Performability Modelling of Distributed Systems using Layered Queueing Networks

Olivia Das and C. Murray Woodside

Department of Systems and Computer Engineering,
Carleton University, Ottawa, Ontario, Canada K1S 5B6
email: odas@sce.carleton.ca, *cmw@sce.carleton.ca*
Date: August 25, 1998

Abstract

Proliferation of large and complex fault-tolerant distributed systems in recent years has stimulated the combined modelling of performance and dependability of such systems. For large systems it may be very expensive to compute valid performance estimates to be used in the combined performability measures. This work considers two different classes of fault-tolerant client-server systems, in which efficient Layered Queueing Network models can be combined with AND-OR graph analysis. The fault-tolerance in the first class is achieved by incorporating stand-by servers and a mechanism to re-direct the service requests at the time of failure whereas fault-tolerance in the second class is achieved by incorporating redundant servers that can all be used at the same time with the service load being distributed equally among them. The model solutions in both the cases can be obtained without solving very large Markovian models.

Keywords: Fault-tolerance, distributed systems, performability, layered queueing networks, client-server, AND-OR graphs, dependability.

1. Introduction

The increased use of complex mission-critical distributed systems has motivated the study of performance and dependability modelling of such systems in an unified manner, popularly known as performability [1]. Since the failure of such systems incurs high costs, there has been a great emphasis on the formal modelling of their fault-tolerant aspects in order to anticipate their expected performability measure. Such systems may be made fault-tolerant by incorporating stand-by servers and a mechanism to re-direct the service requests at the time of failure. They are usually structured in layers, with some kind of operational control or user-interface tasks as the topmost layer, making requests to various layers of servers. Examples of such systems include telecommunication systems, avionic systems, banking systems and airline reservation systems. The purpose of this paper is to describe a performability model termed as Fault-Tolerant Layered Queueing Network (FTLQN) model that can efficiently predict the performability of such fault-tolerant client-server-like distributed systems. This paper further extends the FTLQN modelling concept to model those client-server systems which involve the usage of redundant servers that can all be used at the same time with the service load being distributed equally among them in order to increase the system processing power.

The common approach in performability modelling is to describe the system by a Markov Reward model (e.g., [2], [3], [4], [5], [6], [7]) in which the dependability aspects of the system are modelled by a continuous time Markov chain and a reward rate is associated with each of the states of the Markov chain that denotes the performance level in that state. However, for large systems, the state space in a Markov Reward model becomes very large so that to obtain the solution of such models becomes difficult. Another version of this approach is based on the Dynamic Queueing Network concept [8], [9], [10] that combines the queueing network analysis with the generalized stochastic Petri nets (GSPN) to obtain the performability measures. In this approach, a parameterized queueing network is used to model the performance aspect of the system and a GSPN is used to describe the dependability aspect of the system. The technique comprises of a function that maps the possible markings of the

GSPN, denoting the system configurations, to the queueing network parameters. Then, a reward rate is associated with every possible GSPN marking by solving the corresponding queueing network model. Finally, the actual calculations for performability are done by translating the dynamic queueing network model to a Markov Reward model. This technique is similar to the approach presented in this paper in that both of them

- first determine the possible configurations of the system by using a dependability model describing the system changes due to failure, and
- second assign a reward rate to each of those configurations by solving a performance model corresponding to that configuration.

However the present work takes advantage of the layered structure and the stand-by server strategy, to avoid solving a large Markov Reward model, and to minimize the number of performance solutions required.

Since the performance aspect of a client-server-like distributed system can be conveniently modelled using Layered Queueing Network (LQN) concept [11], [12], [13] and since AND-OR graphs [14] may provide a convenient means to model the dependencies of the service failures in such systems, we propose to combine these two concepts to develop a new performability model called an FTLQN model, by adding the dependability-related parameters to the existing LQN model. We then present a solution strategy for the FTLQN model which can be summarized in the following steps:

- translation of an FTLQN model description of a system into a specialized AND-OR graph
- the generation of all the possible distinct operational configurations of the system
- the probability calculation of the system being in each of those configurations, from the AND-OR graph
- assignment of a reward rate to each of the operational configurations by applying the Layered Queueing analysis
- the determination of the desired performability measures of the system.

The strategy has been automated in a software tool called FTLQNS, the Fault-Tolerant Layered Queueing Network Solver [15].

In section 2, we briefly describe the LQN modelling concept. In section 3, we present the FTLQN modelling technique. In section 4, we describe and compare two solution strategies for our FTLQN model. In section 5, we provide examples illustrating the use of our solution strategy. In section 6, we extend the FTLQN model to allow the modelling of the systems involving redundant servers which can all be used at the same time with the service load being divided equally among them.

2. The Layered Queueing Network Model

A Layered Queueing Network (LQN) model [12] is a performance modelling framework for client-server-like distributed systems that use a style of synchronous inter-task communication, such as remote procedure calls (RPCs), where the sender is blocked until the receiver replies. The framework can also accommodate asynchronous interactions (with no reply), but they are not considered here. An LQN model can be solved for performance metrics by the tool LQNS (Layered Queueing Network Solver) [16].

An LQN model describes objects called "tasks" which may represent software processes (clients and servers) or hardware devices (e.g. processors, disks). Tasks may send or receive service requests. Each of these service requests are handled by a service handler, known as an "entry", embedded in the task accepting the request. One task may have one or more entries providing various services. In object-oriented terms a task is an object and an entry is a method. Every task is allocated to a processor which is also a server. An entry may have a first phase of service before sending the reply, and an optional second phase after the reply. While in the first phase, the client task making the request is blocked and the server merely continues the thread of control of its client, but in the second phase it has an independent thread of control of its own. A task which originates requests, such as a user interface task or a system control task, is called a "reference task", and has only a second phase. Figure 1 shows a Layered Queueing Network model with one reference task (Task 1), four "active servers" (Tasks 2, 3, 4, 5) (which also make requests), and three pure servers (Processors 1, 2 and 3). In this figure, the numbers in square brackets are the execution times of each entry, by phase, while those in round brackets are the mean numbers of messages from entry to entry, by the phase of the sender.

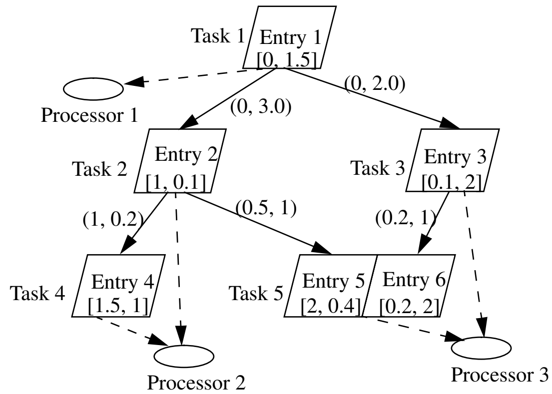

Fig. 1 A Layered Queueing Network model

The present research extends the LQN performance model by adding dependability-related parameters to it, in a way that will support performability analysis for distributed software systems.

3. The Fault Tolerant Layered Queueing Network (FTLQN) Model

An FTLQN model describes distributed systems which employ alternative servers and routing of requests to mask failures. It modifies the LQN model to express the strategy to be used in case of failure and to generate the different configurations in which the system may be fully or partly operational. The layered structure of an FTLQN model describes the dependencies that determine the service failures, based on service dependencies.

In an FTLQN model a service request arc may be replaced by a set of alternative arcs with an order of preference. Figure 2 shows the same system as Figure 1, with Task 4 (now named Task 4a) having a stand-by server, Task 4b. Such a set of arcs is shown attached to a solid black rectangle on Entry 2, with the order of preference indicated by #1, #2. That is if possible, Entry 2 gets this service from Entry 4a, but if Entry 4a is unavailable, it will switch to Entry 4b.

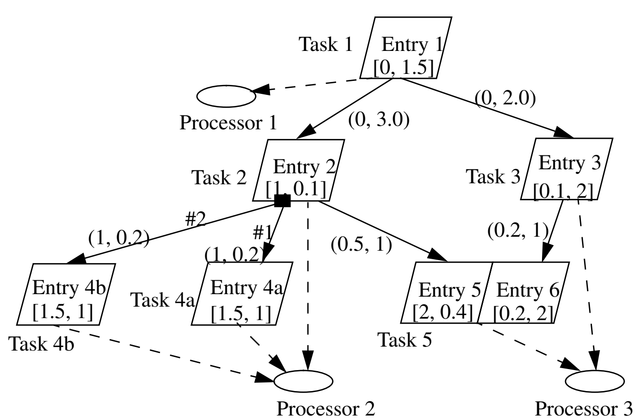

Fig. 2 A Fault Tolerant Layered Queueing Network model

This section describes how failures are propagated from lower layers to upper layers, provides the definition of AND-OR graphs and describes the graph-based representation of an FTLQN model.

A. *Fault Propagation in an FTLQN model*

In an FTLQN model, it is assumed that any of the entities can undergo a failure, which is independent of the failure of other entities in the model (where an entity refers to a software task or a processor). Each entity i has its own *component state* s_i (either 1 or 0) corresponding to its working state or failed state, and is governed by a separate Markov chain with a working state ($s_i = 1$) and a failed state ($s_i = 0$), with rates of failure and repair which gives its probabilities $Prob(s_i = 0)$ and $Prob(s_i = 1)$. Let us consider that there are N entities in the system. The *system configuration state S* is a vector of *component states* of entities, given by $S = \{s_1, s_2, ..., s_N\}$.

Because of the layered structure of an FTLQN model, the failure of a service provided by an entry depends on the failure of the services provided by its processor, its task and by other lower-level entries in the model. We define an entry e of task t to be *working* if

- the task t is working (i.e. $s_t = 1$),
- and the processor p allocated to task t is working (i.e. $s_p = 1$),
- and all the services that the entry e uses are working. A service used by an entry is said to be working if any of the alternative entries providing that service is working.

The entry e is failed otherwise. The dependencies on t and p are represented separately because a crash of task t may be independent of the failure of processor p. We term a working entry to be an *active entry* if it is being used for service by the system.

Let us consider that there are E entries in the FTLQN model. We define the *operational state* s_e* of an entry e, to be either 0 or 1 corresponding to its failed or working state as defined above. Then the *system operational state S** can be defined as a vector of *operational states* of entries given by $S^* = \{s_1^*, s_2^*, ..., s_E^*\}$. Because of the dependencies in the layered structure of the FTLQN model, many *system configuration states* would correspond to a single *system operational state.*

We define a *system activation state S*** to be a set of *system operational states* which have the same *active entries* and thus have the same performance model and the same reward. A state S^{**} includes "*don't cares*" for the entries whose status does not affect the choice of *active entries*. Thus it follows from above that a single *system activation state* would also correspond to a set of *system configuration states*.

The dependencies among the *operational states* of the entries can be computed using an AND-OR graph.

B. *AND-OR Graphs*

AND-OR graphs [14] are used in Artificial Intelligence for the purpose of solving a problem by reducing it to a set of alternative subproblems. The graph of Figure 3 represents a problem A that can be solved by either solving both the subproblems B and C or one of the subproblems D or E. A group of subproblems that must be solved in order to imply a solution to the parent node is indicated by an arc joining the respective edges. By introducing dummy nodes F and G, as in Figure 4, all nodes can be made to be such that their solution requires either all descendents to be solved or only one descendent to be solved. Nodes of the first type are known as AND nodes and those of the latter type OR nodes. Nodes A and G in Figure 4 are OR nodes while node F is an AND node. AND nodes are usually drawn with an arc across all edges leaving the node. Graphs like Figure 4 are called AND-OR graphs.

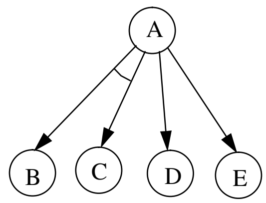

Fig. 3
Graph showing alternative subproblems for problem A

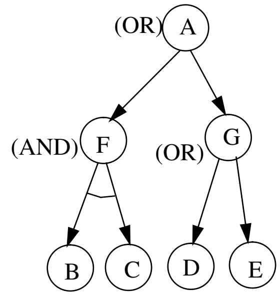

Fig. 4
An AND-OR graph for Fig. 3

In our case, the meaning of a node being "true" is "service is active", rather than the "subproblem is solved".

C. *Graph Representation of an FTLQN model*

The fault dependencies in an FTLQN model can be represented by a prioritized, labelled and directed AND-OR graph termed as a *fault-propagation AND-OR graph*, which we define as a six-tuple $G = (V, E, r, L, priority, \Pi)$ where

1) V is the set of nodes in the graph G. V contains four types of nodes, namely, *entry* node, *service* node, *processor* node, *task* node which are defined as follows:

(i) an *entry* node is an AND node that describes an entry of the FTLQN model. We denote the set of entry nodes by V_{entry}.

(ii) a *service* node is an OR node that describes the preference order of the alternative targets for a requested service. All of its successors represent the alternative targets while its parent represents the entry requesting the service. This node corresponds to the solid black rectangle in the FTLQN model and is represented as a shaded node in G. We denote the set of *service* nodes by $V_{service}$.

(iii) a *processor* node is a leaf node that contains the information about a processor of the FTLQN model. We denote the set of processor nodes by V_{proc}.

(iv) a *task* node is a leaf node that contains the information about a software task of the FTLQN model. We denote the set of task nodes by V_{task}.

V also has one special start node r which is an OR node representing the overall state of the system. Its successors represent the reference entries of the FTLQN model, a reference entry being an entry of a reference task.

2) E is a subset of $V \times V$, the set of edges.

3) $L = V_{proc} \cup V_{task}$ is the set of leaf nodes, which can fail independently.

4) *priority* is a function $priority: E \to W$ that labels a whole number $priority(e)$ to each edge $e = (v, w) \in E$, where $v, w \in V$. If $v \in V_{entry}$, then $priority(e) = 0$ so that all the edges outgoing from an *entry* node would have equal priorities. If $v \in V_{service}$, then $priority(e)$ is the preference order of the target w (lower value for higher preference).

Also if $v = r$, then *priority*(e) = 0 so that all the edges outgoing from r have equal priorities.

5) Π is a function $\Pi : L \rightarrow \{\pi | (\pi \in \Re) \wedge (0 \le \pi \le 1)\}$ that assigns a real number $\Pi(l)$ to each leaf node $l \in L$, representing the probability of node l being in the "working" state. These leaf nodes represent entities that may fail independently.

Figure 5 shows the *fault propagation AND-OR graph* corresponding to the FTLQN model in Figure 2. An edge e is labelled with #*priority*(e) in Figure 5. An edge with priority 0 is not labelled. The node Service-4 is a *service* node that describes the preference order of the alternative targets Entry 4a and Entry 4b for the service requested by Entry 2.

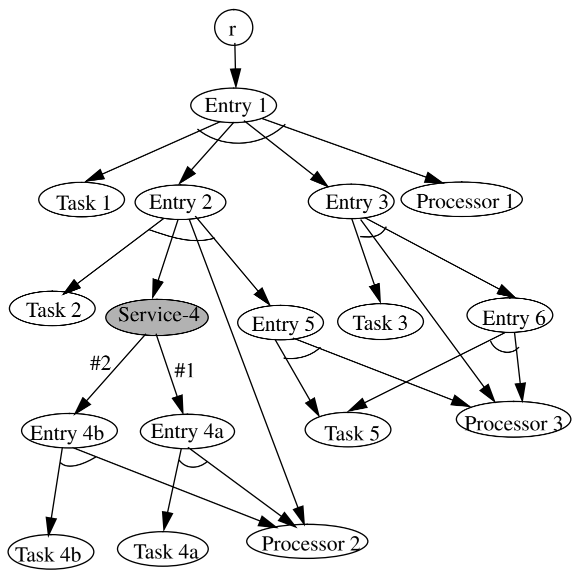

Fig. 5 Fault Propagation AND-OR graph corresponding to Fig 2

The *fault propagation AND-OR graph* is used in the algorithm, described in section 4, to generate the distinct *system activation states* and the probabilities of the system being in those states. We restrict our attention to acyclic graphs in order to avoid circular chain of blocked tasks causing deadlocks.

4. The FTLQN Performability Algorithm

The following algorithm calculates the performability measure of an FTLQN model:

1) Obtain the *fault propagation AND-OR graph*, G, corresponding to the FTLQN model, as described above.

2) Determine all possible distinct *system activation states*, $S_1^{**}, S_2^{**}, ..., S_n^{**}$, from G and calculate the steady-state probability, $Prob(S_i^{**})$, of the system being in state S_i^{**} for $i = 1, 2, \ldots, n$ by one of two approaches described later in this section.

3) Generate an LQN model for each S_i^{**} and apply the Layered Queueing Analysis to obtain its performance measures. Select a performance measure, R_i, and assign it as a reward to S_i^{**}.

4) Compute the expected steady state reward rate of the system given by $\sum_{i=1}^{n} R_i Prob(S_i^{**})$.

Next, we elaborate on the step 2. One way to achieve this step involves the enumeration of all the $2^{|L|}$ *system configuration states* as described below in approach 1. A better way is attempted under approach 2 which avoids such enumeration.

A. *Approach* 1: *Enumeration Approach*

Let $l_1, l_2, ..., l_{|L|}$ denote the $|L|$ entities in the system represented by the leaf nodes in graph, G. If one were to generate the $2^{|L|}$ *system configuration states* of the system, then because of the dependencies in the layered structure of the FTLQN model, many *system configuration states* would correspond to a single *system activation state*. Let $A = \{(a, Prob(a)) \mid a$ is a *system activation state*$\}$.

(a) Initialize A to be empty.

(b) For each *system configuration state* $S_j = \{s_1, s_2, ..., s_{|L|}\}$, where $j = 1, 2, ..., 2^{|L|}$ and $s_i = 0$ or 1,

- find the corresponding *system operational state* S_j^* by traversing G in depth-first-search fashion [17] taking into account the priorities associated with the edges.
- compute $Prob(S_j) = \prod_{i=1}^{|L|} Prob(l_i \text{ being in state } s_i)$,

  where $Prob(l_i$ being in state $s_i) = \Pi(l_i)$ if $s_i = 1$
  $= 1 - \Pi(l_i)$ if $s_i = 0$
- if $\exists (a, Prob(a)) \in A$ such that S_j^* and the *system operational states* in a have the same *active entries*, then $Prob(a) = Prob(a) + Prob(S_j)$ and S_j^* is included in *system activation state a*

  else $A = A \cup \{(a', Prob(a'))\}$ where $a' = \{S_j^*\}$ and $Prob(a') = Prob(S_j)$.

B. *Approach* 2: *State Aggregation Approach*

This approach creates the distinct *system activation states* directly without generating the *system configuration states*. Before describing this approach, we provide some notations and terminologies to be used in the description as follows:

Child(v) denotes the set of children of node v, *i.e. Child*(v) is the set of nodes w, such that $(v, w) \in E$.

Define a *Status Graph* G_i of the *fault propagation AND-OR graph* $G = (V, E, r, L, priority, \Pi)$ to be a subgraph of G with an additional status value associated with its nodes representing their state (failed or working). Thus $G_i = (V_i', E_i, r, L_i', priority, \Pi)$ where

- $V_i' = \{(v, stat(v)) \mid v \in V_i ; V_i \subseteq V ; stat(v) \in \{0, 1\}\}$, *stat(v)* representing the status (failed or working) of the node v
- $E_i \subseteq E$
- $L_i' \subseteq V_i'$ and $L_i' = \{(l, stat(l)) \mid l \in L_i ; L_i \subseteq L ; stat(l) \in \{0, 1\}\}$

We term a *Status Graph* G_s to be an *Activation Graph*, if the following conditions hold:

(a) r is the start node of G_s and $\exists v \in Child(r)$ such that $(v, 1) \in V_s'$

(b) For an *entry* node v:

- if $(v, 1) \in V_s'$, then $\forall(w \in Child(v))$, $(w, 1) \in V_s'$
- if instead, $(v, 0) \in V_s'$, then not all $w \in Child(v)$ have *stat*(w) equal to 1.

(c) For a *service* node v:

- if $(v, 1) \in V_s'$, then $\exists w \in Child(v)$ with *priority*$((v, w)) = m$, such that $(w, 1) \in V_s'$ and $\forall(t \in Child(v))$ with *priority*$((v, t)) < m$, $(t, 0) \in V_s'$.
- if instead, $(v, 0) \in V_s'$, then none of the children w of v has *stat*(w) equal to 1.

Each *activation graph* G_s of G corresponds to a distinct *system activation state*, which is found by associating the status values of the *entry* nodes of the *activation graph* to the corresponding entries in the *system activation state* and associating *don't care* values to all other entries which are not in the *activation graph*. We can therefore determine all possible distinct *system activation states* by generating all the possible *activation graphs* of G. We divide this approach into three phases:

I. *Status Graphs Generation Phase*
II. *Activation Graphs Determination Phase*
III. *Probability Computation Phase*

Phase - I:

This phase generates a set of *status graphs*, $Z' \supseteq Z$, where Z is the set of all possible *activation graphs* of G, by traversing G once. It involves the following steps:

(1) Traverse G in the breadth-first-search fashion [17]. During traversal, process a node v as follows:

If $v = r$,

For each of the combination of status values 0 and 1 for the children of r,
Generate a *status graph* G_i with:

- V_i' consisting of the node r with status 1 (or 0 for all zero combination of status values of its children) and the *i-th* combination of the children of r with their status values as 1 or 0 and
- E_i being the set of edges connecting r to the children of r.

If $v \in V_{entry}$,

In each *status graph* G_i where $(v, 1) \in V_i'$,

If $\exists w \in Child(v)$ such that $(w, 0) \in V_i'$ then

Abort construction of G_i and remove it from the set of *status graphs*

Else

Expand G_i in such a way that:

- V_i' becomes $V_i' \cup \{(w, 1) | (w \in Child(v))\}$ and
- E_i becomes $E_i \bigcup_{w \in Child(v)} (v, w)$.

If $v \in V_{service}$,

In each *status graph* G_i where $(v, 1) \in V_i'$,

Replace G_i by 0 or more expanded *status graphs* $G_{i,k}$, depending on the children w_k of v, $k = 1, 2, ..., m$, where $m = |Child(v)|$

Consider each child w_k of v in turn, for $k = 1, 2, ..., m,$ where w_k denote the child of v with *priority* value k.

If w_k has already been included in G_i with status 0, or
if w_j $(j < k)$ has been included in G_i with status 1, do not generate $G_{i,k}$
Else
Generate $G_{i,k}$ in such a way that:

- $V'_{i,k}$ becomes $V_i' \cup \{(w_j,0) | (j < k)\} \cup \{(w_k,1)\}$ and
- $E_{i,k}$ becomes $E_i \bigcup_{j = 1,2,\ldots k} (v, w_j)$

If $v \in V_{proc} \cup V_{task}$, no action is taken.

Z' is the set of *status graphs* found by these steps. The elements of Z' will henceforth be denoted by G_i.

Phase - II:

This phase removes those *status graphs* generated in Phase-I which are infeasible. A *status graph* G_i is *infeasible* if it contains an inconsistency between the status of a node v (where $(v, 0) \in V_i'$) and the status of its descendants. An inconsistency is detected if the status of the node v determined from the status of its descendants combined with the *fault propagation AND-OR graph*, disagrees with its recorded status (which is "failed"). Since there can be multiple parents of a node, an *infeasible status graph* may get generated in Phase-I. In the algorithm, the infeasibility is actually tested while building the fault tree in Phase-III.

The set of remaining *status graphs* after removing the *infeasible* ones will be Z, the set of *activation graphs*.

Phase - III:

This phase calculates the probability of each of the *activation graphs*, $G_s \in Z$ as follows:

$$Prob(G_s) = \prod_{(v, stat(v)) \in V_s'} Prob(v \text{ being in state } stat(v))$$

$$= \prod_{(l, stat(l)) \in L_s'} Prob(l \text{ in } stat(l)) \cdot \prod_{(v, 0) \notin L_s'} Prob(stat(v) = 0) \ldots (1)$$

$$= \prod_{(l, 1) \in L_s'} \Pi(l) \cdot \prod_{(l, 0) \in L_s'} (1 - \Pi(l)) \cdot \prod_{(v, 0) \notin L_s'} Prob(stat(v) = 0) \ldots (2)$$

Since in the *activation graph* G_s, the *entry* or *service* nodes with status 1 (i.e. $(v, 1) \notin L_s'$) are expanded, the probability calculation does not involve them explicitly; they depend on their children. However for the failed *entry* or *service* nodes (i.e. $(v, 0) \notin L_s'$), the probability calculation must be made separately to give $Prob(stat(v) = 0)$ in (2), and this must be done using the information contained in G. A fault tree is created using both G and G_s to model the simultaneous failure of all these nodes $(v, 0) \notin L_s'$, whose probability is the third product in (2). This is a conventional fault tree (e.g. [18], [19]) so the details of its construction are not given here. The failure probability of the root node of the fault tree is obtained using the Sum-of-Disjoint-Products approach (e.g. [20], [21], [22], [23]).

In the FTLQNS tool, the fault tree calculation for the third product in (2) has been accomplished by using the SHARPE tool [19], [24], [25].

The implementation of Phase - II and III have been merged in the FTLQNS tool in such a way that the feasibility testing of a *status graph* is carried out while trying to build the corresponding fault tree for its probability calculation.

The probability of each of the *activation graphs* thus computed can then be associated with its corresponding *system activation state*.

5. Discussions

A. *Example*

We consider a model of a fault-tolerant client-server system with one reference task as shown in Figure 6 in order to explain our solution algorithm. In this figure, there are five tasks t1-t5 with entries e1-e5 running on the processors p1-p5. The independent failure probabilities for (p1-p5, t1-t5) are in order (0.03, 0.05, 0.07, 0.09, 0.11, 0.1, 0.08, 0.06, 0.04, 0.02).

We translate the FTLQN model in Figure 6 into the corresponding *fault propagation AND-OR graph* as shown in Figure 7. Next, we determine all the possible distinct *System Activation States* (SAS) and the probabilities of the system being in those states from the *fault propagation AND-OR graph* in Fig. 7 as given in Table 1. The number of *System Configuration States* (SCS) corresponding to each SAS is also shown in the table. Notice that there are four *system operational states* in S_1** corresponding to two don't cares. However, each of the SCS being a 10-tuple, there are many more of these SCS in S_1**.

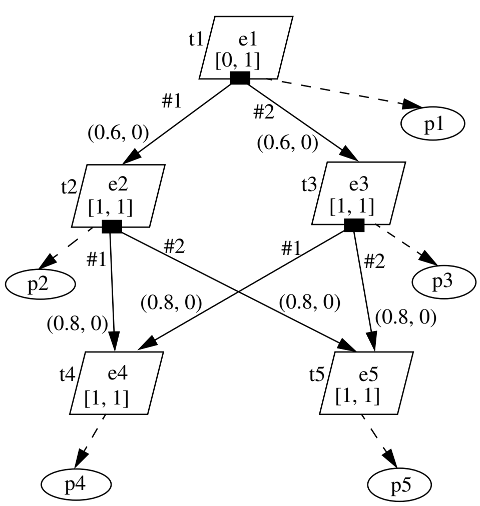

Fig. 6 An FTLQN model of a client-server system with one reference task

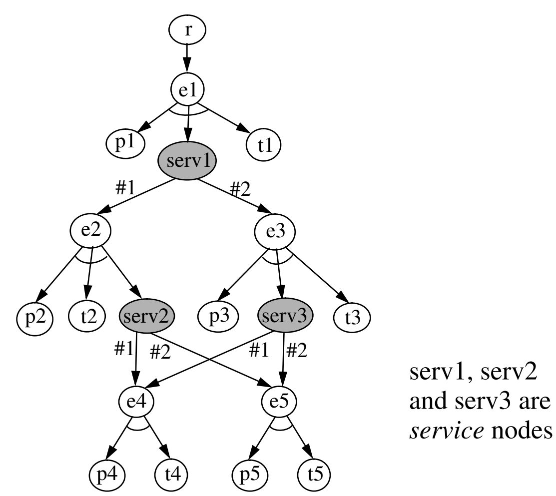

Fig. 7 Fault-propagation AND-OR graph corresponding to Fig. 6

Table 1: System Activation States and their probabilities

SAS S_i**	S_i** = {S_i*} = {[s_1*, s_2*, s_3*, s_4*, s_5*]}. s_1*- s_5* are the *operational states* of entries e1-e5. *d* represents *don't-care*.	No. of SCS in S_i**	*Prob*(S_i**)
S_1**	{ [1, 1, *d*, 1, *d*] }	16	0.6666
S_2**	{ [1, 0, 1, 1, *d*] }	12	0.084
S_3**	{ [1, 1, *d*, 0, 1] }	12	0.0841
S_4**	{ [1, 0, 1, 0, 1] }	9	0.0106
S_5**	System Failed State	1024 - 49 = 975	0.1548

We generate an LQN model for each of the *System Activation States* S_1**, S_2**, S_3** and S_4** and solve those models using LQNS tool [16] for determining their performance measures. Since S_5** corresponds to the fully failed state of the system, we avoid the generation of its corresponding LQN model. In this example, we select *throughput* of the reference task t1 to be the reward which we associate with the corresponding *system activation state* as shown in Table 2.

Table 2: Rewards associated with the System Activation States of Table 1

SAS S_i**	Reward R_i = *throughput* of task t1
S_1**	0.3912
S_2**	0.3912
S_3**	0.3912
S_4**	0.3912
S_5**	0

Finally, from Table 1 and 2, we calculate the expected steady state reward rate of the system as:

$$\sum_{i=1}^{5} R_i Prob(S_i^{**}) = 0.331 \text{ /sec.}$$

B. *Scalability Analysis*

In this section, we attempt to compare the effort of solving using the two approaches of our algorithm by scaling-up a simple client-server system. We do this in order to demonstrate the efficiency of the *State Aggregation Approach* over the *Enumeration Approach*.

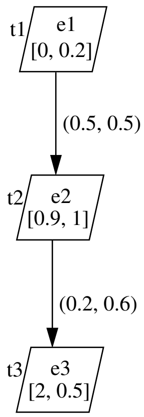

Fig. 8

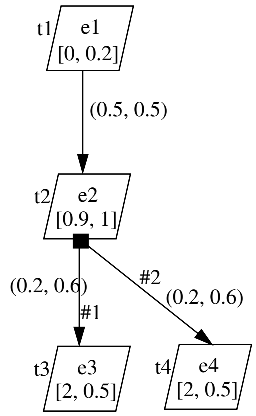

Fig. 9

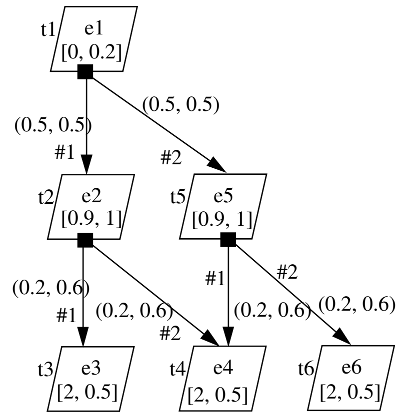

Fig. 10

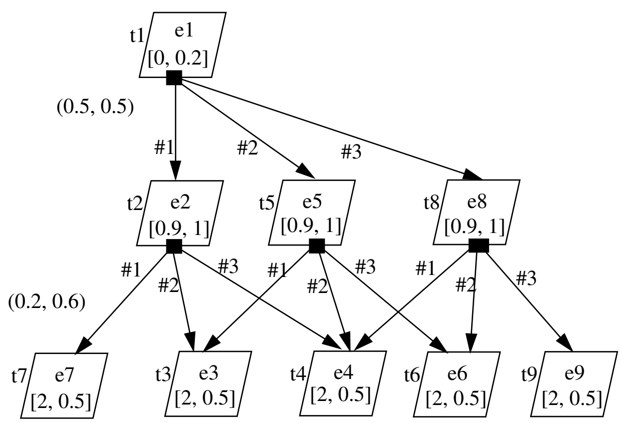

Fig. 11

We consider a client-server system as shown in Figure 8 which consists of three tasks t1, t2, t3 running on their own private processors (not explicitly shown in the figure). We scale-up the system by adding stand-by servers in three stages. In the first stage, we add a single alternative task t4 and its own processor where t4 acts as the stand-by server for the task t3 as shown in Figure 9. In the second stage, we add two more alternative tasks t5, t6 and their own private processors as shown in Figure 10. In the third stage, we scale-up the system further by adding three more tasks t7, t8 and t9 and their own private processors as shown in Figure 11. For simplicity, the independent failure probabilities for all the tasks and the processors in these models has been taken to be 0.1 except the reference task t1 and its processor which have been assumed to be fully reliable.

Table 3: Execution times of System Activation States construction by the Enumeration Approach and the State Aggregation Approach for the Figures 8, 9, 10 and 11

FTLQN Models	No. of System Activation States	No. of System Configuration States	CPU time (in secs) for *Enumeration Approach*	CPU time (in secs) for *State Aggregation Approach* (SAA)	FTLQNS Model Solution time (in secs) including SAA	Mean Throughput Reward (/sec)
Fig. 8	2	64	0.5	0.4	0.6	0.183
Fig. 9	3	256	3.6	1.2	1.6	0.217
Fig. 10	5	4096	902.2	2.9	3.5	0.264
Fig. 11	10	262144	~ 66 hours (only enumerated 15% of states in 11 hrs)	6.8	8.7	0.276
Figure 8 with no failures (perfectly reliable service)						0.278

Next, we compare the performance of the *Enumeration Approach* and the *State Aggregation Approach* of our FTLQN Performability Algorithm in Table 3 by running the four models. We also provide the model solution time using the *State Aggregation Approach* and the mean throughput reward for these models. These results were computed using the FTLQNS tool in which both the approaches are implemented. The execution times shown in the table are in seconds (unless and otherwise stated explicitly), measured on a SUN Sparcstation-20. We observe a marked increase in the CPU time for the *Enumeration Approach* since it has to enumerate a large number of *system configuration states* for a large system. Since the *State Aggregation Approach* avoids such enumeration, the CPU times are much smaller for it. It must also be noted that if we were to formulate a Markov Reward model for Figure 11, whose states are the *system configuration states*, it would have 262144 states similar to the *Enumeration Approach*, and it would take a long time to solve.

If we consider the system in Figure 8 to be a fully reliable system, we obtain its mean throughput (reward) to be 0.278 as shown at the bottom of Table 3. This is an upper bound for the mean throughput reward of the systems with failures. We observe from Table 3 that as the number of stand-by servers are increased, the mean throughput reward increases to 0.276 for Figure 11 which is close to 0.278, the mean throughput reward of Figure 8 with no failures.

6. Extension to the FTLQN model

So far, we have considered a class of systems which assumes that the stand-by servers are not used until needed. In this section, we consider another class of systems where the redundant servers for a service request can all be used at the same time with the load being distributed equally among them. In order to model this class, we next extend the semantics of the FTLQN model. We henceforth term this extension as *load-balanced* FTLQN model.

In a *load-balanced* FTLQN model, a service request arc may be replaced by a set of arcs which can all be used at the same time with the load being equally distributed among them. Figure 12 shows an example of a *load-balanced* FTLQN model. Such a set of arcs is shown attached to a rectangle with thick black border on Entry 2, with the service load written beside it. That is, Entry 2 gets this service from both Entry 4a and Entry 4b with the load being equally divided among them, but if one of them fails, it will switch to the other with the total service load.

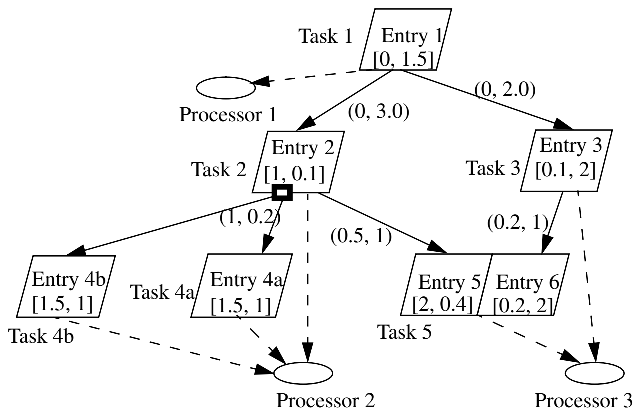

Fig. 12 A *load-balanced* FTLQN model

The graph representation of a *load-balanced* FTLQN model would be the same as described in Section 3(C) except that there would be no priority function associated with its corresponding *fault propagation AND-OR graph*, G. This means G would now become a five tuple, $G = (V, E, r, L, \Pi)$ where V, E, r, L and Π have the same definition as described in Section 3(C) except that the definition of a *service* node now becomes as: "a *service* node is an OR node that describes the redundant targets (having no preference order associated with them) for a requested service". This node corresponds to a rectangle with thick black border in the *load-balanced* FTLQN model. We next describe the performability algorithm for the *load-balanced* FTLQN model.

The algorithm for determining the steady-state performability of a *load-balanced* FTLQN model would be the same as described in Section 4 except that the approaches for achieving step 2 need to be modified. Since the *State Aggregation Approach* is more efficient than the *Enumeration Approach*, we present only the modification of the *State Aggregation Approach* for achieving step 2. The modification is required in Phase-I (i.e. *Status Graphs Generation Phase*) in the processing of a node $v \in V_{service}$ during the graph traversal which is given as follows:

If $v \in V_{service}$,

In each *status graph* G_i where $(v, 1) \in V_i'$,

Replace G_i by 0 or more expanded *status graphs* $G_{i,k}$, depending on the *k*-th combination of status values 0 and 1 for the children of v (except all zero combination) as follows:

If $\exists w \in Child(v)$ such that w is present in V_i' with status 0(1) although its status value in the *k*-th combination is 1(0), then do not generate $G_{i,k}$

Else

Generate $G_{i,k}$ in such a way that:

- $V'_{i,k}$ becomes $V_i' \cup \{children\ of\ v\ with\ their\ status\ values\ as\ 1\ or\ 0\ corresponding\ to\ k\text{-}th\ combination\}$ and
- $E_{i,k}$ becomes $E_i \bigcup_{w \in Child(v)} (v, w)$

Rest of the algorithm remains the same as illustrated before in Section 4.

7. Conclusion

In this paper, we have developed a new performability modelling framework known as Fault-Tolerant Layered Queuing Network for the evaluation of performability of fault-tolerant client-server systems whose fault-tolerance is achieved by incorporating stand-by servers and a mechanism to re-direct the service requests at the time of failure. This modelling technique allows us to model large and complex fault-tolerant systems. The advantage of our model over the well known Markov Reward model lies in avoiding the generation, storage and solution of large Markov models. We have developed a solution strategy for our model that involves AND-OR graph theoretic technique, sum-of-disjoint-products approach and the layered queuing network analysis for the computation of steady-state performability. As part of our solution strategy, we have devised two different approaches, *Enumeration Approach* and *State Aggregation Approach*, for the generation of all the possible distinct operational configurations of the system and their probability calculations and have demonstrated the efficiency of the *State Aggregation Approach* over the *Enumeration Approach* for scaled systems using an example. Both of these approaches have been implemented in the FTLQNS tool. We have also extended the FTLQN modelling concept in order to model those systems which involve the usage of redundant servers that can all be used at the same time with the service load being distributed equally among them. We have further provided the performability algorithm for this extended model.

Our work is similar to [10] in the way that both of them first determine the possible system configurations by using a dependability model and then assign a reward rate to each of those configurations by solving a performance model corresponding to that configuration. However, in [10], the question of combining configurations that give the same performance model is not addressed, whereas we have been able to determine a minimal set of system configurations each corresponding to a distinct performance model.

References

[1] J. F. Meyer, “On Evaluating the Performability of Degradable Computing Systems”, IEEE Trans. on Computers, vol. 29, no. 8, Aug 1980, pp. 720-731.

[2] J. F. Meyer, “Closed-Form Solutions of Performability, IEEE Trans. on Computers, vol. 31, no. 7, 1982, pp. 648-657.

[3] R. M. Smith, K. S. Trivedi and A. V. Ramesh, “Performability Analysis, Measures, an Algorithm and a case study”, IEEE Trans. on Computers, vol. 37, no. 4, 1988, pp. 406-417.

[4] A. L. Reibman, R. Smith and K. S. Trivedi, “Markov and Markov Reward models transient analysis: An overview of Numerical Approaches”, European Journal of Operations Research, vol. 40, 1989, pp. 257-267.

[5] G. Ciardo, A. Blakemore, P. F. Jr. Chimento, J. K. Muppala and K. S. Trivedi, “Automated Generation and Analysis of Markov Reward Models using Stochastic Reward Nets”, in Linear Algebra, Markov Chains and Queueing Models, Carl Meyer and Robert Plemmons (eds.), Springer-Verlag, 1992.

[6] K. S. Trivedi, J. K. Muppala, S. P. Woolet and B. R. Haverkort, “Composite Performance and Dependability Analysis”, Performance Evaluation 14(1992), pp. 197-215.

[7] B. R. Haverkort and K. S. Trivedi, “Specification and Generation of Markov Reward Models”, Discrete-Event Dynamic Systems: Theory and Applications, vol. 3, 1993, pp. 219-247.

[8] B. R. Haverkort and I. G. Niemegeers, “Performability Modelling using Dynamic Queueing Networks”, ACM Performance Evaluation Review 17(1), 1989, pp. 225.

[9] B. R. Haverkort and I. G. Niemegeers, “On the Mutual Performance-Dependability Influence in Dynamic Queueing Networks”, in Proc. International Workshop on Performability modelling of Computer and Communication Systems (1991), pp. 33-40.

[10] B. R. Haverkort, I. G. Niemegeers and P. V. van Zanten, “DyQNtool - A Performability Modelling Tool Based on the Dynamic Queueing Network Concept”, in Computer Performance Evaluation, G. Balbo and G. Serazzi (Eds.), Elsevier Science Publishers B. V. , 1992, pp. 181-195.

[11] C. M. Woodside, “Throughput Calculation for Basic Stochastic Rendezvous Networks”, Performance Evaluation, vol. 9, no. 2, April 1989, pp. 143-160.

[12] C. M. Woodside, J. E. Neilson, D. C. Petriu and S. Majumdar, “The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-like Distributed Software”, IEEE Trans. on Computers, vol. 44, no. 1, January 1995, pp. 20-34.

[13] J. A. Rolia and K. C. Sevcik, “The Method of Layers”, IEEE Trans. on Software Engineering, vol. 21, no. 8, August 1995, pp. 689-700.

[14] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press, Rockville, MD, 1978.

[15] O. Das, “Performance and Dependability Analysis of Fault-Tolerant Layered Distributed Systems”, Master’s Thesis, Department of Systems and Computer Engineering, Carleton University, 1998.

[16] R. G. Franks, A. Hubbard, S. Majumdar, J. E. Neilson, D. C. Petriu, J. A. Rolia and C. M. Woodside, “A Toolset for Performance Engineering and Software Design of Client-Server Systems”, Performance Evaluation, vol. 24, no. 1-2, 1995, pp. 117-136.

[17] N. J. Nilsson, *Principles of Artificial Intelligence*, Tioga Publishing Company, Palo Alto, California, 1980.

[18] J. B. Dugan, “Fault Trees and Imperfect Coverage”, IEEE Transactions on Reliability, vol. 38, no. 2, June 1989, pp. 177-185.

[19] R. A. Sahner, K. S. Trivedi and A. Puliafito (1996), *Performance and Reliability Analysis of Computer Systems: An Example-Based Approach Using the SHARPE Software Package,* Kluwer Academic Publishers, Boston.

[20] J. A. Abraham, “An Improved Algorithm for Network Reliability”, IEEE Transactions on Reliability, vol. R-28, April 1979, pp. 58-61.

[21] S. Hariri and C. S. Raghavendra, “SYREL: A Symbolic Reliability Algorithm based on Path Cutset Methods”, IEEE Transactions on Computers, vol. C-36, Oct. 1987, pp. 1224-1232.

[22] M. Veeraghavan and K. S. Trivedi, “An Improved Algorithm for Symbolic Reliability Analysis”, IEEE Transactions on Reliability, R-40(3), August 1991, pp. 347-358.

[23] S. Rai, M. Veeraghavan and K. S. Trivedi, “A Survey of Efficient Reliability Computation using Disjoint Products Approach”, Networks, Vol. 25, 1995, pp. 147-163.

[24] R. A. Sahner and K. S. Trivedi, “Reliability Modeling using SHARPE”, IEEE Transactions on Reliability, June 1987, pp. 186-193.

[25] R. A. Sahner and K. S. Trivedi, “Performance and Reliability Analysis using Directed Acyclic Graphs”, IEEE Trans. on Software Engineering, vol. 13, no. 10, October 1987, pp. 1105-1114.