A Petri Net Model for FDDI

Nicole Sergent
Laboratoire de Systèmes d'Exploitation
Département d'Informatique
Ecole Polytechnique Fédérale de Lausanne
CH-1015 Switzerland
phone: (+41) 21 693 52 68
fax: (+41) 21 693 67 70
sergent@lse.epfl.ch

Abstract

Fiber Distributed Data Interface (FDDI) is a high speed local area network operating at 100Mbps using a fiber optic ring transmission medium. The purpose of this paper is to build a Petri net model for the FDDI media access control (MAC) protocol. This model is hierarchically integrated into the Petri net model of the two-phase commitment (2PC) algorithm. By simulating the composed Petri net model we evaluate the latency of the 2PC algorithm. The FDDI model is validated by comparing the data obtained from performance measures and the numerical results derived from simulation.

Keywords: FDDI, Petri Nets, modelling, simulation.

1 Introduction

In distributed systems there is neither shared memory nor global clock, so the communication and the synchronisation between processes on different nodes are done by message passing. The distributed algorithm latency cannot be measured if computation begins and ends at different nodes because the clocks are usually not synchronized. Another major impediment in the performance measures approach is that results are biased by unpredictable network and stations loads, unrelated to the studied algorithm. From these two reasons stems the interest in modelling the distributed algorithm and the underlying network, using a formal description technique. By simulating this model it is possible to evaluate the algorithm latency with good accuracy.

A distributed algorithm may be seen as an upper layer process based on a lower layer, the network. Between these two layers there is a strong interconnection. The number of messages generated by the algorithm has a direct influence on the end-to-end delay experienced by a message in the underlying network. This is particularly obvious in Ethernet based networks, where allocating the bus becomes a bottleneck, but this is also true for networks like FDDI. Inversely, the communication delays may greatly affect the number of messages generated during the execution of a distributed algorithm. This interrelation between the network and the algorithm makes it difficult to build an analytical model from which to derive the algorithm latency.

A more general approach is to use a formal description technique, such as Petri nets [10], for modelling both the network and the distributed algorithm and then to derive the algorithm latency using simulation.

Simulation is a well known approach within software engineering disciplines. It has the major drawback of being expensive, in terms of time and computing resources, but its advantage is to allow to solve complicated models which are analytically intractable [3].

In a previous work [11], a Petri net model for Ethernet (CSMA/CD) was built and validated using, as an application, the 2PC algorithm. The Petri net model for Ethernet was reused for evaluating the performances of a consensus algorithm [12]. As more and more Ethernet networks are replaced by FDDI, we aim to build a reusable FDDI Petri net model, which will allow to study the performances of various distributed algorithms, running with this kind of network.

The latency of such an algorithm is mainly determined by the communication costs, i.e., we may reasonably assume that all the operations at the process level are executed in a null time delay. Since only the communication costs are taken into account, the network model must include many details on the communication protocols, in order to be accurate. We have used Hierarchical Coloured Timed Petri nets [7] to build our model, since the classical Petri nets [10] have a low expressiveness and modelling power.

As an application, we have chosen the 2PC algorithm [5], for its simplicity, but the performance of any other distributed algorithm may also be studied within this framework [12]. Using the hierarchy concept, we have built an unified Petri net model, by integrating the FDDI and the 2PC sub-systems. The 2PC algorithm can be easily implemented and evaluated in a real system. Hence, our FDDI model is validated by comparing the simulation results with the data obtained from performance measures. The main interest of this FDDI model is to be reused for evaluating non-deterministic algorithms, such as the Chandra-Toueg consensus algorithm [2].

Our Petri net model is scalable with respect to the number of participant processes, i.e., we may increase the number of processes without modifying the model structure, which would be time-consuming. This is an important asset of the modelling approach over the more difficult performance measures.

The rest of the paper is organized as follows. A description of the FDDI network is given in Section 2. The corresponding Petri net model is presented in Section 3. Section 4 contains the simulation results and some experimental data, which validate our FDDI model. Section 5 concludes the paper.

2 The communication network

2.1 The FDDI protocol

The FDDI [13] is an ANSI standard for high-speed local area networks with a bandwidth of 100 megabits per second. The normal token access method has the disadvantage that a station may have to wait a long time, depending on the number of stations, before receiving the token. This maximum access delay may be unacceptable for some applications.

The timed-token access method, invented by Grow, overcomes this problem by imposing to all the stations to limit their transmissions so that the token rotation time does not exceed a preset target time, the $TTRT$ (target token rotation time). Each station measures the time between successive arrivals of the token, i.e., the TRT (token rotation time). If the measured TRT is equal to or more than the target time (the token is late), the station is not allowed to transmit. If the measured TRT is less than the target (the token is early), it transmits but limits its transmission so that the target is not exceeded. This is achieved by limiting the token holding time (THT) to the difference between $TTRT$ and TRT: $THT = TTRT - TRT$. If the token holding timer expires during the transmission of a frame, the frame is allowed to complete (the yellow light rule). Stations release the token immediately after the transmission of the last frame. The frames go around the ring and are removed by the transmitting station.

There are two modes of transmission: synchronous and asynchronous. The asynchronous mode cannot guarantee bounded access delay, while the synchronous mode does. The more stations on the network, the longer the delay might be. However this happens only when the network is heavily loaded. In a local area network, FDDI is hardly saturated, since the bandwidth is higher than usually required. As a consequence, vendors usually consider the synchronous mode to be optional. Very few commercial FDDI products implement the synchronous mode. This is the reason why, in this paper, we only model the asynchronous mode of FDDI.

Since FDDI protocols are fair only if the priority levels are not implemented, these are not considered in our model.

Let us assume the following notations:

n_s	The number of stations connected on the ring
l_c	The cable length
L	The frame length, in bits
d_s	The station delay, i.e., the delay between receiving a bit and repeating it on the transmitter side ($1\mu s$)

The latency of the links can be computed from the propagation delay of light through the fiber, which is $5.085\mu s/km$. The total ring latency D (the time it takes for a bit to travel around the ring) is the sum of the latencies of the individual stations and the latency of the links: $D = n_s * d_s + l_c * 5.085\mu s/km$.

2.2 The end-to-end delay

In a distributed algorithm, the processes communicate with each other by exchanging messages through a network protocol. We assume point-to-point communications, implemented with UDP sockets. As a network architecture we assume a LAN (Local Area Network), where all the stations are connected on the FDDI ring.

To simplify our model, we make the following assumptions:

- The stations connected on the ring are identical and uniformly distributed along the ring
- There is at most one process running on a station
- The LAN is private, i.e. there is no other message passing on the network except those generated by the considered processes
- There is no message loss, nor transmission error

The end-to-end delay for a message is the time elapsed since the first packet of the message is generated at the sender host until the last packet of the message is received by the destination process on the receiver host. Usually, the messages exchanged in distributed algorithms are small enough to be carried by a single packet, thus the end-to-end delay is defined as the time span between the instant the sender host starts processing a message and the instant the message is received at the destination host. Both times are measured at the process level.

The delay experienced by a message can be decomposed according to the decomposition of the communication path into three subsystems: the sender host subsystem, the network subsystem and the receiver host subsystem.

During the sending operation, a message is processed continuosly by the sending CPU, until it is ready to be transferred to the network controller, by the DMA unit. Sending a message from the process 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 processor speed, the UDP/IP implementation of the UDP/IP protocol and the characteristics of the operating system. It also depends on the size of messages, but we assume that all the messages of the 2PC algorithm have the same size. If a process 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}.

At the receiving host, the process is slightly different. The processing of a message may be interrupted by the arrival of new messages. In general, such interrupts can be amortized by simply adding a delay. For receiving a message, a process 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, the message will wait t_{wR} time units before being received by the destination process.

For sending the bits of a frame on the bus, the controller needs t_t time units, which represents the transmission time; t_t can be computed as $L/100Mbps$, where L is the frame length in bits. The propagation time t_p represents the time it takes for a bit to pass on the ring from the sending to the receiving station. This time depends on the geographical distance separating the two stations and on the number of stations connected between them. We denote by

t_n the time spent by the message in the network subsystem, which is the sum of t_t and t_p. In order to start the transmission of a message, the controller has to wait until it receives the FDDI token. This waiting time is denoted by t_{wn}.

The end-to-end communication delay T experienced by a message exchanged between two distant processes is $T = t_S + t_{wS} + t_n + t_{wn} + t_R + t_{wR}$. While t_S and t_R are constants, t_{wS}, t_{wn} and t_{wR} are variable delays depending on the network and stations loads [1]. This model is similar to those presented in [11] and [8]. For an Ethernet network, t_p can be reasonably assumed constant [11], since it is determined by the length of the bus. For FDDI, t_p is not a constant, we should count the latencies of the stations located between the sending and the receiving hosts. Typical FDDI networks have a ring latency in the range of 100 microseconds [13].

3 The FDDI Petri Net model

3.1 Petri nets

The classical Petri net [10] [9] is a directed bipartite graph with two kinds of nodes, *places* and *transitions*. The nodes are connected by directed, weighted *arcs*. Arcs do not connect nodes of the same type. Places are graphically represented by circles and transitions by rectangles. Places may contain a number of *tokens*, drawn as black dots. A place p is called an *input place* for a transition t if there exists a directed arc from p to t. A place p is called an *output place* of t if there exists a directed arc from t to p. The disposition of tokens on the places is called a marking. $M(p)$ is the number of tokens on the place p. The initial marking of a net is the contents in tokens of each place before executing the net. During the execution the number of tokens may change.

A transition is enabled if each input place contains *enough* tokens, corresponding to the input arc weights. An enabled transition is eager to fire. When a transition fires, it removes the enabling tokens from the input places and produces new tokens for the output places.

Petri nets have been used in many application areas, such as communication protocols, distributed systems, flexible manufacturing systems, hardware design, information systems. Classical Petri nets have been extended by introducing *colours*, *time* and *hierarchy* [7].

3.2 The FDDI model

We have built our Petri nets using ExSpect, a software tool developed by the Eindhoven University of Technology [4], [15], [14]. 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 corner denote transitions, rectangles denote systems, circles denote places, crossed circles denote one-capacity places (*stores*). Exspect allows interactive net simulation.

The *communication* Petri net model is presented in Figure 1. For readability reasons we do not include arc inscriptions, written in a functional language, together with the nets. Instead we present in detail the functioning of the net.

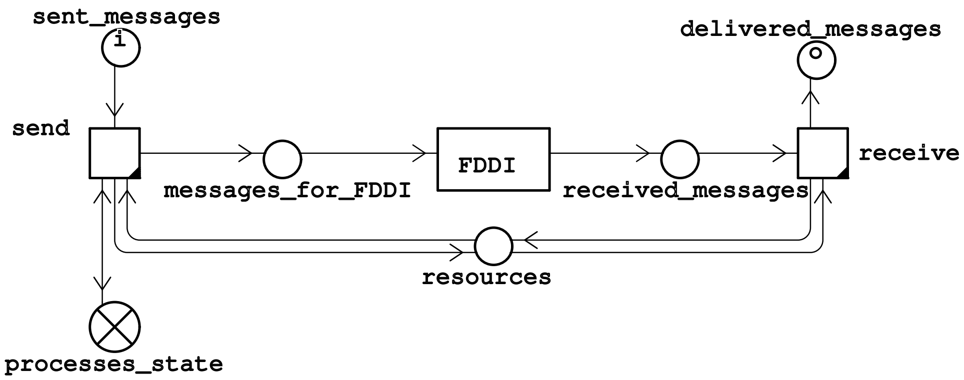

Figure 1: The *communication* system

When a process sends a message to another process, it puts a token in the *sent_messages* place. The token carries several data, such as the source and destination process identifiers, the message type and the message contents. The tokens at the *messages_for_FDDI* place represent messages that are waiting for the FDDI token at the sending controller level. Messages received at the controller level but not yet received by the process are modelled as tokens in the *received_messages* place. Tokens in the *delivered_messages* place represent messages received at the process level.

To reach the controller, a message from the *sent_messages* place needs the station's CPU, which is modelled by a token residing in the *resources* place. This place contains a token for each station CPU. Colours allow to distinguish the different stations. The *send* transition models a send operation from the process 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, for the *resources* and *messages_for_FDDI* places. The FDDI system models the FDDI token allocation and the messages transmission. The *receive* transition models the message reception at the process level and the station's CPU allocation during t_R time units.

The *processes_state* store contains several data which completely describe the state of the Petri net at any moment. This store corresponds to a global view of the system, hence the access at these data is restricted, i.e., a process is allowed to consult information only about his own state variables (otherwise it would be a violation of the non shared memory assumption).

The communication Petri net model supports also IP multicasts. A multicast message has more than one destinations. The message passes once on the network and is received by all the destination hosts.

The components of the FDDI system depicted in Figure 1 are presented in Figure 2. The Petri net model of FDDI (Figure 2) contains one input pin, (*messages_for_FDDI*) and one output pin (*received_messages*), which are connected to the corresponding places in Figure 1 (we are using here the hierarchy concept [6] of Petri nets to hide the complexity of the FDDI system in the communication model).

Tokens put on *messages_for_FDDI* place represent frames arrived at the queue that contains frames to be transmitted on FDDI. The tokens on the *received_messages* output pin model frames received by the controller of the destination host. The *token* place always has a single token, the FDDI token,

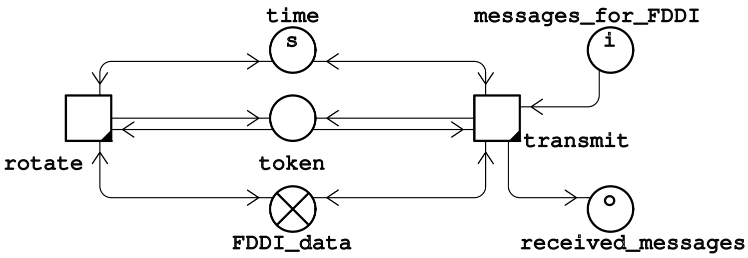

Figure 2: The FDDI system

which carries two informations: the identifier of the station which currently has the FDDI token and whether a transmission is currently taking place or not.

Each station registers the number of messages waiting to be transmitted at its queue and the moment of the last arrival of the FDDI token. These data are updated in the *FDDI_data* store. The current time in the system (the simulation time) is always given by the value of the token from the *time* store. This time is used for computing the TRT.

The *rotate* transition models the progression of the FDDI token through the network ring when no message is transmitted. The condition for the firing to take place is: *[(the current station has not reserved the FDDI token for a second transmission) or ($THT < 0$)] and (the current station has no message to transmit).* It fires by consuming the token from the *token* place and by producing a new one, in the same place, delayed by D/n_s time units. This firing models the action of passing the FDDI token from the current station (the one which holds the FDDI token) to the next station on the ring. It also updates the timing information from the *FDDI_data* store by recording the current time, which is the moment of the last arrival of the FDDI token at the current station.

The *transmit* transition models the action of transmitting a message on the ring. The firing condition is: *($THT > 0$) and (the sender of the message has the FDDI token).* When this transition fires, it removes a message token from the *messages_for_FDDI* place and deposes it in the *received_messages* place, delayed with $t_t + t_p$ (t_p depends on the number of stations connected between the sender and the receiver hosts). It also updates, in the *FDDI_data* store, the moment of the FDDI token arrival and the number of messages to be transmitted by the current station. If the station has more than one message to transmit and $THT > 2t_t$, then the station reserves the token for another transmission, otherwise it releases the token and passes it to the next station, thus delaying the message for extra $t_t + D/n_s$ time units.

3.3 The 2PC model

To ensure consistency in a distributed data base, transaction updates involving several sites must be integrated in the database by an atomic action. Protocols for ensuring transaction atomicity are called commit protocols. The Two-Phase Commitment protocol (2PC) is proposed in [5]. We denote by n the number of participant processes.

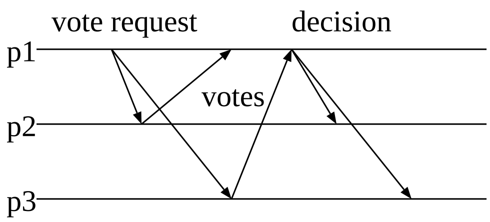

Figure 3: The 2PC algorithm with 3 processes

In the first step, the coordinator sends $n-1$ *vote-request* messages, to all the other participant processes, and each process votes on whether to commit (*yes*) or to abort (*no*) the transaction (see Figure 3). In the second step, the coordinator gathers all the votes and informs each participant on the outcome. The termination time (latency) for this protocol is the time elapsed from the moment the coordinator sends the *vote-request* message to the first participant until all the participants have received the coordinator's decision.

The Petri net model for the 2PC algorithm is depicted in Figure 4 [11]. The *communication* system in Figure 4 represents the Petri net depicted in Figure 1. The *sent_messages* and *delivered_messages* places are the connections between the two Petri nets.

Before executing the net, the n processes, modelled by n coloured tokens,

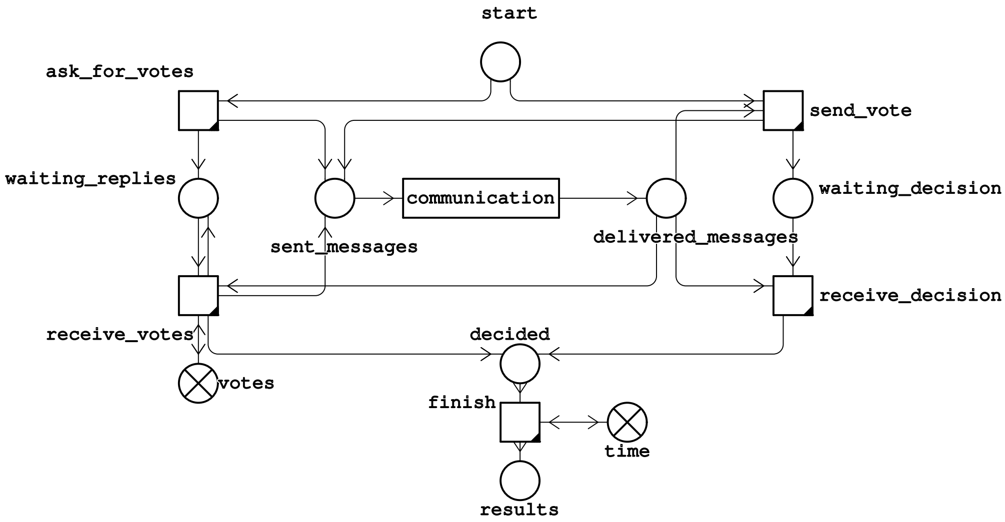

Figure 4: The 2PC Petri Net model

are located at the *start* place. Each token has a colour which models the *pid* of the corresponding process. One of the processes (for instance, the one which has the greatest *pid*) is the coordinator. The *ask_for_votes* transition firing consumes the coordinator token and produces $n-1$ token messages, one for each participant, in the *sent_messages* place. The coordinator token is put in the *waiting_replies* place, reflecting the new state of the coordinator process. Each participant waits in the *start* place until it detects a *vote_request* message token in the *delivered_messages* place. Colours allow a participant to detect whether the message is addressed to him or not. Then the *send_vote* transaction fires and the vote token is put in the *sent_messages* place. A message token contains the same data as in the FDDI Petri net model, i.e. the source and destination identifiers, the message type and the message contents. After sending his vote, a participant proceeds to a new state, reflected by a token output in the *waiting_decision* place.

The coordinator updates the number of votes he has currently received in the *votes* store. After receiving $n-1$ votes, the *receive_votes* transaction will output $n-1$ message tokens in the *sent_messages* place and a coordinator token in the *decided* place. The *finish* transition detects the algorithm end, when all the participants have decided. It records the current time in the *results* place, using the current time value from the *time* store, updated by the ExSpect simulator.

4 Simulation and experimental results

The model was validated by the invariant analysis technique and by performance measures. A special analysis tool, the IAT, found the place invariant M(start)+M(waiting_replies)+M(waiting_decision)+M(decided)=constant. This invariant proves that the number of processes does not change during the execution of the net.

The performance measures we have made to evaluate the algorithm latency are approximate: we measure the time elapsed from the moment when the coordinator sends the transaction to all the participants until the coordinator gathers all votes from the participants (this is possible since the measurements are made on the same station) and we add half of the round-trip time. In order to minimise the bias from extra-loads on stations and network, the approximate performance measures were done at unsocial hours.

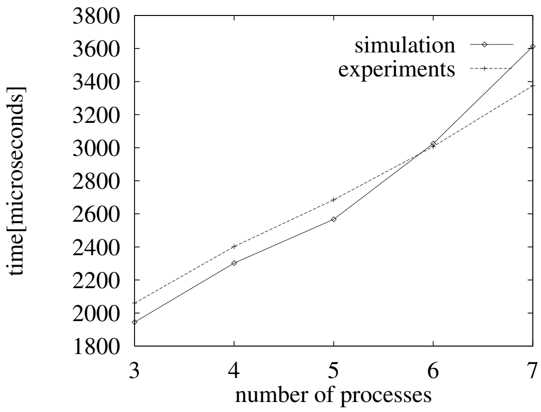

Figure 5: 2PC latency - Simulation and experimental results on FDDI

The simulations were made for $t_S = 180\mu s$, $t_R = 180\mu s$, $n_s = 64$, $l_c = 10km$, $d_s = 1\mu s$, $L = 64bytes$, $TTRT = 8ms$. The experimental and the simulation results obtained for the latency of the 2PC algorithm, running on FDDI, are presented in Figure 5. The latency (in μs) is shown as a function of the number of participant processes.

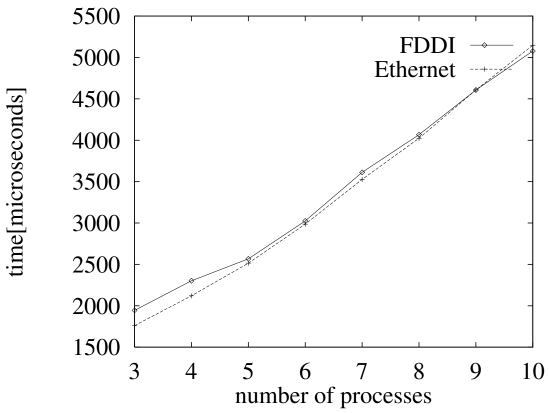

Figure 6: 2PC latency - Simulation results on FDDI and Ethernet

It is possible to compare the performances of the 2PC algorithm running on different networks. The simulation results for the two types of networks, FDDI and Ethernet, are presented in Figure 6. Again, the latency of the 2PC is shown as a function of the number of processes. The parameters injected in the Ethernet model are $t_S = 180\mu s$, $t_R = 180\mu s$, $t_n = 107\mu s$ [11]. For a small number of participants (< 9), the 2PC algorithm runs faster on Ethernet. The explanation is that our FDDI ring, with 64 stations connected on it, has a latency $D = 115\mu s$, so a station may have to wait quite a long time before receiving the token. Even if the transmission time is 100 times smaller than for Ethernet, the increased propagation time leads to poor performances. As the number of participant processes goes larger, the probability that several messages are transmitted during a single token rotation over the ring is increased, so the 2PC algorithm performs better. For more than 8 processes the algorithm runs faster on FDDI than on Ethernet.

5 Conclusions

The Petri nets framework offers ample opportunities for graphical modelling of the various causal dependencies between the steps taken by the processes in a distributed system. Simulation makes it possible to predict an algorithm latency or to compare the performances of several algorithms that are solving the same problem. This approach is very attractive at the design stage, before starting to implement a distributed system, since it provides valuable information of the system at an early stage.

In this paper, we have developed an accurate Petri net model for the FDDI network. The FDDI model was integrated into the 2PC algorithm model, in order to be validated. The reason for this is the facility to make performance measures at the process level rather than at the network controller level. The model was validated by simulation. The performance results obtained by simulating this model were proved to be close enough to the performance measures derived from a real system.

Simulating the model is more attractive than performance measures, since the results are not biased by unpredictable system loads or by approximations due to the lack of global time. The model is scalable; while the size (the number of processes involved in the 2PC algorithm) of the system increases substantially, the size of the model remains constant and the growth of the modelling time is negligible.

In further work, the FDDI model will be integrated in the consensus algorithm model [12], in order to compare the performances of the consensus running in two different networks: Ethernet and FDDI. Using the same hierarchical framework, the Petri net model of the consensus will be integrated into the atomic broadcast model, gaining new insights on the behaviour of this algorithm.

References

[1] L. F. Cabrera, E. Hunter, M. Karels, and D. Mosher. User-Process Communication Performances in Networks of Computers. *IEEE Transactions on Software Engineering*, 14(1), 1988.

[2] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of ACM*, 34(1):225–267, 1996.

[3] T. Delaitre, G. R. Justo, F. Spies, and S. Winter. EDPEPPS: An Environment for the Design and Performance Evaluation of Portable Parallel

Software. In *Proceeedings of the 5-th Euromicro Workshop on Parallel and Distributed Processing*, London, January 1997.

[4] Bakkenist ExSpect. *ExSpect Reference Manual-Release 5.0.* The Netherlands, November 1994.

[5] J. N. Gray. *Notes on Database Operating Systems*, volume 60 of *Lecture Notes in Computer Science*, pages 393–481. Springer-Verlag, 1978.

[6] P. Huber, K. Jensen, and R. M. Shapiro. Hierarchies in Coloured Petri Nets. In *Advances in Petri Nets.* Springer-Verlag, 1990.

[7] K. Jensen. *Coloured Petri nets : basic concepts, analysis, methods and practical use*, volume 2. Springer-Verlag, 1994.

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

[9] M. K. Molloy. Petri Net Modelling, The Past, the Present and the Future. In *Proceedings of the International Conference on Petri Nets and Performance Models*, 1989.

[10] J. L. Peterson. *Petri Nets Theory and the Modelling of Systems.* Prentice-Hall, 1981.

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

[12] N. Sergent. Performance Evaluation of a Consensus Algorithm with Petri Nets. In *Proceedings of the 7th International Conference on Petri Nets and Performance Models*, Saint-Malo (France), June 1997.

[13] A. Shah and G. Ramakrishnan. *FDDI - a high speed network* . Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1994.

[14] W. M. P. van der Aalst. Analysis of Railway Stations by Means of Interval Timed Coloured Petri Nets. *Real-Time Systems*, 9:241–263, 1995.

[15] M. Voorhoeve and B. M. de Reus. *ExSpect Language Tutorial - Release 5.0*. Bakkenist ExSpect, The Netherlands, December 1994.