Reliability and Performance of Component Based Software Systems with Restarts, Retries, Reboots and Repairs

Vibhu Saujanya Sharma  
Dept. of Computer Science and Engineering,  
Indian Institute of Technology Kanpur,  
Kanpur, UP, INDIA 208016  
vsharma@cse.iitk.ac.in

Kishor S. Trivedi  
Dept. of Electrical and Computer Engineering,  
Duke University,  
Durham, NC 27708-0291, USA  
kst@ee.duke.edu

Abstract

High reliability and performance are vital for software systems handling diverse mission critical applications. Such software systems are usually component based and may possess multiple levels of fault recovery. A number of parameters, including the software architecture, behavior of individual components, underlying hardware, and the fault recovery measures, affect the behavior of such systems, and there is a need for an approach to study them. In this paper we present an integrated approach for modeling and analysis of component based systems with multiple levels of failures and fault recovery both at the software, as well as at the hardware level. The approach is useful to analyze attributes such as overall reliability, performance, and machine availabilities for such systems, wherein failures may happen at the software components, the operating system, or at the hardware, and corresponding restarts, retries, reboots or repairs are used for mitigation. Our approach encompasses Markov chain, and queuing network modeling, for estimating system reliability, machine availabilities and performance. The approach is helpful for designing and building better systems and also while improving existing systems.

1 Introduction

Software systems these days are being used in diverse fields and handle many mission and time critical jobs. It is important for such systems to be highly reliable and responsive. As these systems are mostly component based, important attributes like reliability and performance depend on the characteristics of the individual components, the way they interact with each other, and upon the underlying hardware infrastructure on which the components are deployed. Moreover, as failures can happen at the software components as well as the hardware, the way in which these failures are resolved, also has a direct bearing on the overall reliability and performance.

Failures at software components are usually resolved by rebooting their respective machines, and restarting the system. However this adversely affects the performance and also makes the system unavailable. Recent empirical studies [2, 3] show that successfully restarting just the software components (as opposed to rebooting the machines) is an effective way to handle transient software failures and increase system reliability, and simultaneously reduce the performance overhead. Other levels of fault recovery can also be present [27], and these affect the performance as well as the reliability of the system.

As the overall behavior of such complex component based systems depends on a number of different factors, modeling and analyzing such systems for attributes like reliability and performance has become important to ensure their efficient and sound operation. If such an analysis can be performed early in the software life-cycle, it can facilitate in making key decisions regarding the software design so that the final product performs better. Similarly, this activity is equally important for existing systems to help improve them. In general, questions such as these become pertinent, while studying such systems:

- How does the system perform if one or more software components are unreliable?
- How does unreliable underlying hardware affect the system, and where to improve?
- How do multiple fault recovery measures such as restarts, retries, reboots and repairs, affect the system reliability and performance?
- What are the various tradeoffs that exist?

Answering such questions, requires an approach that takes into account the software architecture and deployment
of the system, as well as allows for modeling multiple levels of fault recovery. In this paper we present such an approach, which aims at helping the software engineers build reliable and better performing component based systems.

The main contributions of this work are along the following four dimensions: (i) Extending the existing Discrete Time Markov Chain (DTMC) based reliability modeling techniques [8, 7, 21] to allow component level restarts and application retries and evaluate the increase in system reliability, (ii) Allowing for performance modeling for such systems having unreliable components and fault recovery techniques, (iii) Incorporating the effects of operating system and hardware level failures and corresponding reboots and repairs using a Continuous Time Markov Chain (CTMC) based hardware availability model, and (iv) Assessing the relationships between the performance, reliability and machine availability for such systems and study the variation in these attributes with changes in the system.

We discuss some related work in this field in the next section. In section 3, we introduce the modeling of component restarts and application retries by extending the conventional DTMC model for a component based systems as well as the associated performance model. In section 4, we present the incorporation of machine level reboots and repairs due to operating system and hardware failures, respectively. We apply the approach to an example system in section 5, and study the effect of changes in the system on various attributes. Subsequently, we conclude the paper.

2 Related work

The architecture of a software system greatly influences its performance and reliability. Thus architecture based evaluation has been an active area of research in the recent past. The most prevalent method for architecture based reliability assessment is based upon the work by Cheung [4] wherein a system is modeled using a Discrete Time Markov Chain (DTMC) [25] with one or more absorbing states. Some recent applications of the DTMC based approach for reliability prediction of component based systems, along with ways to ascertain the various parameters such as component reliabilities and transition probabilities, are given in [8, 7, 18]. Some of these approaches do calculate the average execution time of the application for a job, but they do not address the performance evaluation of the whole system under simultaneous load from a large number of clients, wherein resource contention causes queueing delays. There have been various formal models which are concerned only with performance modeling and evaluation also [22, 23, 14, 17, 20].

Empirical studies in [2, 3] have shown the effectiveness of component based restarts in software systems. However these studies are measurement-based and the authors do not perform analytical modeling of the software architecture and the hardware of the system under study. Thus these approaches cannot help in ascertaining the system behavior if some changes are made, neither can they help in predicting where and by what amount will successful component restarts be helpful, if at all. On the other hand, the existing analytical modeling approaches for reliability or performance modeling and analysis, as mentioned earlier, do not take into account the restarts that may happen at the component level, application level retries, and OS and machine level reboots (and hardware repairs).

A related work in [5] focuses on performance and availability modeling of a server with cold and warm replication mechanisms and analyzes the system for tradeoffs between availability and performance. The system under study however, is not component based and the software architecture of the system has not been incorporated in the approach. Another work on multiple levels of restarts is presented in [27], but this also does not consider the architecture of the software systems. Moreover it does not address performance analysis of such systems under different workloads.

In general, there is a need for an integrated modeling and evaluation approach that allows the study of the reliability and performance of such component based software systems with unreliable components wherein one or more levels of software restarts and retries are possible, while also incorporating machine repairs and reboots. This is the focus of this paper. We describe the details of our modeling and evaluation approach in the ensuing sections.

3 Modeling Component Based Systems with Restarts and Retries

Traditionally transient software component failures are handled by rebooting the machine on which the failed component is deployed upon. However, this is a rather inefficient way to handle such failures and newer systems are being designed, with many levels of fault recovery to minimize the need for such wasteful reboots, wherever possible.

When a failure is detected at a software component, first a restart of only that component could be tried [11, 3]. With some probability, this restart can turn out be successful and thus the failure may not reoccur upon restart. However if this is not the case, an application level retry can be attempted. In this case, after the component restart, the user request again traverses the software system from start (from the component that received the control flow first) as a normal request. One way of achieving this is by caching the user request till it successfully traverses through the whole system. The other and an easier way is by instructing the client/browser at the user’s end to automatically retry the request. Transparent component restarts and application level retries, enhance the overall reliability of the system. In both
the cases, the failure may not reoccur and thus the reliability of the system as perceived by the user, increases. However, note that a restart or a retry may not always be successful. We model such component based software systems using DTMCs and in this section we present our modeling approach for quantifying reliability and performance of such systems.

3.1 Creating the DTMC model

For the purpose of evaluation, component based systems have been modeled using DTMCs by [8, 21, 7, 18]. In such DTMC models (Figure 1), the state at any point in time is determined by the component in execution. Transitions between states are governed by the transfer of control from one component to the other. In such models, reaching an absorbing state indicates the successful completion of a user request. This is shown by the state $S$ in Figure 1. The information about the transition probabilities between components for existing systems can be found using profiling tools like gprof [6], ATOM [24], etc. as shown in [8]. For systems under design, data from similar previous systems could be used to estimate these values. Assume that each of the component has a unreliability $p_{fi}$ (or an equivalent reliability $r_i = 1 - p_{fi}$) attached with it. A composite reliability model to capture the overall unreliability of the system can be created for the software system as shown in Figure 2 [25]. This model has two absorbing states, with $S$ signifying the successful completion of the user request and $F$, signifying its failure.

Such DTMC models assume that a failure at any software component will cause overall failure. The overall system unreliability is thus computed from the steady state probability of the DTMC ending up in the failure state $F$ [25]. However, if we consider that a component level restarts and application level retries could be done, failure may not reoccur. Thus, if restarts and retries are allowed, the probability that the component failure will cause the whole system to fail will in general, be lesser.

Assume that the probability that if the component fails, the component is successfully restarted and the problem is resolved is $p_{cri}$. We denote the probability that a restart needs to be followed by an application level retry by $p_{ari}$. Also, $p_{mi} = 1 - (p_{cri} + p_{ari})$ is probability that the software component failure cannot be handled by either a local restart or a restart along with an application level retry.

We show the modified DTMC model for a software system with component restarts and application retries in Figure 3. One can create this modified DTMC model starting from the absorbing DTMC model specifying the architecture of a component based system as shown in Figure 1 as follows:

**Step 1:** For each state $C_i$, except the absorbing state:
- Add another state in the DTMC with the label $F_i$.
- Add an arc from state $C_i$ to $F_i$ with probability $p_{fi}$.
- Adjust the transition probabilities between components in the DTMC accordingly.

**Step 2:** From each state $F_i$, add a transition:
- To $C_i$ with probability $p_{cri}$.
- To $C_i$ with probability $p_{ari}$.
- To $F_i$ with probability $p_{mi} = 1 - (p_{cri} + p_{ari})$.

The model can be used to calculate the average visit counts [25, 21] $V_i, s$ and $V_i, F_i$ to various states of this DTMC and the steady-state probabilities of states $S$ and $F_i$. For a component $C_i$, $V_i, s$ represents the average number of times per job, this component be visited, before the completion of the job. Note that the extra states ($F_i, s$) are needed so that we can separately find out the average number of times ($V_i, F_i$) the restarts are done for each component $C_i$, (otherwise a self loop in each state $C_i$ would have sufficed). This facil-
initiates calculating the restart performance overhead for each component. The overall software reliability of this component based system (Figure 3), $R_s$, is given by the probability of system eventually ending in the state $S$, which can be calculated by solving the DTMC. This is the reliability of the software system as perceived by the user and will in general be more than the reliability for a system without any restarts or retries, as was modeled in Figure 2.

One should note the component failures that could be mitigated using restarts and retries are essentially transient in nature. These failures are caused by software errors which are classified as Mandelbugs [10]. One should note that the probabilities ($p_{mi}$ × $p_{fi}$) associated with each component signify the probability that a failure cannot be handled at the software component level. Such failures may be due to operating system level Mandelbugs, aging-related Mandelbugs or due to Bohrbugs [9]. In the latter case, restarts at any level may not help and the software code has to be debugged.

### 3.2 Creating the Performance Model

For creating the performance model of the system, we need the deployment information for the software components onto the available machines, the machine specifications and the individual average resource demands of the components on the hardware, and the client workload information. We make the assumption that the queuing model is a closed product form network where it is known that only the total service requirement at each resource is enough to determine the steady state measures; in particular, the topology of the network and individual service demands do not matter explicitly [25]. Hence for simplicity we assume a tandem topology for the network. For a system with $n_m$ machines, this model is a $(2n_m+1)$ stage tandem queueing network [25]. Each machine’s CPU and disk are modeled as First Come First Serve (FCFS) queueing stations in this network and the clients are modeled by an Infinite Server (IS) station [19]. Here the service rates of the stations representing the CPUs and disks are set as the inverse of the total average demands per job at that resource and the rate of the IS station is given by the inverse of the average thinktimes of the clients and the total number of jobs in the system is kept the same as the client population.

Using the visit counts, the average resource demands per execution and the resources used for each component’s failure detection and restart, we can find the total average resource demands per job for each component. Assume that for component $C_i$, $cpu$ and $disk$ are the average demands per visit, for CPU processing and disk I/O respectively. Let $rcpu_i$ and $rdisk_i$ be the respective average resource demands for the restart procedure and let $acpu_i$ and $adisk_i$ be the respective additional overheads for the application retry from the component. Recall that $p_{cri}$ is the probability that the failure is handled using a component restart, and $p_{ari}$ the probability that a restart is followed by an application retry at $C_i$. The total average resource requirements ($tcpu_i$ and $tdisk_i$) per job, of $C_i$ are given by:

$$
tcpu_i = V_i \times cpu_i + V_{F_i} \times [(p_{cri} + p_{ari}) \times rcpu_i + p_{ari} \times acpu_i]
$$

$$
tdisk_i = V_i \times disk_i + V_{F_i} \times [(p_{cri} + p_{ari}) \times rdisk_i + p_{ari} \times adisk_i]
$$

Note that as we assume that failures across components are independent, except for the component where the failure has been detected, other components need not incur the restart overheads. Depending upon the deployment of the components, the total average resource demands for the various machines can also be calculated, which in turn decide the service rates for the nodes of the performance model. For any machine $j$ with the set $L(j)$ of software components allocated to it, the total average CPU and disk service requirements are given by the aggregates:

$$
tcpu{j} = \sum_{i \in L(j)} tcpu_i
$$

$$
tdisk{j} = \sum_{i \in L(j)} tdisk_i
$$

We map the total average demands on the CPUs and disks onto a queueing model of the system as shown in Figure 4. For simplicity, we have not included the connectors between the machines in this model and are assuming that the connectors have sufficient capacity and are reliable. These can also be incorporated as presented in [20]. Note that we can relax the constraint of using the product-form network above in two possible ways: using the Whitt’s
QNA approach to numerically solve non-product form network or resort to discrete event simulation [28, 1]. The

solution gives us the system performance in presence of software failures and restarts. The throughput \( (\lambda_s) \) of this queueing model along with the system reliability \( (R_s) \) computed from the DTMC model is then used to compute the rate of the correctly completed requests, also called goodput, \( \lambda_s \) as:

\[
\lambda_s = \lambda_d \times R_s
\]

Next we look at incorporating operating system level and hardware level failures and reboots/repairs in this model.

## 4 Incorporating Operating System and Hardware Level Reboots and Repairs

In this section we extend our approach further to incorporate failures that may happen at the operating system (OS) level as well as at machine hardware. Such failures and the corresponding mitigation, has a direct bearing on the corresponding machine’s availability as well as the overall system performance, and are thus important to model.

System failures may not always be due to problematic software components, but due to problems in the OS which is hosting the components. In such cases, restarts at software components would not be help. One common example of such failures is, when a problem in the OS of a particular machine renders all the applications running on the machine unresponsive. In all such cases, the problematic machine usually has to be shut down and rebooted. This causes the components deployed on that machine to be unavailable for a short period of time. We classify such failures as OS level transient failures which tend to go away if the machine is rebooted.

The availability of a machine \( j \) subject to such OS failures can be modeled as [25]:

\[
A_j = \frac{MTTF_{j,OS}}{MTTF_{j,OS} + MTTR_{j,OS}}
\]

Here \( MTTF_{j,OS} \) and \( MTTR_{j,OS} \) refer to the mean time to failure and mean time to reboot, respectively, for the machine running the OS. The mean time to reboot for the machine can be easily measured for each machine/OS pair but the mean time to failure is more elusive. There is some published data available but in any case, it can be argued that the mean time to OS failure is a function of the load on the system [26, 12]. The load on the system may be characterized the utilization of the server, the arrival rate and such other factors. Note that these values can be computed by solving the queueing model discussed in the last section.

Another interesting observation is that unavailability of a server reduces its service rate [16] and hence adversely affects the performance model. Thus we have an interdependence between the machine availabilities and rate at which jobs are serviced by the system. To deal with this interdependence, we used fixed-point iteration [15, 13]. Consider that the service rates of the CPU and disk of the \( j^{th} \) machine under consideration has been found to be \( s_{cj} \) and \( s_{dj} \) while constructing the performance model using the software architecture and the component deployment. We propose the following algorithm for finding the steady-state goodput \( (\lambda_s) \) of the system:

**Step 1:** For \( k = 0 \), choose initial values for \( \lambda_s^{(0)}, s_{cj}^{(0)}, s_{dj}^{(0)}, A_j^{(0)} \) as the ones assuming no failure \((1 \leq j \leq n_m)\).

**Step 2:** Until \( |\lambda_s^{(k)} - \lambda_s^{(k-1)}| < \epsilon \) repeat {

- Increment \( k \)
- Solve the individual machine availability models to get the new availability \( A_j^{(k)} \) for each machine.
- Adjust the service rates \( s_{cj}^{(k)}, s_{dj}^{(k)} \) for each machine as
  \[
  s_{cj}^{(k)} = s_{cj} \times A_j^{(k)}
  \]
  \[
  s_{dj}^{(k)} = s_{dj} \times A_j^{(k)}
  \]
- Solve the performance model for the new value of goodput, \( \lambda_s^{(k)} \).
- Re-adjust the \( MTTFs \) in each of the availability models accordingly.
}

This algorithm utilizes the results from the performance model described in the previous section and calculates the performance of the whole system, in presence of failures and restarts and retries in the software components as well as the OS. Note that in each iteration, this algorithm updates the rates of CPU and disk at each of the \( n_m \) machines that constitute the system. The other outputs are the updated individual availabilities of the machines.

In many cases, problems may arise in one or more hardware devices on the constituent machines causing the system to fail. In such cases, the system would no longer be available until the problematic device is repaired or replaced. This adversely affects the availability and thus the performance of the system. Using continuous time Markov chains or CTMCs, availability model for each machine can
be extended to account for such failures also. Let \( MTTF_h \) and \( MTTR_h \) refer to the mean time to failure and mean time to repair, respectively, for a particular machine hardware \( h \).

Figure 5 shows a CTMC model of a machine which can have hardware failures due to the CPU and disk as well as OS failures and the corresponding repairs and reboots. Note that the OS and/or hardware \( MTTFs \) may depend on the rate of arrival of jobs (and hence depend on \( \lambda \)), or may be specified values for the systems under consideration. It can also be argued that these \( MTTFs \) may be a function of the machine utilization as Iyer has mentioned in [12].

![Figure 5. CTMC model for a machine incorporating OS and hardware failures and corresponding reboots and repairs](image)

The solution of this CTMC model provides us with the steady-state probability of a machine being in the 'UP' state or the availability of that machine, and this value can then be used in the algorithm given previously. The overall performance of the system can then accordingly be found as before. Next we look at an example system and analyze its performance and reliability for different variations in the relevant parameters.

## 5 An Example

For illustrating the approach, we take the example system with the software architecture as given in [25], which is shown in Figure 3. This consists of 5 components, each of them having a reliability of 0.9. We assume that the software system is deployed onto 3 machines, each having a single CPU and disk. For the purpose of estimation of performance of the software system, we assume the values of average resource demands, reliabilities, and the deployment as shown in Table 1.

<table>
<thead>
<tr>
<th>Component Number</th>
<th>Deployed on</th>
<th>Reliability</th>
<th>Demands/visit (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>Mc1</td>
<td>0.9</td>
<td>0.0050 0.0025</td>
</tr>
<tr>
<td>C2</td>
<td>Mc2</td>
<td>0.9</td>
<td>0.0025 0.0025</td>
</tr>
<tr>
<td>C3</td>
<td>Mc1</td>
<td>0.9</td>
<td>0.0050 0.0050</td>
</tr>
<tr>
<td>C4</td>
<td>Mc2</td>
<td>0.9</td>
<td>0.0025 0.0050</td>
</tr>
<tr>
<td>C5</td>
<td>Mc3</td>
<td>0.9</td>
<td>0.0075 0.0050</td>
</tr>
</tbody>
</table>

We have implemented our approach as a set of scripts and C programs. The implemented tool provides a command line interface to specify data files that describe the system. For the solution of the queueing model, the tool uses SHARPE [19] which is a versatile package for analyzing performance, reliability and performability models. The tool outputs the measures of interest including individual visit counts to the various DTMC states, the average component resource demands for a job (using the per visit demands), the restart overheads accrued for each component, the overall software reliability, the individual availabilities of the machines and the goodput of the system. Note that all the models used, can also be created and solved using the newer version of SHARPE.

For this example, we consider that the system is serving 100 clients each with an average thinktime of 1 second for constructing and solving the queueing model. Initially we assume that the machines and the OS are failure free. Next we assign the probabilities of successful component restarts and application retries and found the effect on the goodput of the system.

In all the configurations, we found that increasing the probability of successful restart or retry increases the overall reliability of the system, as shown in Figure 6, but at the same time the throughput as given by the queueing model solution, went down. This was due to two reasons. Firstly a component restart as well as a application retry increases the average number of visits to the different components. This is due to the fact that these actions causes a portion of the requests, that were previously going to the failure state, to go back to the transient states in the DTMC, thus increasing the average visits to a certain part of the system. This causes the total load on the CPU and disk by the components to increase, thus adversely affecting the throughput. Secondly, a component restart incurs some overhead, both at the CPU as well as the disk. Each time a component restart or the restart along with application retry is done, these resources are used and thus add up to the total average resource demands of the component(s). Thus the throughput decreases.

However the variation in the measure of interest for performance, the system goodput (which is the product of
throughput and reliability), was not monotonic. A higher degree of successful software restarts and retries, do increase the reliability, but at the same time may decrease the goodput (because of the increased load). This observation points to the fact that there may exist a tradeoff between the performance and reliability for such systems. While for a system being used for mission critical jobs, a configuration with increased reliability is desirable, for a transaction processing system, requiring high goodput, this may not be the best configuration.

For the following study, we used four configurations, with everything else being the same, but with different restart overheads of 0.005 sec, 0.01 sec, 0.025 sec and 0.05 sec for the CPU as well as the disk I/O and no additional application retry overheads. We refer to these as overheads O1, O2, O3, and O4 respectively. We then plotted the system goodput and varied the probability of restarts at all components. Figure 7 shows that for lower overheads, the increase in successful restarts at component C5 provides a higher increase the goodput, but as the overheads are increased, the trend is no longer monotonic. One can notice that for higher overheads, the goodput increases initially, but then decreases as the restart probabilities are increased further. This is because after the restart probabilities reach 0.6 for for O3 and 0.2 for O4, the increase in software reliability no longer compensates for the decrease in throughput due to increased overheads. Thus one can obtain a optimum restart probability value for the system in these cases by using our approach.

We found a similar trend in the case when the system had component level restarts followed by application retries. However as in the case of an application retry, as the ‘retried’ request traverses again through the software components from the start, the overall extra load on the system was found to be more in general. This resulted in lower goodputs than the case where only restarts were allowed. Moreover the goodput curves in this case were again found to have a distinct maxima, corresponding to the optimum application retry probability in each of the four overhead configurations.

Note that it may not be possible that all components in the system have the restart and subsequent application retry capability, as this requires an added effort while designing and implementing the system. If only certain components can posses such capability, the ones which highest influence on improving the performance and/or reliability of the system, should be targeted. We examined this aspect by allowing the restarts only at individual components. The results are shown in Figure 8 for the configuration with overhead O1. It can be seen that an increase in successful restarts at component C5 provides a higher increase the goodput than other components. This is due to the fact that in this example system, C5 has a large visit count and thus making this component more reliable causes a big increase in overall reliability. Also, the requests arriving here do not traverse any other transient component thereafter, so any increase
in restarts here only adds up to the overheads at this component. This is unlike other transient components, where an increase in restarts causes the increase in visits to other components thus adversely affecting performance.

We found that the C5 continues to provide the highest increase in goodput for an increase in the probability of successful restarts even when the restart overheads were increased to O4. This is shown in Figure 9. But the amount of increase is much smaller in this case. Also not that the restarts at C1 are actually not of much use in this case, and cause the system to degrade. So is the case with C3.

We also tried a similar approach for application retries. As an application retry is preceded by component restart, in general the performance was lower than the case if restarts alone were done, however the trends were more-or-less similar. Note that till now, we have assumed that only software component level failures are present. However as we discussed earlier, there may be failures at the OS and the hardware level too. We address this situation next.

We consider the software system without any software component restarts or application retries (i.e. as in Figure 2). We consider that the machines are identical and that a failure will render them unavailable till repaired. We assume that the probability that an arriving request will cause a failure to be 0.000005. Thus the $MTTF$ of each machine will depend on the system goodput ($\lambda_s$) as:

$$MTTF = 1/(\lambda_s \times 0.000005)$$

We varied the $MTTR$ of the machines from 1hr to 10mins, thus stepwise increasing their repair rate from $0.000278\text{s}^{-1}$ to $0.001667\text{s}^{-1}$. We fixed all the machines to have the $MTTR$ of 1hr, and then changed this for a single machine only at-a-time, in decrements of 10mins, keeping the other machines’ $MTTR$s fixed at 1hr. We then applied our approach to find the variation in system goodput.

Figure 10 shows that if only one machine’s repair rate has to be improved, only changes at Machine 1 are useful. This is due to the fact that Machine 1 has the most heavily loaded CPU, and this being the bottleneck of the system, limits the system goodput. When changes to the $MTTR$s of Machine 2 or 3 are done, they affect the availability only of those machines, and as the availability of Machine 1 remains untouched, the goodput remains almost the same. However when Machine 1’s $MTTR$ decreases, the increase in its repair rate, increases its availability and causes the bottleneck to reduce. This increases the overall goodput.

For this example system, we have presented only some of the many possible parameter combinations and variations, for the purpose of illustrating our approach. The approach itself is general in nature and in no way restricted only to the specific analysis we presented in this section. For example, one could use the approach to find out how does the system perform with varying probability of successful restarts if restarts at only two components are possible and only one machine is unreliable. The effects of change in software architecture (as partly studied in [21]), can also be ascertained using this approach.

6 Conclusion

Software systems these days are no longer monolithic and their performance and reliability is affected by the individual attributes of their constituent components, their software architecture and the underlying hardware. Due to the nature of such systems, failures can occur at software components, the operating system or at the machine hardware and thus adversely affect the reliability of the system and the way the system performs. Thus, systems are being built with multiple levels of fault recovery such as restarts and retries at the software level, and reboots and repairs at the machine level.
Existing modeling approaches are not rich enough to model such systems. These approaches usually tend to focus on individual attributes like performance [22, 23, 14, 17, 20], reliability [8, 7, 18] etc., and do not take into account the multiple levels of fault recovery, thus ignoring the effects of the various restarts, retries, reboots or repairs that are used for mitigating failures at various levels.

In this paper, we presented a integrated modeling approach suitable for modeling such component based systems, which can be used for ascertaining attributes like software reliability, system performance, machine availabilities, etc. The approach first models the software system with fault recovery features like component restarts and application retries, using a DTMC model. This model is then solved for finding the overall software reliability as well as finding the parameters for the performance model for the system. A queueing network is used for modeling the performance, and it is parameterized using the hardware and workload specifications, the component deployment and their average resource demands per execution. The presence of failures at the operating system level and at the hardware devices and the corresponding reboots and repairs are also incorporated. Such failures and repairs affect the availability of the individual machines. CTMC models are used to model the availability of various machines in the system. As the parameters of the CTMC and the queueing model are interdependent, a fixed-point iteration is done on the average throughput of the system.

Our approach takes into account a large number of system parameters relating to the software architecture, non-functional attributes of the components, fault recovery strategies, and the underlying hardware, and can be used to find out the effect of changes in these, on the overall reliability and performance of the system. The approach can be used while designing new systems, wherein a large number of alternative system architectures, fault recovery strategies, and deployments can be compared and chosen from. As the approach ascertains both the reliability and performance of the system, various tradeoffs can also be found. The approach can also be applied for modeling existing systems which need to be improved or upgraded. Various changes and additions for improvements can be evaluated to find the ones that are the most effective.

We illustrated the usefulness of the approach by an example. We found that different probabilities of restarts and retries influence the reliability as well as the goodput. The goodput was found to be very sensitive to the overheads that the system incurs for performing component restarts. We found that there indeed may exist tradeoffs between the overall reliability and the performance of such systems. The change in average goodput with varying machine repair rates was also studied.

One direction we are working in is to extend the approach to automatically suggest alternatives to optimize the system for performance and reliability. As we have shown that tradeoffs may exist between the two attributes and it is important to chose a system configuration that optimizes these. This would require forming an optimization problem, where the inputs would be the various software and hardware parameters and different fault recovery strategies, and possibly the constraints on which of these can be varied.

We have currently not included costs in this model. Appropriate costs can be associated with various components, hardware configurations and fault recovery strategies and the approach can be extended to also output the costs of different system configurations. This will be helpful to software engineers and designers to decide the best system, while staying within the budgets. For existing systems, this would allow the engineers to optimize the upgrade process for choosing from among a large number of alternatives to improve the system. This is another avenue for future work.

References


