Analysis and Optimization of Service Availability in an HA Cluster with Load-Dependent Machine Availability

Chee-Wei Ang and Chen-Khong Tham

Abstract—Calculations of service availability of a High-Availability (HA) cluster are usually based on the assumption of *load-independent* machine availabilities. In this paper, we study the issues and show how the service availabilities can be calculated under the assumption that machine availabilities are *load dependent*. We present a Markov chain analysis to derive the steady-state service availabilities of a *load-dependent machine availability* HA cluster. We show that with a load-dependent machine availability, the attained service availability is now *policy dependent*. After formulating the problem as a Markov Decision Process, we proceed to determine the optimal policy to achieve the maximum service availabilities by using the method of policy iteration. Two greedy assignment algorithms are studied: *least load* and *first derivative length (FDL) based*, where *least load* corresponds to some load balancing algorithms. We carry out the analysis and simulations on two cases of load profiles: In the first profile, a single machine has the capacity to host all services in the HA cluster; in the second profile, a single machine does not have enough capacity to host all services. We show that the service availabilities achieved under the first load profile are the same, whereas the service availabilities achieved under the second load profile are different. Since the service availabilities achieved are different in the second load profile, we proceed to investigate how the distribution of service availabilities across the services can be controlled by adjusting the rewards vector.

Index Terms—High availability, cluster computing, Markov chains, Markov decision processes, dynamic programming, neuro-dynamic programming.

1 Introduction

A High-Availability (HA) cluster is a network of computers that are arranged such that the machines can cover each other for redundancy. When a machine fails, the services (for example, Web, FTP, e-mail, and so forth) that it was hosting can be taken over by another machine in the HA cluster. In many current implementations, the backup arrangement (which machine to cover for which machine) is preconfigured. This static arrangement of backups is not adaptable to service load changes and can result in under or overutilization of resources if the service load changes. We can, of course, make changes to the backup arrangement manually when we detect an unbalanced state, but it would be better if the system can self-configure to set up the best backup arrangement automatically. This can reduce human error, which can be a major factor affecting service availability [1].

Today, we are seeing more interest in assembling big clusters of commodity computers, instead of using a supercomputer, to support large-scale services such as search engines, for example, back-end computers at Google [2]. Another example is the growing interest in grid computing [3]. Many of these commodity computers are not specifically designed for high reliability and, thus, backup plans such as redundancy and data replication must be put in place [4]. Managing such a large number of computers manually is tedious and time consuming: A self-configuring backup scheme is thus useful for managing such a big cluster of computers. By self-configuring we mean a *dynamic secondary machine assignment*, where machines are assigned as *secondaries* (backups) to other machines dynamically based on the machines' states. In comparison, static schemes are analogous to reservation schemes, where resources are reserved specifically for each service.

Service availability can be evaluated by model-based and measurement-based approaches [5], [6]. The model-based approaches can be further classified into nonstate-space and state-space approaches. Nonstate-space methods such as Reliability Block Diagrams (RBDs), Fault Trees (FTREEs) [5], and Critical Path analysis [7] can be applied to calculate service availabilities for stateless systems. State-space methods mostly refer to Markovian methods such as Continuous-Time Markov-Chains (CTMC) [5]. Stochastic Petri Nets (SPNs) are also used in availability evaluation of more complex systems. A system is first abstracted in SPN by using software such as SNP Package (SPNP) [8] and Symbolic Hierarchical Automated Reliability and Performance Evaluator (SHARPE) [9] and then analyzed to derive the statistics. Measurements can be performed if the system model is not available, as in [10], [11].

Most analyses assumed the load-independent machine availability [12], [13]. However, in reality, a machine is more

- *C.-W. Ang is with the Institute for Infocomm Research, 21, Heng Mui Keng Terrace, Singapore 119613. E-mail: angcw@i2r.a-star.edu.sg.*
- *C.-K. Tham is with the Department of Electrical and Computer Engineering, National University of Singapore, 10, Kent Ridge Crescent, Singapore 119260. E-mail: eletck@nus.edu.sg.*

Manuscript received 11 May 2006; revised 23 Nov. 2006; accepted 16 Jan. 2007; published online 26 Jan. 2007.
Recommended for acceptance by Y. Pan.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0124-0506.
Digital Object Identifier no. 10.1109/TPDS.2007.1071.

likely to fail if it is highly loaded [14], [10], [15], [11]. This is because a higher system activity means a higher chance of exposing software bugs and hardware transients [16]. In other words, machine availability can be modeled as a function of its average load (for example, CPU utilization, memory utilization, and so forth).

It is interesting to observe that the algorithms discussed in this paper may be similar to load balancing algorithms. However, there are some differences. A few load balancing algorithms are compared with the secondary assignment algorithms in Sections 5 and 7. We first describe the background and assumptions that we made in this paper in Section 2, then we present an analysis on an HA cluster with load-dependent machine availabilities in Section 3. After presenting several key concepts, we proceed to find the optimal secondary machine assignment policy in Section 4. In Section 5, we present two *greedy* secondary machine assignment algorithms. We describe a Reinforcement-Learning-based approach in Section 6 for implementing the optimal policy. We present our numerical and simulation results and discuss the effect of various dynamic secondary assignment schemes under different scenarios in Section 7. Finally, we conclude the paper in Section 8.

Our contributions in this paper are listed as follows:

1. Dynamic secondary machine assignment was introduced in maximizing service availability.
2. Service availability was analyzed for a load-dependent machine availability cluster using CTMC.
3. An optimal secondary machine assignment policy was derived using the Markov Decision Process (MDP) method.
4. New observations are made on service availabilities on an HA cluster.

2 Preliminaries

First, let us define two availabilities referred to throughout the paper. *Machine availability* refers to the probability that a machine is found to be in the running state at a given point in time. It can also be defined as the percentage of time that it is running in a given time interval:

$$a_m = \frac{MTTF_m}{MTTF_m + MTTR_m}, \qquad (1)$$

where a_m is the machine availability of machine m, $MTTF_m$ is the Mean Time To Failure, and $MTTR_m$ is the Mean Time To Recover (or Repair or Replace).

Service availability is similarly defined, but with respect to a service. A service is considered to be still available when the machine running it fails, but there is another machine taking over the running of the service, thereby extending the *MTTF* of the service. A service is down only when there is no machine taking over to run the service. In practice, the failover time can be reduced by having as much state information stored in shared disks as possible. We assume a similar configuration of an HA cluster as in [17], where the disks in each machine are accessible from other machines in the cluster so that when a machine fails, the machine that takes over can continue the services with minimal failover time.

We study two cases of *service load intensities*, defined as a fraction of the capacity of each machine to run this service: total average service load intensity < 1.0 (referred as TLL1 in this paper) and total average service load intensity > 1.0 (TLG1), where each machine is assumed to have the same *capacity* of 1.0 (our definition of *capacity* is the amount of workload intensity that a machine can take before the machine becomes unstable and may fail in a short time). The averaging of the service load intensity can be done with a sliding-window mechanism with an appropriate window size that smooths out instantaneous load fluctuations, but captures the load fluctuations across a longer period of time. We shall also discuss the effect on service availabilities when the total average load intensity equals or is close to 1.0.

2.1 Service Availability in a Load-Independent Machine Availability HA Cluster

We consider a set of machines M set up to run a set of services S. All machines are installed and configured with the application programs of all services so that every machine has the *capability* to take over any of the services $s \in S$. In this section, we study the achievable service availabilities under the *load-independent machine availability* assumption in an HA cluster.

2.1.1 *Total Average Service Loads Less Than 1 (TLL1)*

In this case, where every machine has enough *capacity* to take over *all* service loads, we can analyze this scenario with RBD [6]. The attained service availability A can be calculated as $A = 1 - \prod_{m \in M}(1 - a_m)$. Note that the service-independent availability is represented as A instead of A_s $(s \in S)$, since all services S have the *same* service availabilities. This is because all services start and stop together; that is, as long as one machine is up and running, all services are available, and the services are unavailable only when all machines fail.

2.1.2 *Total Average Service Loads Greater Than 1 (TLG1)*

Since each machine does not have enough capacity to take over *all* services in S, the service availability for each service will be different. For example, the lightly loaded services may have higher availabilities, since they are more likely to fit into the active machines than higher loaded services. The problem is thus state-based and can be solved using the CTMC technique.

2.2 Load-Dependent Machine Availability

The measurements described in [14], [10], and [15] show that a relationship between the average CPU load intensity and its failure rate exists. Reference [14] empirically shows that the number of failures increases with the number of bytes transferred in a Web server farm. Similarly, if the *workload intensity* was to increase, then more failures will be expected. The *hazard functions* described in [10] and [15] show both linear and exponential monotonically increasing relationships of failure rate versus average CPU load intensity[1] under different settings. In our case, we use the "average CPU load level versus machine availability," that is, $a_m(l_m)$, where l_m is

1. We use the term "load level" to mean "load intensity" in this paper.

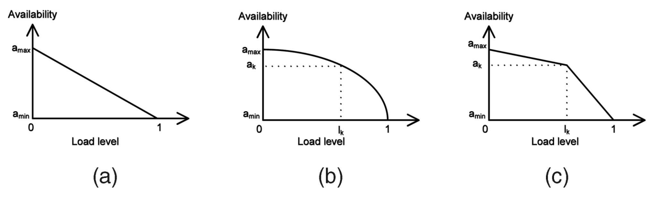


Fig. 1. LLTA functions. (a) Linear. (b) Exponential. (c) Two-piece linear. a_{max} and a_{min} are the maximum and minimum machine availabilities of the machine in its operating range. a_k is the availability at the *knee load* l_k.

the average CPU load level of machine m, instead of the "average CPU load level versus machine failure rate." The conversion is straightforward by using (1) if we assume, which we do in this paper, a fixed value for $MTTR_m$, $\forall m \in M$, and that $MTTF_m(l_m) = 1/\beta_m(l_m)$, where $\beta_m(l_m)$ is the failure rate of machine m. Several possible load-level-to-availability (LLTA) functions are shown in Fig. 1.

2.3 Dynamic Secondary Machine Assignment

Initially, each machine $m \in M$ in the HA cluster is assumed to run one primary service $s_m \in S$, but is also capable of acting as a secondary machine for other services $s \in S$ in the cluster. Secondary machine assignments are made before a machine fails. Upon being chosen as the backup to a service, the machine shall "prepare" the program that runs the service, for example, initializing and loading the necessary files, but will not start the service (that is, not performing hot backup or load balancing). Monitoring (for the event of machine/service failure) is then performed between the primary and the assigned secondary machines by using a *heartbeat* mechanism [18]. A machine running n services will be monitored by up to n machines (a single machine can act as secondary for more than one service). The scheme that we are considering is *dynamic* in that the secondary machine assignments can be changed dynamically based on the machine states. When a machine fails, the services that it was hosting will be taken over by other machines. The "takeover" refers to the takeover of the disks (for example, as in [17]) and its external network connections (for example, Internet Protocol (IP) address takeover). Since the program was "prepared" before the failure, the takeover time is kept minimal. If the session states are also kept in the disks, then ongoing user sessions can be preserved, and thus, no state transfer is necessary. Transport and network layers connections may time out during the takeover, but it should not cause disruptions to the ongoing sessions, as they can be reestablished. High availability and data replication in disks can be addressed by techniques such as the redundant array of independent disks (RAID) [19].

If there is no machine capable of or having enough capacity to take over a service, then the service is deemed *unavailable* and is put into the *waiting services* list W. When machine m recovers, it first performs a *failback* on its primary services $\{s_m\}$, then attempts to take over other waiting services from $W' = W - \{s_m\}$. If the total load of the waiting services W' exceeds the remaining capacity of machine m, then machine m will have to choose among W' the service(s) that it is taking over.

It may be interesting to discuss *performability* [20], [4], [21] briefly. Performability considers not only the binary state of machines, up or down, but also the level of degradation in between the two states. Although this paper does not consider the intermediate levels of degradation, a machine can be defined to be *unavailable* when it *degrades* below some level of fault.

1) Generate CTMC for the given HA cluster configuration and secondary machine assignment policy
2) Form global balance equations
3) With the normalization equation, solve for state probabilities
4) Determine service availabilities from the state probabilities

Fig. 2. Service availability calculation steps.

3 Determining Service Availabilities through CTMC Formulation

With the load-dependent machine availability modeling, the machine failure rate β_m becomes load dependent (that is, state dependent). In this section, we model the load-dependent machine availability system as a CTMC. The steps to determine the service availabilities are outlined in Fig. 2.

As an illustration, the CTMC for a three-machine and three-service example is shown in Fig. 3. The transition graph is generated based on the *least load* secondary machine assignment policy, where the least loaded machine is always chosen whenever a service needs to be reassigned when a machine fails. Since the next states are dependent on the assignment policy used, the transition graphs are different for different assignment policies. The service load profile assumed is {0.3, 0.2, 0.1} for services 1, 2, and 3, respectively (that is, this is a TLL1 case: $\text{total average service load} = 0.6 < 1.0$). The state is represented by a column vector $\overline{x}$ of size $|M| \times |S|$, depicting the services that the machines are hosting:

$$\overline{x} = \left(d_1^1, d_2^1, \ldots, d_{|S|}^1, \ldots, d_1^{|M|}, d_2^{|M|}, \ldots, d_{|S|}^{|M|}\right)^T,$$

where the Boolean $d_s^m = 1$ indicates that machine m is hosting service s, and $d_s^m = 0$ otherwise. T indicates vector transpose. For brevity, the state vectors are presented, with the commas removed between elements for the same machine in Fig. 3. Note that the CTMC is irreducible and positive recurrent; therefore, equilibrium state probabilities can be computed. Although the size of the state space cannot be directly calculated, as it depends on the assignment policy used (the set of states visited by different policies can be different), the state space grows exponentially.[2]

A state transition is triggered by a machine failure or recovery. Upon a machine failure, the services that it was running are reassigned to other machines. The reassignments

2. The number of states S_n for the *least load* secondary assignment policy is given by the recursive expression $S_n = 2 + n(S_{n-1} - 1)$, where $n > 1$ is the number of machines and services, and $S_1 = 2$. Using this expression, we see that the number of states grows exponentially: $S_1 = 2$, $S_2 = 4$, $S_3 = 11$, $S_4 = 42$, $S_5 = 207$, $S_6 = 1{,}238$, $S_7 = 8{,}661$, $S_8 = 69{,}282$, $S_9 = 623{,}531$, and $S_{10} = 6{,}235{,}302$.

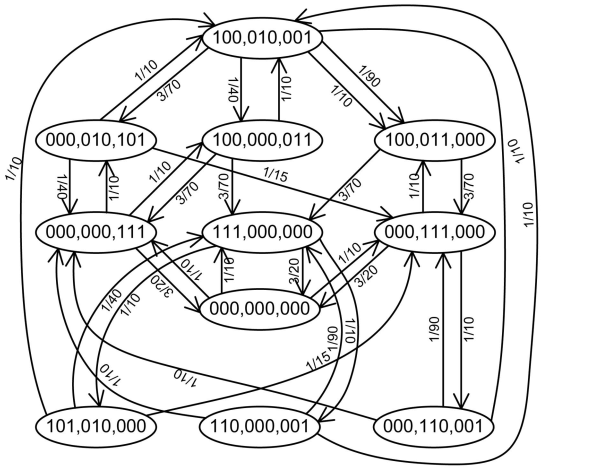


Fig. 3. CTMC of a three-machine/service cluster for the *least load* secondary machine assignment policy.

can be seen in the CTMC by inspecting the next-state vector with respect to the current-state vector. Referring to Fig. 3, say, the current state i is {100, 010, 001}, when machine 1 fails, service 1 is redeployed to machine 3 (because machine 3 was lightly loaded compared to machine 2), the next state j is thus {000, 010, 101}. Note that in this three-machine/service case, the assignment policy plays a part in determining the next state only at one state: {100, 010, 001}. The other state transitions are trivially determined, since there is essentially no other choice. The assignment policy has more influence in determining the achievable service availabilities when there are more machines. The transition rate λ_{ij} is calculated using the LLTA function, with the average load of service 1 (0.3). In this example, we assume a linear function $a_m = a_m^{max}(1 - l_m)$ (shown in Fig. 1a, with $a_m^{min} = 0$ for illustration), where a_m^{max} is set to 1, and l_m is the average load of machine m. Since machine 1 is running only service 1, $l_1 = 0.3$, and $a_1 = 0.7$. With $MTTR_m$, $\forall m \in M$ set to 10 time units, we calculate $MTTF_1(l_1)$ by using (1) to be $\frac{70}{3}$ time units. The transition rate $\lambda_{ij} = \frac{1}{MTTF_1(0.3)}$ is then calculated to be $\frac{3}{70}$. Since $MTTR_m$ is set to the fixed value of 10, the transition rates for all machine recovery events are fixed at 0.1.

With the transition rates determined, the global balance equations [22] can be formed:

$$\pi_j \sum_{i \in C^\mu} \lambda_{ji} = \sum_{i \in C^\mu} \pi_i \lambda_{ij}, \qquad \forall j \in C^\mu, \tag{2}$$

where π_x is the stationary-state probability of state x, $C^\mu \subset C$ is the set of states traversed by policy μ, and C is the set of all states.

The generator matrix $\mathbf{Q}$ is then formed from the set of global balance equations. The matrix element q_{ij} is defined as

$$q_{ij} = \begin{cases} -\lambda_{ji} & i \neq j \\ \sum_{j \in C^\mu} \lambda_{ij} & i = j \end{cases} \qquad i, j \in C^\mu.$$

Note that each column in matrix $\mathbf{Q}$ sums up to 0.0. The balance equation in matrix form is given by $\mathbf{Q}\overline{\pi} = \overline{0}$, where $\overline{\pi}$ is the column vector of state probabilities π_k, $k \in C^\mu$, and $\overline{0}$ is a zero column vector of dimension $|C^\mu|$, which we shall refer to as vector $\overline{b}$. Since these equations are linearly dependent, we include the normalization equation $\sum_{k \in C^\mu} \pi_k = 1$ to solve for the state probabilities π_k. The state probabilities π_k are then determined.[3] Finally, the service availabilities are calculated from the state probabilities. The state vectors $k \in C^\mu$ are scanned to identify the set of states $C_{\overline{s}}^\mu \subset C^\mu$ that do not contain service s, that is, those states k where $d_s^m = 0, \forall m \in M$. The service availability A_s for service s is then obtained by subtracting the sum of the corresponding state probabilities π_k from 1:

$$A_s = 1 - \sum_{k \in C_{\overline{s}}^\mu} \pi_k.$$

Note that although the example assumes a *deterministic policy*, that is, the *least load* policy, the CTMC analysis can be generalized for *stochastic policies*. An example of a stochastic policy is the RANDOM policy discussed in Section 4.2. Note that the number of states may not be larger due to the application of a stochastic policy because a stochastic policy only introduces additional transitions between the same states.

4 Determining Maximum Service Availabilities through MDP Formulation

In the previous section, we show that the service availabilities are dependent on the secondary machine assignment policy used. We also show how the service availabilities can be calculated for an example applying the *least load* policy. The next question to ask is: What is the *optimal policy* that, when applied, returns the maximum achievable service availabilities? In this section, we attempt to find the optimal policy by a *dynamic programming with policy iteration* [24]. First, we formulate the problem into an infinite-horizon MDP with discrete state, event, and action spaces.

4.1 MDP Formulation

The sequence of events and actions in the HA cluster can be modeled as an MDP [25]. Since we have already modeled the system as a CTMC in Section 3, it is relatively easy to transform it into an MDP (a semi-MDP to be exact, since the dwell times of the states are exponentially distributed).

A machine failure or recovery event triggers a state transition. As not all events are possible in each state (for example, a machine recovery event cannot occur on a running machine), the events are state dependent: $\Omega(s) \subset \mathcal{E}$, where $\mathcal{E}$ is the event space. Upon occurrence of an event $\omega \in \Omega(i)$ in a state i, an action $\mu \in U(i, \omega)$ is taken (where $U(i, \omega) \subset \mathcal{U}$ is the set of possible actions when the system is in state i, and event ω occurs, and $\mathcal{U}$ is the action space), which causes the system to transit to a new state j (the transition to j is deterministic with respect to an action μ in this case). A *policy* is a mapping $\mu : C \times \mathcal{E} \mapsto \mathcal{U}$. Together with the transition, a reward or cost $g(j)$ is generated (in our formulation, the reward or cost is dependent only upon the next state). Note that we use the same definition for state as in the CTMC formulation in Section 3. State values $V^\mu(i)$ are assigned to each state i. Each

3. Any method for solving linear simultaneous equations can be used. In our case, LU decomposition [23] was chosen for its speed.

state value can be interpreted as the expected return, given that the system starts in state i and controls the system by using policy μ [26]. Mathematically,

$$V^{\mu}(i) = E\left[\sum_{t=0}^{\infty} \gamma^t g(z_{t+1}) | z_0 = i\right],$$

where $\gamma \in [0, 1)$ is a discount factor, and z_t refers to the state at transition number t. The expectation is taken with respect to next states z_{t+1} because z_{t+1} is stochastic (the next state z_{t+1} depends on s_t and action μ, and μ, in turn, is dependent on the stochastic event ω). We assume a discounted sum of state values in this formulation instead of averaging because the recent past states have more effect in leading the system into the current state.

We can rewrite this expression to form the *Bellman equation* as follows:

$$V^{\mu}(i) = \sum_{j \in C^{\mu}} p_{ij}[g(j) + \gamma V^{\mu}(j)], \forall i \in C^{\mu}, \quad (3)$$

where $j = f(i, \mu)$; that is, next state j is a function of current state i, and the action $\mu \in U(i, \omega)$. p_{ij} is the transition probability of the system going from state i to j, which, in our case, is equal to the probability of the occurrence of event $\omega \in \Omega(i)$ in state i that leads to a transition to state j. This equality holds because given an event ω, the transition from i to j is deterministic, since policy μ is deterministic. Therefore, instead of summing across all possible events $\omega \in \Omega(i)$, we can sum across all possible next states j.

The transition probabilities p_{ij} can be calculated from the transition rates λ_{ij}, as described in Section 3, as follows:

$$p_{ij} = \frac{\lambda_{ij}}{\sum_{j \in C^{\mu}} \lambda_{ij}}, \forall i \in C^{\mu}. \quad (4)$$

Equation (3) can be arranged into matrix form $\overline{v} = \mathbf{P}(\overline{g} + \gamma \overline{v})$, where $\overline{v}$ is the column vector of state values $\{V(i)\}$, $\mathbf{P}$ is the $|C^{\mu}| \times |C^{\mu}|$ matrix of transition probabilities $\{p_{ij}\}$, and $\overline{g}$ is the column vector of rewards $\{g(i)\}$ for general i. Rearranging into standard form[4]

$$(\overline{I} - \gamma \mathbf{P})\overline{v} = \mathbf{P}\overline{g}, \quad (5)$$

where $\overline{I}$ is an identity column vector of size $|C^{\mu}|$.

To steer the system toward maximizing the service availability, rewards of $+1.0$ are given to those states that do not result in loss of services, whereas rewards of -10.0 are given to those states that make services unavailable. Note that as long as the rewards given to transitions to undesirable states are smaller than those given to transitions to the desirable states, the algorithm will converge to the optimal policy [24].[5]

Given policy μ, rewards $g(i)$, and transition probabilities P_{ij}, we can then solve for the state values in $\overline{v}$ by LU decomposition using (5), as we did in Section 3.

4. It is interesting to observe that whereas the computation of state values $V^{\mu}(z_t)$ is based on the state values of its *next states* z_{t+1}, the computation of state probabilities π_{z_t}, as in (2), is based on the state probabilities of its *previous states* z_{t-1}. With this difference noted, we can actually derive $\mathbf{P}$ from $\mathbf{Q}$, and vice versa.

5. We have used rewards of -1.0, 0.0, 0.9, and 0.99 for transitions into undesirable states, and the algorithm converges to the same results. The reason that -10.0 was used was to speed up the convergence.

1) Start with an arbitrary assignment policy
2) Generate the Markov Decision Process (MDP) graph with the current assignment policy
3) Solve for state values
4) If state values are the same as that from the previous iteration, goto step 7, else continue
5) Form greedy policy from the state values
6) Goto step 2
7) Return the last greedy policy as the optimal policy

Fig. 4. Policy iteration.

To determine the optimal policy, we have to first determine the maximum state values by solving the Bellman optimality equation:

$$V^*(i) = \sum_{\omega \in \Omega(i)} p_{i\omega} \max_{\mu \in U(i,\omega)} [g(f(i, \mu)) + \gamma V^*(f(i, \mu))], \quad (6)$$

where $V^*(i)$ is the optimal state values, and $p_{i\omega}$ is the probability that event ω occurs in state i.

What (6) essentially does is that at every event, the action that returns the largest value is chosen. With the optimal state values $V^*(i)$ calculated, we can then derive the optimal policy μ^* as follows:

$$\mu^*(i, \omega) = \arg \max_{\mu \in U(i,\omega)} [g(f(i, \mu)) + \gamma V^*(f(i, \mu))]. \quad (7)$$

We then solve this nonlinear Bellman optimality (6) by *policy iteration* [24], as described next.

4.2 Policy Iteration

The idea of policy iteration is to calculate the set of state values $V^{\mu}(i)$ iteratively by using the previous set of state values for the current iteration until the state values converge. The steps in policy iteration are depicted in Fig. 4.

The service availabilities are then calculated with the optimal policy by using the method described in Section 3. Note that in order to make all states available for selection when the greedy assignment is used, we have to make sure that all states are visited, and values are assigned (in Section 3, we noted that not all states are visited with all policies). In this case, we employed the *exploring start* method, as described in [24], whereby all possible next states are visited from any state in the first iteration. To do this, we start with a *stochastic policy*, which generates actions uniformly across all possible actions $U(i, \omega)$ (that is, the RANDOM policy). This way, when the state space is "walked," all states are visited. Implementing the RANDOM policy in simulation is straightforward: choose an action randomly among all possible actions uniformly every time an event occurs. To compute numerically, the transition rates λ_{ij} have to be computed for the RANDOM policy:

$$\lambda_{ij} = \frac{1}{|C^{\mu}_{i\omega}|} \cdot \left(\frac{1}{MTTF_m(l_m)}\right), i \in C^{\mu}, \forall j \in C^{\mu}_{i\omega} \subset C^{\mu},$$

where $C^{\mu}_{i\omega}$ is the set of possible next states from state i, given that event ω has occurred (in this case, the event is that machine m has failed).

The transition probabilities can then be calculated from the transition rates, as in (4). Steps 2 and 3 are described in Section 4. Step 5 is essentially implementing (7): this step is called *policy improvement* in [24]. The iteration runs until no further improvement can be done to the state values and policy, thereby satisfying (6). With a proper choice of the reward vector $\overline{g}$, we can determine the maximum service availabilities achievable for a given cluster configuration.

4.3 Implementation of the MDP-Based Solution

For practical implementation, the MDP-derived optimal policy can be computed *offline* (that is, concurrent with service provisioning) periodically with the latest measurements of the service loads. After the policy is generated, it can be captured in a mapping table and distributed to all machines in the cluster. By referring to this mapping table, a machine will know whether it is chosen to cover for what services residing in which machines. As mentioned in Section 3, the state space grows exponentially with the number of machines and services. For big installations, we propose to use a *hybrid* approach [27], where the secondary machine assignment first starts with one of the greedy algorithms (Section 5). The assignment policy is then *improved* gradually with the reinforcement learning (RL) technique (Section 6).

5 Greedy Secondary Machine Assignment Algorithms

In this section, we present two *greedy* secondary machine assignment algorithms. By *greedy* we meant that the secondary machine assignment performed at each machine failure event is decided solely by the *desirability* of the next state, without consideration of the future transitions after the next state.

The first algorithm that we study is the *least load*, where the least loaded machine is chosen to take over a downed service. This is similar to some load balancing algorithms where the loads are distributed such that each machine receives about the same amount of workload (for example, the Least Load First (LLF) algorithm [28] and Randomized Load Balancing [29]), since the load is reassigned to the least loaded machine upon a machine failure. However, there are some differences. First, the objectives are different. Typically, load balancing algorithms aim to minimize execution time, communication delays, or maximize resource utilization [30], [28], whereas secondary machine assignment algorithms aim to maximize service availability. Although performing load balancing improves service availability, it is not optimal. Load balancing algorithms consider the distribution of load during job admission (for example, ADAPTLOAD [31], where input workloads are matched to machines at admission based on the estimation of workload size distribution, and the objectives are to minimize the response time and maximize the utilization) or the periodic redistribution of jobs (for example, Adaptive Load Sharing (ALS) [28]), whereas we only consider the reassignment of machines for backup without load redistribution. Only the downed services loads are redistributed when a machine fails.

In the second greedy machine assignment method, we aim to maximize the *dwell time* of each state. Although the *least load* algorithm may appear to do the same, there are some differences. In fact, we shall see that the *least load* policy is a special case of this second algorithm. We shall call this second algorithm the *first-derivative-length (FDL)-based* secondary machine assignment algorithm. Since we assume a fixed machine *MTTR*, our aim is tantamount to maximizing the service uptime. With reference to the state-transition diagram, maximizing the service uptime is, in turn, equivalent to maximizing the time duration before the system transits to an "undesirable" state such as the zero state. The dwell time of state z is given by

$$t_z^{dwell} = \frac{1}{\sum_{k=1}^{N_z^r} \frac{1}{MTTR_k} + \sum_{k=1}^{N_z^f} \frac{1}{MTTF_k(l_k)}},$$

where N_z^r is the number of failed machines waiting to recover, and N_z^f is the number of active machines in state z.

Equivalently, the *dwell rate* of state z is given by

$$r_z^{dwell} = \sum_{k=1}^{N_z^r} \frac{1}{MTTR_k} + \sum_{k=1}^{N_z^f} \beta_k(l_k). \tag{8}$$

We thus want to maximize the dwell time (or, equivalently, minimize the dwell rate) at every state by redistributing the machines' loads. Let us start by assuming that we can redistribute all loads across all machines in *arbitrary-sized pieces*. This is a *convex optimization problem* because the cost is a convex function of load. The problem is akin to that of *optimal routing* [22], since the problem of finding the distribution of traffic load into a number of paths minimizing the overall cost is similar to our problem of finding the distribution of service loads across a number of machines minimizing the state dwell time. As in the network load distribution case, we can use methods like augmented Lagrangian with active sets or FDL [22] to solve for the best distribution. As an example, for a three-machine/service case, if the current state is {100, 010, 001}, and machine 1 fails, then what is the best redistribution of the service loads (in this case, only service 1 needs to be redistributed) across the two active machines (machines 2 and 3)? Since the dwell rate function in (8) is convex, we can find the minimum point by taking the derivative of the dwell rate with respect to the machines' average load and equating to zero. In this simple example, we can substitute machine 3's average load with $L - l_2$, where $L = l_1 + l_2 + l_3$ is the total average service load, differentiate the dwell time function with respect to machine 2's average load, and equate it to zero:

$$\frac{dr_z^{dwell}}{dl_2} = \frac{\partial \beta_2(l_2)}{\partial l_2} + \frac{\partial \beta_3(l_3)}{\partial l_3} \cdot \frac{\partial l_3}{\partial l_2} = 0.$$

Since $l_3 = L - l_1 - l_2$, we have

$$\frac{\partial \beta_2(l_2)}{\partial l_2} = \frac{\partial \beta_3(l_3)}{\partial l_3},$$

which is essentially akin to equating the FDLs of machines 2 and 3. In other words, to achieve the minimum dwell rate, we have to distribute the service loads such that the FDLs of the machines are equal.

Going back to our original secondary machine assignment problem, we cannot redistribute *all* service loads in arbitrary-sized pieces. Therefore, the problem is to find the services-to-machines match at each state so that the FDL equality is as *close* as possible. We can define *FDL distance* as the difference between the largest and smallest FDLs of the active machines. Continuing our example, if the machines are homogeneous, that is, having similar LLTA functions, and assuming a linear relationship, then we have

$$a_m = -(a_m^{max} - a_m^{min})l_m + a_m^{max}, \qquad \forall m \in [1, 2, 3].$$

The FDLs of the machines are then given by

$$\frac{\partial \beta_m(l_m)}{\partial l_m} = \frac{a_m^{max} - a_m^{min}}{\left(a_m^{max} - (a_m^{max} - a_m^{min})l_m\right)^2}. \qquad (9)$$

As an illustration, assume that $a_m^{max} = 1$, $a_m^{min} = 0$, and $MTTR_m = 1$, $\forall m \in [1, 2, 3]$, and assume that the service load profile is {0.45, 0.2, 0.1} for services 1, 2, and 3. If service 1 is assigned to machine 2, then we have $FDL_2 = \frac{1}{(1-(0.45+0.2))^2} = 8.16$ and $FDL_3 = \frac{1}{(1-0.1)^2} = 1.23$. If service 1 is assigned to machine 3, then we have $FDL_2 = \frac{1}{(1-0.2)^2} = 1.56$ and $FDL_3 = \frac{1}{(1-(0.45+0.1))^2} = 4.94$. Notice that the latter assignment (of service 1 to machine 3) gives a smaller difference between FDL_2 and FDL_3, so this assignment is preferred. We can extend this working to more machines and services, but the principle stays the same: choose the service-to-machine assignment that returns the *closest* FDLs. It is interesting to see in this example with the *homogeneous LLTA functions* that the FDLs are *closer* if we choose to assign the load to the *least loaded* machine.

Lemma 1. *Given that the LLTA function $a_m(l_m)$ of machine m is* concave *and* decreasing, *its corresponding FDL function $FDL_m(l_m)$ is* monotonically increasing.

Proof. If $a_m(l_m)$ is *concave* and *decreasing*, then the dwell rate r_m^{dwell} is *convex* and *increasing*, since r_m^{dwell} is essentially the reciprocal of $a_m(l_m)$, assuming a load-independent $MTTR$, as shown in (8). Since FDL_m is the derivative of r_k^{dwell}, and the derivative of an increasing convex function is monotonically increasing, $FDL_m(l_m)$ is *monotonically increasing*. □

Theorem 1. *If all machines $m \in M$, where M is the set of active machines in the cluster in the current state, are homogeneous in that their LLTA functions are the same and* decreasing concave, *then the* FDL-based *policy essentially reduces to the* least load *policy.*

Proof. Let $\bar{l}_a = \{l_1, l_2, \dots, l_{|M|}\}$ be the sorted list of average load levels of the active machines such that $l_1 \geq l_2 \geq \dots \geq l_{|M|}$. The corresponding FDLs of the active machines can then be computed: $\overline{FDL_a} = \{FDL_1, FDL_2, \dots, FDL_{|M|}\}$. Since the machine FDLs are the same, and from Lemma 1, the FDLs are monotonically increasing with respect to their loads, the FDLs can be ordered as $FDL_1 \geq FDL_2 \geq \dots \geq FDL_{|M|}$. By defining *FDL distance* as the difference between the largest and smallest FDLs, we have $FDL\ distance = FDL_1 - FDL_{|M|}$. In order to decrease the *FDL distance*, we have to either decrease FDL_1 by decreasing l_1 or increase $FDL_{|M|}$ by increasing $l_{|M|}$. If there is only one service to be reassigned, then machine $|M|$, which is the least loaded active machine, should be chosen to take over the service so as to decrease the *FDL distance*. The choice of the least loaded machine thus corresponds to that made by the *least load* policy. □

If we have more than one service to be reassigned, then we have to search all combinations of the downed services and the active machines to find the combination that results in the smallest *FDL distance*. The same procedure applies to other cases with heterogeneous machines and the TLG1 load profile.

Note that if the machines' LLTA functions are different, then the *least load* policy is *not equivalent* to the *FDL-based* policy. For example, consider two active machines, with $a_1 = 0.7(1 - l_1)$ and $a_2 = 0.9(1 - l_2)$. The FDLs of the machines are then given by $FDL_1 = \frac{0.7}{(0.7-0.7l_1)^2}$ and $FDL_2 = \frac{0.9}{(0.9-0.9l_2)^2}$. Say, $l_1 = 0.2$, and $l_2 = 0.25$. The *least load* policy will put any new load into machine 1, since its load is smaller. However, the *FDL-based* policy will put the new load into machine 2, since $FDL_2(0.25) = 1.975$ is smaller than $FDL_1(0.2) = 2.232$.

6 Reinforcement Learning-Based Assignment Scheme

In this section, we describe how the optimal policy can be derived using the RL [24] (also called Neuro-Dynamic Programming (NDP) [32]) technique. The crux of RL is the determination of the state values $V^\mu(i)$ online through *simulation* instead of explicitly calculating by solving the system of equations with the state-transition probabilities, as described in Section 4. The steps of determining the optimal policy is similar to that described in Section 4, except that *value iteration* is employed instead of policy iteration.

The RL technique has been applied in many optimization problems where either the state space is intractable or continuous [33] or the state vector cannot be easily determined, as in [27], where the state vector is modeled based on the authors' experience with the problem. The RL technique is thus applicable in our problem of maximizing the service availabilities for bigger systems. In Section 3, we see that the state space grows exponentially with the number of machines and services. With RL, there is no need to precalculate all state-transition probabilities or rates, which requires visiting all possible states. Although the state space is still the same in RL, the optimization need not visit all states, as in *asynchronous dynamic programming* [24], and is thus scalable.

In our problem, a sample can be represented as a 3-tuple $\{i, j, g\}$, where i is the pretransition state, j is the posttransition state, and g is the immediate reward. Note that this representation is different from many other RL formulations, which use $\{i, \mu, g\}$ instead, but since j is deterministic upon an action μ, we choose to use j for easier implementation. We use the TD(λ) update rule [24] (TD stands for *temporal difference*), where $\lambda \in [0, 1]$. λ controls the effect of an update upon the previous states. The larger the λ is, the more the previous states get updated. If we set $\lambda = 0$, then we have the one-backup rule, where only the immediate previous state gets updated. The TD(0) update rule is given by

$$R_t = g(z_t) + \gamma V^\mu(z_t)$$
$$V^\mu(z_{t-1}) \leftarrow V^\mu(z_{t-1}) + \alpha[R_t - V^\mu(z_{t-1})],$$

where $\alpha \in [0, 1]$ is an update factor, and z_t is the state at time t.

As mentioned in Section 4, the state value $V^{\mu}(\overline{x})$ is the expected return of the system if it starts from state $\overline{x}$ and follows a policy μ. The state value of a state can be calculated using the Bellman equation, as in (3): The state value is essentially the weighted sum of the expected rewards plus the next state values. The weights are the transition probabilities to the respective next states from a given state. In the RL case, the transition probabilities are implicitly derived by the frequency of visits to the next states from a given state, following policy μ. Therefore, we can calculate the state value of a given state by averaging the sample updates (immediate reward plus the next state value) when it transits to the next state. We implement an incremental version of the averaging as $\alpha_{\overline{x}} = \frac{1}{1+c_{\overline{x}}}$, where $c_{\overline{x}} \geq 0$ is the number of visits to state $\overline{x}$. When a sample tuple is generated at time t, the previous state value $V^{\mu}(z_{t-1})$ gets updated. As $t \to \infty$, the state values approach $V^{\mu}(i), \forall i \in C^{\mu}$, where C^{μ} is the set of states visited under policy μ.

It is important to find a balance between exploration and exploitation. Exploration is necessary during initial episodes (we define episode as the set of state transitions in between two "undesirable" states) so that as many states as possible can be visited. However, the amount of exploration should be reduced gradually so that the state values can converge to the optimal values. In this paper, we adopted a modified ϵ-greedy method [24], where the ϵ factor is decayed so that the iterations can converge to full exploitation: $\epsilon_{k+1} = \epsilon_k \times \eta$, with $\epsilon_0 = 1$, where $0 \leq \eta \leq 1$ is the decay factor, and k is the episode number. An episode ends when one of the "undesirable" states is visited. "Undesirable" states are those states that have some services unavailable.

7 Numerical and Simulation Results

In this section, we discuss the results obtained by numerical calculation and simulation. First, we make some observations based on numerical results, considering various scenarios with different service load profiles and LLTA functions. Then, we verify the numerical results with simulation. Finally, the simulation of the RL scheme, as described in Section 6, is discussed.

7.1 Numerical Results

A program is written to implement the procedures, as described in Section 3. It essentially "walks" the state space by following the assignment policy. The state-transition rates are then calculated from the given LLTA functions. It then solves for the state probabilities and finally derives the service availabilities. Similarly, another program was written to implement the procedure for finding the optimal policy and maximum achievable service availabilities, as described in Section 4.

7.2 Service Availabilities

We compare five schemes against the optimal service availabilities obtained using the MDP technique: *least load*, *FDL-based*, RANDOM, STATIC, and *max load*. The *least load* and *FDL-based* schemes are implemented as described in Section 5.

The RANDOM scheme provides a benchmark of how different assignment schemes perform. This is akin to the Randomized Load Balancing algorithm [29] in that a machine is chosen randomly to take over a load. It randomly chooses one out of all possible combinations of matching the waiting services with the active machines, taking the machine capacities into account. The STATIC scheme is analogous to the preconfigured secondary machine assignment scheme, which, as far as we know, is the current practice in configuring HA clusters. This scheme is similar to the FF or LLF [28] load balancing algorithms in that the choice of machine is made in a predefined way, for example, round robin. The secondary service backup arrangement is predetermined and stays unchanged throughout the operating lifetime. In our experiments, the static service backups are arranged this way: When a machine fails, the services that it was running are to be reassigned to the next machine, say, the right-hand-side (RHS) machine, where the machines are logically arranged in a circle. The to-be-redeployed services are checked one at a time against the RHS machine. If the RHS machine has no capability or capacity to take over the service, then the next RHS machine in line is checked. This way, all the resources are checked and used to support the services. The *max-load* scheme always chooses the most loaded machine to take over a downed service and is intuitively expected to perform the worst: we want to see how bad the service availability can go. We use three sets of service loads: TLL1, TLG1, and $\text{total average service load} = 1.0$ (TLE1). We study three to six-machine/service cluster configurations.

7.2.1 Total Average Service Load Less Than 1 (TLL1)

When the total average service load is less than the capacity of any one machine, the services would have the same service availabilities. We study the cases with homogeneous and heterogeneous machines (that is, machines with different LLTA functions). The average service loads used in our studies are listed as follows ($\text{total average load} = 0.75$):

- three machines: {0.45, 0.2, 0.1},
- four machines: {0.3, 0.2, 0.15, 0.1},
- five machines: {0.25, 0.2, 0.15, 0.1, 0.05}, and
- six machines: {0.24, 0.19, 0.14, 0.09, 0.05, 0.04}.

The machines' LLTA functions are set to be the same, that is, $a_m = a_m^{max}(1 - l_m)$, with a_m^{max} set to 1, and $a_m^{min} = 0$ for illustration.[6] The results are plotted in Fig. 5 and tabulated in Table 1.

The graphs show the monotonically increasing relationship between service availability with respect to the number of machines. As the number of machines is increased, the difference between the schemes decreases, as there is not much room for improvement. As expected, the *FDL-based* and *least load* schemes perform best and are very close to the optimal. The RANDOM scheme works in between the *least load* and STATIC schemes. Although we see only small differences among the schemes, in the service availability industry [34], we are concerned with small availability improvements, as that translates into considerably long

6. Although in real cases, a_m^{min} may not be 0, that is, the machine fails immediately, a_m^{min} should be much smaller than a_m^{max} when the machine is fully loaded.

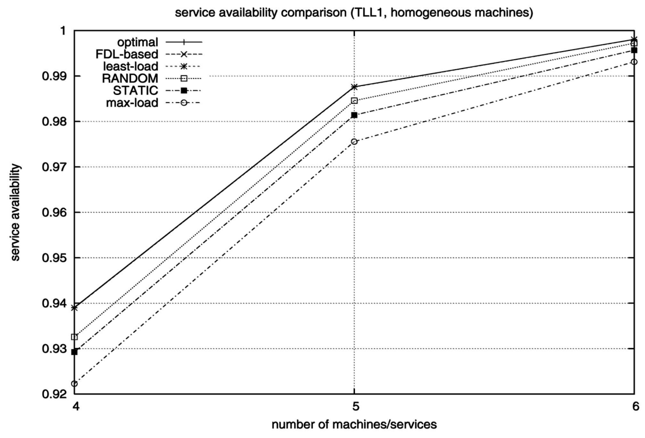


Fig. 5. Service availability comparison (TLL1, homogeneous machines).

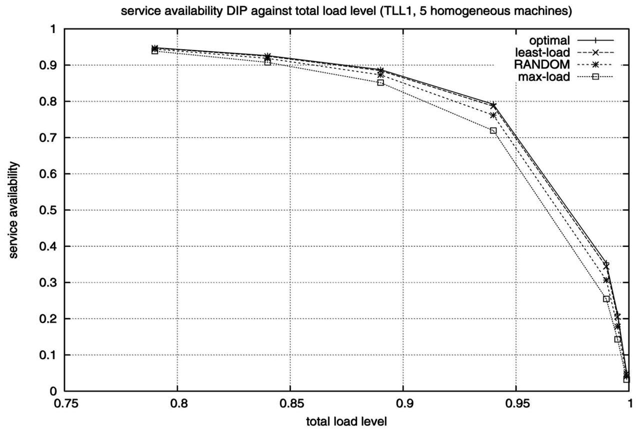


Fig. 6. The DIP effect (TLL1, homogeneous machines).

period of time when we consider *MTTF* to be in the order of weeks or months.

It is interesting that even the RANDOM scheme outperforms the STATIC scheme. However, we should note that the result does not necessarily imply that *all* STATIC schemes are inferior to the RANDOM scheme. The performance of the STATIC scheme depends a lot on its design. For example, if a heavily loaded machine happens to be assigned as the backup machine for another heavily loaded machine, then the service availability achievable is going to be low. This is the case here: the RHS machines, which act as the backup to left-hand-side (LHS) machines, are always the next-in-line lower loaded machines.

When the total average load approaches capacity (TLE1), the service availability drops drastically, as shown in Fig. 6. We call this service availability drop the *DIP* effect. Note that this effect occurs *only if* the machine availability is modeled to be load dependent. As the total average service load approaches 1.0, if a machine were to take over all the service loads, then its machine availability will approach a^{min}, as modeled by its LLTA function. This machine will thus fail with a short *MTTF*. When another machine recovers, it will again take over all service loads and cause it to fail with a short *MTTF* as well. Overall, this results in a drastic drop in service availability.

To avoid the *DIP* effect, we can keep the average load on each machine below some threshold by *rejecting* a new load before its capacity is reached (similar to the random early detection (RED) algorithm used in active queue management [35]). For example, the threshold can be calculated such that we still achieve an improvement in service availability over the no-take-over case:

$$threshold = \arg_{tr}\left\{ A_s(tr) > \kappa\left(\frac{1}{|M|}\sum_{m\in M} a_m(s_m)\right)\right\},$$

where $\kappa \geq 1.0$ is the service availability gain that we want to achieve, and tr is the machine average load threshold, above which it will stop taking over any service.

Note that rejecting the waiting services before a machine's capacity is reached will not drastically reduce their service availabilities; on the contrary, its helps the whole system from collapsing into the *DIP*.

Let us now introduce heterogeneity into the system by assuming different LLTA functions for different machines. We want to see how heterogeneity affects the service availabilities. For this heterogeneous LLTA study, we use the following two-piece linear function for each machine:

$$a_m = \begin{cases} a_m^{max} - \left(a_m^{max} - a_m^k\right)\frac{l_m}{l_m^k}, & \text{for } l_m \leq l_m^k \\ a_m^k\left(\frac{1-l_m}{1-l_m^k}\right), & \text{for } l_m > l_m^k \end{cases}$$

with parameters $a_m^{max} = \{0.9, 0.8, 0.7, 0.6, 0.5, 0.4\}$, $a_m^k = 0.8a_m^{max}$, and $l_m^k = \{0.4, 0.5, 0.6, 0.7, 0.8, 0.9\}$, where a_m^k is the knee availability, and l_m^k is the knee load of machine m. The results are plotted in Fig. 7 and tabulated in Table 2.

The results show that the *FDL-based* policy can achieve better service availabilities than the *least load* policy if the machines are heterogeneous. This reaffirmed the analysis, as presented in Section 5, that the *least load* policy is not the same as the *FDL-based* policy when the LLTA functions of the machines are different and, thus, the *least load* policy is less optimal.

TABLE 1
Service Availability (TLL1, Homogeneous Machines)

Assignment	*Number of machines*			
schemes	*3*	*4*	*5*	*6*
Optimal	0.778852	0.939044	0.987617	0.998049
FDL-based	0.778852	0.938987	0.987600	0.998043
least-load	0.778852	0.938987	0.987580	0.998042
RANDOM	0.771841	0.932575	0.984571	0.997237
STATIC	0.769450	0.929236	0.981391	0.995669
max-load	0.765173	0.922295	0.975566	0.993122

7.2.2 Total Average Service Load Greater Than 1 (TLG1)

In the TLG1 case, a single machine does not have the capacity to take over all services. Some services may have to wait to be taken over. The results are plotted in Figs. 8 and 9 and tabulated in Tables 3 and 4 for homogeneous and heterogeneous LLTA functions for the five-machine/service cluster configuration. The average service loads are given by $\{0.45, 0.34, 0.23, 0.13, 0.1\}$ $(\text{total average service load} = 1.25 > 1.0)$.

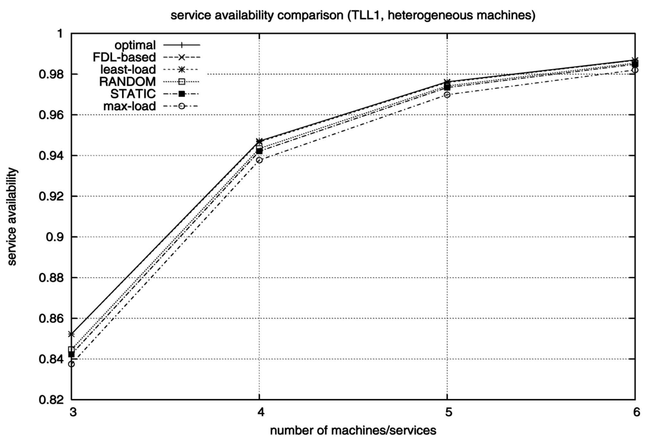


Fig. 7. Service availability comparison (TLL1, heterogeneous machines).

TABLE 2
Service Availability (TLL1, Heterogeneous Machines)

Assignment	*Number of machines*			
schemes	*3*	*4*	*5*	*6*
Optimal	0.852251	0.947075	0.976217	0.986966
FDL-based	0.852251	0.947071	0.976156	0.986818
least-load	0.852251	0.946665	0.975901	0.986768
RANDOM	0.844653	0.943424	0.974154	0.985450
STATIC	0.842279	0.942084	0.973387	0.984828
max-load	0.837508	0.937734	0.969892	0.982031

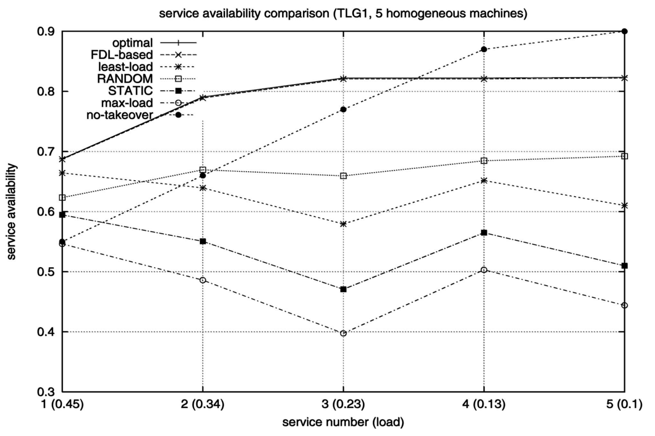


Fig. 8. Service availability comparison (TLG1, five homogeneous machines).

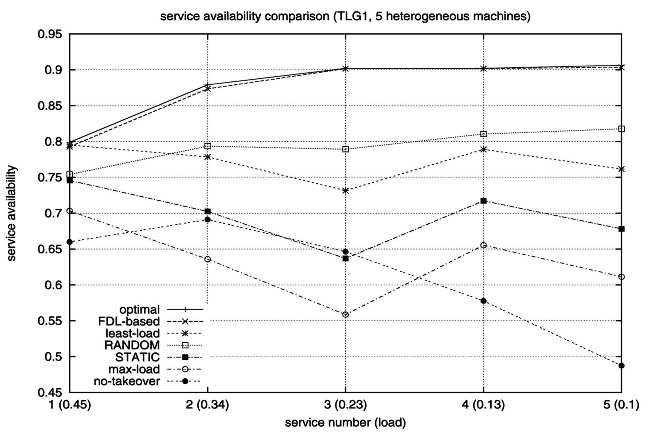


Fig. 9. Service availability (TLG1, five heterogeneous machines).

TABLE 3
Service Availability (TLG1, Five Homogeneous Machines)

Assignment	*Service number (decreasing average load)*				
schemes	*1 (0.45)*	*2 (0.34)*	*3 (0.23)*	*4 (0.13)*	*5 (0.10)*
Optimal	0.687708	0.790585	0.822176	0.822176	0.823346
FDL-based	0.687088	0.788957	0.820746	0.820746	0.822224
least-load	0.664223	0.639453	0.579443	0.651725	0.610178
RANDOM	0.623276	0.669561	0.659403	0.684521	0.692206
STATIC	0.594506	0.550529	0.470718	0.564867	0.509925
max-load	0.546220	0.486063	0.397358	0.503125	0.443970
no-take-over	0.550000	0.660000	0.770000	0.870000	0.900000

TABLE 4
Service Availability (TLG1, Five Heterogeneous Machines)

Assignment	*Service number (decreasing average load)*				
schemes	*1 (0.45)*	*2 (0.34)*	*3 (0.23)*	*4 (0.13)*	*5 (0.10)*
Optimal	0.798949	0.879018	0.901984	0.901984	0.906257
FDL-based	0.792746	0.873412	0.901613	0.901613	0.903535
least-load	0.795075	0.778582	0.731582	0.789024	0.761674
RANDOM	0.754010	0.793562	0.789304	0.810505	0.817763
STATIC	0.745750	0.702449	0.636909	0.717287	0.678240
max-load	0.703281	0.635991	0.558740	0.655501	0.611395
no-take-over	0.660000	0.691200	0.646333	0.577714	0.487500

The results show bigger differences in service availabilities among different assignment policies. The two TLG1 plots show a similar ordering of the achieved service availabilities, except that the case with heterogeneous machines achieves higher service availabilities. The no-take-over case is also shown for reference in Figs. 8 and 9. The heterogeneous case shows that we can better utilize the resources to achieve higher service availabilities by dynamic secondary machine assignment (that is, pooling and sharing of resources) as compared with the no-take-over policy.

Unlike the case in TLL1, the *least load* policy does not perform close to the optimal. From the plots, we see that even the RANDOM policy outperformed the *least load* policy. In the TLL1 case, the assignment policy only selects among the active machines to take over a set of services. In the TLG1 case, the assignment policy has to also decide which services to *reject*, since the resources do not have the capacity to take over all the waiting services. In the *least load* policy, the waiting services are processed in a First-Come, First-Served (FCFS) way; thus, the lower numbered service gets assigned first, leaving the other higher numbered services with less or insufficient resources. Obviously, this FCFS way of deciding which service to reject is not optimal.

The *FDL-based* policy again exhibits a close-to-optimal performance. The close-to-optimal performance also shows that the effect of considering future states is small. That is, we can make good secondary machine assignment based solely on current information, without considering the desirability of the future states, as considered in the MDP-based algorithm. However, we should note that the *FDL-based* scheme relies on the computation of FDL values, which requires that the LLTA functions be differentiable.

Another observation is that all but the optimal and *FDL-based* policies show a nonmonotonous relationship between service availability and average service load. Service 3 is seen to have the lowest availability among the five services under

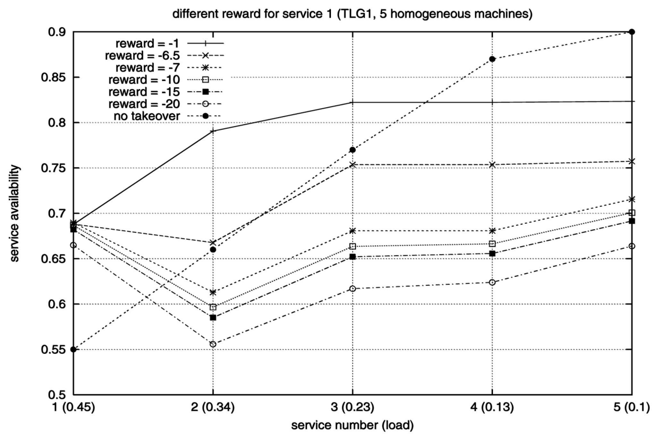


Fig. 10. Service availability comparison (TLG1, five homogeneous machines biased toward service one).

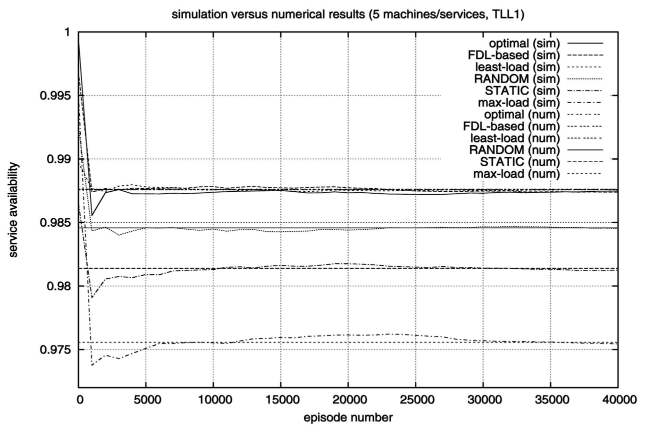


Fig. 11. Convergence of various machine assignment schemes.

the *least load* and *max-load* schemes. This is due to the FCFS policy used in both schemes. When the system is in a state with one machine running services 1, 2, and 3, and this machine fails, service 3 is always rejected, since services 1 and 2 $(\text{total average load} = 0.45 + 0.34 = 0.79)$ are processed first and can find the capacity in either machines 4 or 5. As for machines 4 and 5, with small average loads, it is relatively easier to find the required capacity when the machines that they run on failed. Therefore, service 3 has a relatively poor availability. The RANDOM policy shows less of this problem and shows an increasing service availability against the average service load.

7.3 Service Availability Distribution Control

Both MDP-based and RL-based optimizations are steered by the design of rewards (or penalties). In this section, we study the feasibility of steering the optimizations by redesigning the rewards such that certain service availability distribution can be achieved. For example, is it possible to achieve the same service availabilities among the services in the TLG1 case? Is it possible to achieve higher availabilities for higher loaded services? In this experiment, we vary the reward for service 1 to see how it affects the service availability distribution among the services. The result is shown in Fig. 10.

Depending on the amount of penalty that we impose, we obtain different distributions of the service availabilities. Contrary to what one would expect, instead of increasing the service availability of service 1 by increasing the penalty for the occurrence of service 1's failure, the service availabilities of the other services are reduced to achieve the availability distribution that we want. This effect can be explained as follows: If we were to increase the penalty when service 1 is unavailable, then ultimately, service 1 will always be selected for reassignment (when a machine that it runs on fails) in favor of other waiting services, leaving other services with fewer choices, thus reducing their achievable service availabilities.

7.4 Verification of Numerical Results through Simulation

To show that the concepts work, we build a testbed with three machines and three services. The testbed was set up as *proof of concept* to verify the protocol signaling and that the service takeover time is small; that is, a service session is not disrupted when the machine running it fails. Experimental results on the achieved service availability were not practically obtainable because it would take a long time for a machine to fail. What we did was to *inject* failure events into the machines randomly (failure occurrence in each machine, modeled as a Poisson process with the calculated failure rate) and observe how the machines react. The machine failure rate was modeled by a linear LLTA function, as described in Section 2.2, with the *measured* workload intensity as input. In order to speed up the experiments, a^{max} and a^{min} were made small. However, since the failure events occur *artificially* based on the LLTA function, the results are akin to that obtained using a discrete event simulation. A discrete event simulator (using *smpl* [36]) is thus written to verify the numerical results.

Since the LLTA function is not made known to the decision algorithm, the RL technique is used to derive the optimal policy. The simulation results were generated based on sliding-window averaging, with a windows size of 100,000 samples. Each sample is generated upon completion of an episode. That is, rather than *ensemble averaging* by running many independent simulations, we run one simulation and obtain the final value by *time averaging*, since the system is *ergodic*. The reason for doing this is that we are only interested in the final converged value and not the convergence characteristic, since our purpose is to verify the numerical results. Fig. 11 shows the convergence of the various machine assignment schemes, excluding the RL scheme.

The simulation of the optimal scheme, as shown in Fig. 11, is done by importing the numerically calculated state values of the same configuration to the simulator. The secondary machine assignment is then done greedily with respect to the state values and the immediate rewards. We can see that the simulated results agree with the numerical results, thus verifying the correctness of our CTMC and MDP formulations. Note that although it appears that the system takes about 10,000 episodes to reach the steady-state availabilities, it should not be interpreted as such. The "convergence" is due to the averaging of the past 100,000 samples, and at the beginning of the simulation, we do not have enough samples

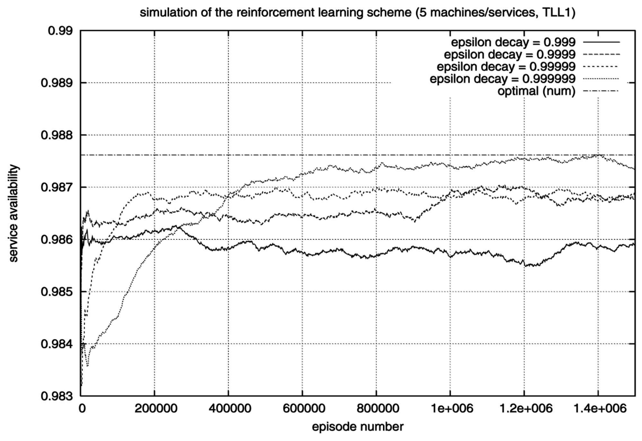


Fig. 12. Convergence of the RL scheme.

to smooth out the instantaneous availabilities. The convergence curve of the RL scheme is shown in Fig. 12.

As mentioned in Section 6, the selection of the ϵ-decay η affects the achieved service availabilities. Fig. 12 shows that the smaller the η, the faster the convergence, but the poorer the performance. A faster decay leads to insufficient state explorations, which results in a suboptimal performance. We thus have to choose a decay value such that we can converge to the optimal performance as fast as possible. In this case, the choice of $\eta = 0.999999$ allows us to reach the optimal service availabilities. To hasten the convergence of the RL scheme to the optimal availabilities, we can *pretrain* the system with estimates of the machine LLTA functions. In the above RL experiment, we started off with essentially the RANDOM assignment policy (note that the graphs start off with service availabilities of around 0.984, which is close to the steady-state availabilities achieved with the RANDOM policy). The RANDOM policy was used so as to promote the exploration of the state space. If we were to start with a policy that is close to the actual optimal policy, then it would be able to converge faster and experience good performance during convergence when the RL agent is learning.

8 Conclusions

We have presented an analysis and an optimization for a load-dependent machine availability HA cluster. With the load-dependent machine availability assumption, we applied the Markov chain theory to the analysis. We showed how the service availabilities could be determined through the formulation of the problem into a CTMC. We also showed that the service availabilities are policy dependent: different policies generate different CTMC graphs, which lead to different results. We then proceeded to derive the optimal policy. By formulating the problem into an infinite-horizon MDP, we derived the optimal policy through policy iteration. Two *greedy* secondary machine assignment schemes were presented, and we showed that in order to achieve a high service availability, we have to assign the services such that the FDL of the machines are as *close* as possible. We then presented an online implementation of the optimal policy by using the RL/NDP technique. Experiments were conducted to evaluate the performance of the various assignment policies. We showed that the *least load* policy works very well in the TLL1 case, but not in the TLG1 case. The *FDL-based* scheme displayed a close-to-optimal performance in both TLL1 and TLG1 cases. We discussed the *DIP* effect and proposed a simple method to avoid it. The effect of the reward vector used in the generation of the optimal policy on the distribution of service availabilities was also discussed. Finally, we verified the numerical results with simulations and discussed the convergence of the RL/NDP secondary machine assignment scheme.

References

[1] D. Scott, "NSM: Often the Weakest Link in Business Availability," http://www.gartner.com/DisplayDocument?id=334197, July 2001.
[2] M. Loney, "The Magic That Makes Google Tick," http://www.zdnet.com.au/insight/software/0,39023769,39168647,00.htm, Dec. 2004.
[3] *The Grid: Blueprint for a New Computing Infrastructure,* I. Foster and C. Kesselman, eds. Morgan Kaufmann, July 1999.
[4] Y.S. Dai and G. Levitin, "Reliability and Performance of Tree-Structured Grid Services," *IEEE Trans. Reliability,* vol. 55, pp. 337-349, June 2006.
[5] K. Trivedi, *Probability and Statistics with Reliability, Queuing and Computer Science Applications.* John Wiley & Sons, 2001.
[6] A. Sathaye, S. Ramani, and K. Trivedi, "Availability Models in Practice," *Proc. Int'l Workshop Fault-Tolerant Control and Computing (FTCC-1),* May 2000.
[7] Y.S. Dai, M. Xie, K.L. Poh, and G.Q. Liu, "A Study of Service Reliability and Availability for Distributed Systems," *Reliability Eng. and System Safety,* vol. 79, pp. 103-112, Jan. 2003.
[8] G. Ciardo, K.S. Trivedi, and J.K. Muppala, "SPNP: Stochastic Petri Net Package," *Proc. Third Int'l Workshop Petri Nets and Performance Models (PNPM '89),* Dec. 1989.
[9] K. Trivedi and C. Hirel, "Sharpe—Symbolic Hierarchical Automated Reliability and Performance Evaluator," http://amod.ee.duke.edu/software_packages.htm, Dec. 2004.
[10] K. Iyer, E. Butner, and E.J. McCluskey, "An Exponential Failure/Load Relationship: Results of a Multi-Computer Statistical Study," Technical Report CSL-TR-81-214, Computer Systems Laboratory, Stanford Univ., July 1981.
[11] B. Schroeder and G.A. Gibson, "A Large-Scale Study of Failures in High-Performance Computing Systems," *Proc. Int'l Conf. Dependable Systems and Networks (DSN '06),* June 2006.
[12] D. Heimann, N. Mittal, and K.S. Trivedi, "Availability and Reliability Modeling for Computer Systems," *Advances in Computers,* M. Yovitts, ed., vol. 31, pp. 175-233. Academic Press, 1990.
[13] R. Robinson and A. Polozoff, "IBM WebSphere Developer Technical J.: Planning for Availability in the Enterprise," http://www-128.ibm.com/developerworks/websphere/techjournal/0312_polozoff/polozoff.html, Oct. 2003.
[14] J. Tian, S. Rudraraju, and Z. Li, "Evaluating Web Software Reliability Based on Workload and Failure Data Extracted from Server Logs," *IEEE Trans. Software Eng.,* vol. 30, no. 11, pp. 754-769, Nov. 2004.
[15] R.K. Iyer and D.J. Rossetti, "A Statistical Load Dependency Model for CPU Errors at SLAC," *Proc. 12th Int'l Symp. Fault-Tolerant Computing (FTCS-12),* pp. 363-372, June 1982.
[16] K. Vaidyanathan and K.S. Trivedi, "A Measurement-Based Model for Estimation of Resource Exhaustion in Operational Software Systems," *Proc. 10th Int'l Symp. Software Reliability Eng. (ISSRE '99),* 1999.
[17] "IBM DB2 V7 Administration Guide Part 12 Chapter 35: DB2 and High Availability on SUN Cluster 2.2," http://publib.boulder.ibm.com/infocenter/db2v7luw/topic/com.ibm.db2v7.doc/db2d0/db2d0273.htm, 2001.
[18] "Linux-HA Heartbeat Program," http://www.linux-ha.org/HeartbeatProgram, 1999.
[19] D.A. Patterson, G.A. Gibson, and R.H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '88),* June 1988.
[20] A. Heddaya and A. Helal, "Reliability, Availability, Dependability and Performability: A User-Centered View," technical report, Boston Univ., 1997.

[21] K. Nagaraja, G. Gama, R. Bianchini, R.P. Martin, W. Meira Jr., and T.D. Nguyen, "Quantifying the Performability of Cluster-Based Services," *IEEE Trans. Parallel and Distributed Systems,* vol. 16, no. 5, pp. 456-467, May 2005.

[22] D. Bertsekas and R. Gallager, *Data Networks,* second ed. Prentice Hall, 1992.

[23] W. Press, B. Flannery, S. Teukolsky, and W. Vetterling, *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge Univ. Press, 2002.

[24] R.S. Sutton and A.G. Barto, *Reinforcement Learning—An Introduction.* MIT Press, 1998.

[25] M.L. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming.* John Wiley & Sons, 1994.

[26] B. Van Roy, "Neuro-Dynamic Programming: Overview and Recent Trends," *Handbook of Markov Decision Processes: Methods and Applications,* E. Feinberg and A. Shwartz, eds. Kluwer Academic Publishers, 2001.

[27] G. Tesauro, N.K. Jong, R. Das, and M.N. Bennani, "A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation," *Proc. Third Int'l Conf. Autonomic Computing (ICAC '06),* pp. 65-73, June 2006.

[28] J. Guo and L.N. Bhuyan, "Load Balancing in a Cluster-Based Web Server for Multimedia Applications," *IEEE Trans. Parallel and Distributed Systems,* vol. 17, no. 11, pp. 1321-1334, Nov. 2006.

[29] M. Adler, S. Chakrabarti, M. Mitzenmacher, and L. Rasmussen, "Parallel Randomized Load Balancing," *Proc. 27th Ann. ACM Symp. Theory of Computing (STOC '95),* pp. 238-247, 1995.

[30] B.A. Shirazi, A.R. Hurson, and K.M. Kavi, *Scheduling and Load Balancing in Parallel and Distributed Systems.* Wiley–IEEE CS Press, May 1995.

[31] Q. Zhang, A. Riska, W. Sun, E. Smirni, and G. Ciardo, "Workload-Aware Load Balancing for Clustered Web Servers," *IEEE Trans. Parallel and Distributed Systems,* vol. 16, no. 3, pp. 219-233, Mar. 2005.

[32] D.P. Bertsekas and J.N. Tsitsiklis, *Neuro-Dynamic Programming.* Athena Scientific, 1996.

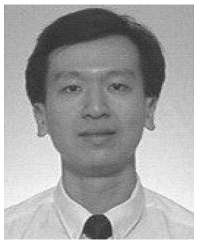
[33] L.P. Kaelbling, M.L. Littman, and A.P. Moore, "Reinforcement Learning: A Survey," *J. Artificial Intelligence Research,* vol. 4, pp. 237-285, 1996.

[34] *Service Availability Forum,* http://www.saforum.org, 2006

[35] S. Floyd and V. Jacobson, "Random Early Detection Gateways for Congestion Avoidance," *IEEE/ACM Trans. Networking,* vol. 1, pp. 397-413, Aug. 1993.

[36] M. MacDougall, *Simulating Computer Systems.* MIT Press, 1987.

Chee-Wei Ang received the BEng degree (first class honors) in electrical and electronic engineering from Nanyang Technological University, Singapore, and the MSc degree in electrical engineering from the National University of Singapore (NUS). He is currently a PhD candidate in the Department of Electrical and Computer Engineering (ECE), NUS. He is also a senior research officer in the Department of Network Technology, Institute for Infocomm Research, Singapore. His research interests include quality-of-service (QoS) control in distributed systems, HA clusters, and mesh networks.



Chen-Khong Tham received the MA and PhD degrees in electrical and information sciences engineering from the University of Cambridge, United Kingdom. He is an associate professor in the Department of Electrical and Computer Engineering (ECE), National University of Singapore (NUS). He is the supervisor of the Computer Networks and Distributed Systems (CNDS) Laboratory, Department of Electrical and Computer Engineering (ECE), NUS. His research interests include coordinated quality-of-service (QoS) management in wired and wireless computer networks and distributed systems, as well as distributed decision making and machine learning. In 2004, he was a visiting research fellow at the University of Melbourne under an Edward Clarence Dyason Universitas 21 Fellowship.