Dependability Modeling and Analysis of Distributed Programs

Noé Lopez-Benitez, Member, IEEE

Abstract—This paper presents a modeling approach based on stochastic Petri nets to estimate the reliability and availability of programs in a distributed computing system environment. In this environment, successful execution of programs is conditioned on the successful access of related files distributed throughout the system. The use of stochastic Petri nets is demonstrated by extending a basic reliability model to account for repair actions when faults occur. To this end, two possible models are discussed: the global repair model, which assumes a centralized repair team that restores the system to its original status when a failure state is reached, and the local repair model, which assumes that repairs are localized to the node where they occur. The former model is useful in evaluating the availability of programs (or the availability of the hardware support) subject to hardware faults that are repaired globally; therefore, the programs of interest can be interrupted. On the other hand, the latter model can be used to evaluate program reliability in the presence of hardware faults subject to repair, without interrupting the normal operation of the system.

Index Terms—Reliability/availability models, distributed systems, stochastic Petri nets, file distribution

I. INTRODUCTION

In a distributed computing system (DCS), programs and data files are distributed among several processing units that may contribute to the simultaneous execution of a single program, or, alternatively, several programs can be executed concurrently. Higher performance and reliability can be obtained not only through fault-tolerant nodes and communication links but also in the form of redundant distribution of programs and data files throughout the system.

A DCS can be represented by a graph \( G(V, E) \), where the set \( V \) of vertices corresponds to the nodes (computational resources) and the set \( E \) corresponds to the communication links. Fig. 1 illustrates a typical representation [1] of a DCS as a set of resources (e.g., processing units, data files, and programs) interconnected as shown. In this figure, a given node can host a set \( F \) of files and a set \( P \) of programs. Node \( i \) consists of \( x_i \) number of processing units, and \( x_{ij} \) represents the number of bidirectional links that connect nodes \( i \) and \( j \). Several copies of files and programs can be distributed in the system. The evaluation of distributed program reliability and distributed system reliability measures the successful execution of a single program and a set of programs, respectively, in a distributed environment. Programs are executed successfully as long as access to the set \( FN \) of needed files is guaranteed. In this environment, the probability of successful execution of a single program defines the distributed program reliability and the distributed system reliability as the probability that all programs in the system are operational [2].

Because of the modeling complexity of large distributed systems, Markov modeling becomes impractical as a result of the generation of a large number of states even for moderate size systems. To circumvent this problem, several reliability evaluation methods [3], [4] have been proposed based on graph theory, Boolean algebra and probability theory, such that closed-form expressions can be obtained. Some of these techniques rely on path enumeration (cutset, spanning trees) [5]–[8], which involve, first, finding all possible paths between a pair of nodes and, second, mapping this set of paths into Boolean expressions that represent mutually exclusive paths. Probability theory is then applied to generate symbolic expressions for exact reliability estimations. The evaluation procedure outlined in [2] to evaluate program and system reliability relies on graph theory to derive minimal file spanning trees (MFST’s). Although these models are useful, they are limited to the evaluation of reliability only. Consequently, the flexibility of Markov models cannot be used to extend the analysis to other measures of interest. To make use of the flexibility of Markov models, it is common to use stochastic Petri nets (SPN’s) to automatically generate them. Several tools already exist for this purpose [9]. Note that the user is shielded only from the large size problem, because a reliable
solution of very large models is still an open problem. In this paper, SPN's are used to represent a DCS subject to link and node failures. It is shown that at the SPN level, state reduction can be achieved by generating only those states that satisfy functional requirements specified in the form of symbolic Boolean expressions. These expressions may correspond to those proposed in the literature used for the reliability evaluation of large systems [3], [4]. To model the effects of global and local hardware repairs, extensions to the basic reliability model of a small system are discussed in detail. Software failures either in the system software or the programs themselves are not directly addressed in this paper. However, evaluation results obtained by using independent models can be aggregated to the models generated by the scheme presented. A generalization for complex models and the results reported were obtained by using the stochastic Petri net package (SPNP) [10].

II. STOCHASTIC PETRI NETS

Petri nets are graphical models used to represent and analyze complex systems with interdependent components [11]. The nodes of a Petri net are referred to as places (circles) and transitions (bars), which represent conditions and events, respectively. Each place may contain tokens (small dots), which are moved from place to place when transitions fire. Places and transitions are connected by arcs that determine the movement of tokens throughout the net. A place p is an input place for a transition t if there is an arc from p to t. An arc from t to p identifies an output place for t. A transition is enabled if each input place contains at least one token. Once enabled, the transition fires by removing a token from each input place and placing a token in each output place. A multiplicity can be associated with each arc; this is an integer that determines the number of tokens removed from an input place or deposited in an output place; in this case, a transition is not enabled if any input place does not have enough tokens. A marking is the collection of tokens in each place. When a transition fires it generates a new marking. The set of markings generated are the nodes of a reachability graph, which are linked by the firing of transitions.

An stochastic Petri net (SPN) mode associates with each transition an exponential firing time [12]. This feature allows an isomorphism with Markov chains and thus provides a useful interface in the case of models with a very large number of states. The models in this paper are based on generalized stochastic Petri nets (GSPN's), which allow two types of transitions: timed transitions (small rectangular bars) with exponentially distributed firing times and immediate transitions (thin bars) that fire as soon as they are enabled [13]. The tool of choice to solve the models developed in this paper is the stochastic Petri net package (SPNP) [10], which relies on GSPN's, plus other features such as marking dependency, variable cardinality of arcs and enabling functions. Marking dependency allows the specification of firing rates and probabilities of transitions in terms of markings. The variable cardinality defines the multiplicity of input and output arcs in terms of markings; i.e., places can be flushed or updated with a desired number of tokens. Enabling functions are Boolean functions associated with transitions; they represent conditions that must be satisfied before transitions can fire. It is shown that these features add valuable flexibility to the modeling power of SPN's.

III. A RELIABILITY MODEL

Consider the graph in Fig. 2, which represents a small distributed system with three nodes interconnected as shown. One objective is to estimate the reliability of the program P1, which requires access to files F1, F2, and F3, i.e., FN1 = {F1, F2, F3}. The program and files are distributed as shown. If the system is repairable, it is desirable to estimate the availability of the system that supports the execution of the programs of interest.

The SPN model shown in Fig. 3 describes the fault model of the three-node system of Fig. 2. A place is assigned to each node and links. These places are p0, p1, p2, ..., p5, respectively. Each place may contain more than one token that corresponds to the number xi of processing units or the number xij of links. The timed transitions t0, ..., t5, represent the exponential failure occurrence of each component type. Each transition tij fires at a rate λij, which is the failure rate of the corresponding component. The failure rate of links xij will also be denoted λij. To avoid the generation of a large state space, it is desirable to generate only those states that correspond to a configuration that supports the execution of a single program or a set of distributed programs that require several files distributed in the system. Such discrimination of states can easily be carried out by associating a common enabling function with each timed transition in the model. For example, for the program P1, which resides in all nodes, the following logical expression can be used as an enabling function of each timed transition:

\[ g = g_{12} g_{13} + g_{12} g_{13} \]

(1)

where

\[ g_{ij} = \begin{cases} 1 & \text{if } x_{ij} > 0, \\ 0 & \text{otherwise}; \end{cases} \]

(2)

Note that if \( x_{ij} > 0 \) (with at least one token in the corresponding place), then the node \( i \) is active. Likewise, if \( x_{ij} > 0 \), then there is at least one active link between nodes \( i \) and \( j \).

Since links are assumed to be full duplex, then \( g_{ij} = g_{ji} \).

Equation (1) can be obtained by using the notion of minimum file spanning trees (MFST). An MFST, as introduced in [2], is a file spanning tree (FST) not embedded in any other FST. An FST connects the node that executes the program (under consideration) with every other node that contains at least one
Fig. 3. SPN of a three-node system with global repair.

The file required for execution. Thus, each term $m, 1 \leq m \leq M$ in (1) corresponds to an MFST such that the following expression is true:

$$g = \sum_{m=1}^{M} \text{MFST}_m,$$

where $M$ is the number of MFST's. Note that the function $g$, defines the type of reliability measure performed. In this case, because $P_1$ is the only program in the system, the same function can be used for both program reliability and system reliability.

Any marking generated that does not satisfy (1) is an absorbing marking. Therefore, all absorbing markings can be merged into one easily identifiable marking by firing transition $t_o$; this is an immediate transition that will empty all places when the following enabling function is satisfied:

$$a = g \sum_{i \neq j} (g_i + g_j).$$

If $t_o$ fires before any timed transition fires, then only non-absorbing markings are left, and there is no need to use the function $g$ to enable each timed transition. For the model in Fig. 3, $t_o$ has the following enabling function:

$$a = g(g_1 + g_2 + g_3 + g_{12} + g_{13} + g_{23}).$$

To generate a single absorbing marking, all input places of $t_o$ have arcs with variable cardinality equal to the number of tokens in the corresponding input places, such that when $t_o$ fires, all places are flushed creating a null marking. Thus, $t_o$ is enabled only when $g$ is not satisfied and at least one place is not empty; in order to avoid absorbing loops, null markings should not enable $t_o$.

The immediate transitions $t_{i0}, t_{i1},$ and $t_{i2}$, model the effects of faulty nodes or links; all arcs have variable cardinality, such that when they fire, their input places are flushed. For example, if $x_i = 0$, i.e., node 1 has failed, there is no need to preserve its connecting links. Therefore, $t_{i0}$ fires to flush the input places for the connecting links $x_{i12}$ and $x_{1i3}$; likewise, if these links have failed, node 1 is not reachable, and transition $i_{i0}$ fires to flush $p_0$ where $x_1$ tokens reside. Thus, the firing of an immediate transition is conditioned on the logical value of an enabling function of the following form:

$$f_i = l_i \oplus g_i,$$

where $l_i = \sum_{i \neq j} g_{ij}$ for $1 \leq i, j \leq n$, and $n$ is the number of nodes.

In Fig. 3, the enabling functions associated with $i_{i0}, i_{i1},$ and $i_{i2}$ are, respectively:

$$f_1 = l_1 \oplus g_1, f_2 = l_2 \oplus g_2, f_3 = l_3 \oplus g_3,$$

where $l_1 = g_{12} + g_{13}, l_2 = g_{21} + g_{23},$ and $l_3 = g_{31} + g_{32}.$

The inclusion of the immediate transitions $t_o, 0 \leq i < n$ has the effect of reducing the number of tangible markings and thus the number of states in the underlying Markov chain. For example, $i_{i0}$ is enabled if $x_{i12} = 0$ and $x_{i13} = 0(l_i = 0)$ and $x_1 > 0(g_1 = 1)$, or, if any link is active, $(l_i = 1)$ and $x_1 > 0(g_1 = 0)$.

If only permanent faults are considered, then recovery mechanisms by which a program can resume execution after a crash-restart sequence can be ignored. Furthermore, we can assume that nondetected faults may cause the execution of the program under consideration to be aborted, because at any time a sequence of nondetected faults may have occurred such that the function $g$ is no longer satisfied. A nondetected fault causes a program to fail with probability $1 - c$; this probability is associated with the immediate transition $t_o$, which fires after a timed transition has fired and a token is placed in $p_0$. Transition $t_{e}$ models covered faults and fires with probability $c$. Both transitions $t_o$ and $t_{e}$ fire simultaneously (i.e., with the same priority). Because SPNP does not allow zero probabilities to be associated with immediate transitions, enabling functions are provided to explicitly enable or disable these transitions, depending on the value of $c$. Thus, the enabling functions for the coverage transitions are given as follows:

$$cf = (c > 0)g \text{ and } af = (c < 1)g.$$

Hence, the effect of firing transition $t_o$ and $t_{e}$ is simply to aggregate the values $c$ and $1 - c$ to the underlying Markov chain. Note that transition $t_o$ does not generate a new marking and that the value $c$ is aggregated to transitions between operational states. On the other hand, transition $t_{e}$ generates a failure marking and the value $1 - c$ is aggregated to transitions to the failure state in the corresponding Markov model; both transitions fire only when $g$ is satisfied.

The enabling functions $(e_i)$ associated with timed transitions are determined as follows:

$$e_i = \begin{cases} 1 & \text{if } \lambda_i > 0, \\ 0 & \text{otherwise.} \end{cases}$$

The notation \((\ast)\) is used to identify enabling functions in the SPN model.

Assuming one processing unit in each node and one link between nodes, the Markov model generated by the SPN in Fig. 3 is shown in Fig. 4. Each state is labeled as \((s : x_1, x_2,\ldots, x_n)\).
$x_3, x_{12}, x_{23}, x_{13}$, where $x$ identifies the state number. The label $F$ is used to identify the failure state which is reached when there are no operational units left.

The operational states correspond to tangible markings generated by the SPN. Note that $g$ in (1) evaluates to 1 in each operational state. All components are assumed to have the same failure rate $\lambda$. Note, for example, that transition $2x_3$ from state 1 to state 4 corresponds to the failure of node 1 or to the failure of the link between nodes 1 and 3.

IV. A GLOBAL REPAIR MODEL

If the system is repairable, it is desirable to estimate the availability of the system that supports the execution of the programs of interest. For this purpose, it is assumed that a single repairman is available for the overall system. This assumption is reasonable for distributed systems confined to small geographic areas. However, the repair team is activated until a series of nodes and links fail; otherwise, it disables the program supporting hardware. When an absorbing marking is reached because the function $g$ is not satisfied, a token is transferred to place $p_1$ to indicate that the system is now under repair. Transition $t_e$ fires with a rate $\mu$ that corresponds to the system repair rate. After repair, the system is then restored to its original status by associating a variable cardinality in each output arc that takes the value of the initial number of tokens in the corresponding output place. An additional arc from $t_a$ to $p_7$ indicates that the repair team can also be activated if the program crashes because of nondetected faults or because software faults have occurred. In this case, a cyclic Markov chain results and the instantaneous and interval availability of the support system can be obtained. To model the effect of hardware repairs on the reliability of programs running in the system, the output arc of $t_a$ can be removed according to a user-supplied value of a binary variable.

V. A LOCAL REPAIR MODEL

The model shown in Fig. 5 is a modification of the one shown in Fig. 3. Fig. 5 includes timed repair transitions $r_{10}, r_{11}, \cdots, r_{15}$. Each $r_{1i}$ has a firing rate of $m_i$. These firing rates correspond to the repair rate of the respective component (node or link). The presence of a token in places $r_{p0}, \cdots, r_{p0}$ indicates that the respective component is under repair.

The immediate transitions $i_{t0}, i_{t1}$, and $i_{t2}$, perform the same modeling function described for the model in Fig. 3, and fire when $f_1, f_2$, and $f_3$ are satisfied. Whenever any of these conditions is satisfied, however, a token is transferred to place $xp_0, xp_1$, or $xp_2$, respectively. Consider the case when $i_{t0}$ fires and places a token in $xp_0$. This condition indicates that either node 1 is on repair or both links $x_{12}$ and $x_{13}$ are on repair. If $x_{1}$ is on repair, then the links are also disabled; but they must be restored when node 1 is repaired, i.e., after transition $r_{10}$ fires. To capture this behavior, the model is extended to include immediate transitions $x_{10}, x_{11}$, and $x_{12}$. Thus, for node 1, $x_{10}$ will fire when the following enabling function is satisfied:

$$h_1 = g_1 + g_2 + g_3 = g_1 + g_2$$

which indicates that as soon as either component $x_1, x_{12}$, or $x_{13}$ become active, the status previous to firing $i_{t0}$ is restored; i.e., all removed tokens are replaced. Thus, for any node $i$, the following condition exists:

$$h_i = g_i + l_i$$

If $m_{x_{10}, p_0}$ denotes the cardinality of the output arc from $x_{10}$ to $p_0$, then, in order to restore retrieved processing units to node 1, the following conditions must exist:

$$m_{x_{10}, p_0} = ix_1 - \#r_{p0} - \#p_0$$

where $ix_1$ denotes the initial number of active units in node 1, i.e., the initial number of tokens in place $p_0$. The notation $\#p$ denotes the number of tokens in place $p$. If the number of units on repair plus the currently active units is not equal to the initial number of units, then the difference was retrieved when $i_{t0}$ fired. Likewise, to restore retrieved links, the following cardinalities are determined similarly:

$$m_{x_{10}, p_3} = (ix_{12} - \#r_{p3} - \#p_3)g_3$$

and

$$m_{x_{10}, p_5} = (ix_{13} - \#r_{p5} - \#p_5)g_3$$

Note, however, that links should be restored only when the respective connected nodes become active. For example, links $x_{12}$ connect nodes 1 and 2. If node 2 is active, $(g_2 = 1)$ and $x_{12} = 0$ imply that all links are on repair or were temporarily retrieved when $i_{t0}$ fired. If node 2 now becomes inactive,
then links $x_{12}$ will be restored only until node 2 is active again. Likewise, to restore processing units, at least one of the connecting links should be active. To ensure that this is so, (6) is modified accordingly. For example, for node 1 in Fig. 5, the enabling function $h_1$ for $x_{10}$ becomes:

$$h_1 = g_1[(m_{x_{10},p_1} > 0) + (m_{x_{10},p_2} > 0)] + l_1(m_{x_{10},p_3} > 0).$$  

(10)

The first term of (10) indicates that if node 1 is active (repaired), and if there is at least one link to restore, then $x_{10}$ is enabled. The second term checks that if at least one link is active (repaired) and if there is a processing unit to restore, then $x_{10}$ is enabled. However, this will restrict the firing of transition $x_{10}$ even when the conditions to fire $x_{10}$ are created, causing the anomalous situation of accumulating tokens in $x_{10}$. To eliminate this problem, the output arc of $x_{10}$ is made variable and subject to the nonzero number of tokens in $x_{10}$. To also use this model as a reliability model, it suffices to check that at least one of the components corresponding to the input places of $x_{10}$ has a zero repair rate. Let $s_i$ denote the sum of all repair rates involved; then the cardinality of the output arc of $x_{10}$ is determined as follows:

$$m_{x_{10},xp_i} = \begin{cases} 1 & \text{if } (s > 0) \text{ and } (\#xp_i = 0), \\ 0 & \text{otherwise}. \end{cases}$$  

(11)

If $\mu_i$ denotes the repair rate of the $k$th component, then $s_i = \sum_j \mu_j$ where $j$ identifies all the input places to transition $x_{10}$.

The enabling functions of repair transitions $r_i$ are specified as follows:

$$cr_i = \begin{cases} 1 & \mu_i > 0, \\ 0 & \text{otherwise}. \end{cases}$$  

(6)

VI. MODELS OF LARGE SYSTEMS

Using (2)–(11), large systems can be modeled via built-in functions provided in SPNP. These functions include trans, place, voarce, viarce, enabling, and priority, and are used for defining transitions, places, variable output arcs, variable input arcs, enabling functions, and firing priority, respectively. The SPN models in Figs. 3 and 4, exhibits regular structures that can be easily described by using vector array structures. Let $np$ denote the total number of places required for the global repair model shown in Fig. 3, such that all places are indexed $p_0, p_1, \cdots, p_{np-1}$. In a system with $n$ components (nodes and links), a place $p_i$ and a timed transition $t_i, 0 \leq i < n$ are assigned to each component that may fail. The immediate transitions $it$ in this model, simply describe the interconnection structure of the nodes. To model uncovered and covered faults, one additional place with index $np - 2$ and two immediate transitions $t_u$ and $t_c$ are required. To model repairs, one more place with index $np - 1$ and a timed transition $t_r$ are added. Finally, to capture absorbing markings the immediate transition $t_a$ is included.

To model local repairs (Fig. 5), it suffices to add repair places and repair transitions for those components subject to failure. The repair places are identified as $r_{p0}, r_{p1}, \cdots, r_{pnp-1}$, where $np$ corresponds to the number of links plus the number of nodes in the system. The repair transitions are denoted as $rt_0, \cdots, rt_{np-1}$.

The auxiliary immediate transitions $xt$ are defined for each node as $x_{i0}, \cdots, x_{in-1}$, and the auxiliary places $xp$ are defined as $xp_0, \cdots, xp_{np-1}$. Built-in functions provided by SPNP can be used for systems of several sizes, and different structures as places, transitions, and arcs can be specified as one or 2-D arrays [10], [14]. A detailed description is given in the appendix.

Using the function $g$ as a reward rate $X$ assigned to tangible markings, the expected reward rate $E[X] = \sum_i r_i \pi_i$ corresponds to the steady state availability of the system, where $r_i$ is the value of $g$ evaluated for marking $i$ and $\pi_i$ is the steady state probability of tangible marking $i$. In the case of global repairs, availability results can also be obtained by observing the probability that a token is in the repair place $np - 1$. For the case in Fig. 3, this place is $p_r$. This value represents the steady state unavailability of the system.

VII. RELIABILITY ANALYSIS

In this section, reliability results are generated by using either the global repair models of the type shown in Fig. 3 or the local repairs as shown in Fig. 4. These results have been validated [15] by using probability expressions derived based on the MFST’s. For large models, these expressions can be generated by using packages such as SYREL [8] designed to evaluate multiterminal reliability. Consider the six-node system described in Fig. 6, which also shows the place assignment for nodes and links.

The program of interest, $P_1$, runs in node 1 and node 6 with $FN_1 = \{F_1, F_2, F_3\}$. This allocation of programs and files is referred to as configuration A1. The logical expression obtained by enumerating the minimum file spanning trees is as follows:

$$g = g_1 g_2 g_3 g_4 g_5 g_6 + g_1 g_2 g_3 g_4 g_6 g_5 + g_1 g_2 g_3 g_4 g_5 g_6 + g_1 g_2 g_3 g_4 g_5 g_6 + g_1 g_2 g_3 g_4 g_5 g_6 + g_1 g_2 g_3 g_4 g_5 g_6.$$  

(A1)

One model direction in the analysis of program reliability is to predict how a given configuration responds to failures of components of different types and to undetected faults, which in this case may be caused by both hardware and software failures. For the configuration A1 shown in Fig. 6, Table I shows the reliability of the program $P_1$, assuming two values of the coverage factor ($c = 1$ and $c = 0.95$) and two failure coefficients ($r = 0$ and $r = 0.001$). The failure coefficient
TABLE I

<table>
<thead>
<tr>
<th>r = 0</th>
<th>r = 0.001</th>
<th>r = 0</th>
<th>r = 0.001</th>
</tr>
</thead>
<tbody>
<tr>
<td>c = 1</td>
<td>c = 1</td>
<td>c = 0.95</td>
<td>c = 0.95</td>
</tr>
<tr>
<td>0.1</td>
<td>0.9571063</td>
<td>0.9570990</td>
<td>0.9337090</td>
</tr>
<tr>
<td>0.3</td>
<td>0.7287716</td>
<td>0.7269987</td>
<td>0.6898890</td>
</tr>
<tr>
<td>0.5</td>
<td>0.4802827</td>
<td>0.4801569</td>
<td>0.4460965</td>
</tr>
<tr>
<td>0.7</td>
<td>0.2915508</td>
<td>0.2914313</td>
<td>0.2679882</td>
</tr>
<tr>
<td>0.9</td>
<td>0.1683837</td>
<td>0.1682796</td>
<td>0.1534405</td>
</tr>
<tr>
<td>1.0</td>
<td>0.1263694</td>
<td>0.1262814</td>
<td>0.1147473</td>
</tr>
</tbody>
</table>

MTTF = 0.5666667 0.5665431 0.5361639 0.5359946

is defined as the ratio between the failure rate of links \( \lambda_i \) (assuming that all \( \lambda_j \) are equal) over the failure rate of the nodes \( \lambda_n \) (assuming that all \( \lambda_j \) are equal); hence, \( r = 0 \) implies that links are assumed fault-free, whereas \( r = 0.001 \) implies that nodes fail 1000 times more frequently than links. In any case, we assume that \( \lambda_n = 1 \). Because of the existence of two copies of \( P_1 \), failures of links have little effect with respect to the results obtained with failures of nodes only. However, the program appears to be more vulnerable to unsuccessful fault detection and to software faults as well, because the reliability and MTTF decrease significantly when \( c \) is reduced from 1 to 0.95.

Another direction in the reliability analysis of this type of systems is to predict the effect of different allocations of programs and files. For this purpose, let us assume a configuration A2 where program \( P_1 \) resides in node 1 only, but all of the files required remain unchanged. This configuration has a function \( g \) as follows:

\[
\begin{align*}
g &= g_1g_2g_3g_4g_5g_6 + g_1g_2g_3g_4g_5g_6 + g_1g_2g_3g_4g_5g_6 + g_1g_2g_3g_4g_5g_6 \\
&= g_1g_2g_3g_4g_5g_6 + g_1g_2g_3g_4g_5g_6 + g_1g_2g_3g_4g_5g_6 + g_1g_2g_3g_4g_5g_6 \\
&= g_1g_2g_3g_4g_5g_6 + g_1g_2g_3g_4g_5g_6 + g_1g_2g_3g_4g_5g_6 + g_1g_2g_3g_4g_5g_6
\end{align*}
\]

(A2)

Consider also a third configuration A3, in which \( P_1 \) resides in nodes \( x_1, x_2 \) and \( x_6 \). Again, the same distribution of files is needed. This configuration has a function \( g \) as follows:

\[
\begin{align*}
g &= g_1g_2g_3g_4g_5g_6 + g_1g_2g_3g_4g_5g_6 + g_1g_2g_3g_4g_5g_6 + g_1g_2g_3g_4g_5g_6 \\
&= g_1g_2g_3g_4g_5g_6 + g_1g_2g_3g_4g_5g_6 + g_1g_2g_3g_4g_5g_6 + g_1g_2g_3g_4g_5g_6
\end{align*}
\]

(A3)

The plots in Fig. 7 show the effect of configurations A1, A2, and A3 under the assumptions of perfect coverage (\( c = 1 \)) and fault-free links (\( r = 0 \)). The failure rate of nodes is assumed to be \( \lambda_n = 1 \). Results are compared with respect to the centralized case (configuration A4), in which a single copy of \( P_1 \) and all required files reside in the same node; if this node is node 1, then \( g = g_1 \), and the program reliability is equivalent to the reliability of this node. Under the given assumptions, the results are expected, because A3 shows a considerable improvement in reliability and MTTF with respect to A1 and A2. The improvement over the centralized case is limited to only a short period of time for the set of parameters chosen. However, that may not be the case if the analysis is extended to more strict assumptions that consider, for example, the age and technology of the equipment used at each node. This condition translates into nodes with different failure rates, which is expected in the case of heterogeneous computer networks. This is illustrated by an additional plot labeled A1* in Fig. 7, where A1 has been evaluated with failure rates: \( \lambda_1 = 1 \) and \( \lambda_i = 0.32 \), \( 2 \leq i \leq 6 \). The links are also assumed to be fault-free (\( r = 0 \)). The lower failure rates of the indicated nodes is reflected in a considerable improvement of the system.

VIII. EFFECT OF HARDWARE REPAIRS

In the case of the global repair model, when failures occur a repair action takes place after the hardware support for a given application has been exhausted, not necessarily after a fault has been detected. The resulting model is useful to estimate program availability, which can be interpreted as the availability of the system hardware support for that particular program or application. For the case of the local repair model, however, repairs occur at the node level as faults are detected. The system can fail when there is not enough active hardware support, because it is on repair. To compare both cases, we refer to the results as corresponding to the dependability of the hardware support system. Assuming that on the average, one fault is detected per month, the configuration A1 described in Fig. 6 is analyzed for a mean time to repair of 1 hr and 24 hr. Only nodes fail at a rate of \( \lambda_i = 1 \), \( 0 \leq i \leq 6 \), and faults are detected 100% of the time; i.e., \( c = 1 \). The numerical results for the configuration shown in Fig. 6 are tabulated in Table II.

As expected, global repair policies exhibit improved dependability, because any detected failure triggers a repair.
action that restores the entire system to its original status. On the other hand, local repairs take place independently, and once faults accumulate and repairs are not performed in a timely manner, the program or programs in question cannot be supported. Global repairs are practical for systems restricted to small geographic areas. However, though local repairs make sense for large systems that cover a large geographic area, the model described in Fig. 5 can be used to estimate the reliability of distributed programs as repairs take place, without interrupting execution of programs. The system fails if the minimum hardware support required is on repair. Thus, in terms of program reliability, Fig. 8 compares results obtained using the local repair model in Fig. 5. The cases plotted correspond to $c = 0.95$ and $c = 0.9$ under the assumptions that repairs take place with a $mtr = 1$ hr. These results are compared with the results obtained when assuming that no repairs take place. Two additional plots in Fig. 8 are obtained by using the global repair model that results after removing the output arc of the immediate transition $t_u$ in Fig. 3. The underlying Markov chain has an absorbing state that is reached from every operational state because of imperfect coverage ($c < 1$), for which, unlike the global scheme, no repair action is accounted for. In this case, it is assumed that the effect is global as the failure state is reached with an aggregated probability of $1 - c$. Thus, the coverage factor aggregated in both models can be evaluated independently to account not only for undetected hardware failures but for software failures as well. The results show that the global repair is the policy of choice if the programs that are evaluated can be interrupted. Otherwise, a local repair policy, as expected, improves the reliability of programs considerably.

IX. CONCLUSION

In this paper, dependability models to analyze distributed computing systems were discussed. These models are based on stochastic Petri nets and rely on previous analytic models to derive reliability and availability estimations of distributed programs. The use of stochastic Petri nets is demonstrated by extending the basic reliability model embedded in Fig. 3 to account for repair actions when faults occur. These extensions result in two repair models. A global repair model that assumes a centralized repair team restores the system to its original status when a failure state is reached. The local repair model assumes that repairs are localized to the node where they occur. The former model is useful in evaluating the availability of programs (or the availability of the hardware support) subject to hardware faults that are repaired globally, and therefore the programs of interest can be interrupted. On the other hand, the latter model can be used to evaluate program reliability in the presence of hardware faults subject to repair without interrupting the normal operation of the system. An advantage in using the models proposed in this paper is their possible use in the study of performability related issues. Also, these models can be extended to account for the likelihood of communication and software related faults.

APPENDIX

Built-in functions provided by SPNP can be used for systems of several sizes and different structures as places, transitions, and arcs can be specified as 1-D or 2-D arrays [10], [14]. This is possible by building a header or a batch file with the appropriate parameters that specify the number of nodes, number of links, number of MFST's, index of places to generate the function $g$, and the input places of the immediate transitions. The set of parameters specified in the header or batch file is unique for each allocation of programs and files (configuration), and it is compiled with the set of SPNP functions to generate the SPN model. Consider, for example, the three-node system with a graph as in Fig. 2. The header built to describe the model for local repairs is as follows:

```
int ndns = 3; /*number of nodes*/
int nlks = 3; /*number of links*/
int np = 7; /*number of places*/
int nrp = 6; /*repair places*/
int nxp = 3; /*auxiliary places*/
int imark[6]; /*initial marking*/
double l[6]; /*array for failure rates*/
double mu[6]; /*repair rates*/
int ntm = 3; /*number of MFST's*/

struct mfsd_t {
  int nps; /*number of places involved*/
  int pls[3]; /*store indexes of these places*/
} mfsd[3] = {
  {3, {0, 1, 3}}, /*description of mfst's in*/
  {3, {0, 2, 5}}, /*term of place indexes*/
  {3, {1, 2, 4}},
};

struct itr_d_t {
  int nk; /*number of links*/
  int lkp[2]; /*place indexes of links*/
  int to_md[3]; /*nodes connected to current
    node*/
} itr[3] = {
  /*structure description*/
  {2, {3, 5}, {1, 2}}, /*in terms of input places to*/
  {2, {3, 4}, {0, 2}}, /*immediate transitions*/
  {2, {4, 5}, {0, 1}},
};
```
Note that some of these parameters are derived from the number \textit{nnds} of nodes and the number \textit{nlks} of links. Therefore, it is necessary only to use the graphical description of the system (Fig. 2) to specify the SPN model, because the header for the global or local repair model can be generated by an external program. Once the places are assigned, the MFST’s and the structure of the system are specified via the respective \textit{C}-structures shown and the appropriate built-in SPNP functions. Consider, for example, the array structure \textit{itr}, which is used to implement the enabling function \textit{f}_i in (5) as follows:

\begin{verbatim}
    enabling.type f(i) {
        int li, gi;
        li = 0;
        for(j=0; j < itr[i].nlk; j++)
            li = li + mark.1("p", itr[i].lkp[j]);
        gi = mark.1("p", i);
        return(gi & li); }
\end{verbatim}

Consider now the use of the array structure \textit{mfst} to implement the function \textit{g} given by (3) and mainly involved with the firing of the absorbing transition \textit{t}_a:

\begin{verbatim}
    int g() {
        int s, m;
        s = 0;
        for(i=0; i < nmt; i++) {
            m = 1;
            for(j = 0; j < mfst[i].nps; j++)
                m = m*mark.1("p", mfst[i].psl[j]);
            s = s + m;
        }
        if(s) return(1);
        else return(0); }
\end{verbatim}

Finally, the following code generalizes (7)-(10) for each and any number of nodes:

\begin{verbatim}
    enabling.type h(i) {
        int li, gi, tkl, tkn;
        gi = mark.1("p", i);
        tkn = chk.n(i, i);
        li = 0; tkl = 0;
        for(j = 0; j < itr[i].nlk; j++) {
            li += mark.1("p", itr[i].lkp[j]);
            idx = j;
            tkl += chk.l(i, i, itr[i].lkp[j]);
        }
        if((tkn > 0 && li > 0) || (tkl > 0 && gi > 0))
            return(1);
        else return(0); }
\end{verbatim}

\begin{verbatim}
    int chk.n(i, j, k) {
        return(mark[k] - mark.1("p", k)
                 - mark.1("p", k));
    }
\end{verbatim}

\begin{verbatim}
    int chk.l(i, j, k) {
        if(mark.1("p", itr[j].to.nl[idx])
            return(mark[k] - mark.1("p", k)
                    - mark.1("p", k));
        else return(0); }
\end{verbatim}

\section*{Acknowledgment}

The author gratefully acknowledges the useful contributions of four anonymous reviewers. Their comments and suggestions greatly improved this paper.

\section*{References}


