Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

STOCHASTIC PETRI NET MODELING OF FAULT TOLERANT MULTI-COMPUTERS

M. Loving
D. Rennels

March 1993
CSD-930008

Stochastic Petri Net Modeling of Fault Tolerant Multi-Computers *

Michael Loving
David Rennels
6291 Boelter Hall
Computer Science Department
University of California
Los Angeles, CA, 90024
Phone: (310)825-4033 FAX: (310)825-2273
Email: rennels@cs.ucla.edu

Abstract

Colored petri nets have been shown to create greatly simplified models that have many fewer places and transitions and can be applied to families of systems with minimal changes. This paper presents a modeling tool that uses a form of colored stochastic petri nets to model the performance of fault-tolerant multicomputer systems as resources fail. The model can be viewed as having tokens that collect to form compound tokens in a manner somewhat analogous to the formation of molecules from atoms. Therefore we call it the Polyvalent Stochastic Petri Net (PSPN) modeling system.

Using this approach simple and relatively similar models can be used for systems of varying sizes and configurations. Complex actions such as backtracking and routing around failed links in a communications system can be expressed in the transitions. Failed parts are modeled by simply removing a resource token from a place.

Models are presented of several multicomputer systems, and their performance is determined with and without various failed parts. Simulations are also shown that model the effect of locality of message destinations on performance. The paper concludes with a discussion of planned extensions of the modeling technique to include new applications and building analytic models as simplifications of the simulation models.

1 Introduction

In large multicomputer systems, the probability of faults and errors increases commensurate with their increased complexity. It is desirable to be able to model the performance of

*This research was sponsored by the Office of Naval Research under grant (N00014-91-J-1009).

such systems as their functionality degrades due to failed components or recovery actions. Stochastic Petri Nets have proven to be a useful tool for modeling many aspects of fault tolerant distributed computers but the models are sufficiently complex that they are often avoided by systems builders.

We have been exploring the use of colored Petri nets to create greatly simplified models, that are hoped to be more expressive and of greater use to designers. A modeling system has been developed and called the Polyvalent Stochastic Petri Net (PSPN) model. Models use tokens with list attributes of variable length and the effect is to have tokens combine to form compound tokens (analogous to molecules) when resources are acquired and separate when resources are released. Transitions implement generalized firing rules that vary with the current state of each compound token. This results in extremely compact models, and homogeneous systems of varying size have the same model. The size difference is reflected by initializing with different numbers of resource tokens. Similarly faults are implemented by removing resource tokens from the model.

Before describing PSPN, it is important to summarize the previous work that led up to this approach.

1.1 Petri Nets

Petri introduced the graphical models[6] which now bear his name. Petri Nets (PNs) are defined as follows[2]:

$$PN = (P, T, A, M)$$

where P is the set of places, T is the set of transitions, A is the set of arcs, and M is the initial marking.

Places are represented as circles; transitions are represented as bars and arcs link places

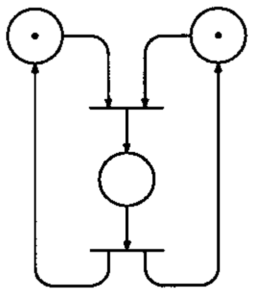


Figure 1: A simple Petri Net.

to transitions and transitions to places. Places may contain tokens which are represented as black dots inside the place in which they reside.

A place is said to be an input place to a given transition if there is an arc from that place to the transition. Similarly, a place is said to be an output place of a given transition if there is an arc from the transition to that place. If all input places to a transition contain at least one token, the transition is enabled. Enabled transitions are said to fire and, upon doing so, they remove a token from each input place and introduce a new token into each output place. With PNs, there is no timing; the order of transition firing can be known, but the amount of time between firings is not part of the model. Similarly, individual tokens have no identity, nothing distinguishes one token from another in the same place.

1.2 Stochastic Petri Nets

Stochastic Petri Nets (SPNs) are obtained from PNs by associating with each transition in a PN an exponentially distributed firing rate. This gives a slightly larger formal definition.

$$SPN = (P, T, A, M, \lambda)$$

where P, T, A, and M are as above and λ is the set of firing rates. It has been shown[1] that SPNs are isomorphic with continuous time Markov Chains where each Petri Net marking is equivalent to a state in the equivalent Markov Chain. Petri Nets are, however, generally a much more compact and intuitive model.

Molloy has also shown that, using the method of stages [4], it is possible to model transitions with abitrarily distributed firing rates provided that the distribution has a rational LaPlace transform. In principle, analytic solution is possible for all such systems. Dugan et. al. [10] allow arbitrary distributions for firing rates and, where analytic solution is not feasible, suggest solution by simulation. Sanders and others have investigated stochastic activity networks for performability modeling[7].

1.3 GSPNs and SHLPNs

Marsan, Conte, and Balbo [2] present extensions of SPNs which they call Generalized Stochastic Petri Nets (GSPNs). They allow nets to have both timed and immediate transitions, thus the formal definition is identical to that for SPNs with the exception that the set λ may have fewer elements than T. They specify that if more than one immediate transition is enabled, a probability distribution is used to select which transition among those enabled will fire. They also present inhibitor arcs, which can often achieve the same result except

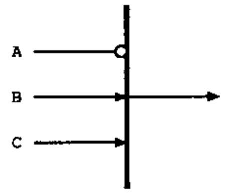


Figure 2: A transition with inhibitor arc.

that they require that there be *no* token in the inhibited input place.

Lin [11] presents Stochastic High Level Petri Nets where he investigates the simplifications that can be obtained by combining into one transition(place) the transitions(places) which have the same or similar function but differ in name. He shows that the philosophers problem can be compactly represented (see figure 3) with a much smaller net where where all E places have been combined into one E place and similarly for the transitions and other places. The tokens now have identities and the transitions have predicates which must be satisfied in order for the transition to be enabled. Also presented is the idea of using compound markings to reduce the size of the equivalent Markov chain.

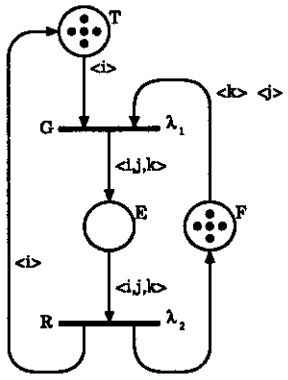


Figure 3: Philosophers High Level Petri Net.

1.4 Colored Petri Nets

Colored Petri Nets[3] give structure to tokens and greatly enhances the decision making power of transitions. Jensen defines CPNs as

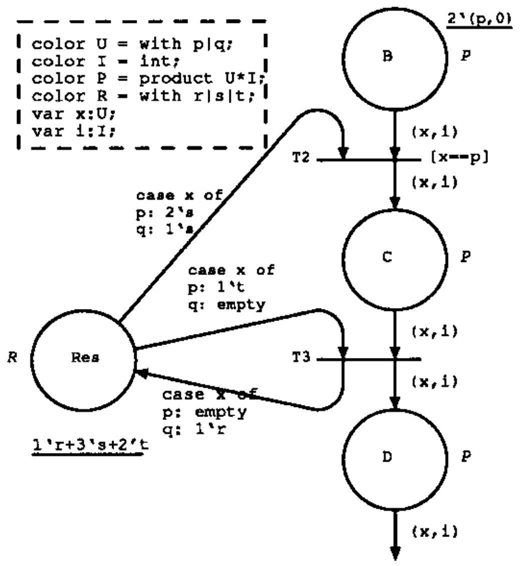
$$CPN = (P, T, A, IN, N, \Sigma, C, G, E)$$

While this initially seems imposing, there are really only three new pieces to this system as compared to PNs. P, T, and A, are as above, and IN in CPN's is the same thing as M above (i.e. the initial state of the system). This leaves us with

1. N is the node function.
2. Σ is a finite set of types, called color sets.
3. C is the color function.
4. G is a set of transition guard functions.
5. E is a set of arc expression functions.

The node function N exists as a tacit assumption in the previous models. It is simply the specification of the source place(transition) and destination transition(place) for each arc. Σ is one of the new features. It is a set of data types (like Pascal records) each of which must be both finite and non-empty. As previously mentioned, tokens in CPNs have structure to them; each token has one of the structures enumerated in Σ. Similarly, each place can hold only one type of token, hence we have C which specifies which type structure is associated with each place. Another new feature is the transition guard functions, G, these are boolean predicates which must be satisfied in order for the transition to fire. Guard functions appear in square brackets next to the transition. Guard functions which always evaluate to true are omitted. Finally, there are the arc expressions, E. These expressions evaluate to a value of the same type as the place from which they emanate and define, with the guard functions, how many and which tokens are needed from each input place in order for a transition to fire.

Figure 4 is an example of CPN's. Upon initial inspection, one finds much new information. Places, transitions and arcs take the usual form. The node function, N, is defined pictorially in the usual way. In the upper left corner of the picture is the definition of the color sets



```
color U = with p|q;
color I = int;
color P = product U*I;
color R = with r|s|t;
var x:U;
var i:I;
```

Figure 4: A portion of a CPN.

(Σ). It is quite simply a list of declared types with type U being an enumerated type whose allowed values are p or q. Type R is similar, but with a different value set. Type I is the integers and type P is a record with two fields, one field is of type I, the other of type U. Two variables, x and i, are also declared.

Next to each place, in italics, is a declaration of its type; this is C, the color function, which defines the type of the tokens which occupy each place. The initial marking of the net is denoted by the underlined inscriptions next to the places. In this case, place Res initially has six tokens (one with value r, three with value s, and two with value t) and place B initially has 2 tokens (both with the value p for the first field and zero for the second). There is one transition guard function in the example and several arc expressions.

T2 illustrates the use of both arc expressions and guard functions. It is said that T2 has two variables; this is because when we examine the guard function of T2 and the arc expressions of all input arcs, we find a total of two distinct variables. These are x and i. The guard function for T2 specifies that x must have the value p in order for the transition to be enabled. Examining the arc expression for the arc from place B tells us that x gets its value from the first field of the token from place B and i gets its value from the second field of that token. The arc expression from place Res tells us that if the value of x is p, then we need two tokens of value s from place Res and that if the value of x is q, then we need only one token of value s from Res. Thus in order for T2 to be enabled, there must be a token in B whose first value is p and there must be at least 2 tokens with value s in place Res.

2 PSPN Modeling Techniques

Our modeling system can be viewed as an amalgamation of Colored Petri Nets (CPNs) [3] and Generalized Stochastic Petri Nets (GSPNs) [2] with some extensions. A formal definition of our system is thus the following.

$$PN = (P, T, I, O, H, M, \lambda, R, p)$$

1. P is the set of places.

2. T is the set of transitions.

3. I is the set of input arcs.

4. O is the set of output arcs.

5. H is the set of inhibitor arcs.

6. M is the initial marking.

7. λ is the set of transition timing distributions.

8. R is the set of firing rules.

9. p is the set of transition predicates.

All Petri Nets also include tokens which are not explicitly mentioned in the above list since they fall more properly under the heading of initial marking. A marking (initial or otherwise) of a Petri net is a distribution of tokens among the places. Tokens reside only in places.

Tokens: Tokens in our PSPN Petri nets as in CPNs have attributes or data values which, like a record in Pascal, may be of whatever structure one needs. All tokens in a given place have the same attribute structure (color in Jensen's work), and an attribute may be a list of varying length. The value of various attributes distinguishes one token in a given place from another. The implementation of list attributes is probably the most unique features of our modeling technique. As mentioned above, tokens have whatever attribute structure one needs, and we allow fields in the attribute structure to be lists of more primitive types.

For example, a token representing a request for service may reach a transition that also has an input from a place with resource tokens. The transition function will select a resource token, add its identity to the list in the requesting token, and fire by passing along the modified compound token. This can be done in a looping fashion so that the same transition is used repeatedly to collect new resources such as communication links, with the transition function examining what new link is needed on the basis of those already collected. At another transition, the list will be modified, and the resource token returned to its home place.

This feature has allowed us to keep our graphical models quite small, but still very expressive. It also allows the use of one graphical model for systems of varying size. List attributes are alluded to in [3], but no extensive use is made of them.

Transitions: Transitions define the manner and timing of token motion about the net. Stochastic Petri Nets associate a set of rates with the transitions such that each transition has a firing rate (usually exponentially distributed) unless it is an immediate transition. In our model, all transitions have a timing distribution. This distribution may be exponential, deterministic, or any other distribution that is codeable in C. Immediate transitions are thus just a special case of deterministic timing distributions; i.e. they are transitions with deterministic distributions and the time to fire is zero.

Predicates: Each transition has associated with it a predicate (in special cases there are more, as is discussed below) which specifies a relationship between the various attributes of the input tokens. This predicate must be satisfied (evaluate to "true") in order for the transition to be enabled. At the moment the transition becomes enabled, it is scheduled to fire at a time in the future that is determined by the transition's timing distribution. These predicates work in the same manner as guard expressions in CPNs.

Firing Rules: In addition to predicated transitions, we have what we call a firing rule. This is a specification (one for each transition) that tells what data values are taken from the various input tokens and where they assigned in the output token(s). This concept is roughly equivalent to the arc expressions in CPNs.

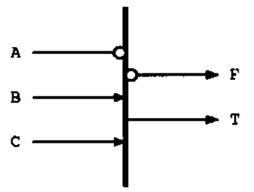


Figure 5: Complementary Output Transition (COT)

Arcs: Arcs are usually specified as being either input arcs or output arcs depending on their relationship to a transition. Input arcs define on which places a transition depends (input places). Output arcs define to which places transitions introduce tokens upon firing (output places).

We use a generalization of inhibitor arcs to expand the modeling power of our nets without greatly increasing their size. A typical transition with inhibitor arc is shown in figure 2; it functions as follows: if there is a token in places B and C and no token in place A, the transition may fire. When it fires, it consumes a token from places B and C and places a token in the output place or places.

In order to incorporate the general idea of inhibitor arcs into our framework of predicated transitions, it was necessary to modify and extend the idea somewhat. The result is the Complementary Output Transition (COT) shown in figure 5. A COT may fire in two different ways. It may fire along its "true" path and consume a token from all input places, or it may fire along its "false" path and consume tokens from only the uninhibited input places. A COT has two predicates associated with it. The first predicate references only the uninhibited input places (B and C in this example). The second predicate may make reference to any or all input places. In order for the T branch to be taken, both predicates must be satisfied, if only the first predicate can be satisfied, the F branch is taken. Of course if it is not possible to satisfy even the first predicate, the transition may not fire. Additionally, a COT has two timing distributions, one for each firing branch.

The COT extension brings together useful features from Colored Petri Nets [3] and Generalized Stochastic Petri Nets [2].

3 PSPN Models

In order to illustrate the use of the PSPN tool, a 4 place model is presented in figure 6 below. The same net is used for modeling Toroid and N-cube architectures of varying sizes.

3.1 Modeling Assumptions

The systems we have modeled have many common elements. Since our primary goal is to measure average message latency, a number of simplifying assumptions have been made.

All processors act identically. Each processor spends a random amount of time executing locally and then needs to send a message. Unless otherwise specified, the destination chosen at random, uniformly distributed over all other processors in the system. The processor is assumed to be busy until the message is delivered and upon delivery it resumes executing locally.

In order to deliver a message, it is assumed that a continuous chain of links from node to node through the system must be acquired and held until the message is delivered (circuit switched system).

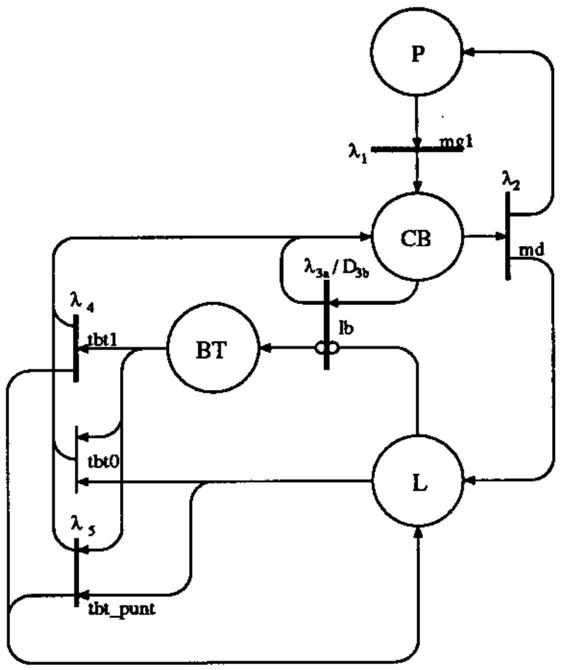
Neither the destination node nor any of the intermediate processor nodes need to participate in building the links; it is assumed that each node has a "link controller" which is capable of accepting link requests and granting link usage without interrupting the main processor of the node. Upon delivery of a message, all links in the chain are released.

3.2 Model Overview

The Petri Net model used is the same for all systems modeled. It consists of four places and six transitions. The tables in figure 6 list the places and transitions and describes them.

Each of the places corresponds to a physical aspect of the system being modeled. Tokens in the P place represent processors executing privately; tokens in the CB place represent messages in the process of building a link chain; tokens in the L place represent available links, and tokens in the BT place represent messages who, for various reasons, were unable to obtain the next link in the chain are attempting some sort of backtracking. The types (or colors) alluded to previously are shown in the token structure portion of the table. For instance tokens in place CB have three attributes, *msrc, mdst,* and *lchain.* The first two attributes represent the processor number of the source and destination processors and the last is a list which contains all of the intermediate nodes in the link chain. Tokens in the L place represent links and the connection between a pair of nodes i and j is represented by two links, one from i to j, the other from j to i.

Each of the transitions, together with their associated predicate(s), represents a change in the physical systems being modeled. While the PSPN graph we are using is the same for all systems, the predicates sometimes vary to reflect the differing requirements of the systems being modeled.



Place	Purpose	Token Structure
CB	connection building	<mssrc,mdst,lchain>
P	private processing	<pid>
L	idle links	<src,dst>
BT	backtracking	<mssrc,mdst,lchain>

Trans	Purpose
mg1	message generation
md	message delivery
lb	lchain building
tbt0	backtracking - chain empty
tbt1	backtracking - chain non empty
tbt_punt	backtracking - give up and start over

Figure 6: Net Model

Processors deciding they need to send a message are represented by **mg1**; since there are no needed conditions on this transition, its predicate is always true. The output token represents a message starting through the system, and it is deposited in the Chain Building (CB) place where it (i) gathers links by rules determined in transition **lb**, and (ii) returns to place CB. The transition **md** represents succesful delivery of those messages; tokens in the CB place that have acquired all links needed to make a continuous chain from the source to the destination will satisfy the **md** predicate and enable this transition.

Transition **lb** represents the attempted acquisition of links and is implemented with a COT. If all of the **lb** transition's predicates are satisfied (a forward link is available), **lb** attaches the new link to the token and returns it to the CB place. If no forward links are available, the token is sent (via the F branch of the **lb** transition) to the BT place where the three tbt transitions are used to implement backtracking.

As explained previously, transitions of this type have two predicates; tokens from the uninhibited input places must satisfy the first predicate; tokens from all input places must satisfy the second predicate in order to fire the true branch. If no combination of tokens from the CB and L places can be found to satisfy both predicates, the false branch is taken (assuming that a token from the CB place satisfies the first predicate). The first predicate on **lb** checks that the link chain is still incomplete. The second predicate checks three things. First, the link must originate at the current last link of the chain; second, the link must go in the correct direction. The third criteria relates to backtracking and is best discussed with the various tbt transitions.

The three tbt transitions implement a rudimentary backtracking algorithm. When no links are in the link chain, the **tbt0** predicate will be satisfied. In this case we simply take any available link that originates at the source node – going off in a random direction. When at least one link is in the link chain, the **tbt1** predicate will be satisfied. Here, we simply give up the last link in the chain and send the token back to CB to try again. The **tbt_punt** transition, as the name implies, takes a rather drastic action; it gives up all links in the chain, obtains a link that goes in a random direction, then sends the token back to CB to continue building its link chain. This only occurs when a token is making its second try at backtracking without making any significant progress in building a link chain.

Since both **tbt0** and **tbt_punt** cause links to be selected which possibly go in the "wrong" direction, it is necessary for the **lb** transition to be constrained to not back up over the last link (even though that might be a step in the "right" direction).

It is customary to represent immediate transitions as thin lines and timed transitions as thicker lines or as boxes, we have adhered to that custom in the figure. Of the five timed transitions, all but **lb** have only a single timing distribution associated with them. We chose, as implied by the λs associated with those transitions, to use exponential distributions for them. The bifurcated **lb** transition has one distribution for its true firing branch and a second for its false firing branch. For the true branch we again chose an exponential distribution and for the false branch a deterministic distribution to demonstrate its availability.

3.3 A Detailed Example

The operation of the model is best understood by example. The 8 node N-cube system is the simplest of the three we studied and demonstrates all of the features in our system. We

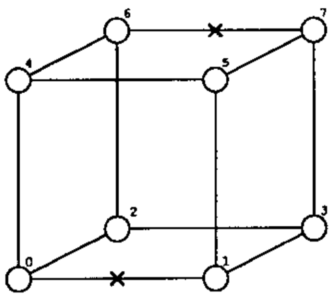


Figure 7: 8 node N-cube with broken links

will follow processor 1 as it sends a message to processor 6. Additionally we will assume that the links (0,1) and (6,7) are broken. The resulting cube is shown in figure 7. This is implemented by simply removing the tokens representing those links from the L place prior to simulation.

We initialize our systems with all processors in the P place and all available links in the L place. The simulation then starts and all enabled transitions are scheduled. For our 8 node N-cube system, this results in 8 instances of **mgl** one being scheduled, one for each processor in the P place. At some time in the future, determined by its timing distribution, **mgl** will fire for processor 1. The token representing p_1 is removed from the P place and a destination address for the message is selected which is uniformly distributed on [0,7]. As mentioned above, we are assuming that the destination is node 6. Tokens in CB also have a chain of links, which may be empty, thus the new token, $p_1m_{16}lc_1$ is inserted in the CB place. The last item in the name of the token, lc, is the link chain which begins at the source node and extends through all the links specified in the chain. For now, the last node in the chain is 1.

	Predicates
mgl	1
md	last(cb.lchain) == cb.mdst
lb0	last(cb.lchain) != cb.mdst
lb1	last(cb.lchain) == l.src && direction of l is correct && l does not reverse the last step made
tbt0	lchain is empty && m.src == l.src
tbt1	lchain not empty && btcntr == 0
tbt_punt	lchain not empty && btcntr > 0 && m.src == l.src

Since the net has changed state, it is necessary to schedule all newly enabled transitions. The only possible ones are **md** and **lb** since these are the only transitions whose input places have had new tokens added. Checking the predicate for **md**, we find that its predicate, which specifies that the message destination must be equal to the last node in the link chain; this is not the case so **md** is not enabled. We then check the predicates for **lb** and find that the first predicate satisfied by the $p_1m_{16}lc_1$ token and so we check links available in the link place trying to find one which, together with $p_1m_{16}lc_1$, satisfies the second **lb** predicate. Since the (1,0) link is broken, it is unavailable, but either of the two links (1,5) and (1,3) will satisfy the predicate. Our model specifies no preference and our simulator simply selects the first token which satisfies the predicate. We assume that link (1,5) is selected; the transition is scheduled to fire, and some time later will fire. The token $p_1m_{16}lc_1$ is removed from CB, l_{15} is removed from L, and a new token $p_1m_{16}lc_{15}$ is placed into CB where the process starts over.

If the $p_1m_{16}lc_{15}$ token selects link (5,4) next it will proceed without incident on to node 6. Let us assume that link (5,7) is selected instead. We then have the situation of $p_1m_{16}lc_{157}$ in CB which still does not satisfy the **md** predicate, but does satisfy the first **lb** predicate. Unfortunately, link (7,6) is unavailable and so there is no link which, when taken with $p_1m_{16}lc_{157}$, will satisfy the second **lb** predicate. It does not matter that the link is broken; the situation would be the same if the link was merely in use by another processor trying to send its own message. The complementary output of **lb** is now scheduled since it is enabled and the primary is not.

This removes the $p_1m_{16}lc_{157}$ token from CB and places it in BT where we find that neither the **tbt0** predicate nor the **tbt_punt** predicate is satisfied. Thus, **tbt1** is scheduled and fires (immediately). This gives up the most recent link, returning it to the L place and the reduced token $p_1m_{16}lc_{15}$ is returned to CB. Since, in the case where more than one token in a place would satisfy a transitions' predicate(s), the token is selected at random, the $p_1m_{16}lc_{15}$ will eventually select the (5,4) link and continue to deliver the message.

If the (4,5) link were also broken, it would be possible for the $p_1m_{16}lc_{15}$ to run through a cycle obtaining and giving up the (5,7) link. We prevent this by having and additional backtracking transition **tbt_punt**, which as the name implies takes drastic action to try to get the message delivered. A counter is maintained (not shown in the figure) which counts passages through **lb**. This counter is set to 2 by **tbt1**; it is decremented (with 0 as a floor) at each passage through **lb**. If a token with a backtrack counter value greater than 0 arrives in BT, it will satisfy only the **tbt_punt** predicate. **tbt_punt** returns all current links in the link chain to the L place and selects a new link which goes in any direction.

4 Simulation Software

Our simulator is more precisely a simulator generator. Descriptions files (shown in the appendix) are processed by the generator which outputs a C program to simulate the described PSPN. The description files consist of 2 major sections: declarations and definitions. In the declarations portion, all places and transitions are declared. The definition portion is where the real definition of the model occurs.

The place definitions define which places are inputs and outputs to which transitions. This section could actually be dispensed with since the same information is included in the transition definitions.

The transition definitions reiterate the input and output places, define the timing distribution(s), the predicate(s) for each transition, and the firing rules. The reader of the appendix will notice that the variable names used in the paper (l.src for example) are higher level language constructs than are used in the actual description files (l[0] for example). The predicates can be any arbitrarily complex C boolean expression. The body of each transition definition, which defines the firing rule, allows some high level language constructs too though it is not yet as extensive as that allowed in the predicates.

The final section of a description file defines the initial marking of the net. For each place, all tokens that are to exist in that place at the start of the simulation are declared.

The flow of the software is relatively straightforward. After each transition firing (and at the start of simulation), all possibly live transitions are checked to see if they may be scheduled, if so, the tokens which will participate are marked (so that they won't be scheduled for something else), the timing distribution is evaluated and a firing event for each scheduled transition is place in an event queue. Next, the event on the front of the queue is executed. The firing rule for the appropriate transition determines how tokens are moved around in the net, transitions are marked as being potentially live based upon changes in their input places and the process repeats itself.

5 Results

We model two different systems using PSPNs. The first is the binary N-cube, the second is the toroidal mesh. Two different sized N-cubes are modeled, the first with 8 nodes, the second with 64; for the toroid, only size 64 is investigated. In each case we measure the average time to deliver a message with varying numbers of broken links. While our models actually use two distinct links (one in each direction) for each internode link, when the word "link" is used in this section, we mean both the ij and ji links.

The mean times for all transitions except **mg1** were fixed. The exponentially distributed transitions all had a mean time of 1 (with no units); the deterministic transitions were 0 in the case of **tbt1** and 0.5 in the case of the F branch of **lb**. The mean time for the **mg1** transition was varied in order to vary the the utilization factor ρ, and was a parameter used to generate a family of curves in the other graphs. Except where noted, destination nodes are chosen at random, uniformly distributed over all nodes in the system other than the originating node.

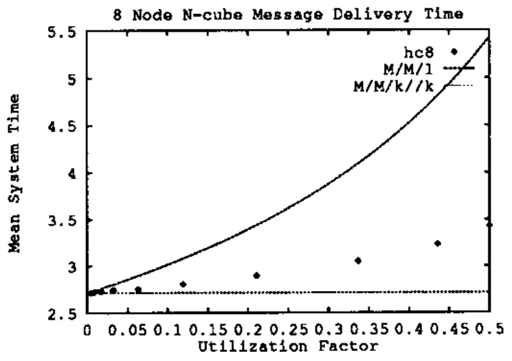


Figure 8: Service Time in 8 node N-cube as a function of ρ.

As a base line test of our system, we wanted to use it to investigate how average message delivery time varied as a function of the system utilization factor ρ. In typical queueing systems, ρ is usually just λ/μ using the notation of [8]. However our system is structured such that the average interarrival time of customers depends on both their average system time T and their average generation time $1/\lambda_{mg1}$. Thus we have

$$\rho = \frac{1/\mu}{1/\lambda} = \frac{\overline{x}}{\overline{t}} = \frac{\overline{x}}{T + 1/\lambda_{mg1}}$$

Where $\overline{x}$ is the average service time. We take this to be the average time for a message to be delivered when there is no competition for the systems resources. Thus $\overline{x} = 1/\lambda_{md} + 12/7\lambda_{lb}$, and, since all of the λs except λ_{mg1} are equal to 1, $\overline{x} = 2.71$. The graph in figure 8 shows how the service time in the system we modeled compared to both the M/M/1 and M/M/k//k queueing systems. One would expect, that with very long interarrival times, the Ncube 8 system would act very much like the *k*-server *k*-customer queueing system and that it would always be better than the single server system and that is just what we found.

For the 8 node binary cube, one cannot break many links before the structure ceases to be connected. If more than 5 links are broken, the structure becomes disconnected, and it is possible for the structure to become disconnected with only 3 broken links. One *can* break 5 links and still have a connected structure, but it has degenerated into a one dimensional array. We chose to break links lying in the (0,1) direction and simulate the system with up to 3 links broken. The links in question are (0,1), (7,6), and (2,3). Among these three links, the choice of the second link to break makes a significant difference in the system's performance. The (0,1) and (2,3) links are on the same face and breaking these two gives much lower performance than if (0,1) and (7,6) are selected.

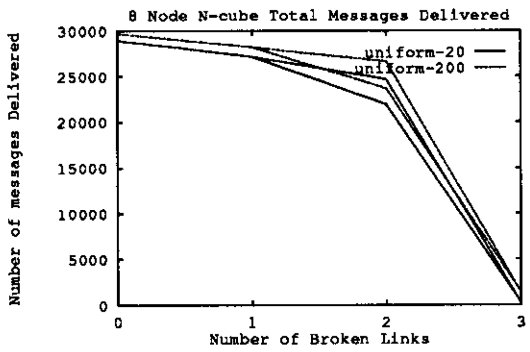


Figure 9: Total Messages Delivered as a function of Number of Broken Links

Figure 9 shows the total messages delivered with varying numbers of broken links for $\bar{t}_{mgl} = 200$ (relatively sparse message traffic) and $\bar{t}_{mgl} = 20$. We found that the simple backtracking algorithm which we chose to model was totally inadequate for the 8 node cube with 3 links broken. When 3 links were broken, competition for the one available link between the two halves of the cube reduced message throughput to nearly 0 Performance of the system varied with 2 links broken depending on the configuration selected; with two links broken on the same face performance was degraded in terms of both throughput and delay. The graphs have divergences in them where multiple configurations with the same number of broken links was investigated.

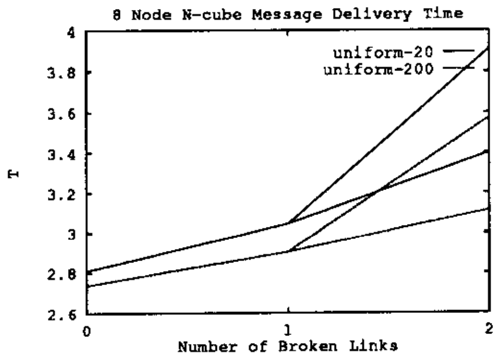


Figure 10: Service Time in 8 Node N-cube

What can be seen in figure 10 is that message delivery times are always lower for the system with large interarrival times for messages. The 3 broken link case is not included in the figure since message traffic was so low that the results were erratic. A better backtracking algorithm (or different selection of which links are broken) is needed to study this case. As previously mentioned, the data for 2 broken links varies depending on the relative location of the links. Figure 10 reflects this and we see that with two parallel links on the same face broken, average message latency increases dramatically compared to breaking two parallel links that are opposite each other across the body diagonal.

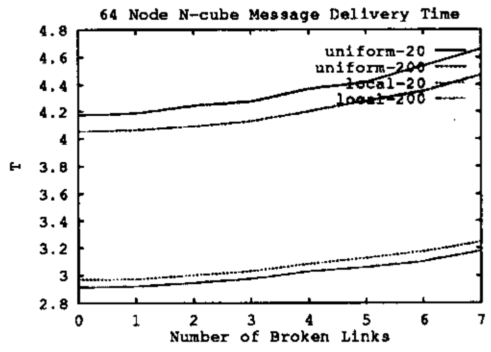


Figure 11: Service Time in 64 Node N-cube

For the 64 node N-cube we also looked at message delivery times with varying numbers of broken links. The links we chose to break were the (0,1), (16,17), (32,33), (48,49), (8,9), (24,25), (40,41), and (56,57). If one divides the 64-cube up into 8 8-cubes, these are the (0,1) links of all of the subcubes. Again we measure delay with $\bar{t}_{mg1}$ set to both 20 and 200. We also varied the selection of the destination node for messages. In the top set of set of curves, destination nodes are selected at random, uniformly distributed over all other nodes in the system. In the bottom set of curves, the destination is selected with a geometric distribution such that nodes a distance 2 away are only 1/2 as likely as nodes of distance 1 and similarly, with decreasing probability out to distance 6. In the upper curves, the mean distance is 3.04 (we exclude the originating processor as a destination); in the lower curves, it is 1.90. It might be expected that a system with greater locality of message traffic would experience less impact from link breakage, and, in absolute terms, this is true. As a percentage of the average message delay, the difference, while still there, is very small. With 8 links broken, there is a 9% degradation in service time for the system with locality and compared to 11% degradation with uniformly distributed destinations.

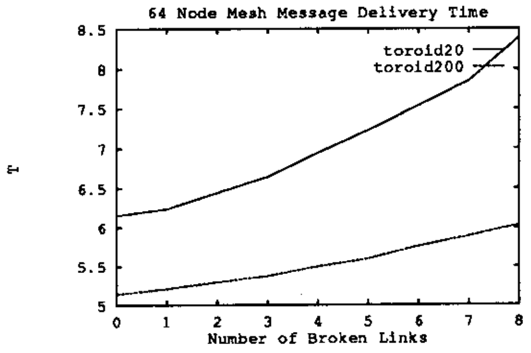


Figure 12: Service Time in 64 Node Toroid

For the toroid, we looked at service time as a function of the number of broken links. In this case the destination nodes are again uniformly distributed. This gives a mean distance of 4.06 and hence $\overline{x} = 5.06$ at zero load (**md** has mean time of 1 also). Here we see that the toroid, at light loads, suffers somewhat more (17%) degradation in delivery time with 8 links broken, but with the heavier traffic ($\bar{t}_{mgl} = 20$), the delivery time is seriously degraded with a 36% increase.

6 Conclusion

The PSPN modeling techniques presented in this paper represent a concise and powerful tool for the analysis of computer systems. PSPN models of several computer systems have been developed to demonstrate their utility in the area of fault tolerance modeling. PSPNs are a natural extension of CPNs and GSPNs and, with appropriate timing distributions, should be amenable to analytic solution of the corresponding Markov Chain; this need to be investigated.

Other areas for future work include improvements in the capabilities of our software system and modeling of other interesting systems such as distributed shared memory multicomputers.

References

[1] M. K. Molloy On the integration of delay and throughput measures in distributed processing models. Ph.D. dissertation, University of California, Los Angeles (1981).

[2] M. A. Marsan, G. Conte, and G. Balbo "A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems" vol. 2, no. 2, pp. 93-122.

[3] K. Jensen "Coloured Petri Nets: A High Level Language for System Design and Analysis" *High Level Petri Nets: Theory and Application.* Springer-Verlag; Berlin, Germany; 1991.

[4] D. R. Cox "The Analysis of Non-Markovian Stochastic Processes by the Inclusion of Supplementary Variables." *Proc. Camb. Phil. Soc.*, 51, pp. 433-441, 1955

[5] J. A. Carrasco "Automated Construction of Compound Markov Chains from Generalized Stochastic High-level Petri Nets" *Proc. 3rd International Workshop on Petri Nets and Performance Models*, IEEE Computer Society Press, Kyoto, 1989, pp. 93-102.

[6] C. A. Petri *Communications with Automata*, Ph.D. dissertation, Tech. Rep. RADC-TR-65-377, Rome Air Development Center, Rome, NY, 1966.

[7] W. Sanders *Construction and Solution of Performability Models Based on Stochastic Activity Networks*, Ph.D. dissertation; Tech. Rep. CRL-TR-9-88, University of Michigan, Computing Research Laboratory; Ann Arbor, Michigan; 1988.

[8] L. Kleinrock *Queueing Systems, Volume I: Theory*, John Wiley & Sons, New York, 1975.

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

[10] J. B. Dugan, K. S. Trivedi, R. M. Geist, and V. F. Nicola "Extended Stochastic Petri Nets: Applications and Analysis" *Performance 84*, North-Holland; Amsterdam; 1985.

[11] C. Lin and D. C. Marinescu "Stochastic High-Level Petri Nets and Applications" *IEEE Transactions on Computers*, vol. 37, no. 7, pp. 815-825.

```
/* PPN desc file for 8 node N-cube */
places: p,cb,plr,bt,cb_p,l;
trans: mg0,mg1,lb\,md,tlr,tlr_last,bogus_msg,tbt0,tbt1,tbt_punt;

place p(md,bogus_msg;mg0);
place cb_p(mg0;mg1,bogus_msg);
place bt(lb;tbt0,tbt1,tbt_punt);
place l(tlr,tlr_last;lb,tbt0,tbt_punt);
place cb(mg1,lb,tbt0,tbt1,tbt_punt;md,lb);
place plr(tlr,md,tbt_punt;tlr,tlr_last);

trans lb(cb,l\;cb;bt):exp(1.0):determ(0.5);
{
/*  predicate checks three things
  1: that the link starts at the current last node
  2: that the link goes in the right direction  */
/*3: that the link does not reverse the last link */
  predicate:{ (cb[1] != cb[(3)]) };
  predicate:{ (cb[(3)] == l[0]) &&
              ( ( (l[0]^l[1]) & (cb[(3)]^cb[1]) ) != 0) &&
              ( cb[3] == 4 ? 1 : (cb[(3)-1] != l[1]))
            };
  success{
    for(i=0;i<=cb[3];++i)
      cb[i] = cb[i];

    cb[3] = cb[3] + 1;
    cb[(3)] = l[1];
    cb[2] = (0 == cb[2]) ? 0 : (cb[2] - 1);  /* decrement tbt cntr */
  };
  failure{
    for(i=0;i<=cb[3];++i)
      bt[i] = cb[i];
  };
}

/*  tbt0 is the transition which deals with m,p tokens which have not gone
    anywhere yet and could not find a link in the right direction

    for these, we simply grab a link in any direction and send them
    back to place cb */
trans tbt0(bt,l;cb):exp(1.0);
{
  predicate:{ (bt[3] == 4) &&
              (l[0] == bt[(3)])
            };
    for(i=0;i<=bt[3];++i)
      cb[i] = bt[i];

    cb[3] = bt[3] + 1;
    cb[(3)] = l[1];
}

/* tbt1 handles all those m,p tokens that have travelled at least one leg of
   their journey, for these, we back up one link and send them back to CB */
trans tbt1(bt;cb,l):determ(0.0);
{
  predicate:{ (bt[3] != 4) &&
              (bt[2] == 0)   /* first pass through */
            };
  l[1] = bt[(3)];
  l[0] = bt[(3)-1];
  for(i=0;i<=bt[3]-1;++i)
    cb[i] = bt[i];
```

```
  cb[3] = bt[3]-1;
  cb[2] = bt[2] + 2;
}

trans tbt_punt(l,bt;cb,plr):exp(1.0);
{
  predicate:{ (bt[3] != 4) &&
              (bt[2] > 0) &&
              (l[0] == bt[0])
            };
  cb[0] = bt[0];
  cb[1] = bt[1];
  cb[2] = 0;
  cb[3] = 5;
  cb[4] = bt[0];
  cb[5] = l[1];

  plr[0] = bt[3] - 3;
  for(i=1;i<=bt[3]-3;++i)
    plr[i] = bt[i+3];
}

trans mg0(p;cb_p):determ(0.0);
{
  predicate:{1};
  cb_p[0] = p[0];
  cb_p[1] = funct t_message_gen1;
  cb_p[2] = 0;  /* start the count at 0 */
  cb_p[3] = 4;
  cb_p[4] = p[0];
}

trans bogus_msg(cb_p;p):determ(0.0);  /* send bogus msgs back to p */
{
  predicate:{cb_p[0] == cb_p[1]};
  p[0] = cb_p[0];
}

trans mg1(cb_p;cb):exp(20.0);
{
  predicate:{cb_p[0] != cb_p[1]};
  cb[0] = cb_p[0];
  cb[1] = cb_p[1];
  cb[2] = cb_p[2];
  cb[3] = cb_p[3];
  cb[4] = cb_p[4];
}

/* the message delivery place */
trans md(cb;plr,p):exp(1.0);
{
  /* predicate checks to see if last node in link chain is destination */
  predicate:{ (cb[1] == cb[(3)]) };
  p[0] = cb[0];
  plr[0] = cb[3] - 3;
  for(i=1;i<=cb[3]-3;++i)
    plr[i] = cb[i+3];
}

trans tlr(plr;plr,l):determ(0.0);
{
  predicate:{plr[0] != 2};
  plr[0] = plr[0] - 1;
  for(i=1;i<=plr[0]-1;++i)
```

```
    plr[i] = plr[i];
  l[1] = plr[(0)];
  l[0] = plr.out[(0)];
}

trans tlr_last(plr;l):determ(0.0);
{
   predicate:{ plr[0] == 2};
   l[0] = plr[1];
   l[1] = plr[2];
}

l_marks{
  p : (0),(1),(2),(3),(4),(5),(6),(7);
  l : (0,1),(1,0),(0,2),(2,0),(0,4),(4,0),(1,3),(3,1),(1,5),(5,1),(2,3),(3,2),
      (2,6),(6,2),(3,7),(7,3),(4,5),(5,4),(4,6),(6,4),(5,7),(7,5),(6,7),(7,6);
}

======================================================================
/* PPN desc file for 64 node N-cube*/
places: p,cb,plr,bt,cb_p,l;
trans: mg0,mg1,lb\,md,tlr,tlr_last,bogus_msg,tbt0,tbt1,tbt_punt;

place p(md,bogus_msg;mg0);
place cb_p(mg0;mg1,bogus_msg);
place bt(lb;tbt0,tbt1,tbt_punt);
place l(tlr,tlr_last;lb,tbt0,tbt_punt);
place cb(mg1,lb,tbt0,tbt1,tbt_punt;md,lb);
place plr(tlr,md,tbt_punt;tlr,tlr_last);

trans lb(cb,l\;cb;bt):exp(1.0):determ(0.5);
{
/*  predicate checks three things
  1: that the link starts at the current last node
  2: that the link goes in the right direction  */
/*3: that the link does not reverse the last link */
  predicate:{ (cb[1] != cb[(3)]) };
  predicate:{ (cb[(3)] == l[0]) &&
              ( ( (l[0]^l[1]) & (cb[(3)]^cb[1]) ) != 0) &&
              ( cb[3] == 4 ? 1 : (cb[(3)-1] != l[1]))
            };
  success{
    for(i=0;i<=cb[3];++i)
      cb[i] = cb[i];

    cb[3] = cb[3] + 1;
    cb[(3)] = l[1];
    cb[2] = (0 == cb[2]) ? 0 : (cb[2] - 1);  /* decrement tbt cntr */
  };
  failure{
    for(i=0;i<=cb[3];++i)
      bt[i] = cb[i];
  };
}

/*  tbt0 is the transition which deals with m,p tokens which have not gone
    anywhere yet and could not find a link in the right direction

    for these, we simply grab a link in any direction and send them
    back to place cb */
trans tbt0(bt,l;cb):exp(1.0);
{
  predicate:{  (bt[3] == 4) &&
               (l[0] == bt[(3)])
            };
    for(i=0;i<=bt[3];++i)
      cb[i] = bt[i];

    cb[3] = bt[3] + 1;
    cb[(3)] = l[1];
}

/* tbt1 handles all those m,p tokens that have travelled at least one leg of
   their journey, for these, we back up one link and send them back to CB */
trans tbt1(bt;cb,l):determ(0.0);
{
  predicate:{  (bt[3] != 4) &&
               (bt[2] == 0)   /* first pass through */
            };
  l[1] = bt[(3)];
  l[0] = bt[(3)-1];
  for(i=0;i<=bt[3]-1;++i)
    cb[i] = bt[i];
  cb[3] = bt[3]-1;
  cb[2] = bt[2] + 2;
}

trans tbt_punt(l,bt;cb,plr):exp(1.0);
{
  predicate:{  (bt[3] != 4) &&
               (bt[2] > 0) &&
               (l[0] == bt[0])
            };
  cb[0] = bt[0];
  cb[1] = bt[1];
  cb[2] = 0;
  cb[3] = 5;
  cb[4] = bt[0];
  cb[5] = l[1];

  plr[0] = bt[3] - 3;
  for(i=1;i<=bt[3]-3;++i)
    plr[i] = bt[i+3];
}

trans mg0(p;cb_p):determ(0.0);
{
  predicate:{1};
  cb_p[0] = p[0];
  cb_p[1] = funct t_message_gen1;
  cb_p[2] = 0;  /* start the count at 0 */
  cb_p[3] = 4;
  cb_p[4] = p[0];
}

trans bogus_msg(cb_p;p):determ(0.0);  /* send bogus msgs back to p */
{
  predicate:{cb_p[0] == cb_p[1]};
  p[0] = cb_p[0];
}

trans mg1(cb_p;cb):exp(20.0);
{
  predicate:{cb_p[0] != cb_p[1]};
  cb[0] = cb_p[0];
  cb[1] = cb_p[1];
  cb[2] = cb_p[2];
  cb[3] = cb_p[3];
  cb[4] = cb_p[4];
}

/* the message delivery place */
trans md(cb;plr,p):exp(1.0);
{
  /* predicate checks to see if last node in link chain is destination */
  predicate:{ (cb[1] == cb[(3)]) };
  p[0] = cb[0];
  plr[0] = cb[3] - 3;
  for(i=1;i<=cb[3]-3;++i)
    plr[i] = cb[i+3];
}

trans tlr(plr;plr,l):determ(0.0);
{
  predicate:{plr[0] != 2};
  plr[0] = plr[0] - 1;
  for(i=1;i<=plr[0]-1;++i)
    plr[i] = plr[i];
  l[1] = plr[(0)];
  l[0] = plr.out[(0)];
}

trans tlr_last(plr;l):determ(0.0);
{
   predicate:{ plr[0] == 2};
   l[0] = plr[1];
   l[1] = plr[2];
}

i_marks{
  p : (0),(1),(2),(3),(4),(5),(6),(7),(8),(9),(10),(11),
      (12),(13),(14),(15),(16),(17),(18),(19),(20),(21),(22),(23),
      (24),(25),(26),(27),(28),(29),(30),(31),(32),(33),(34),(35),
      (36),(37),(38),(39),(40),(41),(42),(43),(44),(45),(46),(47),
      (48),(49),(50),(51),(52),(53),(54),(55),(56),(57),(58),(59),
      (60),(61),(62),(63);
  l : (0,1),  (0,2),  (0,4),  (0,8),  (0,16),  (0,32),
      (1,0),  (1,3),  (1,5),  (1,9),  (1,17),  (1,33),
      (2,0),  (2,3),  (2,6),  (2,10),  (2,18),  (2,34),
      (3,1),  (3,2),  (3,7),  (3,11),  (3,19),  (3,35),
      (4,0),  (4,5),  (4,6),  (4,12),  (4,20),  (4,36),
      (5,1),  (5,4),  (5,7),  (5,13),  (5,21),  (5,37),
             [55 lines deleted for brevity]
      (61,29),  (61,45),  (61,53),  (61,57),  (61,60),  (61,63),
      (62,30),  (62,46),  (62,54),  (62,58),  (62,60),  (62,63),
      (63,31),  (63,47),  (63,55),  (63,59),  (63,61),  (63,62);
}

========================================================================
/* PPN desc file for 64 node toroidal mesh */
places: p,l,cb,plr,bt,cb_p;
trans: mg0,mg1,lb\,md,tlr,tlr_last,bogus_msg,tbt0,tbt1,tbt_punt;

place p(md,bogus_msg;mg0);
place cb_p(mg0;mg1,bogus_msg);
place bt(lb;tbt0,tbt1,tbt_punt);
place l(tlr,tlr_last;lb,tbt0);
place cb(mg1,lb,tbt0,tbt1,tbt_punt;md,lb);
place plr(tlr,md,tbt_punt;tlr,tlr_last);

trans lb(cb,l\;cb;bt):exp(1.0):determ(0.5);  /* meshed */
{
```

```
  predicate:{ (cb[1] != cb[(3)]) }; /* not at destination */
  predicate:{ (cb[(3)] == l[0]) &&  /* starts at right place */
              ( md(cb[1],l[1]) < md(cb[1],cb[(3)]) ) &&  /* must write md */
              ( cb[3] == 4 ? 1 : (cb[(3)-1] != l[1]))  /* no backtrack here */
            };
  success{
    for(i=0;i<=cb[3];++i)
      cb[i] = cb[i];
    cb[3] = cb[3] + 1;
    cb[(3)] = l[1];
    cb[2] = (0 == cb[2]) ? 0 : (cb[2] - 1);  /* decrement tbt cntr */
  };
  failure{
    for(i=0;i<=cb[3];++i)
      bt[i] = cb[i];
  };
}

/*  tbt0 is the transition which deals with m,p tokens which have not gone
    anywhere yet and could not find a link in the right direction
    for these, we simply grab a link in the wrong direction and send them
    back to place cb*/
trans tbt0(bt,l;cb):exp(1.0);
{
  predicate:{ (bt[3] == 4) &&
              (l[0] == bt[(3)])};
    for(i=0;i<=bt[3];++i)
      cb[i] = bt[i];
    cb[3] = bt[3] + 1;
    cb[(3)] = l[1];
}

/* tbt1 handles all those m,p tokens that have travelled at least one leg of
   their journey, for these, we back up one link and send them back to CB */
trans tbt1(bt;cb,l):exp(1.0);
{
  predicate:{ (bt[3] != 4) &&
              (bt[2] == 0)   /* first pass through */ };
  l[1] = bt[(3)];
  l[0] = bt[(3)-1];
  for(i=0;i<=bt[3]-1;++i)
    cb[i] = bt[i];
  cb[3] = bt[3]-1;
  cb[2] = bt[2] + 2;
}

trans tbt_punt(bt;cb,plr):determ(0.0);
{
  predicate:{ (bt[3] != 4) &&
             (bt[2] > 0) };
  cb[0] = bt[0];
  cb[1] = bt[1];
  cb[2] = 0;
  cb[3] = 4;
  cb[4] = bt[0];

  plr[0] = bt[3] - 3;
  for(i=1;i<=bt[3]-3;++i)
    plr[i] = bt[i+3];
}

trans mg0(p;cb_p):determ(0.0);
{
  predicate:{1};
  cb_p[0] = p[0];
  cb_p[1] = funct t_message_gen1;
  cb_p[2] = 0;  /* start the count at 0 */
  cb_p[3] = 4;
  cb_p[4] = p[0];
}

trans bogus_msg(cb_p;p):determ(0.0);  /* send bogus msgs back to p */
{
  predicate:{cb_p[0] == cb_p[1]};
  p[0] = cb_p[0];
}

trans mg1(cb_p;cb):exp(20.0);
{
  predicate:{cb_p[0] != cb_p[1]};
  cb[0] = cb_p[0];
  cb[1] = cb_p[1];
  cb[2] = cb_p[2];
  cb[3] = cb_p[3];
  cb[4] = cb_p[4];
}

trans md(cb;plr,p):exp(1.0);
{
  predicate:{ (cb[1] == cb[(3)]) /* at destination? */ };
  p[0] = cb[0];
  plr[0] = cb[3] - 3;
  for(i=1;i<=cb[3]-3;++i)
    plr[i] = cb[i+3];
}

trans tlr(plr;plr,l):determ(0.0);
{
  predicate:{plr[0] != 2};
  plr[0] = plr[0] - 1;
  for(i=1;i<=plr[0]-1;++i)
    plr[i] = plr[i];
  l[1] = plr[(0)];
  l[0] = plr.out[(0)];
}


trans tlr_last(plr;l):determ(0.0); /* meshed */
{  predicate:{ plr[0] == 2};
   l[0] = plr[1];
   l[1] = plr[2]; }


i_marks{

p :    (0),(1),(2),(3),(4),(5),(6),(7),(8),(9),
       /*[4 lines deleted for brevity]*/
      (50),(51),(52),(53),(54),(55),(56),(57),(58),(59),
      (60),(61),(62),(63);

l: (0,1),(0,7),(0,8),(0,56),
   (1,0),(1,2),(1,9),(1,57),
   (2,1),(2,3),(2,10),(2,58),
   (3,2),(3,4),(3,11),(3,59),
   (4,3),(4,5),(4,12),(4,60),
   (5,4),(5,6),(5,13),(5,61),
   (6,5),(6,7),(6,14),(6,62),
   /*[50 lines deleted for brevity]*/

   (57,1),(57,49),(57,56),(57,58),
   (58,2),(58,50),(58,57),(58,59),
   (59,3),(59,51),(59,58),(59,60),
   (60,4),(60,52),(60,59),(60,61),
   (61,5),(61,53),(61,60),(61,62),
   (62,6),(62,54),(62,61),(62,63),
   (63,7),(63,55),(63,56),(63,62);
}
```