delays are strongly correlated, these two propositions are likely to significantly improve the termination time of the consensus.The idea is that a process p_i, which is executing the consensus, needs the information supplied by its local FS_i only if it is in the third phase of the algorithm. Every time a process p_i enters the Phase 3 of the consensus, FS_i starts a timer. If the processor p_i has not received any message from the coordinator, after a time-out delay $\Delta^{(2)}$, FS_i adds the coordinator to the set of suspected processes (see Figure 3). This is the second possible implementation of the Failure Suspectors. Note that, in this case, the Failure Suspectors do not send any message on the network.

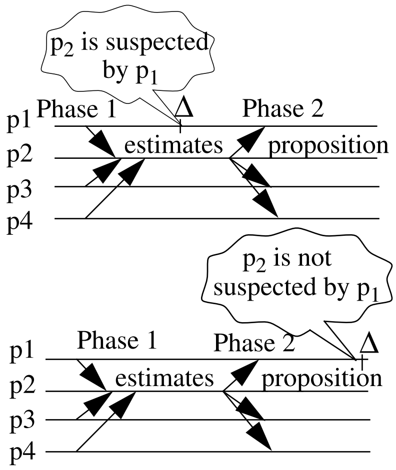

Figure 3 *The second implementation of the FS*

The third possible implementation of the Failure Suspectors is obtained by extending the mechanism of the second one. The basic idea is the same: when a process starts executing the Phase 3, it also starts a timer. Additionally, whenever a coordinator c receives a proposition from p_j, sent in the current round r of p_j, its Failure Suspector FS_c sends to p_j an «I_am_alive» message each $\Delta^{(3)}$ time units. Sending «I_am_alive» messages goes on until the coordinator sends to p_j, in the round r, its estimation $estimate_c$ (see Figure 4). This prevents a process p_i to suspect a correct coordinator c, which is simply waiting for other propositions, before being able to send his estimation.

We assume that all the delays required to perform the consensus local computations are null. Only the transmission cost is included in the computation of the termination time. This assumption is justified since the cost of an assignment statement represents less than 0.1% of the cost of a *send* or *receive* operation.

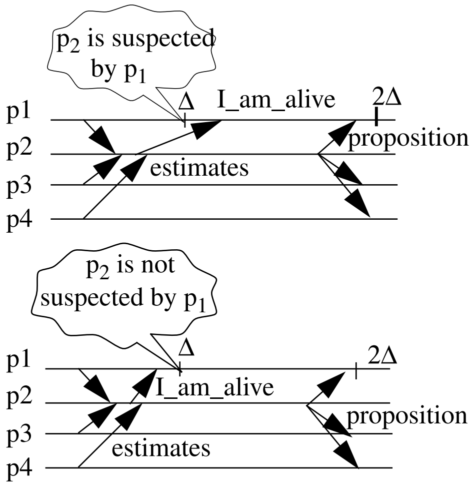

Figure 4 *The third implementation of the FS*

2.4. The network model

The processes participating in a consensus communicate with each other by exchanging messages through a reliable network protocol. We have focused on UDP/IP, built on top of an Ethernet CSMA/CD bus. First, a simple model for UDP/IP in a private LAN was built. The idea is the end-to-end delay experienced by a message in the underlying network can be decomposed into the time spent on the sending station's CPU (t_S), the network delay (t_n) and the time spent on the receiving station's CPU (t_R). If the network or the needed CPUs are busy, a message has to wait, which adds some waiting times to the end-to-end delay. We assume point-to-point communications. The network controller has its own processor which is responsible for implementing the Ethernet protocol [11]. To simplify our model, we make several assumptions:

1. the stations connected on the bus are identical.
2. there is at most one instance of the consensus running on a host station.
3. the LAN is private, i.e., there is no other message passing on the network except those generated by the considered applications.
4. there is neither message loss nor message collision.

Even if UDP/IP may loose some packets, the 4-th assumption is realistic. In fact several experimental studies [1] prove that the actual loss of packets with current hardware interfaces is one out of two million. The "no-message collision" is a strong assumption for a network like Ethernet, where the achievable throughput is limited to about 35% of the available bandwidth due to collisions [10]. It may be realistic if we assume the nodes spatially closed along the Ethernet bus and the offered load by far below the achievable throughput.

A similar model may be found in [8]. The validation of this network model and the identification of the constant parameters t_R, t_S and t_n of a real network are presented in [13].

3. Petri net model

The models in this paper were constructed using Hierarchical Coloured Timed Petri nets [7], incorporating coloured tokens, timed firing and hierarchical refinement.

Each token may carry complex typed information and this information may influence the firing rule of transitions. We consider the model where each token has a timestamp which indicates when the token becomes available. The timestamp of an output token, produced by a transition t, is equal to the sum of the firing time and a delay specified by the transition t.

We have built our Petri nets using EXSPECT, a software tool developed by Eindhoven University of Technology [5]. The EXSPECT language consists of two parts, a functional part and a dynamic part. The functional part is used to define types and functions. The dynamic part is used to specify the net. EXSPECT provides a hierarchy construct, called *system* (or *substitution transition*, in [7]). A system is an aggregate of transitions, places and (possibly) subsystems. Graphically, squares with a marked bottom right-hand corner denote transitions, rectangles denote systems, circles denote places, crossed circles denote one-capacity places (*stores*). Each store contains exactly one token, which cannot be delayed. A system is connected into a net by *pins*. Pins have types and must be connected to places or stores of the corresponding type. There are three kinds of pins: input, output and store pins, corresponding to input places, output places, and stores, respectively. Pins are depicted by circles containing "i", "o" or "s" marks, for input, output and store pins. For readability reasons we do not include arc inscriptions, written in a functional language, together with the nets.

The top-level system is the *consensus* system (Figure 9), which includes the *phase1* transition, the *phase2*, *phase3*, *phase4* sub-systems (modeling the corresponding phases of the algorithm), the *crash* transition (modeling random uniformly distributed crash failures), the *Failure_Suspectors* sub-system, the *broadcast* sub-system (which models the Reliable Broadcast protocol) and the *network* sub-system (which models UDP/IP).

The flow of the tokens through the Petri net models the behaviour of the consensus protocol implemented using UDP/IP communications. There are two types of tokens: process tokens and message tokens. The process tokens carry complex information which completely describe a process state at any moment. For instance, a token in the state place *begin_phase3* models a process which is waiting for the proposition of the coordinator (see Figure 1). The process identity, its current round number, estimation and estimation timestamp are given by the token value. A process token can reside only in a state place. In Table 1 there is a short description for each state place.

State place	Description
start	process ready to start a new round
begin_phase3	process waiting for a proposition from the coordinator or a suspicion from FS
propositions	coordinator waiting for n-f propositions
begin_phase4	coordinator waiting for n-f acks or nacks

TABLE 1. *State places*

Similarly, a message token carries complex information, which uniquely identifies a message (sender identity, receiver identity, message type, message content). The message tokens flow through the *network* system, they model the message exchange between the processes.

Initially, the net contains n tokens at the place labelled *start* (Figure 9), representing the n processes before launching the consensus. A token without type which resides in the *next_crash* place will enable after some time the *crash* transition, which models the crash failures generation. The *crash* transition fires according to a uniform random distribution, i.e., processes crash randomly. During a consensus execution, the maximum number of firings of the *crash* transition is f, according to the requirements of a non-blocking algorithm. Inside the *network* system there is also a *resources* place which initially holds some tokens, we will discuss it later in this section. All the other places of the net are initially empty.

The *processes_state* store always contains one token and this token cannot be delayed by transitions (it is always available). It contains all data that define the processes state during the execution of a consensus algorithm: the list of suspected processes for each participant, whether a process crashed or not, whether a process reached a decision, internal processes information (current round number, estimation and estimation timestamp), broadcast related data and others. This corresponds to a global external view of the system, process tokens do not have access to these data (otherwise this would correspond to a violation of the non shared memory assumption).

A process token from the *start* place may follow two sequences, depending whether the process is a coordinator or not. If the process is a coordinator, it will go through the *phase2* subsystem, the *begin_phase4* place, the *phase4* subsystem and back to the *start* place. If the processor is not a coordinator, it will go through the *phase1* transition, the *begin_phase3* place, the *phase3* subsystem and back to the *start* place. This cycle, which models a complete round of the consensus algorithm, is repeated by each process token until the corresponding process delivers the decision. For instance, if a process is not the coordinator of its current round, its process token (initially in the *start* place) enables the *phase1* transition, which fires by consuming the corresponding token from the *start* place and produces an output token in the *begin_phase3* place (the process is now waiting in Phase 3) and a message token in *sent_messages* place (its proposition sent to the coordinator).

The *network* system (Figure 10) models the UDP/IP protocol from Section 2.3. When a process sends a message to another process, it deposits a token in the *sent_messages* place. All the messages exchanged between the Failure Suspectors or generated by the Reliable Broadcast protocole transit via the network system, thus generating an extra-load on the bus which may increase the messages delays of the consensus.

The *sent_messages* and *delivered_messages* places shown in Figure 9 are identical to the corresponding places on the Figure 10. This is graphically represented by ¨i¨ or ¨o¨ inside the corresponding circles (from *input* and *output pins*, corresponding to the *port* places in [7]). All the places in Figure 10, except the *resources* place, may only contain message tokens. The token put in the *sent_messages* place carries several data, such as the source and destination process identifiers, the message type and the message content. The tokens from the *messages_for_bus* place model messages which are waiting for the bus at the sending controller level. Messages received at the controller level but not yet received by the application are modeled as tokens in the *received_messages* place. Tokens in the *delivered_messages* place model messages received at the application level.

To reach the sending controller level, a message from the *sent_messages* place needs the station's CPU, which is modeled as a token residing in the *resources* place. This place contains a token for each station CPU and a token which models the bus. Colours allow to distinguish the different resources. The *send* transition models a send operation from the application level to the controller level. It consumes two input tokens, from the *sent_messages* and *resources* places and produces two output tokens, both delayed with t_S, representing the released CPU and the message sent at the controller level (for *resources* and *messages_for_bus* places). The *send_on_bus* transition models the bus allocation and the message transmission. This transition may fire if it can consume the *bus* token from the *resources* place (the bus is idle) and a message token from the *messages_for_bus* place (message waiting for the bus). Firing will produce a message token (in the *received_messages* place) and the bus token (in the *resources* place), both delayed with t_n. The *receive* transition models the message reception and the station's CPU allocation during t_R time units (its behaviour is similar to that of the *send* transition).

The *broadcast* system (Figure 11) executes the Reliable Broadcast protocol. Whenever a process delivers a decision message (a process token may be in one of the places *start*, *propositions*, *begin_phase3*, *begin_phase4*), it sends a decision message to each participant. It may crash between two successive sends. Sending messages is executed by the four *broadcast_** transitions, four installations of the same transition definition. After a process has sent the decision to all the other participants, it adopts this decision. Hence a process might send a decision message to all the other processes and crash just before adopting the decision.

The *suspector_1* system (Figure 12) models the first implementation of the Failure Suspectors presented in Section 2.2. The *interrogation* transition sends interrogation messages each Δ_i time units, the *receive_answer* transition delivers the answer messages and the *t_o* transition updates the list of suspected processes. When a process p sends an interrogation message to a process q, the *interrogation* transition fires and produces an output token in the *timeout* place, delayed with Δ_{to}. The *t_o* transition fires when the token in the *timeout* place becomes available and a flag from the *processes_state* store indicates that no message was received from process q, i.e., the timeout on q has expired. Then p will suspect q. The lists of suspected processes are updated in the *processes_state* store.When the algorithm begins, these lists are empty.

As stated in Section 2.2, the second implementation of the Failure Suspectors is a particular case of the third one. The PN model corresponding to these two kinds of Failure Suspectors is presented in Figure 13. Whenever a process p enters in the third phase of the consensus, a flag is updated in the *processes_state* store, which allows the *start_monitoring* transition to fire. As a result, a delayed output token is put in the *time_out* place. The *time_out* transition fires whenever a token from the *time_out* place becomes available and makes use of the information from the *processes_state* store for updating the list of suspected processes. The *receive_messages* transition makes the demultiplexing of the messages received by a process

The *phase2* system (Figure 14) includes the *begin_ph2*, *receive_propositions* and *end_phase2* transitions. The *begin_ph2* transition initialises the list of proposed estimates with the current estimation of the coordinator. Whenever *receive_propositions* is enabled by a message token from the *delivered_messages* place, it fires and consumes the message, updating the list of proposed estimates. The *propositions* place contains the number of estimates received and the estimate with the highest timestamp. The update is made only if a bigger estimate timestamp is detected. After n-f estimates are gathered, the *end_phase2* transition will fire n-1 times, each time sending $estimate_c$ to a participant. If the coordinator crashes during this operation, the *end_phase2* transition is disabled.

The *phase3* system (Figure 15) includes the *suspect*, *receive_estimate* and *update_flag* transitions. The *update_flag* transition is enabled by a message token representing a proposition from the coordinator. When it fires, it updates a flag from the *processes_state* store, thus avoiding to send a nack to a suspected coordinator from which the process has already delivered a proposition. If the *suspect* transition fires, enabled by a suspicion, it sends a nack to the current coordinator. If the *receive_estimate* transition fires, this means that the process delivered the coordinator decision, so it will send an ack message.

The *phase4* system (Figure 16) contains two transitions: *receive_ack_nack* and *send_decision*. The first one gathers n-f votes from the participants. If the result is *ack*, the *send_decision* transition starts the reliable broadcast algorithm executed by the *broadcast* system. If the result is *nack* (at least one process sent a *nack* vote), the *send_decision* transition puts the process token back in the *start* place.

Our Petri net model has a deadlock, the state when all processes have decided. At the end of a simulation, when the deadlock arises, the simulation time indicates the termination time for the consensus. To obtain statistically significant results on the termination time we need to run many simulations, so we have constructed another PN, at a higher level than the consensus PN, which is responsible to initialise the consensus net and to store the termination time when a deadlock of the consensus PN arises. The complete description of this PN is beyond the scope of this paper.

4. Simulation results

Analytical techniques (invariants, occurrence graph) [7] are not able to provide a formal validation for the consensus algorithm. The reason for this is that our model includes only an approximation of $\Diamond S$ Failure Suspectors [2], close to what would be a real Failure Suspector. The consequence is that the proof of termination for the consensus [2] is no longer valid, thereby a consensus may never finish (the number of rounds may increase infinitely if a coordinator is always suspected by at least one correct process). After running a few simulations, it is possible to choose a reasonable timeout value, by inspecting the token values. Such a timeout should be large enough to generate only a few erroneous suspicions, which allows the consensus to finish in only one round most of the time.

Since we know that our Petri net model does not guarantee consensus termination, there is no point in inspecting the infinite occurrence graph. Instead, a special analysis tool included in EXSPECT, calculated the place invariant M(start) + M(propositions) + M(begin_phase3) + M(begin_phase4) =constant. This S-invariant proves that, at any time, every process is in some state. That is, the number of processes does not change during a net execution.

The model was validated using simulation, in a similar two-steps approach to that described in [3]. The difference with [3] stems from an additional step in the validation process, based on the fact that, when either no process crash occurs or crashes are deterministic, the consensus algorithm has a predictable behaviour, that depends on the Δ_{to} and Δ_i values, introduced in Section 2.3 (when using the first implementation of the Failure Suspectors). If Δ_{to} is big enough to have only a few erroneous suspicions and if there is no process crash, the consensus terminates in one round. This expected behaviour was verified by simulations with 3, 4 and 5 processes.

Our interest is to use the Petri net model to study the non-deterministic situations, when crashes occur randomly. The simulations were made on Sun SPARCstations 20, under Solaris 2.3. As the number of processes in the consensus increases, the real time for executing a simulation becomes longer. With 3 participant processes, the time needed to perform consensus 5000 times is 4 hours. For 7 participants, the same time becomes 24 hours. This is an important drawback of the simulation as a performance evaluation tool, but several studies [4] prove that it has to be accepted for a detailed behaviour description of parallel algorithms, since the detailed models are too complex to be solved analytically. Using the Central Limit Theorem of statistics and the independent replications method, it is possible to build 95% confidence interval around the termination time values. This methodology was used in [12]. Since the purpose of this paper is to compare the performances of the three possible implementations of the Failure Suspectors, the results presented here were computed by averaging over a large number of simulations (thousands), without trying to provide confidence intervals, which would be too time consuming.

The non-contention assumption for the network is justified by the simulations, since the generated load is under 15% of the available bandwidth (this can be verified by computing the number of messages exchanged during the execution of the PN model).

The simulations were made for n=5, t_S=230 μs, t_R=100 μs, t_n=250 μs. The t_S, t_r and t_n constants correspond to a real network, and were identified using the same approach as in [13]. We have chosen the same t_S and t_R values for all participants, but it is also possible to choose different values for the t_S and t_R parameters for each process, thus considering different processor speeds.

Figures 5 and 6 show the termination time of the consensus, using the first implementation of the Failure Suspectors as a function of the timeout delay Δ_{to}, for several interrogation delays Δ_i. The termination time t plotted in Figure 5 is the best case, when no crash failure occurs during the consensus execution. Figure 6 presents the worst case, when the coordinator of the first round crashes.

From Figure 5 we may see the termination time t decreases as Δ_i increases. The more often interrogations are sent and the shorter timeouts are, the longer the consensus algorithm requires. This may be surprising, as quick interrogations and short timeouts should make the information exchange more rapid. The problem is that the communication is done through a bus which allows only one message at a time. The frequent interrogation messages contribute to the network traffic and thus increase the end-to-end delay. For the four Δ_i values considered the recommended timeout value Δ_{to} is 6 ms. Smaller values increase the termination time t by producing erroneous suspicions and thus increasing the number of rounds.

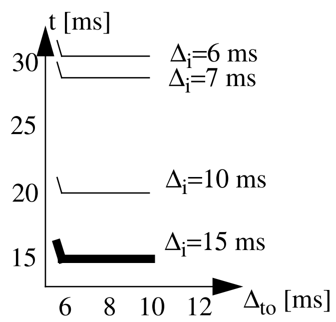

Figure 5 *No crash failures*

From Figure 6 we may see that Δ_i=8 ms or Δ_i=10 ms would be a bad choice, since they generate too much traffic on the network. Indeed, as these two values are smaller than Δ_i=15 ms, they would normally provide a smaller t when crashes occur, since a crash would be more quickly detected. This is not the case in Figure 6, where the Δ_i=8 ms and Δ_i=10 ms curves are above the Δ_i=15 ms and Δ_i=20 ms curves. The explanation is that end-to-end delays are smaller for Δ_i=15 ms and Δ_i=20 ms than for Δ_i=8 ms and Δ_i=10 ms. Acceptable Failure Suspector parameters are Δ_i=15 ms and Δ_{to}=6 ms or Δ_i=20 ms and Δ_{to}=6 ms. The best Δ_{to} value is the smallest which minimizes t when there are no crash failures. These results are very similar to those obtained in [12].

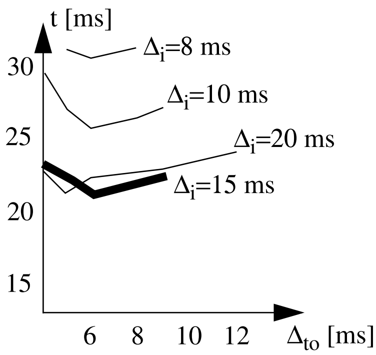

Figure 6 *The coordinator of the first round crashes*

The termination time of the consensus using the second implementation of the Failure Suspectors is plotted in Figure 7. When no crash failure occurs during the execution, the termination time of the consensus is minimized by choosing a $\Delta^{(2)}$ value bigger than 3 ms. Indeed, as Figure 7 shows, for $\Delta^{(2)}$ values less than 3 ms, false suspicions are generated, thus increasing the number of rounds and the termination time. When the coordinator of the first round crashes, we may see from the shape of the above curve in Figure 7 that the termination time reaches the minimum for $\Delta^{(2)}$=3.5 ms.

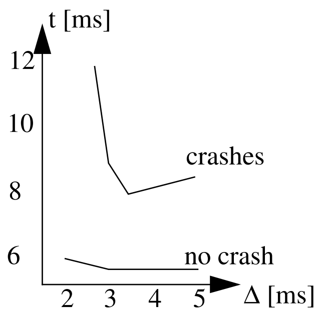

Figure 7 *The second implementation of the FS*

For $\Delta^{(2)}$ values less than 3.5 ms the Failure Suspectors make mistakes, while for $\Delta^{(2)}$ values bigger than 3.5 ms the waiting time in the third phase before detecting a real crash is increased. For the second implementation of the Failure Suspectors it is possible to choose the best $\Delta^{(2)}$ value ($\Delta^{(2)}$=3.5 ms), which optimizes the performance of the consensus.

A similar behaviour is observed with the third implementation of the Failure Suspectors (Figure 8). The same value of Δ^3=3.5 ms is recommended for obtaining the best performances. By comparing the curves «no crash» and «crashes» plotted in Figures 7 and 8, one may notice that using the second implementation of the Failure Suspectors leads to slightly lower termination times, thus providing the best performances. The explanation is that the third

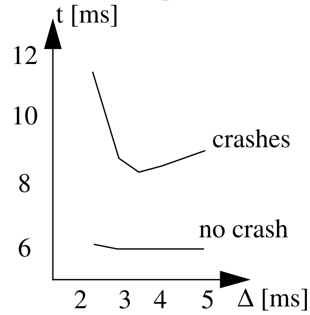

Figure 8 *The third implementation of the FS*

implementation of the Failure Suspectors generates the «I_am_alive» messages, which contribute to the traffic on the network, thus increasing end-to-end delays and the termination time.

Table 2 presents the simulation results obtained with well parameterized Failure Supectors. From the three different implementations considered in this paper, the one which leads to the best performances is the second one. It allows the consensus to finish in about 5.7 ms, when there are no failures, or 8 ms, when the coordinator of the first round crashes.

FS	**parameters**	**t without crashes [ms]**	**t with crashes [ms]**
1	Δ_i=15 ms, Δ_{to}=6 ms	15	21.7
2	$\Delta^{(2)}$=3.5 ms	5.7	8
3	$\Delta^{(3)}$=3.5 ms	6.2	8.6

TABLE 2. *Simulation results*

The simulation results obtained with the second implementation of the Failure Suspectors were compared to the experimental results provided by a real system, the same used for identifying the t_S, t_n and t_R constant parameters. The results obtained by simulating the Petri net model corresponding to the second implementation of the Failure Suspectors are quite close to those obtained from the real system, as shown in Table 3.

The simulation results prove the success of our modeling approach and enlighten the possibility to design asynchronous distributed algorithms to obtain good performances.

n	t simulations [ms]	t experimental [ms]
3	2.74	2.73
4	3.99	3.91
5	5.33	5.17
6	6.83	6.51

TABLE 3. *Experimental and simulation results*

5. Conclusions

The present case study leads to new interesting insights in the Chandra-Toueg algorithm and allows to considerably improve its performances by choosing an appropriate implementation for the Failure Suspector modules.

It should be noticed that all the results obtained in this paper are strongly dependent on the type of network and the network protocols chosen for exchanging messages between the processes. The number of messages generated on the bus has an important influence over the termination time of the consensus, and optimizing the algorithm means reducing as much as possible the traffic on the network. It is not surprising that the best Failure Suspector is the one which does not generate any additional message on the network. If the consensus is built on top of networks like FDDI-II, capable of carrying several messages at the same time, the conclusion may be quite different.

Using Hierarchical Coloured Timed Petri nets, it is possible to make a very detailed model of a high-level distributed algorithm and to take into account the latencies introduced by lower level protocols. This kind of Petri nets becomes a powerful high-level programming language, which allows to distribute the complexity of the code over the net structure and the arc inscriptions and preconditions. The major drawback of using such a powerful formalism is to loose the analysis capabilities of classical Petri nets. The simulation becomes the only way to verify the behaviour of the Petri net model, but this is also the price to pay when using other formalisms, like queuing networks [4].

The Petri net model of the consensus algorithm presented in this paper, built in a hierarchical style, is very flexible and allows, for instance, to modify the network model from Ethernet to FDDI without having to modify the hole net. The model is scalable; when the size (the number of processes in consensus) of the modeled system increases substantially, the size of the Petri net model remains constant and the modeling time (the time needed to adapt the model to different number of processes) grows insignificantly. The simulation time unfortunately increases substantially, as discussed in Section 4.

Further work will include the refinement of the network model and temporal studies of high-level distributed protocols like Atomic Broadcast, which are based on the consensus algorithm, thus allowing the reuse of the consensus Petri net model as a sub-system.

References

[1] Cabrera L. F., Hunter E., Karels M., D. Mosher. User-Process Communication Performance in Networks of Computers. *IEEE Transactions on Software Engineering*, vol. 14, no.1, 1988.

[2] Chandra T. D., S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of ACM*, 34(1), 1996.

[3] Clausen H., P. Jensen. Validation and Performance Analysis of Network Algorithms by Coloured Petri Nets. *Proceedings of the 14th International Conference on Applications and Theory of Petri Nets*, Springer-Verlag, 1993.

[4] Delaitre T., Justo G.R., Spies F., S. Winter. EDPEPPS: An Environment for the Design and Performance Evaluation of Portable Parallel Software. *Proceeedings of the 5-th Euromicro Workshop on Parallel and Distributed Procesing*, London, U.K., January 22-24, 1997.

[5] ExSpect Reference Manual - Release 5.0. Bakkenist ExSpect, Netherlands, november 1994.

[6] Fischer M., Lynch N., M.Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, vol.32, No.2, April 1985.

[7] K.Jensen. Coloured Petri nets: basic concepts, analysis methods and practical use. *Springer Verlag*,1994.

[8] Malcolm N., W.Zhao. Hard Real Time Communication in Multiple-Access Networks. *Real-Time Systems* 8, 35-77, 1995.

[9] Menascé D., Y. Yesha. On a Unified Framework for the Evaluation of Distributed Quorum Attainment Protocols. *IEEE Trans. on Soft. eng.*, vol.20, no. 11, nov. 1994.

[10] M. Nemzow. The Ethernet Management Guide. *McGraw-Hill Series on Computer Communications*, 1995.

[11] M. Santifaller. TCP/IP and ONC/NFS- Internetworking in a UNIX Environment. *Addison-Wesley*, 1994.

[12] Sergent N. Probabilistic Guarantees on the Termination Time of a Consensus Algorithm in Asynchronous Systems. *Proc. of the Communication Networks and Distributed Systems Modeling and Simulation Conference*, WMC´97, Phoenix (Arizona), January, 1997.

[13] Sergent N. Evaluating Latency of Distributed Algorithms Using Petri Nets. *Proc of the 5-th Euromicro Workshop on Parallel and Distributed Processing*, London, January, 1997.

[14] Schiper A., R. Guerraoui. Consensus Service: A Modular Approach For Building Fault-Tolerant Agreement Protocols in Distributed Systems. *Proc. of the 26th IEEE Int Symp on Fault-Tolerant Computing Systems* (FTCS-26), Sendai, Japan, 1996.

[15] Vachon J. Analyse temporelle d'un algorithme de consensus dans un système asynchrone. *Proc. RenPar'8*, Bordeaux, 1996.

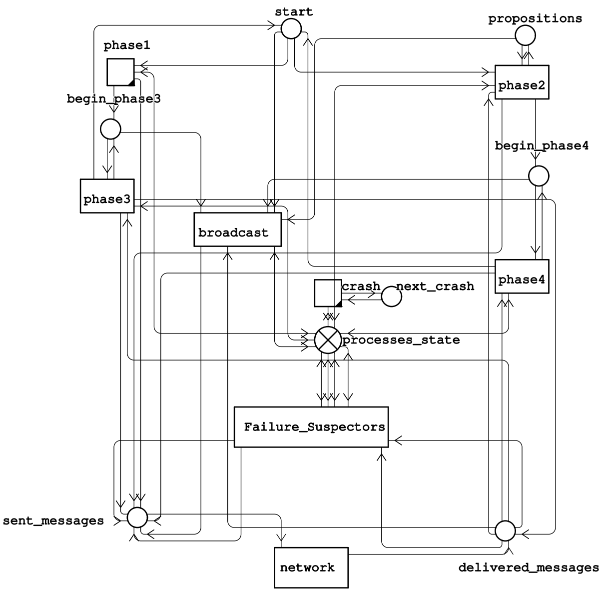

Figure 9 The *consensus* model

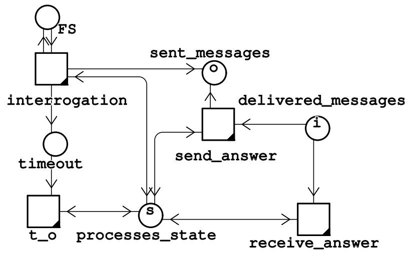

Figure 12 The *suspector_1* system

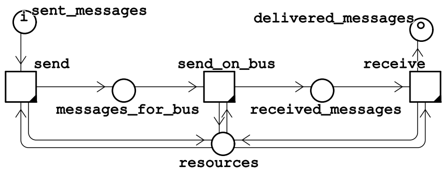

Figure 10 The *network* system

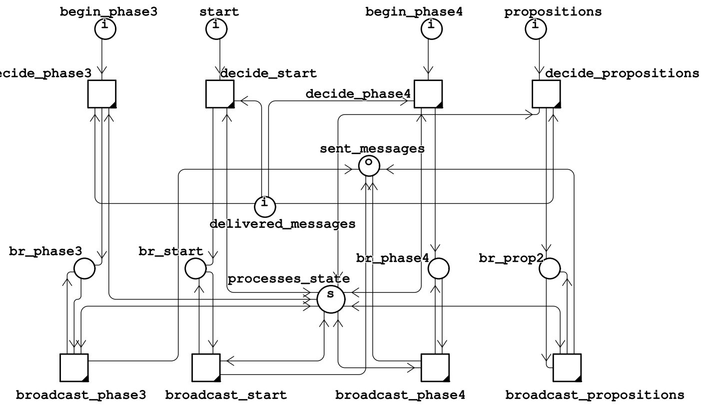

Figure 11 The *broadcast* system

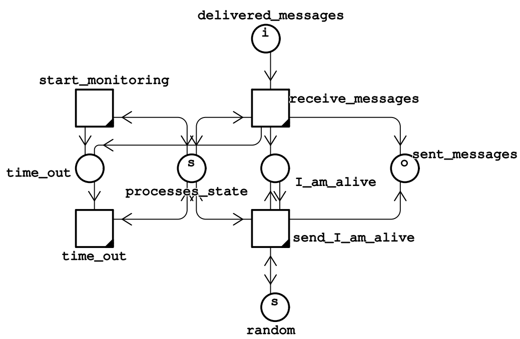

Figure 13 The *suspector_2_3* system

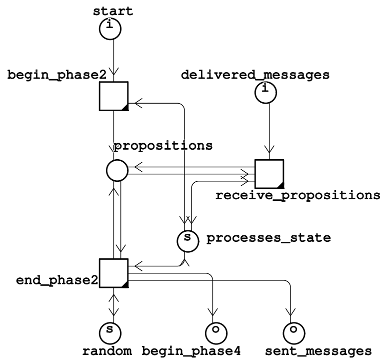

Figure 14 The *phase2* system

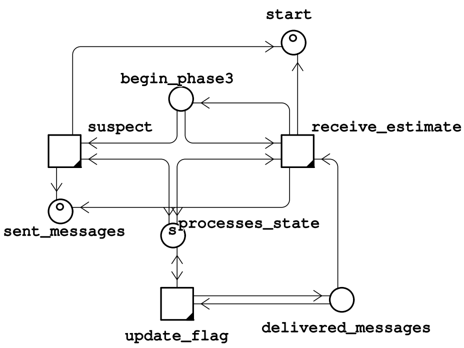

Figure 15 The *phase3* system

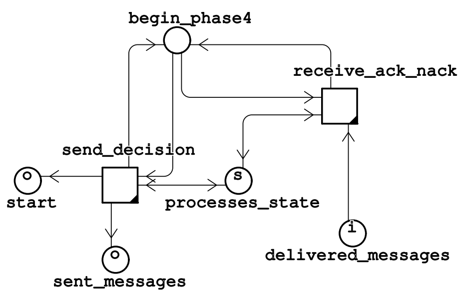

Figure 16 The *phase4* system