Computing the Performability of Layered Distributed Systems with a Management Architecture

Olivia Das and C. Murray Woodside

Dept. of Systems and Computer Engineering
Carleton University, Ottawa, Ontario, Canada K1S 5B6

email: {odas, cmw}@sce.carleton.ca

ABSTRACT

This paper analyzes the performability of client-server applications that use a separate fault management architecture for monitoring and controlling of the status of the application software and hardware. The analysis considers the impact of the management components and connections, and their reliability, on performability. The approach combines minpath algorithms, Layered Queueing analysis and non-coherent fault tree analysis techniques for efficient computation of expected reward rate of the application.

Keywords

System Fault-tolerance, Performability, Distributed Systems, Non-coherent fault trees, Layered Queueing Networks.

1. INTRODUCTION

This work considers efficient evaluation of the effectiveness of fault management architectures in distributed systems. It assumes there is a fault management subsystem (for example [21, 16, 14]) to detect failures, to isolate faults and to coordinate reconfiguration of services after a failure or repair event. The architecture of the fault management subsystem is critical to its capability to respond, by obtaining knowledge of failures and by applying correctional measures. This work considers a broad class of fault management architectures called MAMA, applied to a layered distributed service system that has a layered queueing performance model. It is important to include layered service effects because most distributed systems have some element of layering (which in this work simply means subsystems in service layers, that offer services to other subsystems in higher layers), which means that the effects of a server failure are felt through the inability of its clients to obtain service. Thus failures are propagated by the layered dependencies.

In [4] and [5], the failure dependencies induced by a layered service structure were exploited to create an efficient calculation of failure probabilities, and of combined performance and reliability measures (called performability[19] measures). The model, called Fault-tolerant Layered Queueing networks or FTLQN, assumed perfect knowledge of the failure state. It can be seen as an extension of the Dynamic Queueing Analysis approach of Haverkort et.al. [12, 13] to include layered queueing analysis in place of standard flat queueing models, and the role of layered service dependencies in propagating failures. The FTLQN calculations are highly scalable, essentially because a large number of possible failure states often lead to just a few distinct performance models to be solved.

The great majority of work on reliability modeling is concerned with hardware component failures (e.g., the book by Wallace [1]). Software reliability analysis is largely concerned with analyzing test results to predict failure rates (e.g. [18]). For an extensive survey on software reliability models, the reader is referred to [20]. Most of these models treat the software as a whole considering only its interactions with the outside environment without looking into its internal structure, i.e. its architecture. Detecting faults and working around them in a running system, which is the concern of a fault management subsystem, are described in [21] for a grid system, and in [14] and [16] for cluster systems. These describe prototype systems and measurements. [9] describes a Markov model for software failures based on the system given in [14]. A model for software failures based on the application software architecture was described in [10, 11], however it did not consider a management architecture, or its related failures. Sun *et. al.* investigated the impact of a single centralized fault management server on a highly available communication system in [22] but did not include more general management architectures. As far as we know, [6] was the first attempt to model the effect of management subsystem failures on system reliability.

In [6], the MAMA management architecture was introduced into the FTLQN model and analyzed by enumerating the system failure states. The present paper describes a much more efficient algorithm to determine the configurations and their probabilities. It determines the connectivity between a management element that detects a failure and a point of reconfiguration, by applying Minpath algorithms to a Knowledge Propagation Graph. This is then used in a Fault Propagation Graph, which is an AND-OR-NOT graph that also captures the service dependencies, to give configurations and probabilities.

A kind of non-coherence is introduced into the Fault Propagation Graph because the occurrence of a failure is not enough to trigger a reconfiguration; the knowledge of the failure must reach the decision-making elements. This makes it an AND-OR-NOT graph, instead of an AND-OR graph as used in [5] (which, we remember, assumed perfect knowledge). To compute the probabilities of the operational configurations, we use non-coherent fault trees [7, 17]. They can be constructed so as to model both standby redundancy (redirect requests to a standby server) and load-balancing redundancy (redistribute the load among the remaining operational servers).

The contribution of this work is an efficient algorithm for the probabilities of the operational configurations. To give a performability measure, the following steps were used:

- find the different operational configurations of the system from the non-coherent fault propagation graph
- compute the probability of each operational configuration (using a non-coherent fault tree)
- determine the reward rate for each operational configuration (using a Layered Queueing model for each one)
- combine the probabilities and the rewards to determine the steady-state performability.

The computational tool Dependable-LQNS developed for the calculations uses the Aralia software for non-coherent fault trees [7] and the Layered Queueing Network Solver (LQNS) [8] for the performance calculations for each operational configuration.

Sections 2 and 3 describe the models for the application and the management subsystem. studied in this paper, and section 4 defines a model for knowledge propagation. Sections 5 present the non-coherent fault propagation graph and the performability computation. Experience to evaluate its effectiveness is given in Section 7.

2. LAYERED DISTRIBUTED SYSTEMS DESCRIPTION

Distributed systems with clients and servers are typically constructed with a layered architecture. Client-server architectures are popular for distributed systems such as information processing systems, web applications and database systems. Compared to centralized computing they enhance the system usability, scalability and availability. In these layered applications, failure of a processor or a failure of a process in one layer can cause many other processes that depend on its services to fail, unless they can detect the failure and reconfigure to use a redundant process. This section describes the FTLQN model introduced in [4] to capture the layered operational or failure dependencies among various services.

Figure 1 illustrates an FTLQN model with an example of a layered console application. There are six *tasks* (concurrent operating system processes, represented as rectangles), “Console”, “Application Server”, “Console Server”, “Database-A”, “Database-B” (backup of “Database-A”) and “Data Server”. Each task runs on a processor represented by a dotted ellipse. The task “Application Server” creates a report which involves reading from the database and also requesting some kind of application data from the “Data Server”. Tasks have one or more *entries* which are service handlers (for example, the task Data Server has entries “Get Application Data” and “Get Console Data”), which in turn make service requests (represented by rounded rectangles). A service request has a single target server if there is no server redundancy, or a set of redundant target servers with a policy to decide which one to use. Figure 1 shows the service “dbService” requested by the entry “Create Report” for reading from the database. It has a primary-standby policy and the priority of the target servers are labelled “#n” on the arcs going to the servers. In Figure 1 all the service requests are synchronous so that the sender is blocked until the receiver replies. Asynchronous requests are also possible. The workload is generated by tasks like UI which cycle and create requests, and they are called *reference tasks*. The model is restricted to being acyclic in order to avoid cycles of mutual waiting that may lead to deadlock.

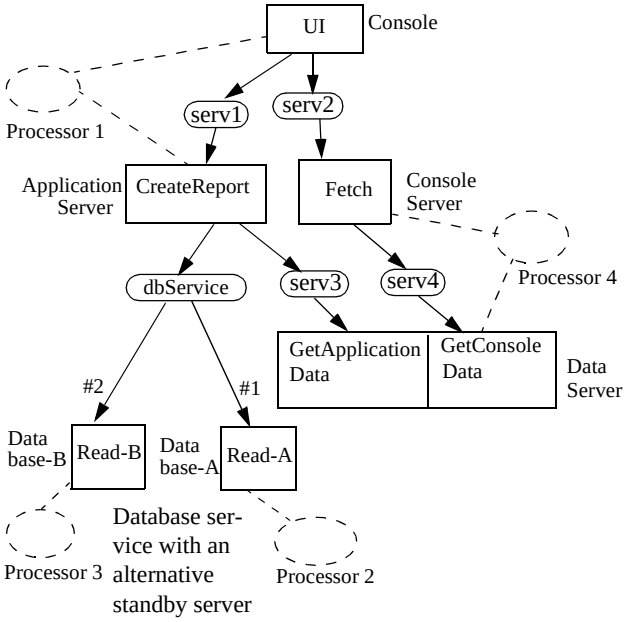


Figure 1. A Fault Tolerant Layered Queueing Network model

FTLQN is a dependability extension of Layered Queueing Networks (LQN) [8], which is a pure performance model for layered applications. The performance parameters are the mean CPU demand of each entry and the mean number of requests for each interaction. The availability related parameters are the probabilities of being in a failed state for each component (either a task or a processor). The performance measures may be defined for components (e.g. utilization) or for requests (e.g. delay).

The FTLQN model shows the policy for redundant servers but the decision about where to forward the request is made by the fault management sub-system (not visible in this model) based on its knowledge of the state of the application components. This resolution of requests gives different *operational configurations* of the application. Each one has an ordinary LQN model which can be solved by the Layered Queueing Network Solver LQNS [8], based on extended queueing approximations.

3. FAULT MANAGEMENT SUBSYSTEM DESCRIPTION

The fault management subsystem controls and monitors the health of a distributed application. It performs activities that include failure detection, fault isolation, and automatic recovery from failures by reconfiguring to use redundant application components.

A common architecture for management subsystems is the manager-agent model [15], which is used for example in Tivoli application management systems [23]. In this model, agents mediate between application components and managers. Each application processor usually runs an agent that monitors all the application tasks running on that processor and also monitors the operating system's health status. An agent is responsible for sending events to the manager, relaying data and command requests from the manager to the application tasks, collecting responses and forwarding them to the requester. There are one or more managers which are responsible for collecting status information from the agents (using a management protocol such as SNMP or JMX), making decisions and issuing notifications for reconfiguration.

A managed application task has a set of responsibilities that include exposing an appropriate interface to the management system, sending heartbeat messages to the local agent, or responding to requests from an agent or a manager. Reconfiguration is handled by an agent which causes its managed task to retarget or forward its requests to appropriate target server.

Failures of application components are detected by timeouts in waiting for heartbeats, or for responses to pings or polls. Heartbeat messages from an application task can be generated by a special heartbeat interrupt service routine which sends a message to the local agent every time an interrupt occurs, as long as the task has not crashed. A component that cannot initiate heartbeat messages (e.g a processor), may be able to respond to messages such as pings or polling messages from an agent or manager. In either case the agent or manager listens for a message and assumes failure if the delay is too long.

3.1 Abstract Management Model

Figure 2 illustrates the manager-agent model described above in an abstract manner using UML class diagram notation [2]. It presents the model at two levels. In the first level (see Figure 2(a)), the manager tasks are shown to manage the application tasks directly while in the second level (see Figure 2(b)), agents come into play mediating between the managers and the application tasks.

This abstract management model is described using the MAMA (***M**odel for **A**vailability **M**anagement **A**rchitecture*) notation described in [6]. It has four classes of components: application tasks, agent tasks, manager tasks and the processors hosting them. They are connected using three different classes of connector:

- Alive-watch connector: This connector is used between two components in cases where the destination component would like to be able to detect whether the source component is alive or not. This may be achieved by periodic polls or "i-am-alive" messages. Usually, the source of this connector are the manageable application components.
- Status-watch connector: In cases where a destination component would like to know about the liveness of the source component and also wants to collect failure data about other components in the system that has been gathered by the source component, this connector is used. An example would be a connector from an agent task to a manager task.
- Notify connector: This connector is used for cases where the source component would like to send or forward reconfiguration commands to its sub-ordinate destination component (for example, a manager sending commands to an agent or an agent forwarding a command to an application task) or conveying management information to its peer (for example, a

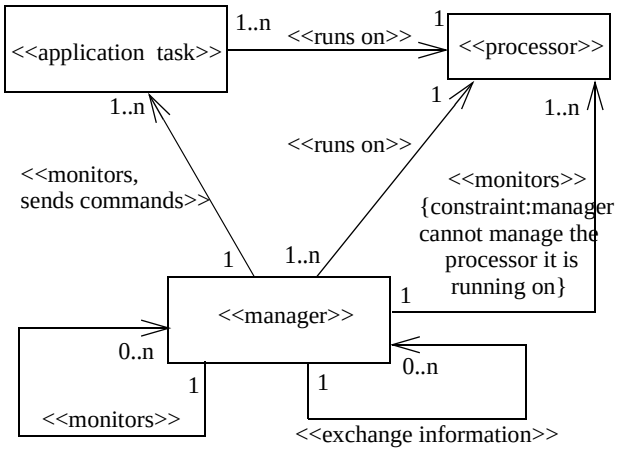


Figure 2(a) Class diagram showing relationship between management components

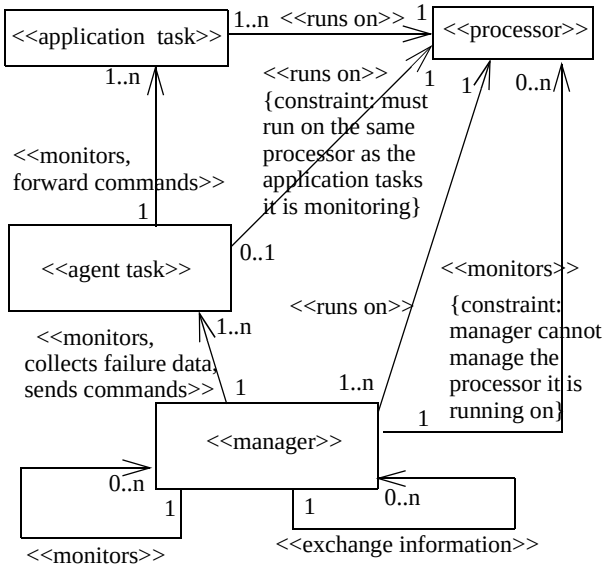


Figure 2(b) An agent component mediating between a manager and an application task

Figure 2. The manager-agent model

manager sending information about the domain it manages to another manager).

Cycles may occur in a MAMA model; we assume that the flow of information is managed in a way so as not to cycle or ping-pong. It is also assumed that if a task watches a remote task, then it also watches the processor executing the remote task in order to differentiate between a task failure and a processor failure.

Figure 3 shows the graphical notation used in this work for MAMA components and connectors. Figure 4 shows a MAMA model for a centralized management architecture for the application of Figure 1, with agents ag1-ag4 running on the processors p1-p4. m1 is the single manager task that indirectly manages all the application tasks through the agents and directly monitors the processors.

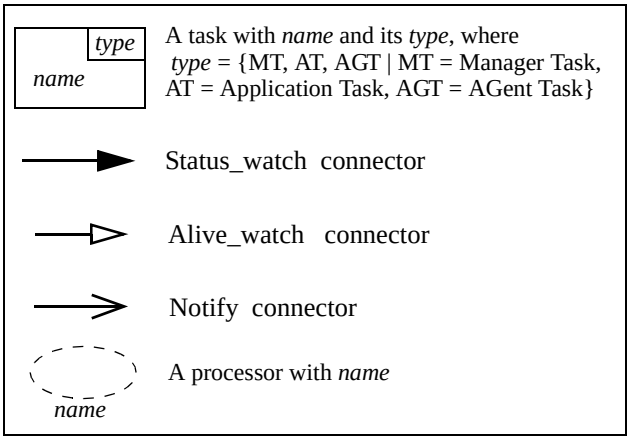


Figure 3. Legend used in this work for MAMA model

The next section describes how the failure information propagates in the management architecture from the source of failure to the point where reconfigurations are executed.

4. FAULT MANAGEMENT INFORMATION PROPAGATION MODELING

The responsibility of the fault management system is to detect component failures, reason about the global system state, and to issue reconfiguration commands to the relevant retargettable application tasks. Successful reconfiguration depends on the connectivity (through the MAMA model) between the source of failure and the point of retargetting services, at the instant of failure. The connectivity is determined by applying minpath algorithms to a derived graph called the Knowledge Propagation Graph for the MAMA architecture.

The Knowledge Propagation Graph has an arc for each component and connector in the MAMA architecture, and vertices representing their points of connection. It is created as follows.

Let us denote iv_i and tv_i to be the initial and terminal vertices respectively of arc i.

For each component i in the MAMA model, the Knowledge Propagation Graph K has

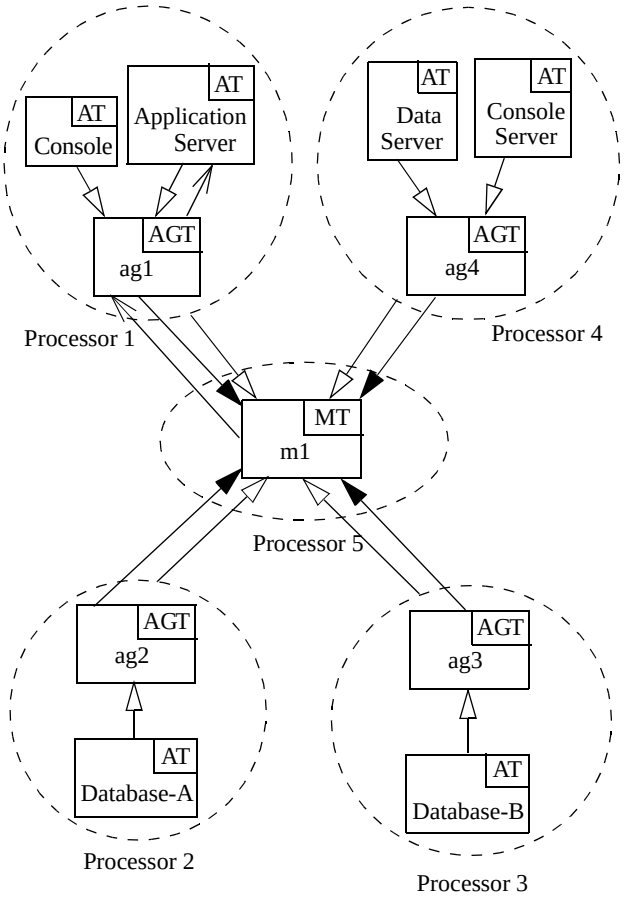


Figure 4. MAMA model for a centralized management architecture for layered application of Figure 1

- two vertices iv_i, tv_i
- a directed arc $i = (iv_i, tv_i)$, of type *component*.

For each connector (i, j) in the MAMA model, K has

- a directed arc w from tv_i to iv_j.
- the type of the arc w is set equal to the type of the connector (i, j). That is, it is one of {*alive-watch, status-watch, notify*}.

Figure 5 shows the knowledge propagation graph corresponding to the MAMA model in Figure 4.

Let o_i denote the state of component i, with

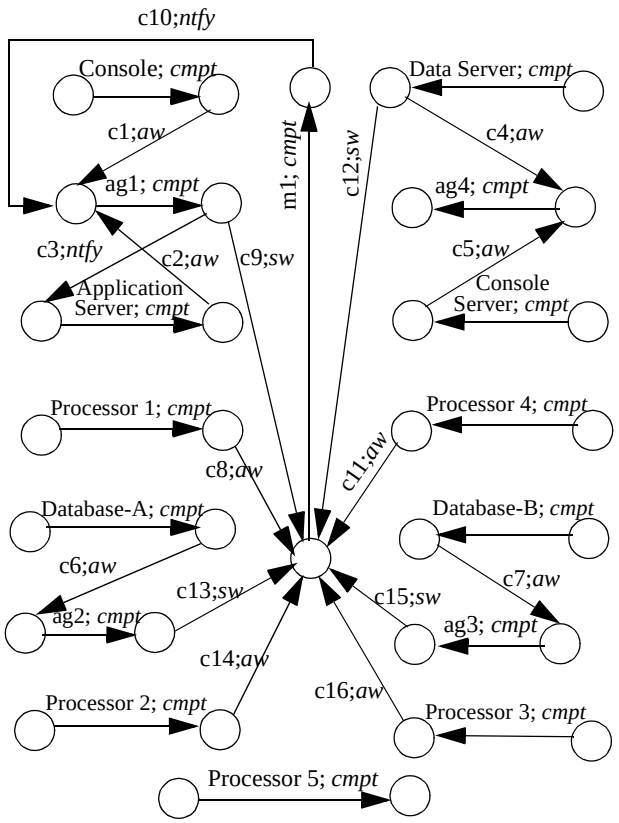
$o_i = 1$ if component i is in working state

$= 0$ if component i is in failed state.

The knowledge propagation graph is used for computing the binary-valued connectivity function, $know_{c,t}$, between two components c and t as follows:

$$know_{c,t} = \bigvee_{q=1}^{m} \left(\bigwedge_{j \in P_q^+} o_j \right)$$

where m is the total number of minpaths $(P_1, P_2, ..., P_m)$ from tv_c



Each edge corresponding to a component is labelled by its name and type as name; *cmpt*.
Abbreviations are: *cmpt = component*; *ntfy = notify*; *aw = alive-watch*; *sw = status-watch*

Figure 5. The Knowledge Propagation graph corresponding to the MAMA model in Figure 4.

to tv_t. A *minpath* P_q is a minimal set of arcs of graph K such that when all the arcs in P_q are operational, then vertices tv_c and tv_t are connected; vertices tv_c and tv_t are disconnected for every proper subset of P_q. A minpath P_q from tv_c to tv_t is obtained from K when c represents a task, or from the reduced graph [K - {*arcs representing task* t_j (*contained in processor* c) *in* K}] when c represents a processor. The analysis can use any standard minpaths algorithm (e.g. [3]), taking into account that the first arc in the path must be of type *alive-watch* or *status-watch* and rest of the arcs should be of type *component*, *status-watch* or *notify.* Define P_q^+ as an augmented minpath obtained from P_q as:

$$P_q^+ = P_q \cup \left\{ \bigcup_{t_j \in P_q} (arc\ p |\ p\ is\ processor\ of\ task\ t_j) \right\} \cup p_c$$

if c is a task, or

$$P_q^+ = P_q \cup \left\{ \bigcup_{t_j \in P_q} (arc\ p |\ p\ is\ processor\ of\ task\ t_j) \right\}$$

if c is a processor.

The knowledge propagation graph is used in the next section while modeling the operational dependencies of FTLQN model.

5. OPERATIONAL SERVICE DEPENDENCIES MODELING

The operational dependencies describe how an operation depends on its processor and on layered services, and on the use of a "working" alternative for a failed service. Earlier analysis, based on perfect knowledge of failures, used an AND-OR graph to model the dependencies [5]. In the present model a "working" system state may depend (through reconfiguration) on the fact that some system element is in a "not working" state (e.g. a backup configuration of an application might corresponds to a primary element being "not working" and the management system has knowledge about the failure and has reconfigured the application to use the backup). This requires an AND-OR-NOT graph, like the *Non-coherent fault-propagation graph* introduced here. It represents the operational dependencies among the entries in the FTLQN model and also takes into account the effect of failures of the management and the agent tasks on the operation of the entries. It combines the management connectivity informations (computed from the knowledge propagation graph) with the structural information extracted from the FTLQN model.

The fault propagation graph is an AND-OR-NOT graph that has a root node *root*, and a node for each processor, task, entry, and service. Some dummy nodes are also added to this graph to account for the knowledge informations. In this graph, edges connecting a node to a group of lower nodes represent an OR function of the values at the lower nodes. If the edges have an arc across them, they represent an AND function. A filled dummy node connecting to one lower node is a NOT function of that node.

Figure 6 shows the fault propagation graph for the model given in Figure 1 and Figure 5. In this figure, the *root* is (by convention) an OR node with the reference entry node "UI" as its only child. An *entry node* is an AND node representing an entry *e* of the FTLQN model. An entry is working if its task is working AND its processor is working AND all the services it uses are working. This implies that the children of an entry node are: node for *e*'s task, the node for *e*'s processor and the nodes corresponding to the services that entry *e* uses. For example, the entry node "UI" is an AND node with the service nodes "serv1" and "serv2", the task node "Console", and the processor node "Processor1" as its children. A *service node* is an OR node representing a service *s* of the FTLQN model. If the service *s* has a single target entry, then the node corresponding to that target entry is the only child of the service node, e.g. the service node "serv1" has only child "CreateReport". Otherwise, if *s* has multiple redundant target entries, then the children of the corresponding service node are dummy nodes which represent the operational logic of the target entries. For example, the service node "dbService" has two

dummy nodes, “d1” and “d2”, as its children. “d1” represents primary branch is working and “d2” represents backup branch is working.

The primary branch, represented by “d1”, is working if

- the entry “Read-A” is working
- AND the task “Application Server” which requires “dbService” has the knowledge about the working status of “Read-A”. Let the dummy node “d3” represents this knowledge information. In order to know that “Read-A” is working (i.e. “d3” is true), the task “Application Server” has to know that the task “Database-A” is working (represented by dummy node “d4”) AND also the processor “Processor2” is working (represented by dummy node “d5”). The node “d4” requires “Database-A” to be working AND there is a connectivity from “Database-A” to “Application Server”, represented by $know_{Database\text{-}A,ApplicationServer}$. A *know* node represents alternative minpaths between two components and thus is an OR node with each child representing a single minpath. For clarity, these know nodes are not expanded in this figure.

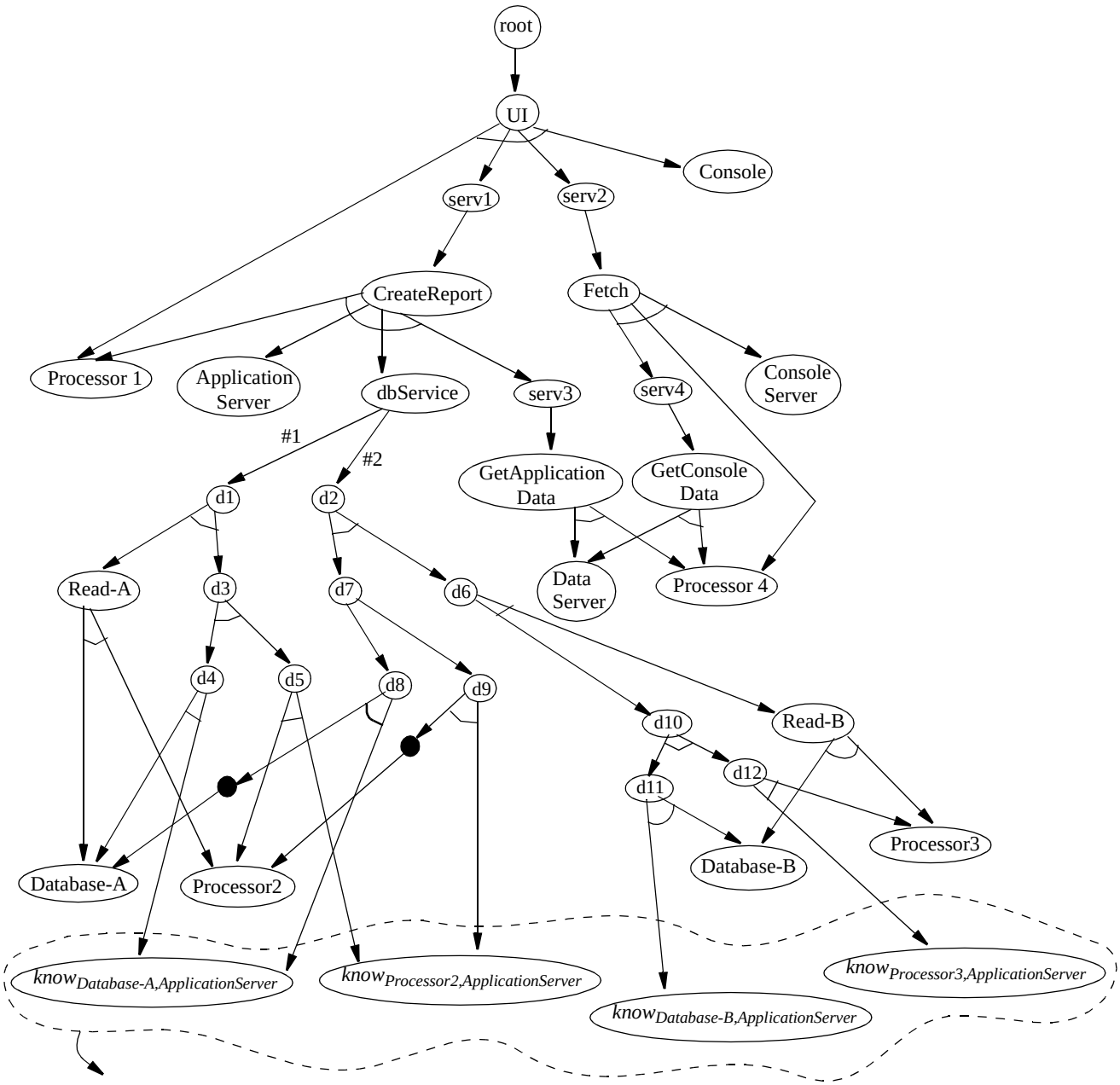


Figure 6. The Non-Coherent Fault-propagation graph corresponding to the FTLQN model of Figure 1 and Knowledge Propagation Graph of Figure 5. $know_{c,t}$ functions are not expanded here for clarity. Each $know_{c,t}$ would be an OR node with AND children (one child for each minpath).

The backup branch, represented by "d2", is working if

- the entry "Read-B" is working and the task "Application Server" knows about it. Let the dummy node "d6" represents this fact. The expansion of "d6" is very similar to the expansion logic of "d1" and is not elaborated here.
- AND the entry "Read-A" is failed and the task "Application Server" knows about the failure. This fact is represented by dummy node "d7". In order to know that "Read-A" is failed (i.e. node "d7" is true), the task "Application Server" has to know that either the task "Database-A" is failed (represented by dummy node "d8") OR the processor "Processor2" is failed (represented by dummy node "d9"). The node "d8" requires "Database-A" to be failed AND there is a connectivity from "Database-A" to "Application Server", represented by $know_{Database\text{-}A,ApplicationServer}$. Similar argument holds for node "d9".

This graph is used to determine the set of distinct operational configurations of the FTLQN model, and their probabilities. From the root down, all distinct sets of alternative working possibilities are created by a breadth-first search. Once the graph is all searched, a complete alternative corresponds to an operational configuration. A full description of the process was given for AND-OR graphs in [5]. The process with NOT functions is almost identical. In practice, nodes that represent the management components are not explored since they are not part of a configuration (although they affect the probabilities of achieving a configuration).

6. PERFORMABILITY COMPUTATION

This section describes an algorithm to compute the mean of any performance measure, averaged over failure states and the probability that a system has failed.

It has following steps:

Step(1):Obtain the knowledge propagation graph *K* corresponding to the specified MAMA model (i.e. the fault management architecture), as described in Section 4.

Step(2): Obtain the non-coherent fault propagation graph *G* using the layered FTLQN model (i.e. the layered application) and the knowledge propagation graph *K*, as described in Section 5.

Step(3): Using the fault propagation graph, determine the set, *Z*, of all the distinct operational configurations C_i of the FTLQN model. A distinct operational configuration corresponds to a distinct choice of alternatives (for a primary-backup policy) or of different combinations of redundant servers (for a load-balanced policy).

Step(4): Compute probability, $Prob(C_i)$, of the system being in each such configuration C_i using non-coherent fault tree analysis described in sub-section 6.1.

Step(5): For each $C_i \in Z$, generate an ordinary Layered Queueing Network model and solve it [8]. From the performance measures assign a reward R_i to configuration C_i.

Step(6): Compute the expected reward rate of the system as

$$R = \sum_{C_i \in Z} R_i Prob(C_i) \ .$$

Next, we elaborate on Step (4).

6.1 Step (4) - Computation of Configuration Probabilities by State Aggregation Approach using Non-Coherent Fault Trees:

This approach computes the probabilities of the distinct operational configurations directly without exhaustively generating all the failure states of the system. Using the graph *G*, it generates a non-coherent fault tree corresponding to each configuration (that has already been obtained in Step(3)) and computes the probability for that configuration using Aralia tool [7]. The fault tree represents a logic function which is true for all states which give the stated configuration, and failed for all other states. Thus, it expresses the conditions under which the configuration occurs. The non-coherent fault tree solution computes the probability of this configuration, from the state probabilities.

Figure 7 provides an example of the non-coherent fault tree generated for a configuration *"dbService is using the backup Server Database-B"* of the FTLQN model in Figure 1. For this configuration to occur, the entries "UI", "CreateReport", "Fetch", "Read-B", "GetApplicationData" and "GetConsoleData" have to be working. This implies that all the components "Console", "Processor1", "ApplicationServer", "ConsoleServer", "DataServer", "Processor4", "Database-B" and "Processor3" have to be working. These eight components correspond to the eight basic events that are direct child of the top AND gate of the fault tree. No gate is explicitly created for the entries. The dummy AND node "d2" in the fault propagation graph in Figure 6 corresponds to the fact that the service "dbService" is using the backup server. Thus, an AND gate corresponding to this "d2" node is added as a child of the top gate in the fault tree. In this particular example, all the OR gates corresponding to the know nodes have only one AND child since there is only one minpath between the relevant components.

As in our earlier models [5], the representation here is logically inverted compared to most fault trees; an edge is true if the corresponding element is operational (rather than failed). However, the fault tree here is quite different from the earlier models because it represents knowledge dependencies and NOT functions.

6.2 Step (4) - Computation of Configuration Probabilities by Enumeration Approach:

From [6], this brute-force approach enumerates all the failure states. It is described here for comparison.

Let the total number of components (i.e. the processors and the tasks) in the MAMA model and the FTLQN model be *N*. Enumerate all 2^N failure states of the system. For each state $\Gamma = \{\gamma_1, \gamma_2, ...\gamma_N\}$, where $\gamma_c = 0$ or 1, if the *root* node of *G* is working (i.e. the state Γ satisfies the non-coherent fault propagation graph *G*), obtain the configuration *C* for that state. Set *Prob*(*C*) =

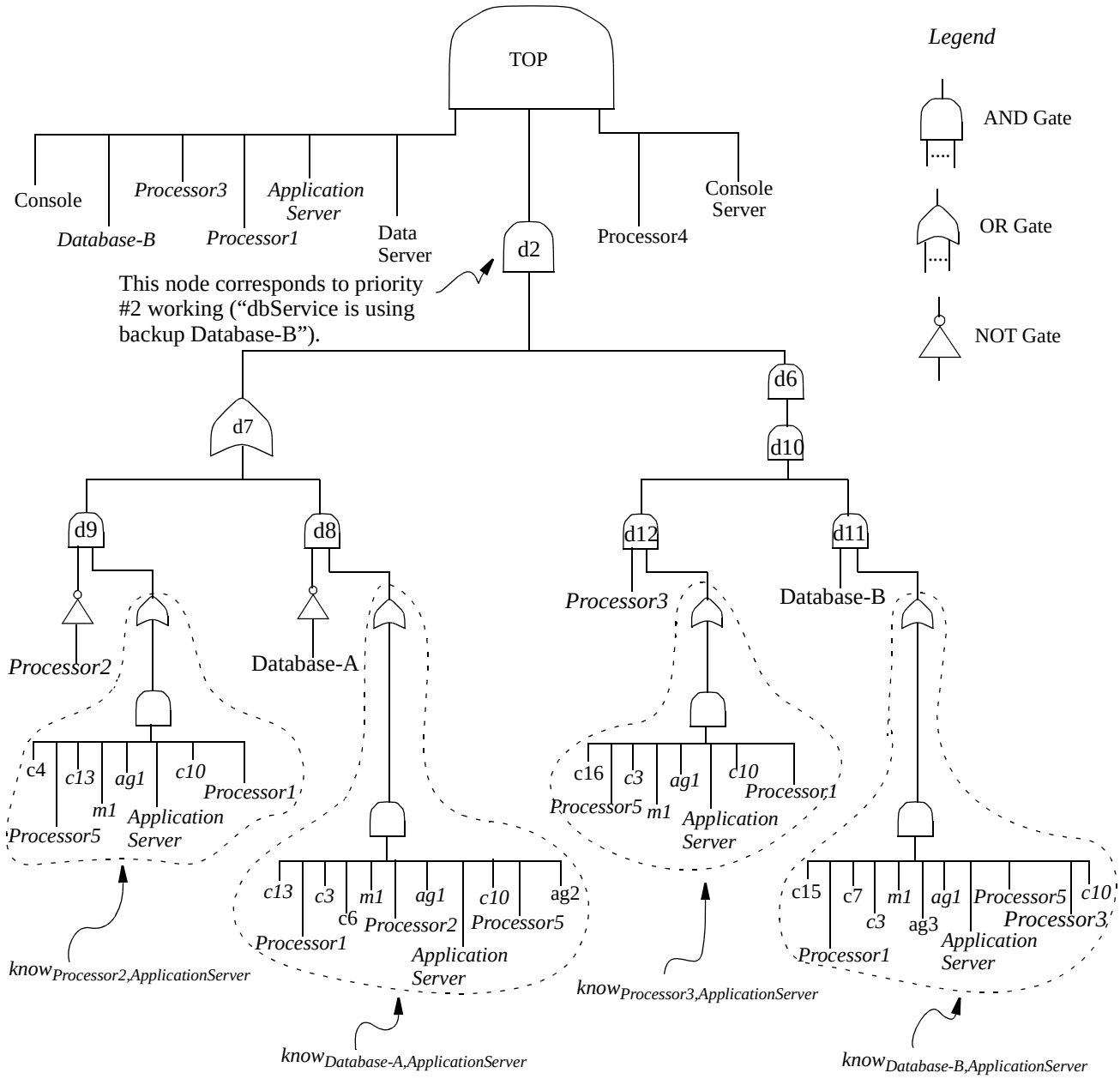


Figure 7. Non-coherent fault-tree corresponding to the operational configuration "*dbService is using the Server Database-B*" of the FTLQN model in Figure 1. Each dotted enclosure realizes a *know* function.The repeated basic events are shown in italics.

$$Prob(C) + \prod_{c=1}^{N} Prob\ (c \text{ is in state } \gamma_c).$$

We observe that the state aggregation approach takes more computational time as the number of operational configurations increases. The number of operational configurations depends on the structure and the number of components of the FTLQN model. The introduction of management components and their connections may exclude some operational configurations due to lack of knowledge in the management system about the status of the application components (but can never add more configurations, since they are a function only of the FTLQN model, not of the MAMA elements).

However, the addition of management components increases the total number of components (contributed by both MAMA and FTLQN) thereby increasing the number of states that has to be enumerated. Overall then the state aggregation approach should have less computation than the enumeration approach.

7. EXPERIENCE

7.1 Example

Let us consider the example of the layered console application shown in Figure 1 and a centralized management architecture managing the layered application as shown in Figure 4, in order to explain our solution approach. The probabilities of being in failed state for each of the components (i.e. all the management tasks, application tasks and the processors) are assumed to be 0.1. Let us consider the mean total demand for execution on the processor for entries "CreateReport", "Fetch", "Read-A", "GetApplicationData" and "GetConsoleData" be 1 seconds and for the thin client entry "UI" to be 0.01 seconds. The execution demand for the backup entry "Read-B" is assumed to be 1.5 seconds. Let us further assume that on average, at every invocation, a caller entry makes one service request to each of the called entries.

The MAMA model in Figure 4 is first translated to the knowledge propagation graph shown in Figure 5. Next, combining the information of the FTLQN model of Figure 1 and the knowledge propagation graph of Figure 5, a non-coherent fault propagation graph (illustrated in Figure 6) is obtained. Then, the set of all distinct operational configurations and their associated probabilities are found, with the results shown in Table 1. For this system, the number of distinct operational configurations is much less than the number of system states, two vs. 2^{16} (=65536).

Table 1. Standby-Redundancy policy: Distinct Operational Configurations and their probabilities obtained using State Aggregation Approach

Configuration C_i	Number of system states mapped to C_i (as found from Enumeration approach)	Prob(C_i)	Reward R_i = mean *throughput* of task "Console"
C_1	16	0.2824	0.1996
C_2	10	0.0511	0.1815
System failed state	65536 - (16 + 10) = 65510	0.6665	0
where, C_1: "dbService is using server Database-A." C_2: "dbService service is using server Database-B and Data-base-A failed."			

An LQN model is created for each of the operational configurations, C_1 and C_2, and solved using LQNS tool [8] to give their performance measures. In this example, we select the mean throughput of the load-generating task "Console" to be the reward. Finally, the expected reward rate of the system is obtained as 0.0656 responses/sec.

If we change the redundancy policy of "dbService" from stand-by to load-balanced, we would get more operational configurations as shown in Table 2. This is because when both the servers, "Database-A" and "Database-B" are operational, "dbService" would use both of them with the workload being equally distributed among the two. The probabilities of the configurations are different in this case because of the following reasons: In the standby case, we assume that a standby server can be used if the failure status of the primary and the working status of the standby is known by the management system. However, in the load-balanced case, we assume that a redundant server can be used if its working status is known. The expected reward rate of the system in this load-balanced case is found to have increased to 0.0683 responses/sec, because the change in the knowledge function makes the system failed state less probable.

7.2 Scalability Analysis (Comparison of Two Approaches)

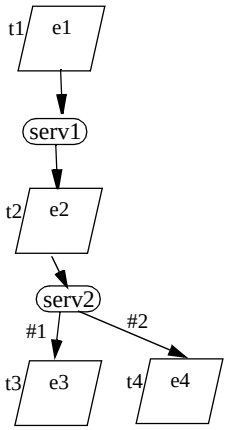
This section demonstrates the efficiency of the *State Aggregation Approach* over the *Enumeration Approach* by comparing them through incremental scaling up of a layered application and its associated management system.

The example is a three-tier client-server system shown in Figure 8(a), with four tasks t1-t4 running on their own private processors (not explicitly shown in the figure). Task t4 is a backup of task t3. The associated management system is a centralized model with one central manager controlling the whole application. Figure 8(b) shows the centralized management model with no agents (manager is directly watching all the application components). Figure 8(c) shows the same management model with agents running on each processor thereby increasing the number of management components (keeping the number of application components same). This is done in order to demonstrate the effect of increasing the number of management components on the performance of the two approaches.

Next, this application is scaled up in two stages; in the first stage (Figure 9) standby-servers are added in both layers and in the second stage (Figure 10) each service has two stand-bys with overlapping use. The management model (not shown here due to lack of space), with or without agents, for these two stages are also centralized and is similar to those used for the application in Figure 8(a), except that in these cases, the tasks interested for receiving notifications from the manager are t1 and t5 for application in Figure 9, and t1, t2, t5 and t8 for that in Figure 10. The probabilities of being in failed state for all the components (i.e. all the management tasks, application tasks and the processors) are assumed to be 0.05. The mean total demand for execution on the processor for all the entries be 0.3 seconds except the thin client entry "e1" whose demand is assumed to be 0.01 seconds.

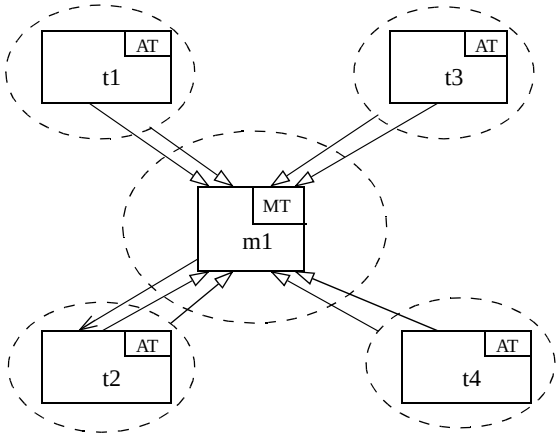
Table 3 shows the computation time taken for determining the distinct operational configurations and their probabilities by both the approaches. The computation time is in seconds and is measured on Windows98 hosted by Pentium (III) processor. Immediately we can see that aggregation is faster except for the smallest system, and that its effort grows much more slowly as the system is made more complex.

For the management model with no agents, as we move from one stage to the next (see row1, row3 and row5 in Table 3), we find an



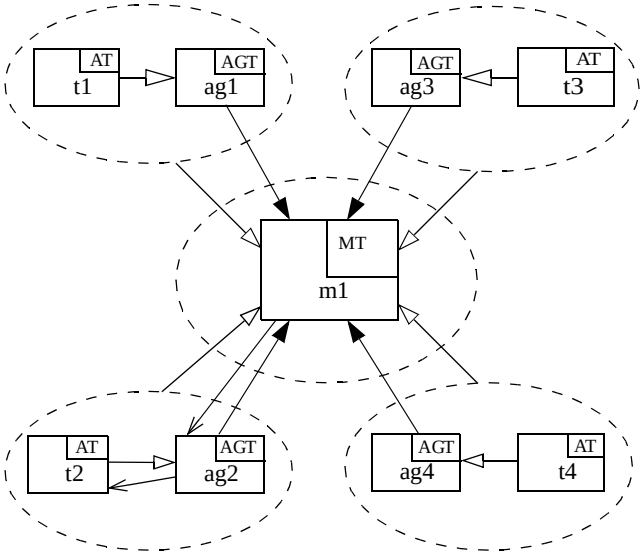
(a)

Original system: Bottom-level stand-by only



(b)

Centralized MAMA model for application in 8(a) without agents



(c)

Centralized MAMA model for aplication in 8(a) with agents

Figure 8. A three-tiered client-server system and its associated central management system (with agents in (b) and without agents in (c))

increase in the number of operational configurations due to the increasing addition of the stand-by servers, resulting in the generation of more non-coherent fault trees (one for each operational configuration), thereby increasing the computation time for the State Aggregation Approach. However for a particular scaled stage of the application (e.g. see row1 and row2 in Table 3), as we add agents to the management model, the number of management components increases (consequently the number of nodes in the knowledge propagation graph gets larger) although the number of operational configurations remains the same. The increase in the CPU time for the State Aggregation approach in this case is due to the minpath determination algorithm which now works on a bigger knowledge propagation graph.

The Enumeration approach, however, performs badly compared to the State Aggregation Approach with the scaling up of application as well as management components. As we add more standby servers to the application, the number of reconfiguration points increases, resulting in a larger number of executions of the minpath algorithm. Therefore, although there is the same number of system states (that need to be enumerated) in row2 and row3 in Table 3, the computation time for the Enumeration approach is higher for the case in row3.

8. CONCLUSIONS

The introduction of management components and their interconnections into the performability analysis of layered systems gives rise to scenarios where an operational state of the system corresponds to a mixture of failed and working states of its

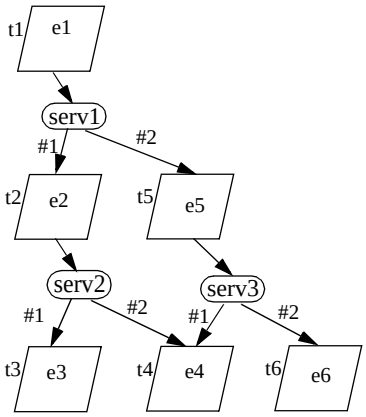


Figure 9. One stand-by server throughout

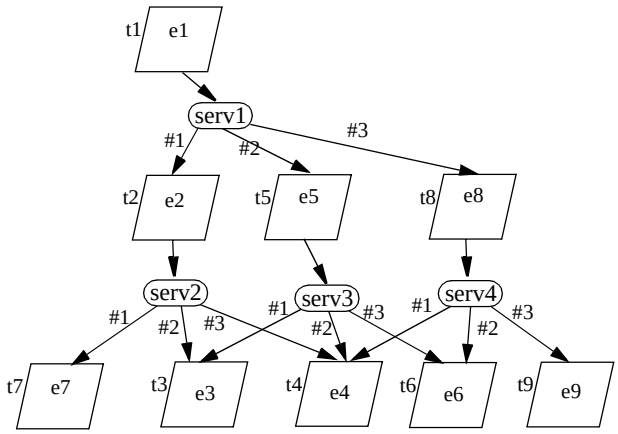


Figure 10. Two stand-by servers throughout

Scaled-up versions of the application in Figure 8. (The corresponding MAMA models (with and without agents) would be similar to those of Figure 8 and are not shown here.

Table 3. Computation time for determining the distinct operational configurations by the Enumeration Approach and the State Aggregation Approach for the models in Fig. 8 and 9

Application model	Centralized Management MAMA model	Number of System states (enumerated by Enumeration Approach)	Number of Operational Configurations (not including System failed state)	CPU time (in secs) for Enumeration Approach	CPU time (in secs) for State Aggregation Approach	Mean Throughput Reward (responses/ sec)
Fig. 8(a)	without agents (Fig. 8(b))	1024	2	0.22	1.42	1.194
	with agents (Fig. 8(c))	16384	2	1.49	1.48	1.075
Fig. 9	without agents	16384	4	3.19	1.81	1.319
	with agents	1048576	4	102.65	2.36	1.125
Fig.10	without agents	1048576	9	252	2.97	1.334
	with agents	~ 5 x 10^8	9	> 12 hrs	5.16	1.138

components, making the analysis non-coherent. This paper presented an efficient approach that takes into account this issue of non-coherence in the performability computation. Initial experience with modest sized examples shows a reasonable growth of effort with system complexity, indicating that the scalability of this approach may be excellent.

The analysis relies on layered queueing analysis, non-coherent fault tree solution and minpath algorithms. Examining a few cases, our observation is that the LQN solutions and non-coherent fault tree analysis scale up well. However, with the increase in the problem size, the minpath algorithms will limit the scalability of the present approach.

9. ACKNOWLEDGMENTS

This research was supported by a grant from NSERC, the Natural Sciences and Research Council of Canada, and by a scholarship from Nortel.

10. REFERENCES

[1] Blischke, W. R. and Murthy, D. N. Prabhakar, “Reliability: Modeling, Prediction, and Optimization”, Wiley, 2000.

Table 2. Load-balanced Policy: Distinct Operational Configurations and their probabilities obtained using State Aggregation Approach

Configuration C_i	Prob(C_i)	Reward R_i = mean *throughput* of task "Console"
C_1	0.0765	0.1996
C_2	0.0765	0.1815
C_3	0.2059	0.1901
System failed state	0.6411	0

where,
C_1: "dbService is using server Database-A, Database-B failed"
C_2: "dbService is using server Database-B, Database-A failed"
C_3: "dbService service is using both Database-B and Database-A with equally distributed load."

[2] Booch, G., Rumbaugh, J. and Jacobson, I., *The Unified Modeling Language User Guide*, Addison-Wesley, 1st edition, 1998.

[3] Colbourn, C. J., *The Combinatorics of Network Reliability*, Oxford University Press, 1987.

[4] Das, O. and Woodside, C. M., "The Fault-tolerant layered queueing network model for performability of distributed systems", IEEE Int. Computer Performance and Dependability Symposium (IPDS'98), Sept. 1998, pp. 132-141.

[5] Das, O. and Woodside, C. M., "Evaluating layered distributed software systems with fault-tolerant features", Performance Evaluation, 45 (1), 2001, pp. 57-76.

[6] Das, O. and Woodside, C. M., "Modeling the Coverage and Effectiveness of Fault-Management Architectures in Layered Distributed Systems", IEEE International Conference on Dependable Systems and Networks (DSN'2002), June 2002, pp. 745-754.

[7] Dutuit, Y. and Rauzy, A., "Exact and Truncated Computations of Prime Implicants of Coherent and non-Coherent Fault Trees within Aralia", Reliability Engineering and System Safety, 58, 1997, pp. 127-144.

[8] Franks, G., Majumdar, S., Neilson, J., Petriu, D., Rolia, J. and Woodside, M., "Performance Analysis of Distributed Server Systems," in the Sixth International Conference on Software Quality (6ICSQ), Ottawa, Ontario, 1996, pp. 15-26.

[9] Garg, S., Huang, Y., Kintala, C. M. R.,Trivedi, K. S. and Yajnik, S., "Performance and Reliability Evaluation of Passive Replication Schemes in Application Level Fault Tolerance", 29th Annual International Symp. on Fault-Tolerant Computing (FTCS'99), June 1999, pp. 322-329.

[10] Gokhale, S. S.,Wong, W. E., Trivedi, K. S. and Horgan, J. R., "An analytical approach to architecture-based software reliability prediction", IEEE Intl. Computer Performance and Dependability Symposium (IPDS'98), Sept. 1998, pp. 13-22.

[11] Goseva-Popstojanova, K. and Trivedi, K. S., "Architecture-based approach to reliability assessment of software systems", Performance Evaluation, 45 (2-3), 2001, pp. 179-204.

[12] Haverkort, B. R., Niemegeers, I. G. and Veldhuyzen van Zanten, P., "DYQNTOOL: A performability modelling tool based on the Dynamic Queueing Network concept", in Proc. of the 5th Intl. Conference on Computer Performance Evaluation: Modelling Techniques and Tools, G. Balbo, G. Serazzi, editors, North-Holland, 1992, pp. 181-195.

[13] Haverkort, B. R., "Performability modelling using DYQNTOOL$^+$", International Journal of Reliability, Quality and Safety Engineering, 1995, pp. 383-404.

[14] Huang, Y., Chung, P. Y., Kintala, C. M. R., Liang, D. and Wang, C., "NT-Swift: Software implemented fault-tolerance for Windows-NT", Proc. of 2nd USENIX WindowsNT Symposium, Aug. 3-5, 1998.

[15] Kreger, H., "Java management extensions for application management", IBM Systems Journal, 40(1), 2001, pp. 104-129.

[16] Laranjeira, L. A., "NCAPS: Application high availability in UNIX computer clusters", Proc. of 28th Int. Symp. on Fault Tolerant Computing (FTCS-28), June 1998, pp. 441-450.

[17] Luo, T. and Trivedi, K. S., "Using Multiple Variable Inversion Technique to Analyze Fault-trees with Inverse Gates", Fast Abstracts, ISSRE'98.

[18] Lyu, M. R., editor., *Handbook of Software Reliability Engineering*, McGraw-Hill and IEEE Computer Society, New York, 1996.

[19] Meyer, J. F., "On Evaluating the Performability of Degradable Computing Systems", IEEE Trans. on Computers, 29(8), Aug 1980, pp. 720-731.

[20] Musa, J. D., Iannino, A. and Okumoto, K., *Software Reliability - Measurement, Prediction, Application*, McGraw-Hill, New York, 1987.

[21] Stelling, P., Foster, I., Kesselman, C., Lee, C. and Laszewski, G. von, "A fault detection service for wide area distributed computations" in Proc. of 7th IEEE Symp. on High Performance Distributed Computations, 1998, pp. 268-278.

[22] Sun, H., Han, J. J. and Levendel, I., "Impact of Fault Management Server and Its Failure-related Parameters on High-Availability Communication Systems", IEEE International Conference on Dependable Systems and Networks (DSN'2002), June 2002, pp. 679-686.

[23] Tivoli Systems Inc., 9442 Capital of Texas Highway North, Arboretum Plaza One, Austin, TX 78759. See http:// www.tivoli.com.