A Two-Level Technique for Modeling Fault-Tolerant Systems

UCLA Computer Science Department Technical Report: TR-980024

Mike Loving and David Rennels
University of California, Los Angeles,CA 90024
rennels@cs.ucla.edu, loving@cs.ucla.edu

Abstract

By extending stochastic Petri nets we believe it is possible to simulate much of the intricate behavior of fault-tolerant parallel processing systems with a concise and relatively simple and intuitive model. A series of augmentations of stochastic Petri net models is described in which color and linked-list attributes are added that allow modeling of detailed behavior of a variety of scalable intercommunication systems of virtually any size with the same small model. A simulation system is described, and it is further augmented using a two-level model to describe and model Birman's ABCAST and CBCAST atomic broadcast protocols. These were chosen both as a complex behavioral example to test our modeling approach and also because atomic broadcast is an essential mechanism in implementing fault-tolerant parallel systems.

1. Introduction

Although Stochastic Petri Net modeling is often a powerful tool in system design, it is often left in the domain of modeling specialist because of the complexity of setting up models. This work was initiated as an attempt to simplify construction of models of this type to make them more accessible for direct use by system designers. The first insight that propelled this work is that by adding color (attributes) to the tokens especially attributes in the form of linked lists, one could build simple but powerful models that express a great deal of the detailed behavior of the most commonly used high performance interconnection networks. The second insight, borrowed from earlier research by Estrin at UCLA was that by coupling a Petri Net model with a second higher level behavioral model, one could perform sophisticated behavioral modeling using a very compact and intuitive specification. To test these techniques, we have developed a simulation system based upon them and applied it to modeling Birman's ABCAST and CBCAST atomic broadcast protocols - one approach to implementing the building blocks needed to construct fault-tolerant parallel systems [Cri 85].

1.1 Adding Color to Stochastic Petri Nets

It has been shown [Pet81] that Petri Nets with inhibitor arcs are equivalent to Turing machines in computational power and can thus perform any computable task. A version called Stochastic Petri Nets (SPN) have found extensive use in modeling because timing distributions can be associated with the firing of each transition thus allowing fault-rates and recovery rates to be incorporated in performability models or the times expected for recovery actions in behavioral models. Adding attributes (colors) to the tokens can greatly simplify these models. Consider a Stochastic Petri Net model for the well known Dining Philosophers problem (Figure 1-1). Large circles represent "Places"; arrowed lines represent "Arcs"; solid lines represent "Transitions", and solid dots represent "Tokens". The state of the system is entirely defined by the distribution of tokens in places; this is known as the "Marking" of the net.

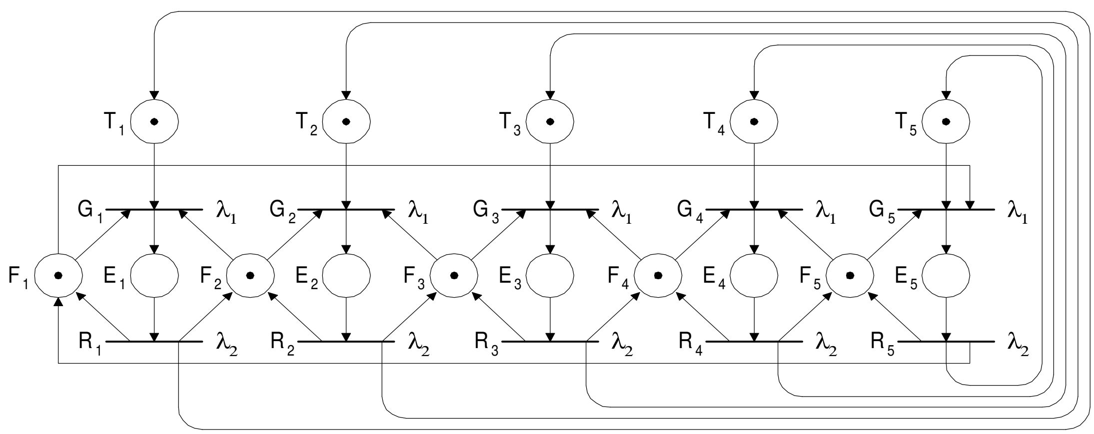

Figure 1-1: SPN for Dining Philosophers

Briefly, each of the five philosophers is either thinking (token in the T place) or eating (token in the E place). The F places hold available forks. In order to move from thinking to eating, the philosopher must acquire the two neighboring forks. This "Get Forks" operation is represented by the G transitions, which require a token from a T place and the two adjacent F places. Lack of available fork tokens will prevent G transitions from firing. When the philosopher has completed a sojourn in the eating place, they return both forks and resume thinking; this is represented by the R transitions which require an E input and introduce an output token into the corresponding T place and to both adjacent F places. The G and R transitions are both "timed" transitions meaning that they do not fire immediately upon becoming enabled. Instead there is a time delay that can be constant (deterministic) or random. Random delays can be of any distribution, but exponential distributions give isomorphism with continuous time Markov Chains.

Now consider a Colored Stochastic Petri Net [Zen85][Lin88] representation of the dining philosophers system as shown in Figure 1-2.. In this model, the tokens are of two types - philosopher or fork, and each token has a single attribute - an index. Predicates are associated with the transitions so that firings only occur if specified attribute conditions are met by the input tokens.

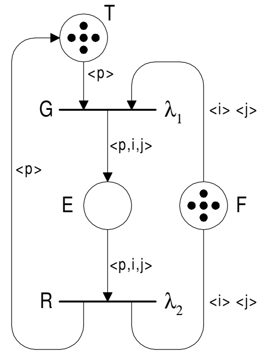

Figure 1-2: CSPN Dining Philosophers

The CSPN Philosophers model is much simpler than the SPN version. There is only one T place where all philosopher tokens spend time thinking. The single G transition allows any philosopher token <p> to obtain two fork tokens <i> and <j> subject to the following constraints:

$$p = i \quad \text{and} \quad p = (j-1) \bmod 5$$

These constraints or "predicates" in CSPN parlance ensure that a philosopher may only use the two neighboring forks. Similarly, there is but one E place where tokens representing a philosopher and his forks reside while he is eating. When the R transition fires, it consumes one token from the E place, returns the two fork tokens to the F place, and returns the philosopher token to the T place. Note that a token in the E place must have attributes from the tokens it absorbed attached so that when the R transition fires, the proper fork and philosopher tokens can be returned with their corresponding index values. The G and R transitions are both timed transitions that have exponentially distributed delays of mean λ_1 and λ_2 respectively.

The use of color can increase the utility of Petri Nets in the same way a high level language can increase one's productivity as compared to programming in an assembly language. Even with Colored Stochastic Petri nets, one soon finds that many modeling tasks would benefit greatly from enhancements to the basic system. For example, figure 1-3 shows the CSPN model for an 8 node circuit-switched or wormhole-routed hypercube.

Place P is the "private" place; tokens in P represent processors executing privately. L is the "link" place; tokens in L represent available links. CB is the "connection building" place; tokens in CB represent messages that need to go to another processor. There are three paths leaving CB. The leftmost path is for those messages that need to travel only one hop to get to their destination. The rightmost path is for those messages that need to travel three hops and the middle path is for those messages that need to travel two hops.

Note that tokens representing messages in the Hypercube must have three attribute fields that hold the identity of the links they have captured so that when a message has been delivered the links can be returned to the L place for other messages to use. The transition predicates implement a routing algorithm that determines what links can be captured to move a message closer to its destination. More sophisticated versions of this with many more transitions and arcs can employ backtracking where links are returned when forward progress is blocked.

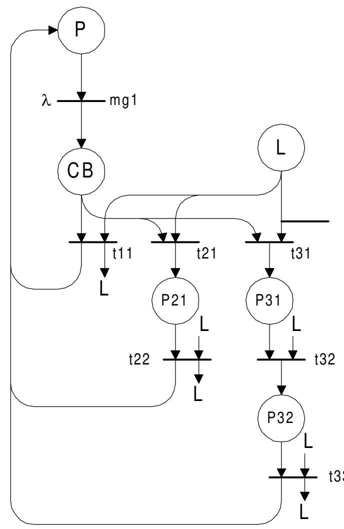

Figure 1-3

1.2 PSPN – An Enhanced Colored Stochastic Petri Net Model

As one can see, even CSPN models can get fairly complex as the intricacy of system being modeled increases, and the size of the model tends to increase with the number of processing nodes in the system being modeled. We have developed a colored stochastic Petri Net model that we call the Polyvalent Stochastic Petri Net (PSPN). PSPN includes timing information as SPNs do and token attributes as CPNs do and includes additional token attribute features that further reduce the complexity of the models.

A principal feature of the PSPN model is that it provides for the use of a linked-list in one or more attributes of a token. This allows models to be constructed whose structure is independent of the number of nodes in a system – no matter how many resources are captured as a token passes through a net, the same set of predicates can be applied to the last n entries on the linked-list attribute. The similarity with the way chemical reactions occur based on outer electrons caused us to possibly engage in hyperbole and choose the descriptive adjective Polyvalent.

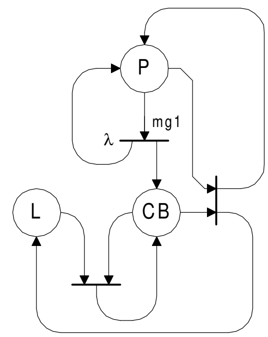

Figure 1-4

To simplify this complexity, we have developed a colored stochastic Petri Net model that we call the Polyvalent Stochastic Petri Net (PSPN). PSPN includes timing information as SPNs do and token attributes as CPNs do and adds token attribute features which reduce the complexity of the models. For example, figure 1-4 shows the PSPN model for the same 8 node hypercube described above. It is a much simpler model than the CSPN model and with the PSPN modeling system, it is also the model for any size hypercube. With appropriate changes to two of the transition predicates, it is also the model for any size toroidal mesh.

1.3 Description of the simulator

Figure 1-5 shows the basic flow of our simulator generator tools. A text description of model to be simulated is written up in a Net Description File (NDF) and used as input to PSPNGEN. PSPNGEN reads the NDF and generates several C language source files (Net Dependent Source or NDS) and these are compiled together with the Net Independent Source (NIS) files and any other User Supplied Source files to generate a PSPN simulator.

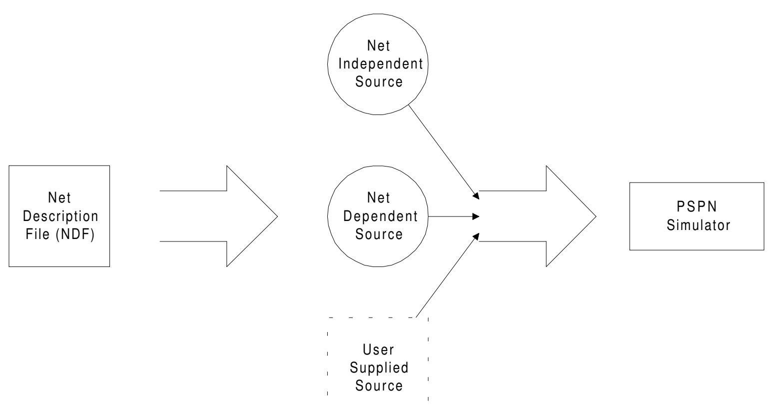

Figure 1-5: Simulator Generation Flow

The NDF contains a complete description of the net with all places, transitions, arcs, and the initial marking of the net. Transition descriptions contain predicates, firing rules, and timing information. Places also define the attributes (color) of the tokens they hold. The NDS generated from the NDF contains the code that sets up the basic data structures of the simulator and the initial marking. It also contains the predicate, firing rules, and timing functions.

The NIS files contain the "engine" that runs all of this, which is primarily the scheduler. The scheduler examines all transitions, finds those that are enabled by applying transition predicate functions, and schedules those that are enabled for firing at a future time determined by the transition's timing function. The first iteration of the scheduler is fairly involved, but, after that, the set of transitions that may be enabled is much smaller and many optimizations are possible. After the initial marking of the net has been fully examined by the scheduler and all enabled transitions are scheduled for firing, execution begins. That portion of the simulator is a straightforward event-driven simulator, which removes the next scheduled transition firing from the front of the firing queue and calls that transition's firing rule function (part of the NDS code). The process then repeats itself.

We have used this simulator generator to develop simulation models of many different types of systems such as hypercubes and toroidal meshes using both packet switching and circuit switching, including the use of various backtracking algorithms. The flexibility of the PSPN system has allowed us to study these systems with various loads and configurations and various numbers of failures to study the fault tolerance properties of the systems. We have also used this system to develop a model for a large Omega network that was being studied by another researcher here at UCLA. The original handcrafted simulator for that project took over a year to write whereas the PSPN model was up and running less than 1 month after we started on it. Certainly the handcrafted simulator would run a given simulation much more rapidly, but one can learn a lot with a slow simulator in the 11 extra months that it took to develop the fast simulator.

2. Coupling PSPN with a High Level Model

Modeling of higher level behavior is extremely cumbersome and counterintuitive if done in a single Petri Net, SPN, or PSPN model. Certainly, the higher level nets allow higher level systems to be modeled more concisely than with the lower level nets, but there is always a more complicated system on the horizon that is beyond the reasonable capabilities of a given type of net. As examples, one only needs to consider modeling the details of a directory-based cache consistency scheme in a scalable computer, or a fault-tolerance scheme. Here, much of the system and the availability and use of resources can more easily be thought of as independent of the tokens which represent actions and messages. Since the Petri net is equivalent to a Turing machine, one could certainly model it in theory, but the result would likely be an intractably large net.

We have investigated the feasibility of extending PSPN to allow coupling a deterministic high level partner model representing system state to the basic PSPN model. The compound model would ideally retain the conciseness of PSPN while allowing study of more complex systems. The value of this type of approach was verified in the UCLA SARA modeling system many years ago although the underlying modeling system was quite different [Est 78]. We then implemented models of the CBCAST and ABCAST protocols over a hypercube connected network in order to experiment with the capabilities of the expanded model.

A brief investigation of the ABCAST protocol reveals that implementing it on a pure PSPN would be a serious endeavor. Closer inspection shows that, with some extensions to PSPN, the model can be implemented much more simply and in a fashion which we believe to be more intuitive to designers.

2.1 Compound PSPN and High Level Models

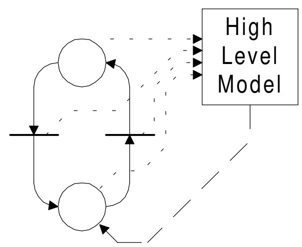

Figure 2-1

Figure 2-1 shows a conceptual depiction of our compound model. The PSPN portion of the model provides the basic "engine" with timing, and such features as it can provide. The High Level Model keeps track of the more intricate information that would be difficult to incorporate into the PSPN portion. As implied by the figure, the high level model exchanges information with the PSPN model, both keeping track of high level information and modifying the action of the PSPN model.

2.2 Exchanging Information Between the Models

When one adds the high level model to the basic PSPN model, it is necessary for information to be exchanged between the two. There must be ways to pass information to the high level model and ways for it to feed information back to the PSPN model.

Since the only time that anything is happening in the PSPN is when a transition fires, that is the only point in time at which it is necessary to pass information to the high level model. If the transition fires at time t, it suffices to allow calls to the high level model at times $t-\varepsilon$ and $t+\varepsilon$. The first possible call allows the high level model to see and operate on the input tokens; the second call allows the same flexibility with the output tokens. Additionally, whenever the high level model is invoked, it may want to examine the state (marking) of the system.

Feedback to the PSPN model is implemented in 3 ways

- **Modification of PSPN marking** -- is implemented by having the high level model construct the appropriate token objects and then return them to the PSPN portion for insertion in the desired place.

- **Modification of transition enabling rules** -- is implemented by having the high level model modify state variables or simply replacing the predicate functions with alternative instances of the predicate function.

- **Modification of transition firing rules** -- Conceptually, modification of the transition firing rules is a useful concept. In actual practice, it is implemented by combining the $t-\varepsilon$ high level call, with the capability of modify the PSPN marking discussed above.

One could certainly use the high level model to effect changes in a transitions firing distribution (timing), however we have not investigated that.

2.3 Implementation of Information Exchange

Examples of the actual implementation of the methods of exchanging information are shown below.

2.3.1 Firing Time Calls to High Level Model

The calls to the high level model are implemented in the C code by allowing the insertion of function calls at two points in each transition firing rule.

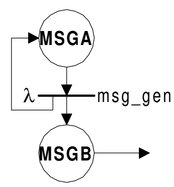

Figure 2-2: A Simple PSPN Transition

Given a simple PSPN transition as shown in figure 2-2, PSPN source for a simple transition definition is of the form:

```
tran msg_gen(MSGA;MSGA,MSGB):exp(1000.0)
{
  predicate:{1};
  MSGA.pid = MSGA.pid;
  MSGB.pid = MSGA.pid;
}
```

The first line of transition definition contains the name of the transition, lists of its input and output places (in parentheses), and the timing distribution. The third line has the predicate (which is always true) and the fourth and fifth lines contain the firing rule. The firing rule assigns values to the attributes of all output tokens (lvalues in assignment statements) using expressions based on the attribute values of the input tokens.

The firing rule is actually just the assignment statements after the predicate, but the complete transition definition is shown for clarity. The C code produced by PSPN for this is:

```
void msg_gen_rule(e)
EVENT *e;
{
/* variable declarations */
TOKEN *MSGA_IN;
TOKEN *MSGB_OUT;
TOKEN *MSGA_OUT;

/* storage allocation for output tokens */
MSGB_OUT = malloc_token();
MSGA_OUT = malloc_token();

/* extraction of input tokens from input places */
MSGA_IN = e_xtract_token(pmat[2],e);

/* 1 */

/* attribute assignments */
MSGA_OUT->attr[0+(0)] = MSGA_IN->attr[0+(0)];
MSGB_OUT->attr[0+(0)] = MSGA_IN->attr[0+(0)];

/* 2 */

/* insertion of output tokens into output places */
insert_token(pmat[3],MSGB_OUT);
insert_token(pmat[2],MSGA_OUT);

/* deallocation of storage for input tokens */
free(MSGA_IN);
}
```

The points labeled as /* 1 */ and /* 2 */ represent the $t-\varepsilon$ and $t+\varepsilon$ points discussed in the previous section.

At point 1, the generated code has allocated space for all output tokens and retrieved all of the input tokens from the appropriate input places. At this point one has essentially a blank slate to work with and can do just about anything. At point 2, the attribute assignment statements have all been executed to reflect the functionality described in the net description file, but the output tokens have not yet been inserted into the output places.

In our studies of ABCAST, we have inserted all of our external function calls at point 2 and then have the high level model implement the intricate bookkeeping for the ABCAST algorithm that is explained in Section 3.

The call to the high level model passes pointers to all input tokens and all output tokens of the currently firing instance of a transition. The high level code may then record whatever information it needs, and take whatever action it desires. These actions, as previously mentioned include modifying the PSPN predicates, marking, and transitions.

2.3.2 Predicate Modification

In any PSPN model, each transition has a predicate which specifies some relationship of the attributes of the combination of input tokens. When a combination of input tokens satisfies (causes it to evaluate to TRUE) the predicate, the transition is said to be enabled and is scheduled to fire and consume those input tokens.

Predicates are implemented in PSPN as indirect calls to predicate functions which return an int. A simple predicate such as

```
predicate:{1};
```

results in C code that looks like:

```
return(1);
```

more complicated predicates like:

```
predicate:{MSGB.pid == p.pid};
```

result in more complicated code i.e.:

```
int mg0_predicate_0(){
return( pmat[3]->current_token->attr[0+(0)] ==
        pmat[0]->current_token->attr[0+(0)]
      );
}
```

The PSPN parser requires that the predicate actually be an expression and this limits what can cleanly be done in the PSPN source code. Certainly the C operators ? and : allow significant additional flexibility, but things like loops, function calls and etc. can greatly add to the decision making power of the predicate.

If one uses PSPN to generate the basic simulator, but then modifies the generated C code it is possible to obtain much greater functionality. When this is done, arbitrarily complex relationships can be checked far more easily than if it must be coded as a C expression. Additionally, since the call to the predicate function is implemented indirectly (i.e. something like `*(pred_fn)();` in C), it is straightforward for the high level model to modify the function pointer to point to an entirely different function.

2.3.3 Marking Modification

Should the high level model need to, it may create new tokens and insert them in any Place in the PSPN model. The code to create and insert tokens is provided as part of the PSPN Net Independent Source code.

```
TOKEN *MSG;
MSG = malloc_token();
MSG->CB_SOURCE = 0;
MSG->CB_DEST = 1;
insert_token(CB,MSG);
```

The PSPN generator produces the appropriate header files so that symbolic names can be used for Places, Transitions, and token attributes. Removal of tokens from a place is somewhat more complicated. It requires first finding the token, extracting it from the place, and then removing any transition firings that are attached to the token.

2.3.4 Firing Rule Modification

As previously mentioned, modification of the semantics of a firing rule is implemented by using the available calls in section 2.3.1 together with the tools for modifying the PSPN marking outlined in section 2.3.3.

3. Modeling of Atomic Broadcast Protocols

As a demonstration of the modeling power of augmented PSPNs, we wanted to select a reasonable real world problem of sufficient complexity to demonstrate their power. We felt that the ABCAST and CBCAST protocols would also serve as excellent examples of the type of underlying mechanism needed to implement fault-tolerant distributed systems.

3.1 Description of the Protocols

ABCAST and CBCAST are message protocols developed by Birman et. al. at Cornell [Bir91]. In brief, applying the CBCAST protocol provides causal ordering; applying ABCAST provides total ordering.

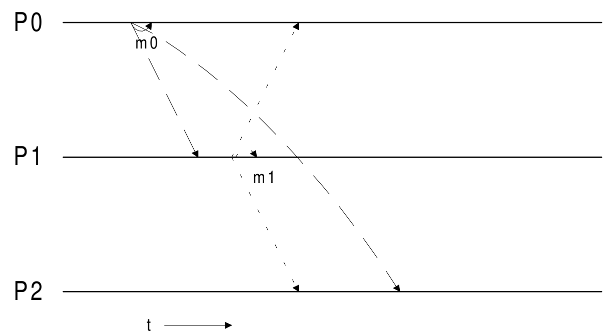

Figure 3-1: Non-Causal Message Delivery

As shown in figure 3-1, the problem addressed by CBCAST is when P_2 receives message m_1 before it receives message m_0. Since m_0 precedes m_1 on P_1, causality would require that m_0 also precede *m1* on P_2.

In order to alleviate this problem, the CBCAST protocol requires that each processor maintain a virtual time vector (vector of sequence numbers). Prior to sending a message, P_i increments $VT_p[i]$, and timestamps *m* with the updated *VT*. Destination processors or processes distinguish between receipt of messages and delivery to the application process. The OS (or whatever piece of low level software is implementing the protocol) queues received messages but defers delivery until the recipient's *VT* catches up to the message's *VT*. The recipient "catches up" by delivering messages that the sender had received when it sent *m*. More formally, the recipient may deliver *m* when the following criteria are satisfied:

$$VT_m[i] = 1 + VT_p[i] \quad \text{and} \quad VT_m[k] \le VT_p[k] \quad \forall k \ne i$$

These two rules are just formal statements of two very simple rules. The first rule states that this message *m* must be the next sequential message from processor *i*. The second rule states that we can't deliver *m* until we have delivered all other messages that processor *i* had received prior to sending *m*. Upon delivering *m*, the receiving processor updates its VT vector, to reflect that it has delivered *m*.

Figure 3-2 shows how CBCAST works. When message m_1 arrives at P_2, delivery is delayed until the virtual timestamp rules are satisfied; this ensures that m_0 gets delivered first.

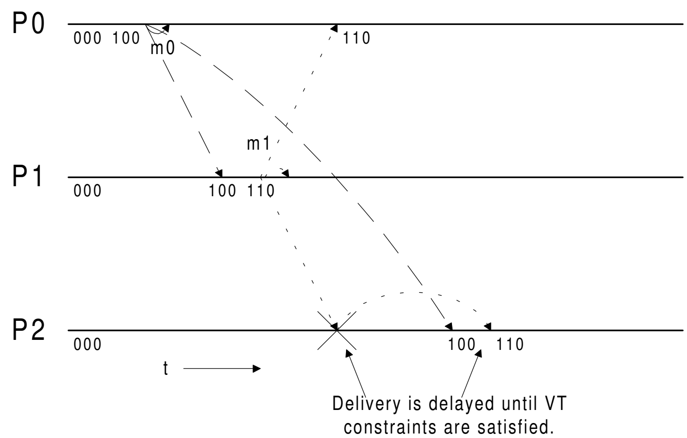

Figure 3-2: Causal Message Delivery

Even with causal message delivery, one could still have the situation depicted in figure 3-3 in which messages m_0 and m_1 are delivered by all the rules of CBCAST, but are not delivered in the same order on the two recipients. P_0 has no knowledge of m_1 to prevent immediate delivery of m_0 and P_1 has a similar lack of knowledge about m_0.

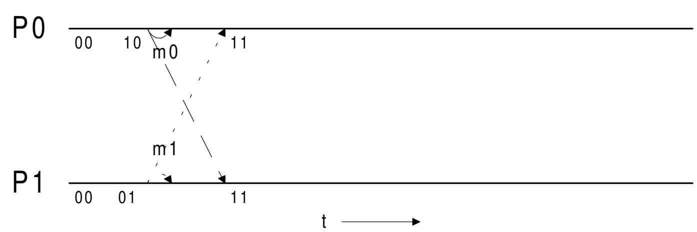

Figure 3-3: Causal but not Totally Ordered Message Delivery

ABCAST addresses this shortcoming by expanding on CBCAST. ABCAST designates one processor as the Token Holder. The token holder is the only processor which can deliver ABCAST messages without a specific authorization; all other processors queue these messages until the token holder notifies them of which messages to deliver and in what order. The token holder waits until it has received and delivered T messages to itself using normal CBCAST rules. The token holder then authorizes other processors of the messages to be delivered and their ordering (via a special message known as a Sets Order message or SO message). T is referred to as the "trigger level" and is a parameter which can be varied to suit a given system. Low values for T increase the overhead for SO messages, but high values for T increase latency.

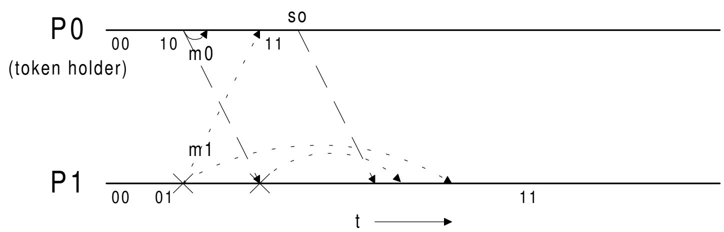

Figure 3-4: Totally Ordered Message Delivery

The operation of ABCAST is depicted in figure 3-4. Since P_0 is the token holder, P_1 cannot deliver any ABCAST messages (including ABCAST messages from itself) until it gets the SO message from P_0. Then P_1 delivers both pending ABCAST messages in exactly the same order as P_0.

3.2 Broadcasts on a Hypercube

Shown in figure 3-5 is a high level graphical depiction of our model for message traffic on a hypercube configured multicomputer. It has three places and three transitions. The places are *P*, *CB*, and *L*. The *P* place is where tokens representing processors reside; the *CB* place is where tokens representing messages which are building a connection (connection building) reside; and *L* is where tokens representing available links reside. The transitions are *mg* (message generation), *lb* (link building), and *md* (message delivery).

The basic operation of the model works like this:

A Processor operates privately for an exponentially distributed time 1/λ and then decides to send a message to one or more other processors. The firing of the *mg* transition represents this "deciding to send a message", and introduces a token into the *CB* place. This token has a source and destination but, as yet, has not acquired any links.

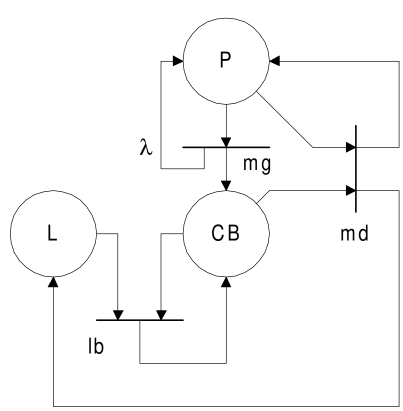

Figure 3-5: High Level HCube Broadcast Model

These message tokens in the *CB* place will reside there until they accumulate all of the links necessary to construct a continuous chain of links from the source to the destination. They accumulate links via the firing of the *lb* transition, which has as inputs both the *CB* place and the *L* place. In order for *lb* to be enabled, its predicate must be satisfied and this requires one message token from the *CB* place and a link token from the *L* place which starts at the current last node in the link chain AND goes in the right direction in the hypercube.

When the message tokens in the *CB* place have accumulated enough links such that the last node in the chain is the same as the destination processor of the message, the *md* transition will be enable to fire. This "delivers" the message and returns all of the links to the pool of free links represented by the *L* place.

3.3 Model Details

The model shown in figure 3-5 is a simplified depiction of the actual model used; it depicts the overall operation of the model but omits low level details. Each of the three transitions shown in figure 3-5 is actually implemented using a more complex subnetwork.

3.3.1 Message Generation

Message generation is actually done as depicted in figure 3-6.

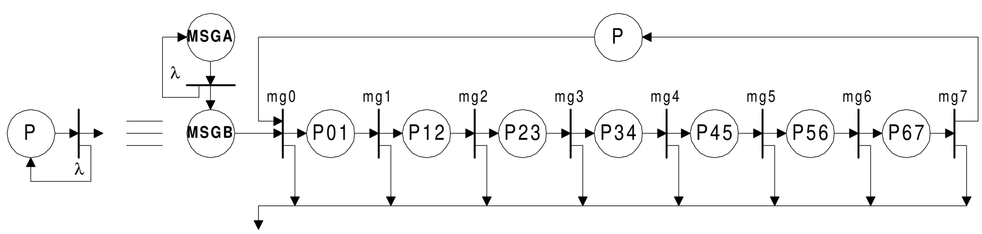

Figure 3-6: Message Generation Detail

While the model shown may look imposing, its operation is really quite simple. Conceptually we want to have broadcast messages generated from each processor at rate λ, but we also want to always have all processor tokens present in the *P* place and available for scheduling. To achieve this goal, we use the *MSGA* place which generates broadcast messages at the required rate and then places them in *MSGB*. All of the *mg*_x places are immediate transitions, so the moment that a token appears in *MGB*, transition *mg_0* is scheduled to fire (with zero delay). The firing of *mg_0* introduces a message token destined for processor 0 into the *CB* place and a message generation token into place *P01*. Subsequent *mg*_x transitions fire one after another but all at the same point in simulation time until finally the processor token is returned to the *P* place.

3.3.2 Link Building

The building of continuous connections from source to destination is also more involved than shown in the high level model. Figure 3-7 shows the actual implementation. Having message tokens simply wait in the CB place for the links they need can easily result in deadlock. This is avoided by having a mechanism whereby messages can give up the links that they have acquired and be delayed for a small amount of time. The delay insures that some other message has an opportunity to acquire the newly freed link(s).

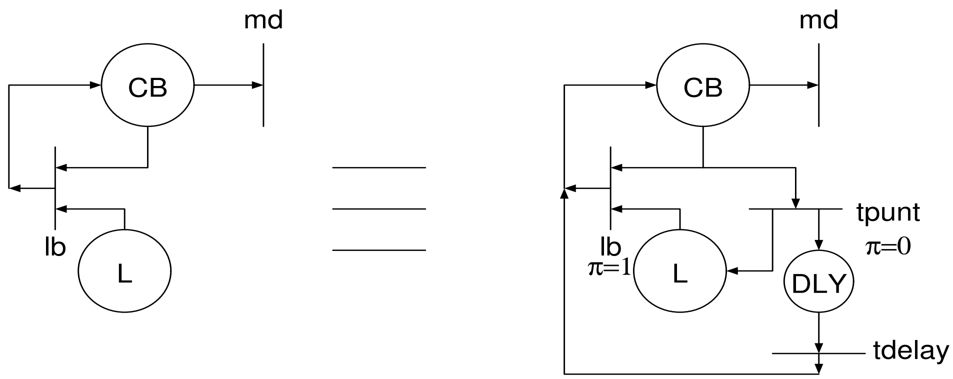

Figure 3-7: Link Building Detail

The *tpunt* transition implements this giving up of links; the lower priority (π=0) on *tpunt* than on *lb* (π=1) ensures that messages do not give up their links unnecessarily. The *tpunt* transition will only be enabled to fire and consume a message token from *CB* if there is no enabling for the *lb* transition for that message token. This only happens when the next link needed is not available. This rudimentary backtracking is not intended as an effective routing algorithm, merely a deadlock avoidance mechanism.

A closer look at the scheduling of the *lb* transition may clarify how this works. Suppose that there is a message *m* from proccesor 0 to processor 7 in the *CB* place and that links <0,1> and <1,3> have been acquired by *m*. as shown in figure 3-8. When the scheduler tries to schedule the *lb* transition (which it will do before *tpunt* since *lb* has a higher priority), the <3,7> link must be available in order to enable *lb*, otherwise we can't schedule a firing of *lb* with *m*'s token. In that case, when the scheduler gets to the *tpunt* transition, the token representing *m* is still unscheduled, and the predicate for *tpunt* places no precluding constraints on the enabling. So *tpunt* is scheduled to fire.

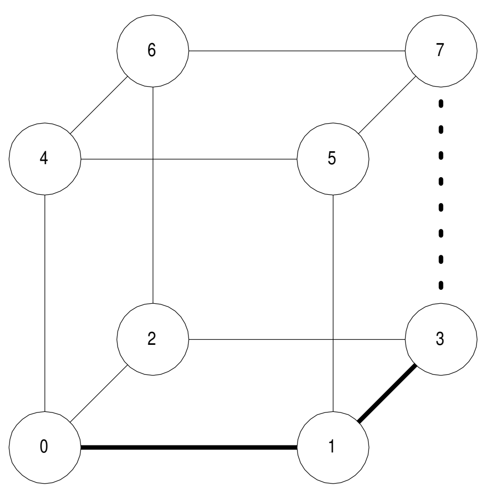

Figure 3-8: Link Building Example

3.3.3 Message Delivery

Message delivery is straightforward; once the complete chain of links from source to destination has been acquired, the *md* transition is enabled. It fires, consumes the message token and returns the links to the *L* place.

3.4 PSPN Model for CBCAST and ABCAST

Shown below, in figure 3-9, is a high level depiction of the PSPN model for CBCAST and ABCAST.

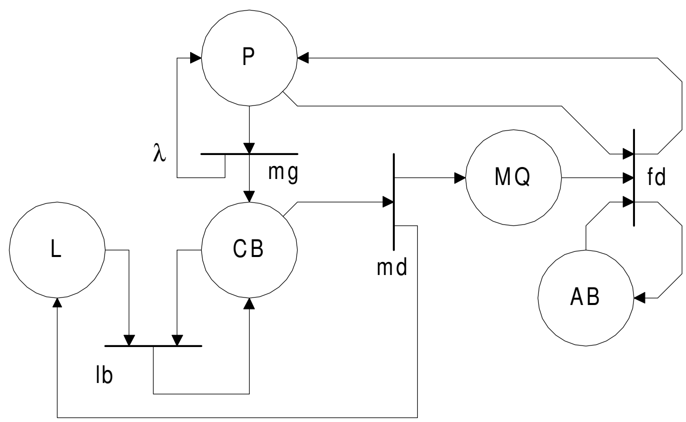

Figure 3-9: High Level View of ABCAST PSPN Network

The basic PSPN model for broadcast on a hypercube holds for both CBCAST and ABCAST type broadcasts and has the following properties.

1) Two new places (the *MQ* and *AB* places) have been added. They are inputs to the *fd* (final delivery) transition. The *MQ* places holds messages which have arrived at their destination but are not yet deliverable. When it is possible to deliver ABCAST messages, tokens will be inserted in the *AB* place by the action of the high level model. This place is only needed for ABCAST messages; for all messages with simpler protocols, it is unused (a dummy token is inserted that is included in all transitions).

2) The *fd* predicate is so greatly expanded that it became more convenient to implement it using the high level model as described in section 2.

3) As with the *fd* predicate, the firing rule became more convenient to implement in the coupled high level model.

4) While it doesn't actually change the structure of the model, it is important to note that each P token and all message tokens have, as one of their attributes, an array which holds their VT (Vector Time). We chose to use the underlying attribute mechanisms of PSPN to embed the timestamp tables needed to schedule message delivery in the individual processor tokens – thus the arc from the processor place to the *fd* transition.

3.4.1 *fd* Transition Predicate

As discussed above, the scheduler cycles through possible combinations of schedulable tokens for each transition. It calls the transition's predicate function to determine if the combination is schedulable so all a transition predicate need do is look at the combination of tokens and return TRUE if the combination is schedulable and FALSE otherwise. The final delivery predicate for CBCAST was complex enough to merit external code, for ABCAST, it is even more so. The basic logic of the *fd* predicate is to first determine that the basic delivery conditions, such as message destination PID is the same as the processor PID, are met. Assuming that the basic conditions are met, the predicate checks the type of message and applies the criteria appropriate for a given message type.

If the message uses none of the protocols, we call it an unprotocoled or regular message. No additional constraints are placed on its delivery. The *fd* transition is immediately scheduled to fire and the message is delivered. The *AB* place is dealt with in the same manner as for SO or CBCAST messages.

If the message is a CBCAST message or an ABCAST to the token holder, deliver by normal CBCAST rules. The rules check the *VT* stamp on the message and compare it to the *VT* of the destination processor and if the *VT* delivery criteria is met, the *fd* transition is immediately scheduled to fire. Like the SO message above, no constraints are placed on the selection of token from the *AB* place.

If the message is an SO message, we want to deliver it immediately (this differs from Birman's paper). The *fd* transition is scheduled to fire and no constraints are placed on the selection of a token from the *AB* place. Some token will be taken, but during firing of the *fd* transition, we specify that whatever *AB* token is taken, it is immediately returned to the *AB* place.

Otherwise, the message is an ABCAST message to a non tokenholder. For these messages, we apply normal CBCAST message delivery rules, plus we require that the token selected from the *AB* place match the message's <src,ssn> (unique message id) pair. The presence or absence of the appropriate delivery enabling token in the AB place is the mechanism for the implementation of ABCAST's total ordering.

It is important to note that since the AB place is an input place to the *fd* transition, a token from that place must be consumed in any firing of the *fd* transition. The AB place is instrumental in allowing a model of the ABCAST protocol to be constructed, but it is slightly in the way for non ABCAST messages. The *fd* transition's firing rule and predicate have to do some creative things to make everything work smoothly.

3.4.2 *fd* Transition Firing Rule

Like the predicate, the firing rule for the *fd* transition is implemented primarily in the high level model. Also like the predicate, the firing rule has four explicit sets of code, one for each type of message.

For all messages, the PSPN model takes care of extracting the appropriate tokens from input places, doing the basic attribute assignments, and inserting result tokens into the output places. Between the second and third steps there, the high level model is invoked. It is passed all the tokens in question so that it may implement the details of the protocol.

For regular messages and CBCAST messages, the firing of the *fd* transition is straightforward. The *MQ* token representing the message is consumed, the processor token from the *P* place is returned to the *P* place and the *AB* token is returned to the *AB* place. If the message is a CBCAST message, the VT of the processor token is updated to reflect the message's delivery. The *AB* place token is unmodified. Since the predicate for the *fd* transition placed no constraints on the selection of the *AB* token, it is quite possible that the *fd* transition was fired using an *AB* token that is necessary for some other message delivery. To cover those situations where no real *AB* tokens exist, the initial marking of the net includes a dummy *AB* token that has attributes that can never match any ABCAST message. This is done so that there will always be at least this one token in the *AB* place to enable delivery of non ABCAST messages.

For ABCAST messages to the token holder, delivery and token handling are identical to that for regular and CBCAST messages. However, each delivery of an ABCAST message to the token holder results in modifications to the global state data structure maintained in the high level model, which may precipitate many other actions. The global data structure is composed of 3 sets of lists, (S,T, and D). There is one S list for the entire network and there is one T list and one D list for each non token holder processor in the network.

ABCAST messages to the token holder can result in a fourth type of message called a Sets Order message (SO message). According to Birman et. al., SO messages are treated just like CBCAST messages. Unfortunately this can result in SO messages never getting delivered, so we found it necessary to treat SO messages as regular messages (i.e. neither CBCAST nor ABCAST) which are deliverable immediately upon arrival.

The S list (Set list) is the list of messages that have been delivered to the token holder, but have not yet been enumerated in an SO message. The token holder accumulates messages on the S list until the trigger level is reached. It then sends an SO message to all other processors enumerating the messages that are now deliverable. To model the SO message, the high level model creates message tokens and inserts them in the CB place.

The T list (Triggered list) represents messages that have been triggered. That is, an SO message has been sent enumerating the messages that are in the T list. It is possible that more than one SO message will be in transit from the token holder to a given destination at a time. The T list holds messages whose SO message is in transit.

The D list (Deliverable list) for each non token holder destination P_d, represents those messages which are deliverable at P_d. For a given message to be deliverable, an SO message enumerating it must have arrived at P_d. When that SO message arrives at P_d, the messages it enumerates are moved from the T list to the D list. If there are no items already on the D list (representing undelivered ABCAST messages), an token representing the first message is constructed and inserted into the AB place. This AB token will allow delivery of the first message enumerated in the SO message once it as arrived at its destination. When all that first message arrives at the destination and there the appropriate AB token is in the AB place, the message may be delivered. When that happens, the AB token is consumed and the high level model takes the next item on the D list and creates its AB token. Since AB tokens for later messages enumerated in the SO message are not inserted in the AB place until the first AB token is consumed, ABCAST total ordering is insured.

3.4.3 ABCAST Example

The entire chain of events leading up to delivery of ABCAST messages to non token holders is rather involved. In the following, we will assume that the trigger level is 2 and examine the sending and delivery of 2 ABCAST messages *m1* and *m2*. We make no assumptions about the origin of these two messages but will look at their eventual delivery at processor *P3* which we assume to not be the token holder.

The sequence of events is as follows:

Message *m1* arrives at the token holder. The token holder delivers *m1* to itself, but, since no other ABCAST messages have arrived, the token holder does not send an SO message. In the PSPN simulation, the high level model notes the event and places *m1* in the S list.

Message *m2* arrives at the token holder. The token holder delivers *m2* to itself and, since the trigger level has now been reached, the token holder sends an SO message to all other processors in the system. In the PSPN simulation, the high level model is invoked and it places *m2* on the end of the S list. It is then noted that the trigger level has been reached so all messages on the S list are copied to the T list of all other processors in the system. The S list is cleared and tokens that represent SO messages to all non token holder processors are constructed and placed in the CB place.

Eventually the SO message destined for *P3* will acquire the necessary links to arrive at *P3* and arrive there. Since it is an SO message, it is delivered immediately and messages *m1* and *m2* are now deliverable. In the PSPN simulation, the high level model copies 2 messages from the front of *P3*'s T list to the tail of its D list. If the D list was empty before the copy, an *AB* token is introduced into the *AB* place (if the D list was not empty, there is already an token for this processor in the *AB* place). This token has the appropriate attributes to allow the delivery of *m1*, i.e. it's attributes allow the satisfaction of the *fd* transition's predicate for message *m1* at processor *P3*.

If *m1* has not already arrived at *P3*, nothing happens. We have to wait for *m1* to arrive at *P3* before we can deliver it. In the pspn simulation, there is no token representing *m1* (destined for *P3*) in the *MQ* place so the *fd* transition's predicate cannot be satisfied.

Once *m1* arrives at *P3* (or if it is already there upon delivery of the SO message), it is immediately deliverable. Delivery of *m1* at *P3* enables delivery of *m2* at *P3*. In the PSPN simulation, the token in place *MQ* (which represents *m1* destined for *P3*) together with the AB token from step 3 satisfy the *fd* predicate. Transition *fd* is scheduled to fire, and, since, *fd* is an immediate transition, it fires immediately. The high level model is again invoked and it notes that an ABCAST message to a non token holder has been delivered, takes the next message off of the D list for that processor, and introduces a token into the AB place to enable delivery of that next message.

4. Simulation Results

We simulated the three different protocols (ABCAST, CBCAST, and no protocol) on a circuit switched eight node hypercube connected multiprocessor. For ABCAST we simulated the system at two different trigger values. As one can see in figure 4-1, ABCAST, at any trigger level, adds a significant amount of latency to the broadcasts. Also, at very low traffic levels, ABCAST trigger values greater than 1 cause great increases in latency. This is because the system is forced to wait a long time on average for the second (in the T=2 case) broadcast message before it can send the Sets Order message and commence final delivery. It is worth noting that the latency penalty for CBCAST is quite small. As one would expect, low trigger values saturate rapidly and higher trigger values are appropriate at higher traffic levels.

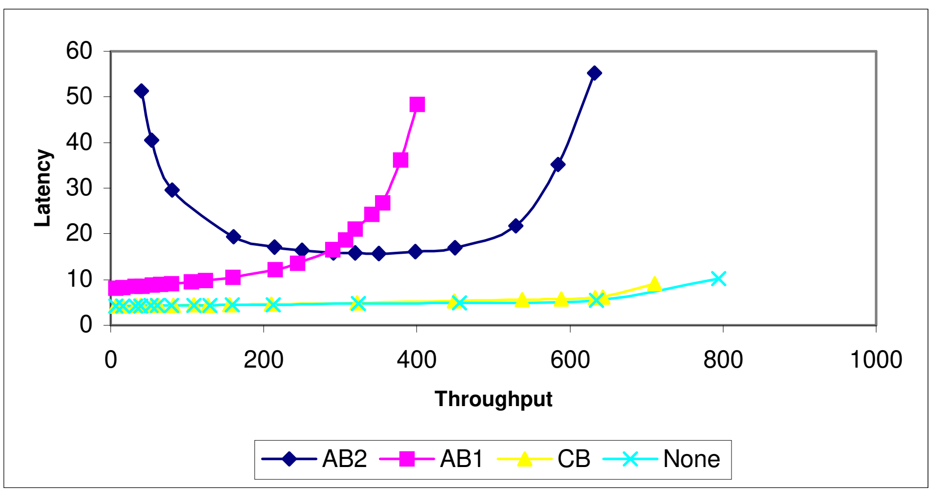

Figure 4-1: 3 Protocols (ABCAST, CBCAST, and None)

]Figure 4-2 shows latency versus throughput for CBCAST and for broadcasts with no ordering protocol. Here we can see that the magnitude of the penalty for CBCAST is quite small. It should be noted that the rudimentary backtracking algorithm we chose resulted in system deadlock at traffic levels higher than those on the graph so that the behavior of CBCAST relative to no protocol is not precisely known.

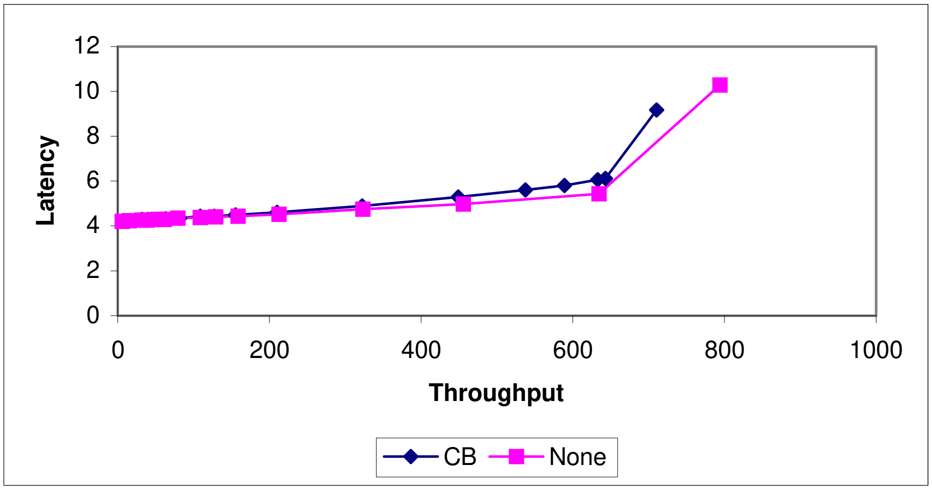

Figure 4-2: Closeup of CBCAST and No Protocol

5. Conclusion

We have believed for a long time that the stochastic petri net models could be extended to do behavioral modeling of complex fault-tolerant systems – both to concisely describe complex systems and to verify their correctness and completeness. But existing models seemed to us to be much too cumbersome to serve in this role. We have shown with the PSPN model by adding attributes (color) to the tokens and by extending the attributes with linked-lists, that complex intercommunication systems can be modeled with very compact and intuitive descriptions. Dynamic routing algorithms with backtracking can also be modeled both simply and concisely. The same model can be used for structures of any size by starting with different numbers of tokens and making simple changes to constants in the expressions describing transitions. Faults can be inserted by simply removing resource tokens. The basic PSPN model is equivalent to Markov models since SPNs are isomorphic with continuous time Markov Chains [Mar95].

In order to model more complex behaviors such as atomic broadcast, a second-level model was introduced where the second-level model can be triggered by specific transitions and can change the markings of the net. This model was demonstrated by modeling the CBCAST and ABCAST Protocols. This approach is different from most other work in that it abandons the direct equivalence with Markov models in order to be able to describe complex behavior with tractable and relatively simple descriptive models. We have shown that the PSPN system can be used to develop a simulation model of a complex system in a relatively short amount of time. Hopefully this will improve the power and accessibility of modeling tools.

REFERENCES

[Bir87] K. Birman and T. Joseph "Reliable Communication in the Presence of Failures" *ACM Transactions on Computer Systems*, 5(1), pp. 47-76

[Bir91] K. Birman, A. Schiper, and P. Stephenson "Lightweight Causal and Atomic Group Multicast" *ACM Transactions on Computer Systems*, 9(3), pp. 272-314.

[Cri 86] F. Cristian, H. Aghili, R. Strong, and D. Dolev, "Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement," *Dig. FTCS-12*, pp. 200-206.

[Est78] G. Estrin, "A Methodology for Design of Digital Systems – Supported by SARA at the Age of One", *Proceedings, National Computer Conference*, pp. 313-324, AFIPS, Los Angeles, California, 1978.

[Lin88] C. Lin and D Marinescu "Stochastic High Level Petri Nets and Applications", *IEEE Transactions on Computers*, 37(7), pp. 815-825.

[Lov93] M. Loving and D. Rennels, *Stochastic Petri Net Modeling of Fault Tolerant Multi Computers*, UCLA CSD-TR93-003.

[Mar95] M. Marsan et. al. (1995) *Modelling With Generalized Stochastic Petri Nets*, John Wiley & Sons, West Sussex, England.

[Pet81] J. Peterson (1981) *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, Englewood Cliffs, NJ.

[Zen85] A. Zenie "Colored Stochastic Petri Nets", *Proceedings of the International Workshop on Timed Petri Nets*, Torino, Italy, pp. 262-271, July 1985.