PROBABILISTIC GUARANTEES ON THE TERMINATION TIME OF A CONSENSUS ALGORITHM IN ASYNCHRONOUS SYSTEMS

Nicole Sergent
Département d'Informatique
Ecole Polytechnique Fédérale de Lausanne
1013 Lausanne (Switzerland)
sergent@lse.epfl.ch

KEY WORDS: Petri nets, UDP/IP, consensus, simulation.

ABSTRACT

This paper is concerned with an application of Hierarchical Coloured Timed Petri Nets to the modelling and the performance evaluation of a distributed consensus algorithm. The asynchronous system model is augmented with Failure Suspectors, which allow to overcome the impossibility of reaching consensus in the presence of crash failures. We have used a top down modular approach to manage the complexity of the considered protocols. 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. Our model establishes a strong interdependence between the end-to-end delays and the amount of traffic on the network. As a network protocol we have considered UDP/IP. Simulating the consensus Petri net model, it is possible to obtain optimal values for the Failure Suspectors parameters. Probabilistic guarantees on the termination time of consensus are provided.

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.

The main concern when designing an algorithm for asynchronous systems is to be fault-tolerant. Usually, the performance of distributed algorithms is measured in terms of the number of messages exchanged during the execution. We assess that the termination time of the algorithms is also important and in this paper we present a modelling approach, based on Petri nets (Peterson 1981), which gives statistically significant results for the termination time of the Chandra-Toueg consensus algorithm (Chandra and Toueg 1996).

Consensus allows processes to reach a common decision that depends on their initial inputs, despite failures. Reaching agreement between several computational nodes is an important problem, it can be used for instance in electing a leader or deciding on a common view of operational processes. In asynchronous systems subject to failures, consensus cannot be solved deterministically (Fischer *et al*. 1985). To achieve fault-tolerance the computational model was extended by including the *Failure Suspectors* concept (Chandra and Toueg 1996).

Evaluating the termination time using the performance measures approach is difficult and not reliable in distributed systems, since computation begins and ends on 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, not related to the studied algorithms. Analytical techniques are normally not adequate for a detailed study of layered protocol architectures, due to their complexity. A successful attempt is reported in (Menascé and Yesha 1994) for an algorithm with a deterministic number of exchanged messages. This is not the case for the Chandra-Toueg consensus algorithm, where the number of generated messages depends on the communication delays. Inversely, the traffic produced by the algorithms 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 (Vachon 1996), but the solution requires the consideration of a maximum message delay. The model proposed in this paper doesn't have to change the asynchronous system model.

Our approach is based on a formal description technique with sound mathematical basis, like Petri nets. A Petri net model has the advantages of graphical representation, simple specification and the ability to model synchronisation and parallelism. In our case, modelling the consensus algorithm using simple Petri nets would be impossible, since there are variables which may vary in an infinite set of values. Another reason is the impossibility to find the termination time of the algorithm if there is no way to introduce time in the model. This leads to extensions of the initial Petri nets, like introducing "colour", "time" and "hierarchy" (Jensen 1994), which are used in this paper for modelling the consensus algorithm and the underlying UDP/IP network protocol. The "hierarchy" extension is particularly useful for a layered modelling approach, it allows to keep the complexity of the net reasonable.

Simulating the model we obtain statistically significant results on the termination time (see section 2.2) of consensus. Simulation is also used to study the consensus protocol behaviour under different input conditions and to choose optimum values for the Failure Suspectors parameters Δ_i and Δ_{to} (see Section 2.1). These optimum values depend on the number of participating processes n, as the network traffic may considerably vary for different values of n. We can thus provide the statistical guarantee that consensus will finish before a certain time with a predefined probability. A small risk percentage has to be accepted as inevitable in an asynchronous system, where there is no upper limit for the communication delays.

The rest of the paper is organized as follows. A description of the modelled system is given in Section 2. The corresponding Petri net model is presented in Section 3. Section 4 contains the simulation results and some considerations on the validation of the model using analytical techniques. 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 through a network. Processes fail by crashing (byzantine failures are not considered) and crashed processes never recover. A process $p_i \in P$ may (1) send a message to another process, (2) deliver a message sent by another process p_j, (3) perform some local computation or (4) crash. Communication is realized by message passing, asynchronous (the transmission delays are unbounded) and reliable. A reliable 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 a local module called Failure Suspector FS_i which maintains a set of processes that it currently suspects to be unreachable. A Failure Suspector FS_i can make mistakes by incorrectly suspecting a process p_j. Suspicions are not stable, i.e. if p_i currently suspects p_j and p_i receives a message from p_j then it will 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.

A Failure Suspector can be characterized by two properties: completeness and accuracy. An Eventually Strong Failure Detector $\Diamond\mathcal{S}$ satisfies *strong completeness* and *eventual weak accuracy.* Strong completeness requires that, eventually, every process that crashes is permanently suspected by every correct process. Eventual weak accuracy requires that there is a time after which some correct process is never suspected by any correct process.

A Failure Suspector FS_i can be implemented as follows. FS_i sends periodically, with an interrogation delay Δ_i, an interrogation message to all FS_j ($i \neq j$). Any time FS_i receives an interrogation message from FS_j, it replies with an answer message to FS_j. If after a time-out delay Δ_{to} after sending an interrogation message to FS_j, the processor p_i has not received any message from p_j, FS_i adds p_j to the set of suspected processes. This class of Failure Suspector satisfies strong completeness but does not satisfy eventual weak accuracy. Eventual weak accuracy is impossible to achieve in real systems with a probability equal to 1, because of the unbounded message delays assumption.

2.2. The consensus algorithm

In the consensus, each participant 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*: no two correct processes decide differently.

We denote by f the maximum number of faulty processes. The Chandra-Toueg algorithm (Chandra and Toueg 1996) requires $\Diamond\mathcal{S}$ and $f < n/2$. The protocol is based on the *rotating coordinator* paradigm. Computation proceeds in asynchronous rounds, where in each round another process is the coordinator. Each coordinator tries to come to 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 S*pontaneous Reliable Broadcast* algorithm. If agreement is not possible, the processes pass to the next round.

Each round is divided into 4 asynchronous phases (Figure 1). 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_i waits for $estimate_c$ or for a suspicion of c from its FS_i. If p receives $estimate_c$, it replies to c with an acknowledgement *ack* indicating that it has adopted $estimate_c$ as its own estimate. If p suspects c, it replies with a negative acknowledgement *nack*. 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, where the decision is taken in the first round, is presented in *Figure 1*.

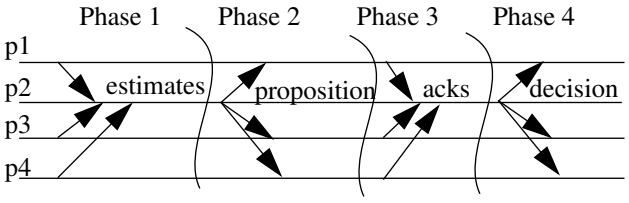

Figure 1 The consensus algorithm

The *Reliable Broadcast* communication primitive (Chandra and Toueg 1996) guarantees that all the correct processes will deliver the decision. This primitive is implemented by the following algorithm: whenever a process receives a message m for the

first time, it diffuses m to all the other processes, and delivers it.

It was formally proved that, assuming $\Diamond\mathcal{S}$ and $f < n/2$, this consensus algorithm satisfies the requirements of validity, agreement and termination. The Failure Suspector $\Diamond\mathcal{S}$ ensures that there is a time after which some correct process is never suspected by any correct process (eventual weak accuracy). When this process becomes the coordinator, it can decide.

The *termination time* of the consensus protocol is defined as the time elapsed since the first process sends a message in Phase 1 of the first round until all the correct processes have decided. The purpose of this study is to give probabilistic guarantees on the termination time of the consensus and to study the influence of Δ_i, Δ_{to} parameters on this time. We assume that all the delays required to perform the internal consensus operations are null, thus only the transmission cost is included in the evaluation of the termination time. This assumption is justified since the cost of an assignment statement represents 3-5% of the cost of a *sendto* or *receivefrom* operation.

2.3. The UDP/IP model

The processes solving the consensus problem communicate by exchanging messages through a reliable network. We focused on UDP/IP, which is unreliable but allows to handle packet losses at the application level. As a network architecture we assume a multi-access LAN (Local Area Network), where all the stations are directly connected on an Ethernet (CSMA/CD) bus, without repeaters. This network implements the completely connected network of our system model since any two processes can communicate by sending messages on the bus. The purpose of this study is to build a simple UDP/IP model which allows to establish a relationship between the traffic on the network and the message delays. This model will be later translated into a Petri net. To simplify the analysis, we make several assumptions:

1. the stations connected on the bus are identical
2. there is at most one consensus application running on a host station
3. the LAN is private, i.e. there is no other traffic on the network except the one generated by the consensus protocol
4. there is no message loss, no message collision, nor transmission error.

Even if UDP/IP may loose some packets, the 4-th assumption is not unrealistic. In fact several experimental studies (Cabrera *et al*. 1988) prove that the actual loss of packets is about one in two million. In future work we will relax this assumption by including message loss in the model.

The size of the messages at the application level is 4 bytes (the size of the consensus messages) and the Ethernet frame size is the minimum allowed, 64 bytes (with some stuff bytes). There is no message fragmentation, each message is carried in a single frame.We assume only point-to-point communications. The network controller has its own processor which is responsible for implementing the Ethernet protocol (Comer and Stevens 1991). As already mentioned, we assume that the instructions at the consensus level are executed in a null time delay, all the timing aspects are included at the network level. They derive from allocating two kinds of finite resources, the CPUs and the bus.

Sending a message from the application level to the network controller requires the allocation of the station's CPU. If the operation succeeds, the CPU of the sending station will be unavailable for t_S time units. The delay for a send operation t_S is a constant depending on the UDP/IP implementation and the operating system characteristics. If an application attempts to send a message while the CPU is busy, the message will have to wait until the CPU becomes idle. We denote this waiting time by t_{wS}.
For receiving a message, an application has to allocate the CPU of the host station for t_R time units, counted from the moment the last bit of the frame has been read by the network controller. If the CPU is busy, a message will wait t_{wR} time units before being received by the application. We assume that the network controller is available immediately after the last bit of a message has been sent, so it can detect a new frame (otherwise some messages may be lost).
For sending a frame on the bus, the controller needs t_t time units, which represents the transmission time (t_t can be computed as L/D, where L is the frame length in bits and D is the number of bits which are transmitted in a second). For D=10 Mbits/s, we have t_t=52 µs. The propagation time t_p represents the time it takes a bit to pass on the bus from one station to another. For a maximal distance on the bus of 500 meters between two stations, t_p is 2,5 µs. The time needed to send a frame on the bus is t_n, which is the sum of t_t, t_p and other memory access delays. This means t_n>54.5 µs. If the bus is busy, a frame will have to wait for t_{wn} time units, before being transmitted by the controller. Sending a message occupies the bus during t_t+t_p time units, but for simplifying we will consider the bus unavailable for t_n time units.

The end-to-end communication delay T experienced by a message exchanged between two distant applications is $T=t_{wS}+t_S+t_{wn}+t_n+t_{wR}+t_R$. While t_S,t_n and t_R are constant, t_{wS},t_{wn} and t_{wR} are variable delays depending on the network and stations loads. A similar decomposition of T may be found in (Malcolm and Zhao 1995).

We validated this model against the performance results of a real system, consisting in a SUN stations network (SS2-GX, 1 x CPU SPARC, 40 Mhz, 64MB RAM), interconnected over an Ethernet CSMA/CD bus at 10 Mbits/s. A detailed presentation of the validation process of the network model may be found in (Sergent 1996).

3. PETRI NET MODEL

Classical Petri nets cannot describe complex systems. The models in this paper are constructed using Hierarchical Coloured Timed Petri nets, incorporating coloured tokens, timed firing and hierarchical refinement (Jensen 1994).
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. In hierarchical Petri nets, complex systems can be decomposed into subsystems, more easy to specify and to design. The modeller may concentrate on the details of a subsystem, ignoring all the other aspects of the

system. We have found this feature very useful for separately specifying the consensus and UDP/IP models.

We built our Petri nets using EXSPECT, a software tool developed by Eindhoven University of Technology (Voorhoeve and Reus 1994). The language EXSPECT 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*. 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 and crossed circles denote one-capacity places (*stores*). For readability reasons we do not include arc inscriptions, written in a functional language, together with the nets. For a detailed description of the model, see (Sergent 1996).

The top-level system is the *consensus* system (Figure 2), which includes the *phase1* transition, the *phase2*, *phase3*, *phase4* sub-systems (modelling the corresponding phases of the algorithm), the *crash* transition (modelling random uniformly distributed crash failures), the *suspector* sub-system (modelling the Failure Suspectors), 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 the state of a process 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. Table 1 gives a short description for each state place. Similarly, a

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.

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, modelling the message exchange between the processes.

Initially, the net contains n tokens in the place labelled *start* (*Figure 2*), representing the n processes before launching the consensus, and n tokens in the *FS* place, modelling the n failure suspector modules. A token without a type, which resides in the *next_crash* place will enable, after some time, the *crash* transition. Inside the *network* system there is also a *resources* place which initially holds some tokens, this will be discussed later in this section. All 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 records all data which define the state of the processes during the execution of consensus: the list of suspected processes, whether a process has crashed or not, whether a process reached a decision, internal processes information, 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).

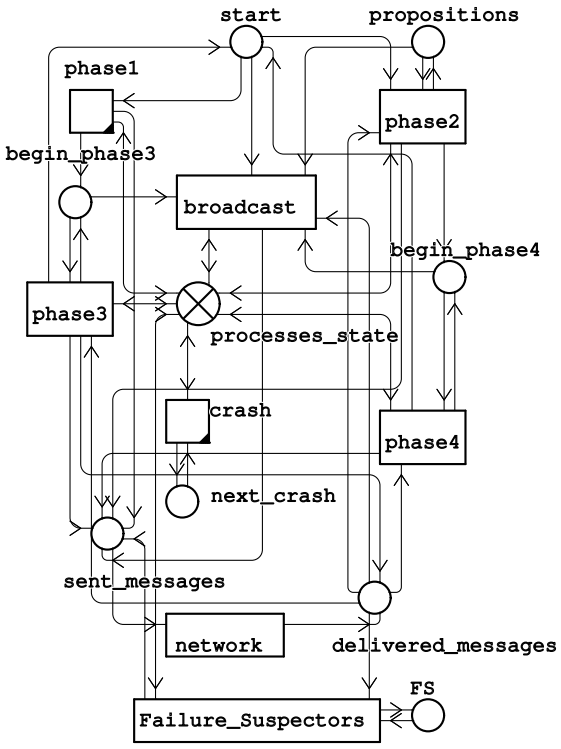

Figure 2 The *consensus* model

A process token from the *start* place can follow two sequences, depending if the modelled process is coordinator or not. If the process is coordinator, it will go through *phase2* system, *begin_phase4* place, *phase4* system and back to the *start* place. If the processor is not a coordinator, it will go through *phase1* processor, *begin_phase3* place, *phase3* system and back to the *start* place. This cycle, modelling one 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 the *sent_messages* place (proposition sent to the coordinator).

The *network* system (*Figure 3*) models the UDP/IP transport layer (see section 2.3). When a process sends a message to another process, it deposes a token in the *sent_messages* place. All the messages exchanged between the Failure Suspectors or generated by the Reliable Broadcast transit through the network system, thus generating an extra-load on the bus which may increase the consensus messages delays.
The *sent_messages* and *delivered_messages* places from *Figure 2*

are identical to the corresponding places in *Figure 3*. This is graphically represented by ¨i¨ or ¨o¨ inside the corresponding circles (for *input* and *output pins*). All the places from Figure 3, except the *resources* place, can contain only message tokens. The token which is 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 modelled as tokens in the *received_messages* place. Tokens in the *delivered_messages* place model messages received at the application level.

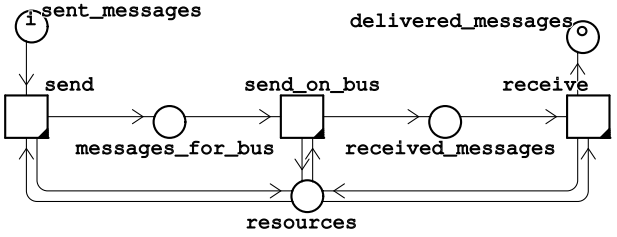

Figure 3 The *network* system

To attain the sending controller level, a message from the *sent_messages* place needs the station's CPU, which is modelled 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, 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.

The *Failure_Suspectors* system is presented in *Figure 4*. The *interrogation* transition sends interrogation messages every 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. At that point p suspects q. The lists of suspected processes are recorded in the *processes_state* store.Upon startup of the algorithm, these lists are empty.
For a detailed presentation of all the transitions and systems of the consensus Petri net model, see (Sergent 1996).

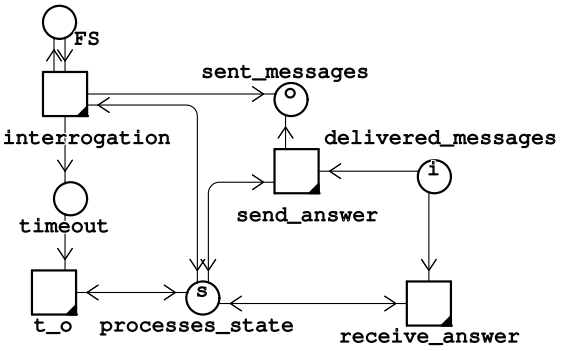

Figure 4 The *Failure_Suspectors* system

4. MODEL VALIDATION AND EXPERIMENTAL RESULTS

Analytical techniques (invariants, occurrence graph) (Jensen 1994) 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 the $\Diamond S$ Failure Suspectors. In practice it is impossible to guarantee the *eventual weak accuracy* property, as discussed in Section 2.1. The consequence is that the proof of termination for the consensus is no longer valid, so a consensus may never finish (the number of rounds may increase infinitely if the coordinator is always suspected by at least one correct process). If we know our Petri net model does not guarantee consensus termination, there is no point to inspect the infinite occurrence graph. Instead, we have found the place invariant M(start)+M(propositions)+M(begin_phase3)+M(begin_phase4) =constant. This place invariant proves that every process is always in some state, i.e. the number of processes does not change.
The model was validated using simulation, in a similar two-step approach to that described in (Clausen and Jensen 1993). Our interest is to use the validated model to study the non-deterministic situations, when crashes occur randomly. Using the Central Limit Theorem from statistics and the independent replications method, we build a 95% confidence interval around the termination time values. The upper bounds of the confidence intervals are plotted in *Figures 9* and *10*.

The simulations were made for n=7, t_S=584 µs, t_R=634 µs, t_n=232 µs. The termination time t plotted in *Figure 9* corresponds to the best case, when no crash failures occur. *Figure 10* depicts the worst case, when the coordinator of the first round crashes. From *Figure 9* we may see that the termination time t decreases as Δ_i increases. The more often interrogations are sent and the shorter time-outs are, the longer the consensus algorithm needs to terminate. This may be surprising, as quick interrogations and short time-outs should make the information exchange more rapid. The problem is that the bus 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 three Δ_i values considered the recommended time-out value Δ_{to} is 20 ms. Smaller values increase the termination time t by producing erroneous suspicions and thus increasing the number of rounds.
From *Figure 10* we may see that Δ_i=25 ms or Δ_i=30 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=35 ms, they would normally provide a smaller t when crashes occur (*Figure*

10), since a crash would be more quickly detected. This is not the case in *Figure 10*, where the Δ_i=25 ms and Δ_i=30 ms curves are above the Δ_i=35 ms and Δ_i=38 ms curves. The explanation is that end-to-end delays are smaller for Δ_i=35 ms and Δ_i=38 ms than for Δ_i=25 ms and Δ_i=30 ms. The conclusion is that adequate Fault Suspectors parameters are Δ_i=35 ms and Δ_{to}=20 ms or Δ_i=38 ms and Δ_{to}=20 ms. If we choose Δ_i=35 ms and Δ_{to}=20 ms the termination time of consensus will be smaller than 76 ms with a probability of 95%. For Δ_i=38 ms and Δ_{to}=20 ms the termination time of the consensus will be smaller than 80 ms with 95% probability. A system designer may choose any Δ_i value greater than 35 ms according to his time-response requirements in the worst case, when crashes occur. The best Δ_{to} value is the one which minimizes t when there are no crash failures. The 5% of risk may be decreased by increasing the guaranteed termination time value.

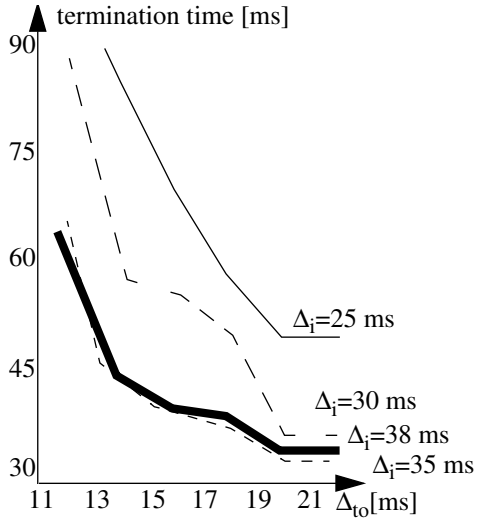

Figure 9 No crash failure

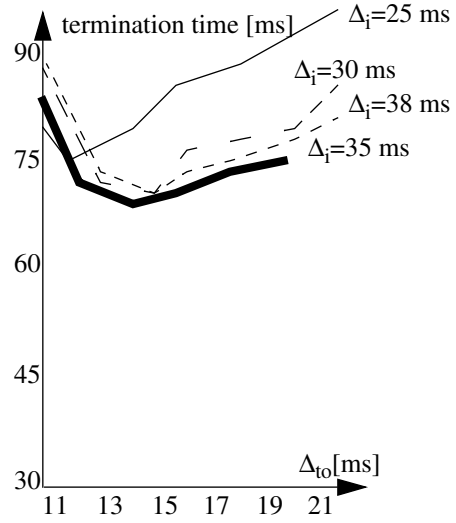

Figure 10 The coordinator of the first round crashes

5. CONCLUSIONS

The advantage of using Hierarchical Coloured Timed Petri nets and the EXSPECT tool is the possibility to make a good trade-off between the size of the net and the length of the code, represented by arc inscriptions and preconditions. Our net size was kept reasonable by writing about 1000 lines of code lines in a functional language. The Petri net model of the consensus protocol presented in this paper, built in a hierarchical style, is very flexible, it allows for instance to modify the network model from UDP to TCP without having to modify the whole net. The model is scalable: while the size (the number of processes in consensus) of the modelled system increases substantially, the size of the model remains constant and the modelling time grows insignificantly. The simulation results prove the success of our modelling approach and enlightens the possibility to design asynchronous distributed systems to obtain the best response time while accepting a pre-defined probabilistic risk. The worst case response time may also be probabilistically evaluated.

References:

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

Chandra T. D. and S. Toueg. 1996. "Unreliable failure detectors for reliable distributed systems". *Journal of ACM*, 34(1), pp. 225-267.

Clausen H. and P. Jensen. 1993. "Validation and Performance Analysis of Network Algorithms by Coloured Petri Nets". *In Proceedings of the 14th International Conference on Applications and theory of Petri nets*. Springer Verlag.

Comer D.E. and L.D. Stevens. 1991. *Internetworking with TCP/IP.* Volume II. Prentice-Hall.

Fischer M., Lynch N.and Paterson M. 1985. "Impossibility of Distributed Consensus with One Faulty Process". *J. ACM*, vol.32, No.2 (April), pages 374-382.

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

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

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

Peterson J.L. 1981. *Petri Nets Theory and the modelling of systems*. Prentice Hall.

Sergent N. 1996. "Performance evaluation of a consensus algorithm with Petri nets". *Rapport technique*, Department of Computer Science, EPFL, Switzerland.

Vachon J. 1996. "Analyse temporelle d'un algorithme de consensus dans un système asynchrone". In *Proc. RenPar'8*, Bordeaux, pp. 65-69.

Voorhoeve M. and B.M. de Reus . 1994. *ExSpect Language Tutorial - Release 5.0*. Bakkenist ExSpect, Netherlands.