Stochastic Modeling of Modern Storage Systems

by

Ruofan Xia

Department of Computer Science
Duke University

Date: __________________________

Approved:

____

Kishor S. Trivedi, Supervisor

____

John Board

____

Xiaobai Sun

____

Krishnendu Chakrabarty

Dissertation submitted in partial fulfillment of
the requirements for the degree of Doctor
of Philosophy in the Department of
Computer Science in the Graduate School
of Duke University

2015

ABSTRACT

Stochastic Modeling of Modern Storage Systems

by

Ruofan Xia

Department of Computer Science
Duke University

Date: ________________________
Approved:

____
Kishor S. Trivedi, Supervisor

____
John Board

____
Xiaobai Sun

____
Krishnendu Chakrabarty

An abstract of a dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science in the Graduate School of Duke University

2015

Abstract

Storage systems play a vital part in modern IT systems. As the volume of data grows explosively and greater requirements on storage performance and reliability are put forward, effective and efficient design and operation of storage systems become increasingly complicated.

Such efforts would benefit significantly from the availability of quantitative analysis techniques that facilitate comparison of different system designs and configurations and provide projection of system behavior under potential operational scenarios. The techniques should be able to capture the system details that are relevant to the system measures of interest with adequate accuracy, and they should allow efficient solution so that they can be employed for multiple scenarios and for dynamic system reconfiguration.

This dissertation develops a set of quantitative analysis methods for modern storage systems using stochastic modeling techniques. The presented models cover several of the most prevalent storage technologies, including RAID, cloud storage and replicated storage, and investigate some major issues in modern storage systems, such as storage capacity planning, provisioning and backup planning. Quantitative investigation on important system measures such as reliability, availability and performance is conducted. Towards this goal, a variety of modeling formalisms and solution methods, such as analytical technique, simulative solutions and Markov Decision process optimization method, are employed based on the matching of the underlying model assumptions and the nature of the system aspects being studied. One of the primary focuses of the model development is on solution efficiency and scalability of the models to large systems. The accuracy of the developed models are validated through extensive simulation.

Contents

List of Tables

List of Figures

Acknowledgements

I would like to express my deepest gratitude to my advisor, Dr. Kishor S. Trivedi, who has always been a significant source of guidance and support for me during my time at Duke. Dr. Trivedi is a man of great wisdom and intellectual capability, the degree of which is only matched by his patience towards those under his advising and the kind of inspiration he gives them. His guidance and assistance are instrumental in my overcoming the many difficulties in a PhD career, my eventual completion of the degree and my acquisition of the valuable research skills and knowledge. It suffices to say that the experience of studying and performing research with him has significantly shaped my future.

I am greatly thankful to my other committee members, Dr. John A. Board, Dr. Xiaobai Sun and Dr. Krishnendu Chakrabarty, for their guidance and kindness during my PhD career. I would not forget the interactions that I had with them, the advice and support that I received during these sessions, and the insights and inspirations I would ultimately draw from such rich experiences. Their assistance is one of the reasons that I was able to surmount the obstacles on the PhD path to reach the final destination.

I would also like to thank the many people I had the opportunity to work with for the help they gave me. I would like to express my gratitude to Dr. Andrew Rindos of IBM RTP, who facilitates close collaboration with IBM which became a valuable experience in my PhD career. Dr. Rindos is also a great source of insights and advice, and I benefited a lot from some very interesting and constructive discussions with him. I would also like to say thanks to Dr. Javier Alonso for the many engaging and beneficial discussions I had with him during his time at Duke, which helps me immensely with the progress of my study. Finally, I would like to thank Mr. Fumio Machida of NEC Japan, with whom I had the honor of over three years' collaboration, for the ideas he shared with me based on his many years' experience in the industry. I also had the opportunity to undertake an internship in Japan, hosted by Mr. Machida, which greatly enriched my PhD experience and for which I am deeply grateful.

My thanks also go to the former and current members of the research group, Dongseong Kim, Rahul Ghosh, Xiaoyan Yin, Stefano Sebastio, Zheng Zheng, Harish Sukhwani and Rafael Ricardo. I learned a lot from my interaction with them and enjoyed very much the friendly environment they created.

Lastly I would express the greatest gratitude to my parents, Luchuan Xia and Jinya Luo, not only for their support during my pursuit of the PhD degree but also for bringing me to this world and shaping my growth with love and patience, so that the I of today came into being with much experience learned from the past and much more to explore in the future. As I stand now at the end of my PhD career and ponder about the trajectory of my life so far, I can't help but deeply thanking (but would never be able to do this enough) them for being the wonderful parents they have always been.

I offer special thanks for the generous financial support for my research work from Duke University Graduate School and Computer Science Department, NEC Corporation and IBM.

1. Introduction

1.1 An Overview of Modern Storage System Complexity

In modern IT systems, storage components are playing increasingly essential roles in facilitating daily operation in virtually any organization. With the onset of the “Big Data” era, storage operations today are faced with rapidly increasing amount of data generated by daily operations, and the complexity of storing such data in a reliable and cost-effective fashion is also quickly escalating. On the one hand, increasing public awareness about data security and privacy, together with a few high-profile data breach incidents, has caused stricter regulations on data protection to be established, and the consequence of losing important business/client data can be devastating to an organization. On the other hand, modern storage systems are also known for their complexity which, combined with their critical roles, means effective design and operation for such systems remain highly important but complex issues. Such complexity arises due to a variety of reasons, such as the sheer large number of devices running (and constantly failing/recovering) and the dependencies among various software/hardware components. As a result, significant expertise is typically needed to properly design and operate a modern storage environment. On top of these, the sheer amount of data and ever increasing demand for high performance and high availability in data access creates tension with limited IT budgets and results in a complicated issue of how to design and operate complex storage environments in a cost-effective manner.

To handle the new development in data volume and complexity, many novel techniques have been developed in the storage system field. While these techniques address many previous issues, they typically achieve so with different complexity of their own. A good example to illustrate the complexity of modern storage technique is the relative new field of cloud storage. While the rapid expansion and development of cloud-related technologies potentially leads to greatly reduced storage management complexities, it can also result in new operational challenges. For example, the adoption of public cloud storage services to meet the organization's storage needs could reduce operational overheads by removing (or reducing) the need to manage and maintain the hardware and/or software pieces. On the other hand, such a move could also reduce one's ability to finely control the storage system and lessen the transparency into operational details. In the case of a cloud storage owner, either one that provides public access or just uses a private cloud for internal purposes, such complexity increases manifold. One of the most important issues in such a context is planning of computation capacity, with minimizing operational cost and maintaining service level agreement (SLA) (e.g., for a public service provider) or providing adequate performance and dependability to internal users (e.g., for a private cloud owner) as the goals. Effective planning in these scenarios requires techniques or tools for predicting the overall cost and metrics related to SLA/performance, something that is nontrivial given the complicated operational dynamics in such storage systems, including the highly diverse usage patterns, frequent system component removal/addition due to the large system size, and the complex dependencies among components. Such complexity only increases further for a user who would like to harness the benefits of both public and private storage clouds by means of a hybrid-cloud setup.

To handle the complexity in storage system design and operation, it would be helpful if there are effective techniques to quantitatively analyze system measures of interest under different scenarios. Ideally, in the system design and implementation phase such techniques should allow different design choices, such as different system structures and optimization options, to be evaluated in terms of their effect on system behavior; while in the operation phase these techniques should facilitate analysis of system dynamics under different runtime configurations and varying workload mixture/usage patterns. Such techniques can be used to analyze a range of scenarios and make predictions on how the system would behave under each, and are important for the analysis of tradeoffs in system design/planning and essential to dynamic adjustment of system configurations during operation. In these respects, such techniques provide valuable assistance to efforts in effectively designing and operating a complex storage system.

1.2 An Overview of Quantitative System Analysis Methods

Broadly speaking, three major approaches exist for quantitatively analyzing system behavior:

- The first direction is to directly measure system runtime behavior and capture statistics of metrics of interest based on such measurement. The measurement may be obtained either from the actual system or from a system prototype (or a scaled-down version of the actual system) deployed in some controlled environment.
- The second direction is to construct a model consisting of the logical components and flows of the system and perform simulation to study system behavior.

Measurements on metrics of interest are taken in the form of simulation statistics, and prediction of the real system behavior is established from such statistics.

- The third direction is to abstract system characteristics relevant to the metrics of interest and describe these characteristics in mathematical terms, resulting in stochastic models that can provide estimates about system behavior.

It is important to note that the first approach (measurement) refers to directly obtaining system statistics from the measurement, instead of feeding the measurement to a simulation or stochastic model. In this respect it is one way to utilize measurement data, and does not imply that such measurement cannot or should not be used to improve simulation or stochastic modeling. Quite to the contrary, successful system prediction/analysis with simulation or stochastic modeling always benefit from availability of measurement data, either for more accurate parameterization or more detailed understanding of system dynamics.

A comparison of the relative merits and drawbacks of the approaches is in order.

- The greatest strength of the direct measurement approach comes from its high fidelity and level of details. Measuring the actual system gives, by definition, the highest accuracy possible. Even actually building a prototype (or a scaled-down version of the actual system) allows much internal details of the actual system to be included. In this case, if a suitably designed experimental environment is also in place to adequately reflect the actual system in operation environment, the result is again a highly accurate representation of the real system behavior. On the other hand, the downside of this approach lies in the large amount of effort required to realize such a potential. For a complex storage system, both direct measurement and building a sufficiently detailed prototype with a suitable operation environment are generally significant undertakings, if possible at all. Furthermore, gathering sufficient behavioral data can also be tricky if the metrics of interest only relate to system states that arise infrequently during operation. The need for such significant investment also makes these approaches unsuitable for exploring multiple major design choices and configuration options as doing so potentially requires re-implementing or heavily modifying the system and the environment to retain fidelity.

- Compared to the prototype approach, constructing a logical model for the system typically requires much less effort and investment. System details that have little impact on metrics of interest can be ignored, and subsystems that are well understood can be replaced with the suitable logical description of their behavior. Accuracy-wise, all the important system internal details can still be included in the simulation model, while the actual operation environment can be captured either through traces of system input or input synthesized from statistics of the real environment. The drawback of the logical modeling approach is that, while constructing the model requires much less effort, running the simulation to gather sufficient data for statistical analysis of the system behavior may not be so. Besides, the problem of gathering data about rare system events remains. Consequently using this approach could still face major challenges if a significant number of design/configuration variants are to be compared.

- Using the stochastic modeling approach has two main advantages. The first is that solution to the models, and hence estimates on system measures, can usually (but not always) be obtained with much reduced effort/time compared to the other two approaches. Moreover, if model construction is accurate, then quantitative evaluation of rare system events and their effect can still be efficiently performed, since in this case these are just part of the results given by the mathematics behind the models. As such the mathematical modeling approach can be very effective in evaluating and comparing a large number of system variants. The second advantage is that the resulting models facilitate formal optimization of the system. In some cases direct mathematical optimization can be carried out; otherwise numerical optimization can be performed which is made possible in part by the relative low cost of evaluating different system variants using the stochastic models. The disadvantages of stochastic modeling come from the fact that the model formalism in use will impose certain assumptions, typically to ensure a set of desired computational/analytical properties in the resulting mathematical expressions. For a specific modeling approach, depending on the system under investigation, the corresponding modeling assumptions can lead to two main issues: the first is inaccuracy since the real system behavior may not conform to such assumptions;

the second is inefficiency as the formalism chosen may not capture some system details in a concise and easy-to-analyze manner.

Given the above merits and drawbacks, it is worthwhile to point out that in practice the various quantitative approaches are usually employed in conjunction, and sometimes in an iterative fashion, to better capitalize on their respective advantages. For example, in the design process of a system, stochastic models are frequently used to quickly compare many alternatives and generate preliminary estimates on resulting system behavior. These "first order" values are useful in narrowing down the set of candidate options to some promising ones. Once this step is achieved, more detailed design evaluation can be carried out by simulation with stochastic models being potentially used to facilitate more efficient simulation execution. Finally, prototypes may be built and deployed to yield system evaluation results in a realistic environment, and measurements on either the prototype and/or the real system (once deployed) may be collected for improving the stochastic and simulation models.

The stochastic modeling approach covers a wide range of formalisms and techniques. As previously described, these methods all make some underlying assumptions, the nature of which can be very different across methods. As such, different formalisms can be appropriate for studying different systems and/or different metrics. A suitable formalism potentially makes the modeling process more accurate and efficient, while an inappropriate one could make the work more difficult. Therefore, when performing stochastic modeling it is important to be aware what assumptions are being made when using a particular formalism, check whether they are acceptable for the system and/or metric under study, and choose the formalism accordingly. As a result, it can be advantageous to apply multiple modeling formalisms in conjunction when studying a particular system, with subsets of techniques handling different system aspects. Meanwhile, for a given formalism there would still be various ways of using it to study the system metric in question, potentially with different effectiveness. All of these aspects combined together make stochastic modeling an approach that requires deliberation to use, but also one that can be very efficient, accurate and flexible if employed appropriately.

Due to the potential power and flexibility of stochastic modeling, in this thesis I will focus on using this approach to conduct quantitative studies of complex storage system behavior, with some use of simulation for comparison/validation purposes. The main goal is to investigate how to appropriately employ multiple modeling formalisms to efficiently and accurately study and optimize systems that are important for today's storage operations. Along the way I make several contributions and improvements over existing methods, as are described in the next section.

1.3 Contributions of the Dissertation

In the course of modeling the various storage systems, difficulties may arise due to the mismatch between the assumptions of particular modeling formalisms and the behavior of the system under study. To resolve such issues, it is frequently necessary to develop new ways of applying the formalism or using multiple formalisms in conjunction. Along this line, this dissertation makes the following contributions.

- Development of a detailed stochastic model for the availability and performability study of different Redundant Array of Independent Disks (RAID) configurations based on the Markov Regenerative Process (MRGP) formalism. The new model removes some of the classic exponential time distribution assumption used in system availability analysis, with the goal of quantify the potential error from such assumption. Detailed closed-form formulas are derived and comprehensive numerical investigation is performed to showcase the utility of the new approach while also providing some evidence that the traditional exponential-time-based models are still useful in some practical scenarios.

- Development of new approaches to handle the scalability and inaccuracy issues in previous models when applying Markov chain formalisms to study cloud storage system provisioning. The previous models attempted to solve the scalability issue in modeling a large cloud system through a simple decomposition approach that turns out to be inaccurate in some cases. In this dissertation I first adopt a new perspective to look at the cloud storage provisioning performance problem, which allows me to develop a much different model that is highly accurate. To make additional improvement on the scalability of this new model, I make further progress along two directions. The first is to use a hybrid analysis approach and jointly analyze the model with both numerical solution and simulation, which proves to have good solution efficiency and hence improved scalability. The second direction is to employ Markov Chain Monte Carlo (MCMC) method, where I prove the correctness of the method in this context and then develop the algorithm details for this problem. Comparison with the hybrid solution method shows that the MCMC method provides a useful alternative in solving the model that is complementary to simulation.

- Development of a simple state-space model for analyzing different data backup configurations and policies, in terms of their effect on system performance and data availability, in a storage system. The simplicity of the model facilitates its (re)use in system analysis, while accuracy of the model is validated through simulation.
- Development of a Markov Decision Process (MDP) framework for optimizing storage system backup planning. The framework optimizes storage availability while taking into account important business and system capacity constraints in the planning process. To address the so called "curse of dimensionality", i.e. an exponential growth in model size w.r.t. the problem size, that plagues exact MDP models in many practical situations (as in this one), I develop a simple but effective approximation approach that allows the overall planning problem to be decomposed into multiple smaller problems, each of which can be studied as an MDP planning problem separately. This approach significantly reduces the problem size and allows system of much larger size to be studied. Application of the resulting approximation scheme is carried out through a combination of stochastic model and simulation approaches, and effectiveness of the MDP-based planning is compared with that of several common planning heuristics and found to yield better results.
- Development of a detailed and comprehensive model for investigating the performance and availability of replicated storage systems. The model utilizes

multiple formalisms in conjunction to study different aspects of the system dynamics in the most appropriate fashion. To provide distributional information about system performance, which is generally not available from stochastic models of complex systems, I develop an approximation scheme based on queueing networks. Through validation with simulation, the approximation method proves to be highly effective at providing distributional information on system processing time of latency-sensitive tasks, and the availability of such distributional information allows system performance prediction to be carried out on task execution time quantiles, which could yield significant benefit for system capacity planning and runtime optimization.

1.4 Outline of the Dissertation

This dissertation is structured around the different aspects of modern storage systems that are investigated and the stochastic models constructed for them. Different formalisms and techniques are employed in different cases based on appropriateness of the model assumptions and efficiency of using the specific modeling methods to study the particular system under investigation.

Chapter 2 presents background information on several stochastic modeling formalisms used in this dissertation. I will first cover Markov chain models, several of its variants, and the related Markov reward modeling approach. These formalisms are useful for studying a system whose behavior can be described by a set of reasonably sized states (i.e. an example of "state-space models") and the dynamics of transitions among the states can have complicated dependence on the states. The Stochastic Reward Net (SRN) formalism is also introduced which is effective at assisting construction and analysis of a Markov reward model. Then I will describe queueing network models, which are non-state-space models and follow different assumptions about the nature of the state transitions. These different assumptions result in better efficiency at studying some important system performance metrics. Next I will describe Markov decision processes, which are a type of optimization formalism based on the Markov models and useful for planning and optimizing system operations. Finally I will briefly review a few sampling approaches that are useful in solving certain stochastic models in this dissertation, as well as simulation techniques.

Chapter 3 presents a set of stochastic models for comparing different Redundant Array of Independent Disks (RAID) configurations in terms of their availability and access performance. One of the foci of this chapter is using a Markov Regenerative Process (MRGP) model to avoid the potential inaccuracy in using a continuous-time Markov chain (CTMC) model. Detailed formulas for the MRGP model are derived, and its results are compared with those from corresponding CTMC models to illustrate the improvement in accuracy.

Chapter 4 presents a stochastic performance model for the provisioning aspect of a cloud storage system. The model builds upon previous work [1] to significantly improve the accuracy of the existing model, at the expense of some scalability gains. To compensate for the (slightly) reduced scalability, two additional approaches are presented. The first is a hybrid solution method that combines simulation and numerical solution, while the second is a sampling-based procedure. The accuracies and efficiencies of the two methods are then investigated and compared to that of the original model.

Chapter 5 presents a stochastic model for analyzing and comparing different data backup configurations and policies. The model is simple and easy to use, and its accuracy is established via comparison with simulation.

Chapter 6 describes an optimization framework based on Markov Decision Process (MDP). It then presents details on how to use this framework to optimize storage backup planning while satisfying important business and resource constraints. The chapter further describes a novel approximation scheme that allows the overall optimization problem to be decomposed into multiple smaller independent ones. The approximation is shown to be effective at generating backup plans that are competitive versus many commonly employed heuristics, while it also greatly improves the scalability of the framework that tends to be one of the greatest hurdles in applying MDP-based techniques in practice.

Chapter 7 presents a detailed and comprehensive model that investigates the reliability, availability and performance aspects of replicated storage systems. Clear discussions about the assumptions of different modeling formalisms and their applicability in the various system components are provided as the basis of employing the various formalisms, and the result is a model that is accurate in its study of different system parts. The chapter also describes a novel approximation scheme that allows distributional information on the performance of some classes of tasks to be obtained very accurately, potentially providing great utility to system operation and planning in practice.

Finally, chapter 8 provides summary of this dissertation and some discussion regarding the insights gained from applying stochastic modeling to all the storage system aspects that are investigated.

2. Background on Stochastic Modeling Formalisms

This chapter provides background information about the stochastic modeling formalisms that are used in this dissertation.

2.1 Markov Chain and Related Formalisms

2.1.1 Markov Chains

A Markov chain is a stochastic process defined on a discrete state space that satisfies the property that, given the state the process is in at the current time, the future evolution of the process would not depend on any history of the process beyond the current state and time. Mathematically speaking, a discrete-state stochastic process $\{X(t)|t\epsilon T\}$ is a Markov chain if, for any sequence of time points $t_0<t_1<t_2<\ldots<t_n<t$, the conditional distribution of $X(t)$ depends only on the process state at the last time epoch, i.e.,

$$P(X(t) \le x|X(t_n) = x_n, X(t_{n-1}) = x_{n-1}, \cdots X(t_0) = x_0) =$$
$$(X(t) \le x|X(t_n) = x_n)\mathrm{P}\left(\mathrm{X(t)} \le \mathrm{x}\middle|\mathrm{X(t_n)} = \mathrm{x_n}, \mathrm{X}\left(\mathrm{t_{n\text{-}1}}\right) = \mathrm{x_{n\text{-}1}}, \cdots \mathrm{X(t_0)} = \mathrm{x_0}\right) = \qquad (2.1)$$
$$\mathrm{P(X(t)} \le \mathrm{x|X(t_n)} = \mathrm{x_n})$$

This property is the Markov property, and can be interpreted as “the future depends on the past only through the present”. It is a very important and useful property in applying stochastic processes since it allows a significant simplification of the analysis without taking some rarely valid assumptions, e.g. an independent process.

In the above definition the “present” including both the state and time, so that the same system state at different times could still lead to different future. A further simplification of the process is the notion of "homogeneity", where the dependence on current time is also removed, i.e.,

$$P(X(t+\tau) \leq x | X(t_n+\tau) = x_n) = P(X(t) \leq x | X(t_n) = x_n), \forall \tau \geq 0 \tag{2.2}$$

This thesis will only make use of homogeneous Markov chains.

The time dimension of Markov chains can be either continuous or discrete, resulting in continuous-time Markov chains (CTMC) or discrete-time Markov chains (DTMC) respectively. Also, the number of states can be either finite or infinite. Finite Markov chains have some nice analytical properties facilitating easier analysis, and are generally adequate to capture the dynamics of storage system dynamics that arise in practice. Therefore this thesis will only consider finite Markov chains.

2.1.1.1 Parameterization and Solution

Both DTMC and CTMC may be fully described by a matrix which specifies the probabilities of making specific transitions.

- For a DTMC, the matrix is termed a "transition probability matrix" and is parameterized by the probabilities of the chain going into a particular state at the next time point from the current state. Note that the "new" state at the next time point may be the same as the one the chain is currently in.
- In the case of a CTMC, the matrix is called an "(infinitesimal) generator matrix" and represents the limiting ratio between the probability of going into a new state at a future time point and the distance to that point, when that "future point" becomes infinitely close to the current one. These limiting probabilities are usually called the

"rates" of going from a current state to the new states, with the "rate" of "not leaving the current state" equals the negative of the sums of all the outgoing rates. The generator matrix is then parameterized by such rates.

A Markov chain may be either analyzed for its steady-state/stationary measures, which cover its long-term behavior, and for transient measures which reflect short-term behavior. The basic metrics are the stationary/transient state probabilities. The stationary probabilities would exist under certain structural requirements on the chain, and can be computed via solving a set of linear equations defined by,

$$\begin{aligned} &\text{For DTMC: } \boldsymbol{v} = \boldsymbol{vP} \\ &\text{For CTMC: } \boldsymbol{0} = \boldsymbol{\pi Q} \end{aligned} \tag{2.3}$$

where $\boldsymbol{\pi}$ stands for the stationary probability vector, $\boldsymbol{P}$ the transition probability matrix, and $\boldsymbol{Q}$ the generator matrix. For transient analysis, a DTMC can be solved simply by multiplying the initial probability vector with the transition matrix a number of times (equal to the number of time steps desired), while a CTMC can be solved using methods such as solving ordinary differential equations or uniformization [2][3].

2.1.2 Semi-Markov Chain

Given that in both CTMC and DTMC the Markov property is satisfied at each time point, a natural variant of Markov chain arises in the form of a discrete-state continuous-time stochastic process that satisfies the Markov property at a countable subset of the time horizon. If the Markov property is satisfied at each time when the chain makes a state change, the resulting stochastic process is called a semi-Markov chain (SMC). Mathematically, a discrete-state continuous-time stochastic process is a semi-Markov chain if it satisfies the following two conditions: 1) there exist a sequence of time points $\{T_i\}$, $i \in \mathbf{N}$, such that the Markov property is satisfied at all these points; and 2) for any $t \in [T_i, T_{i+1})$, $X(t) = X(T_i)$.

As the formal definition of SMC suggests, the main distinction of SMC compared to CTMC is that the elapsed time between neighboring state change points follows an arbitrary distribution instead of an exponential one, the latter mandated by the stronger Markov property of CTMC. This “arbitrariness” in candidate distributions is also the main advantage of SMC compared to CTMC, as it allows an SMC model to better describe system dynamics that do not fit into the “memoryless” property of CTMC. On the other hand, the price to pay is a relatively more difficult analysis procedure for SMC, especially for transient analysis.

2.1.2.1SMC Parameterization

The fundamental way to parameterize an SMP is by defining the firing time distribution of each transition. However, in certain cases such information is not necessary. Two cases that are of relevance to this thesis are:

- If only stationary state measures are needed;
- If the SMP is absorbing and only probabilities of and mean time to termination in the various absorbing states are of concern.

In such cases the only information necessary to solve the SMP and obtain the desired measures are the probabilities of transitioning from current states to new states, as well as the expected times until the transitions happen. Then there are two ways to parameterize an SMP:

1) Still parameterize each transition with its firing time distributions; the transition probabilities and mean sojourn time in a state are then computed from such distributions of all the transitions enabled in the current state, via:

$$
\begin{aligned}
P_{ij} &= \int_{t=0}^{\infty} f_{ij}(t) \prod\nolimits_{k\neq j} [1 - F_{ik}(t)]\, dt \\
ET(i) &= \sum\nolimits_{j} \left\{ \int_{t=0}^{\infty} t f_{ij}(t) \prod\nolimits_{k\neq j} [1 - F_{ik}(t)]\, dt \right\}
\end{aligned}
\tag{2.4}
$$

where $f_{ij}(t)$ and $F_{ij}(t)$ are the density and distribution functions of the firing time of a given transition from state *i* to state *j* respectively.

2) Parameterize a transition with its probability of firing in the current state as well as the mean time until its firing. The mean sojourn time of the state can then be computed as the weighted sum of transition-conditional mean times, via:

$$
ET(i) = \sum\nolimits_{j} ET(j|i) \cdot P_{ij} \tag{2.5}
$$

where $ET(j|i)$ and P_{ij} are the probability and expected time of going from state *i* into state *j* respectively.

While the second approach can ultimately be derived from the first one if the distribution information is available, sometimes it is simpler to follow this approach, and it would also work even if the full transition distributions are difficult to obtain/derive.

2.1.2.2 Solving an SMC

Regarding the solution method of SMC, a key concept is that of the transition kernel, which defines the probability that the chain transitions from state *i* to state *j* at or before time *t*:

$$K_{ij}(t) = \int_0^t f_{ij}(\alpha) \prod\nolimits_{k \neq j} [1 - F_{ik}(\alpha)]\, d\alpha \tag{2.6}$$

Then if we define $V_{ij}(t)$ as the probability that the chain is in state *j* at time *t* when it was in state *i* at time zero, i.e.,

$$V_{ij}(t) = P(X(t) = j | X(0) = i), t \geq 0 \tag{2.7}$$

It can be shown [4] that $\{V_{ij}(t)\}$ satisfy:

$$V_{ij}(t) = \left[1 - \sum\nolimits_j K_{ij}(t)\right] \delta_{ij} + \sum\nolimits_k \int_0^t K_{ik}(\alpha) V_{jk}(t - \alpha) d\alpha \tag{2.8}$$

Equation 2.8 is essentially an application of the theorem of total probability, and can be solved either in the Laplace transform domain or the time domain [5]. Solving this equation is mandatory if transient analysis of an SMC is to be conducted. However, in some cases (such as the two in the previous subsection) solving this integral equation is unnecessary:

- If only stationary state probabilities of the SMC are needed, then the solution can proceed in the following steps:
  1) Obtain the quantities in (2.4). These can be computed in a fashion similar to the SMC kernel entries.
  2) The $\{P_{ij}\}$ terms are transition probabilities among states and together specify a DTMC, known as the "embedded DTMC" of the SMC. Solving this DTMC for its stationary state probability gives the (normalized) frequency of visiting the different SMC/DTMC states. Denote these as $\{v_i\}$.

3) The {*ET*(*i*)} terms are the expected sojourn time in state *i* once the state is visited. Then from the definition of stationary state probabilities, these probabilities of the target SMC can be obtained as:

$$\pi_i = \frac{v_i ET(i)}{\sum_i v_i ET(i)} \tag{2.9}$$

- If the SMC is absorbing and only the probabilities and expected times to absorb into specific terminating states are needed, then the solution steps are:

1) Again obtain the quantities in (2.4) and construct the embedded DTMC.

2) Due to the absorbing nature of the DTMC, its transition probability matrix can be (subject to some potential state reordering/relabeling) shown to have the following special structure:

$$\boldsymbol{P} = \begin{bmatrix} \boldsymbol{Q} & \boldsymbol{C} \\ 0 & \boldsymbol{1} \end{bmatrix}$$

where ***Q*** is a substochastic matrix (i.e. with all entries in the range [0, 1] and at least one row sum to less than unity) and ***1*** is an identity matrix whose dimension is the same as the number of absorbing states in the DTMC.

3) Use the ***Q*** matrix, compute the expected visit counts of the non-terminating states until absorption as outlined in [6]:

$$\vec{\boldsymbol{V}} = (\boldsymbol{I} - \boldsymbol{Q})^{-\mathbf{1}} \tag{2.10}$$

4) Use the {*ET*(*i*)} along with {V_i} (computed in (2.10) above) to compute the expected time to absorption via the following, where $S_{nonabsorb}$ is the set of non-absorbing states:

$$ET_{absorb} = \sum\nolimits_{i \in S_{nonabsorb}} V_i ET(i) \tag{2.11}$$

This thesis will primarily utilize these two SMC solution methods.

2.1.3 Markov Regenerative Process

2.1.3.1 Definition

The Markov Regeneration Chain (MRGP) formalism allows state transitions with general distributions in a discrete state space model. An MRGP is a stochastic process { $Z(t)$; t>0 } on the discrete state space Φ. During evolution of the chain, there are regeneration time points at which the process probabilistically restarts itself. The stochastic process between regeneration epochs does not necessarily have Markov property, but the sequence of regeneration time points satisfy Markov property such that the future evolution of the stochastic process, given the process state at a regeneration point, does not depend on the history before that point. Therefore an MRGP may be viewed as having the SMC and CTMC formalisms as its special cases. The formal definition of MRGP is as follows [4]:

Definition: A stochastic process { $Z(t)$; $t > 0$ } defined on a discrete state space Φ is a Markov regenerative process if there exists a Markov renewal sequence { (X_n,T_n), $X_n \in \Omega$ } of random variable pairs, where X_n is the state being visited by and T_n the time of the n-th transition, such that all the conditional finite distributions of { $Z(T_n + t)$; $t > 0$ } given { $Z(u)$; $0 \leq u \leq T_n$, $X_n=i$ } are the same as those of { $Z(t)$; $t \geq 0$ } given $X_0 = i$. The definition implies that:

$$Pr\{Z(T_n + t) = j | Z(u); 0 \leq u < T_n, X_n = i\} = Pr\{Z(t) = j | X_0 = i\} \tag{2.12}$$

2.1.3.2 Steady-State Solution

The stochastic process { X_n; $n > 0$ } over the regeneration time points of an MRGP forms a discrete time Markov chain on a state space Ω ϵ Φ, called the embedded Markov chain of the MRGP. The sequence {T_n; $n > 0$} are the regeneration epochs. The evolution of an MRGP can be summarized by the evolution of the embedded Markov chain along with the stochastic behavior between two successive regeneration points. Both are defined in the form of kernel distributions. For the embedded Markov chain, the kernel distributions are defined by the following conditional transition probabilities:

$$K_{ij}(t) = P\{X_{n+1} = j, T_{n+1} - T_n \leq t | X_n = i\}, i, j \in \Omega \tag{2.13}$$

A distribution $K_{ij}(t)$ specifies the probability of starting in state *i* and the next regeneration state is *j* visited at or before time *t*. The matrix whose entries are specified as the kernel distributions, $K(t) = [K_{ij}(t)]$, is called the global kernel of the MRGP. On the other hand, the behavior between two successive regeneration time points is specified as the probability that, given the process started in regeneration state *i* ϵ Ω at time 0, it is in state *j* ϵ Φ at time *t* without any regeneration occurring in between, i.e.,:

$$E_{ij}(t) = P\{Z(t) = j, T_1 > t | Z(0) = i\} \tag{2.14}$$

The matrix $E(t) = [E_{ij}(t)]$ is called the local kernel of the MRGP. When the embedded discrete time Markov chain is finite and irreducible, its steady-state probability vector ***v*** is given by the solution to the linear system under constraints:

$$\boldsymbol{v} = \boldsymbol{v} \cdot \boldsymbol{K}(\infty) \; subject \; to \sum_{i \in \Omega} v_i = 1 \tag{2.15}$$

Then the steady-state probabilities $\pi_j, j \in \Phi$ of the MRGP are given by:

$$\pi_j = \frac{\sum_{k\in\Omega} v_k \alpha_{kj}}{\sum_{k\in\Omega} v_k \sum_{l\in\Omega} \alpha_{kl}} \tag{2.16}$$

where

$$\alpha_{ij} = \int_0^\infty E_{ij}(t)dt$$

is the mean sojourn time in state *j* before the next regeneration time point, given the initial regeneration state *i* [4].

2.1.4 Markov Reward Models

Markov reward models (MRM) extend the Markov model formalisms (and related ones such as SMC and MRGP) with reward assignment. A reward assignment is a mapping from any given state to a real value. Compared to the Markov models, which in itself only yields state probabilities and visit counts, a suitable reward assignment allows a variety of other important system measures to be captured. Generally speaking, as long as a measure can be expressed as a function of the underlying model state, it can be captured and analyzed through an appropriate reward assignment.

Similar to the steady-state and transient analysis variants for a Markov model, a Markov reward model can also be analyzed for steady-state expected reward rate and transient expected reward rate at and accumulated reward up to a given time t. The former asks about the long-term time-averaged reward accumulation from the evolution of the system, while the latter two capture the instantaneous expected reward at time t, which is determined by the underlying state distribution at that time, and the total expected reward accrued as the system evolve from time 0 to *t*, respectively. In both cases, the reward assignment allows system measures that are more complex than state probabilities and visit counts to be studied through the Markov model.

2.1.5 Stochastic Reward Net

Writing down a Markov chain by hand gets tedious once the number of states grows large. Even if construction of the chain can be automated, the whole modeling and analysis process is still less than intuitive since the Markov chain states may not straightforwardly reflect the conditions of system components, the latter being generally better understood by people working in system design/management fields who may benefit from the analysis results. Thus it would be quite helpful to have a formalism that allows a high-level specification of system components and automatically generates the underlying Markov chain and its solutions.

The Stochastic Reward Net (SRN) [7] provides such a formalism. SRN is an extension to the Petri Net formalism which can be defined mathematically as:

- A Petri net (PN) is a 4-tuple: PN = (*P*, *T*, *A*, *M*), where *P* is the finite set of places (represented by circles), T is the finite set of transitions (represented by bars), *A* is the finite set of arcs (connecting elements of *P* and *T*) and *M* is the set of markings each of which denotes a different distribution of tokens in the places of the net. The initial marking is denoted by M_0.

The notion of tokens in the PN formalism is used to indicate the flow of work/information/etc. throughout the model. The places and transitions formed a bipartite graph, with arcs connecting them. Tokens flow along arcs when transitions fire, and come to rest in places, representing the process of work/info/etc. flowing across the system among its different components (the places) after certain action/processing is done (the transitions firing). Extensions to PN (such as [8]) introduce probabilistic distributions to the transition firing times and immediate transitions, allowing specific time dynamics and conditional branches to be captured in a straightforward and intuitive manner. A Markov chain model can be automatically generated from such a timed PN model (such as implemented in [7]), simplifying the model construction and analysis process.

SRN introduces the additional construct of marking reward assignment that facilitates specifying at a high level many numerical measures to compute. Just as a timed PN model translates to a Markov chain, an SRN model translates to a Markov reward model. It also introduces a richer set of transition-controlling constructs, such as guard functions and marking-dependent firing rates, which allow more complex interdependence among the components to be specified at high level.

2.2 Queueing Network

The queueing network is a type of non-state-space model (meaning that the model analysis process does not require the generation of the underlying state space) that is commonly used in analyzing system performance. In its most generic form, a queueing network is simply a graphical way to represent the steps of processing tasks/information in a system. The formalism consists of two types of entities: stations and jobs. The jobs represent tasks/information that the system (composed of the stations) needs to process. Depending on whether jobs arrive from the outside or indefinitely circulate inside the system, the model can be classified either as an open queueing model or a closed queueing model. The jobs arrive at the stations seeking processing, and the stations adopt certain scheduling policies to allocate its processing power to the jobs. If the distribution of number/type of jobs in all stations in a model is considered as a random variable, then the queueing network also represents a particular stochastic process.

The most generic queueing networks cannot be very efficiently analyzed. A subset of them, where the underlying stochastic process is Markovian, may be analyzed as Markov chains. However, many queueing networks do not fall into this category as the systems they represent exhibit dynamics that are far from being Markovian. Also, the Markov chain formalism itself may experience the state-space explosion problem depending on the scenario.

A very important subset of queueing networks, that do enjoy very efficient solutions, are the product-form queueing networks. These models possess the product-form solution where each station may be analyzed in an (nearly) independent fashion [9]. Such a property greatly simplifies the analysis process at the expense of imposing some additional constraints on the system dynamics. A detailed description of product-form queueing networks and their analysis methods can be found in [10] and the references therein. Here I briefly describe the types of product-form queueing networks used in this thesis.

The definition of a product-form queueing network consists of the following elements:

- A set of queueing stations $\{S_i\}$. Each station has a certain processing capacity (typically represented by the distribution of the time needed to complete a job) and a scheduling discipline. For a station to admit a product-form solution, the possible combinations of the scheduling and capacity are:

  - A first-come-first-serve (FCFS) discipline with an exponentially distributed job service time.
  - A processor-sharing (PS) discipline with a differentiable service time distribution.
  - A pure delay discipline (also known as infinite server (IS)) with a differentiable service time distribution.[11]
  - A last-come-first-serve preemptive-resume (LCFS-PR) discipline with a differentiable service time distribution.

- A set of routing probabilities among the stations and (in the case of an open network) from the outside into the system and back. These probabilities specify how job arrivals are distributed among the stations, how completed jobs flow among the stations, and when jobs leave the network. The probabilities are pre-specified and do not change based on model states (The cases where such probabilities may vary based on the underlying model state, such as those in [12], are not considered in this thesis).
- A set of job chains. Each chain may have its own routing probabilities and service time distributions at the stations (unless the station is a FCFS one), and may either allow job arrival from/leaving to the outside (an open chain) or specifying a fixed number of jobs in the system (a closed chain). In the case of an open chain, the job arrival must follow a Poisson process in order for the product-form property to hold.

The solution to an open product-form queueing network is especially simple. The basic idea is to use the routing probabilities to establish a DTMC and use this matrix to obtain the expected visit count of a job to the stations before leaving the network (in a fashion similar to the second SMC solution in the previous subsection, with states replaced by the stations). Then the total arrival rate to each station can be obtained as the overall external arrival rate (recall that the arrivals form a Poisson process) multiplied with the corresponding expected visit count. Subsequently each station can be studied in isolation. If we denote the total arrival rate to station *i* of class *j* jobs as λ_{ij} and the reciprocal of the mean processing time of a job from class *j* as μ_{ij}, the metrics of station *i* can then be obtained as follows:

Throughput of class *j*: $\mathrm{T}_{ij} = \lambda_{ij}$

Utilization of class *j*: $\rho_{ij} = \lambda_{ij}/\mu_{ij}$

Total utilization: $\rho_i = \sum_j \lambda_{ij}/\mu_{ij}$

Stationary probability of n jobs: $p(n) = (1 - \rho_i)\rho_i{}^n$

Steady-state expected # of jobs: $EN = \rho_i/(1 - \rho_i)$ (2.17)

Steady-state expected response time: $ERT = 1/(1 - \rho_i) \times 1/\mu_i$

Steady-state expected # of class *j* jobs: $EN_j = \rho_{ij}/(1 - \rho_i)$

Steady-state expected response time of class *j*: $ERT_j = 1/(1 - \rho_i) \times 1/\mu_{ij}$

Compared to the open queue, the solution to a closed product-form queueing network is more complex. Nevertheless there exist efficient algorithms, such as those described in [13] and [14]. These algorithms have been implemented in many software packages, such as [15], so this thesis will make use of such tools and skip the descriptions of the algorithms.

2.3 Markov Decision Process

The Markov decision process (MDP) is a state-based dynamic optimization formalism. An MDP instance is defined by the following components:

1) A set of states, *S*, that describe the condition of the system being modeled.
2) A set of actions A_s, $s \in S$, that specify how the system may evolve from the current state *s*. The actions form the 'control variables' of the optimization problem.
3) Decision points, the time points at which an action can be chosen from those available at the current state *s*.
4) A transition probability function $P(s'|s, a)$ which defines the transition probability into a given new state *s*' at the next decision point, conditioned on the current state *s* and the action *a* chosen at the current decision point.
5) A reward function $R(s'|s, a)$ which defines the expected reward that will be accumulated when the system goes into a new state *s*' at the next decision point, conditioned on the state *s* and the action *a* at the current decision point. Optionally, another reward function may also be defined for each action, i.e. $R(s, a)$, instead of being associated with a particular transition under this action.

The optimality in an MDP is in the sense of maximizing or minimizing some reward criteria during the course of system evolution. There are several common choices: the expected total reward, the expected discounted reward, and the expected time-averaged reward. The expected total reward criterion is applicable to finite-horizon MDPs where the number of decision points is finite. In contrast, the expected discounted reward and expected time-averaged reward criteria can be applied to infinite-horizon MDPs, where the number of decision points is infinite (but typically countable). Of these two, the expected discounted reward is easier to apply (and arguably more widely used) than the expected time-averaged reward. This thesis uses the expected discounted reward as the objective function.

As its name suggests, the MDP possesses the Markov property in the sense that the system evolution beyond a decision point depends only on the system state and the action chosen at that point. When the decision horizon is infinite, if the underlying model parameters do not vary with time, the theory of MDP indicates that it is sufficient to locate a stationary policy to achieve optimality subject to some constraints on the model structure and the optimality criteria [16]. In such cases, the existence of stationary policies means that there is no need to consider the past history when making a decision about which action to perform in a given state. Thus the goal of solving an infinite horizon MDP is to generate a mapping from the states to their actions, i.e., a stationary policy, so that in a given state an action is known to be the optimal choice to control system evolution from that point. Once a stationary policy is obtained, the transitions (and their probabilities) from each state become fixed, and the MDP becomes a (discrete-time) Markov chain. For finite horizon MDPs, an optimal policy in general will not be stationary, but can be solved with the same solution method (described later) even in cases the model parameters change with time. A small MDP example, which illustrates how the different choices of actions in a given state translate to different courses of system evolution in the future, by yielding different future states and transition probabilities, is given in Figure 2.1.

- Current state: *1*
- Possible next state: *2,3,4*
- Actions: a_1, a_2
- $P(1|1,a_1) = 0.6$
  $P(2|1,a_1) = 0.1$
  $P(3|1,a_1) = 0.1$
  $P(4|1,a_1) = 0.2$
- $P(1|1,a_2) = 0.4$
  $P(2|1,a_2) = 0.35$
  $P(3|1,a_2) = 0.25$

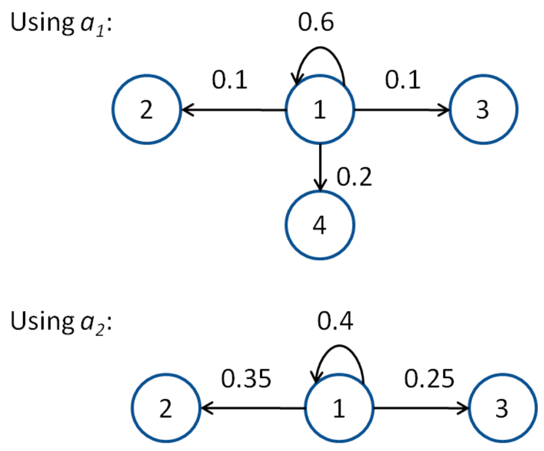


Figure 2.1. Example MDP

Solving an MDP instance involves at its core a set of optimality equations, whose solution provides the optimal action to choose at each system state. Under the infinite-horizon case, using the expected discounted reward optimality criterion, and assuming the objective is minimization, the equations can be described as:

$$\forall s,\ \ V(s) = \min_{a \in A_s}\{R(s,a) + \sum\nolimits_{s'} P(s'|s,a)[R(s'|s,a) + \beta V(s')]\} \qquad (2.18)$$

Here *V*(*s*) is the value function of the system, which maps the state space *S* to real values and describes the (un)desirability of the states. The term *β* is the discount factor, a positive value less than unity that represents the reduced relevance of future situations with respect to the current decision. For a given MDP, the number of equations equals the number of states, and the purpose is to choose an action from those available in the current state *s* so as to minimize the right-hand-side of equation 2.18. The set of equations can be solved iteratively, and convergence is guaranteed under some general conditions [16]. After the convergence the set of minimizing actions at each state form the optimal decisions, while the value function entries provide the optimal expected discounted reward, when starting from a given state and following the optimal decisions along system evolution.

2.4 Simulation and Sampling Methods

2.4.1 Simulation

Simulation, in the context of system modeling and analysis, refers to the method of analyzing a real system through imitating its behavior based on its logical abstraction. Note that simulation is distinct from emulation as the latter focuses on providing a functionality identical or similar to that of the real system using some substitutes. In a simulation model, the simulation program (executed on computers) proceeds by generating and acting upon synthesized events that mirror the corresponding occurrences in the real system, to the extent that the components in the simulation model display behavior and interactions that are close to their real counterparts. This imitation of the real system dynamics provides the basis to study the latter through the simulation model.

As the simulation execution progresses, measures of interest would be gathered which assume some of the possible values that their real counterparts could take. However, unless the model is purely deterministic (which typically has the implication that the system being investigated is simple), the set of values obtained from a single simulation execution reflects only one possible realization of system dynamics when the system is subject to the same input as is used in the simulation. Thus to overcome randomness and accurately evaluate system measures, the simulation needs to be repeated multiple times, and the resulting values, each forming a sample, are combined to generate statistics about system measures. In general such a statistic would be the sample mean of the corresponding system measure and, assuming the simulation runs are conducted independently, relies on some form of the law of large numbers to ensure the system measure is accurately estimated from a sufficiently large sample set.

Regarding application of simulation in practice, one of the primary concerns is that the result is always subject to some amount of uncertainty, which is essentially the variance of the collected samples. Generally speaking it would be necessary to quantify such uncertainty, which is typically done by means of confidence intervals. A confidence interval is a range around the computed statistic value, and its method of construction guarantees that a given percentage of intervals constructed in this way would contain the real system measure value. Such constructions are based on the distribution of the gathered statistic, which may either be derived directly based on known distributional form of the system measure (e.g. the mean of exponentially distributed samples follows an Erlang/Chi-squared distribution) or approximated (e.g. normal approximation based on the central limit theorem).

Sometimes the aforementioned uncertainty can be very large for some system measures, e.g. those that reflect occurrence of rare system events. Due to the infrequent occurrence of such events, there is inadequate information to damp out the randomness without running the execution for very long time. Thus the simulation may become inefficient when used to evaluate such measures. Several “variance-reduction” methods, such as [17] and [18], exist to mitigate such a problem.

2.4.2 Sampling Method

Sampling methods are those techniques that produce random values from given probability distributions. In the context of this thesis, aside from playing a part in simulations, sampling methods also provide a useful technique for solving certain stochastic models. In this respect they are primarily employed in two cases: 1) when the stochastic model involves integrals for which it is difficult to derive closed form formulas; 2) when some stochastic models are composed of submodels whose complex interactions prevent a more direct solution approach from being used. In both cases, the sampling method estimates the quantity of interest by casting the said quantity as an expectation with respect to a probability distribution (with the assumption that computing the expectation directly is difficult but drawing a sample from the probability distribution is easy), and then relying on the law of large numbers to approximate the quantity with the mean of many function value samples, where one value is produced by firstly sampling from the probability distribution and then feeding the probability sample to the function.

The first case arises in Chapter 7 where certain submodels produce results that are expressed in terms of sums of random variables. While in principle closed forms for these results could be obtained via direct convolution, such a method can become infeasible depending on the distributions that the component random variables follow. In that case the sampling method can be a much simpler alternative and, while slower to evaluate than a closed form formula, is nonetheless efficient since the overall model only needs evaluation for a small number of times.

The second case arises in Chapter 4, where one particular sampling method, the Gibbs sampling, is employed to solve a Markov chain model that involves multiple interacting submodels. In that context, the sampling procedure would produce state samples for each submodels based on the states of its peers (so as to capture the interactions among the submodels), and overall system metrics can then be established from the total set of samples. One notable feature of Gibbs sampling, and the family of Markov Chain Monte Carlo (MCMC) sampling methods to which Gibbs sampling belongs, is that the sampling distribution is not directly available. Instead, an MCMC method generates samples by making random walks on a Markov chain, where the probability distribution over the walk directions at each chain state is set up such that the samples generated would (after sufficient numbers of samples are collected) follow the desired distribution.

3. RAID System Modeling[1]

3.1 Overview

In modern dependable storage systems, data availability is treated as an essential aspect. Among the various techniques for improving data availability, RAID is a well-established popular solution to protect the integrity of data from disk failures. Several choices of RAID architectures are available, varying in the number and organization of disks constituting the disk arrays as well as the level of data reliability provided. In much research literature, the reliability of RAID storage systems is often evaluated by the measure mean time to data loss (MTTDL), which is often computed from continuous time Markov chain (CTMC) models by assuming exponential distributions for the times to disk failure and disk rebuild times. The quantification of RAID reliability allows designers to select a proper storage architecture to satisfy given needs.

Literature on disk failure statistics, however, has revealed that in many cases the times to disk failure event do not follow exponential distribution. Schroeder et al analyzed 100,000 disk failures and showed that the annual disk failure rate (AFR) is much higher than the expected values derived from MTTF with exponential assumptions [19]. Disk failures not only consist of operational failures such as bad servo-track and bad electronics, but also include latent defects caused by writing errors or media degradation [20]. The impact of latent sector errors during a disk rebuilding process is not negligible because it may cause a double disk failure leading to data loss. Researchers have made efforts to narrow the gap between the conceptual storage

[1] The work in this chapter has been described in the following document:
Performability Modeling for RAID Storage Systems by Markov Regenerative Process. Fumio Machida, Ruofan Xia and Kishor Trivedi. Submitted to IEEE Transaction on Dependable and Secure Computing (TDSC), under review.

model and the real failure data by improving models and introducing efficient solution methods [20][21][22].

While MTTDL and the assumption of exponential distributions have been the subjects of criticism [23][24], there are also counterarguments that MTTDL is still useful for comparing solution techniques and different architectures of RAID systems. Venkatesan and Iliadis showed that the MTTDL was practically insensitive to the actual failure distribution when failure rate is much smaller than repair rate [25]. Authors in [26] advocate that no study in the literature disproves the validity of MTTDL as a criterion for the comparison of reliability among different data storage schemes. Furthermore, the misconceptions in the criticism of using MTTDL are refuted in [27]. Here I will follow this argument and use Markov models for comparative study of RAID storage systems. However, when exponential distributions are assumed for disk rebuild times, the impact of the memoryless property on data availability may not be negligible, as the sensitivity analysis in [25] points out. In order to compare the data availabilities of different RAID architectures, the time to rebuild a disk should be treated more precisely in the models.

In this chapter I will extend the traditional CTMC models for RAID storage systems to deal with non-exponential disk rebuild time. When a disk fails in a RAID system that could tolerate multiple disk failures, a rebuild process starts to reconstruct the data on a spare disk using the data stored in the remaining disks. Even if another disk failure occurs during the rebuild process, the rebuild process continues the operation until the data is either reconstructed completely or lost. Since the rebuild times no longer have the memoryless property, and that during the rebuild process the stochastic model representing the RAID may change state due to additional disk failures, it is necessary to use the Markov regenerative chain (MRGP) [4] formalism to accurately capture the behavior of the system. In the MRGP model, the rebuild time is not necessarily exponentially distributed and hence it provides a more accurate representation of RAID storage behavior compared to CTMC models.

The MRGP model is used to conduct performability comparison between RAID10 and RAID6 systems which are the two most popular enterprise RAID architectures. RAID10 is sometimes called a hierarchical RAID as it combines mirroring (RAID1) and striping (RAID0). Due to its better write performance, generally RAID10 is preferred over RAID6. However, in terms of disk failure tolerance, RAID10 is inferior to RAID6 as it can lose data by a particular combination of double disk failures while RAID6 tolerates any double disk failure. Depending on the number of disks used, RAID6 may also achieve better sequential read performance compared to RAID10 with the same number of disks [28]. To better characterize the performability of RAID10 and RAID6, in this chapter I employ MRGP models with reward assignment, known as Markov regenerative reward model (MRRM). Reward rates are assigned based on performance benchmark measurements of the two systems, and the resulting MRRMs yield the expected performability of RAID storage systems as its steady-state solution. I will also discuss the tradeoffs between RAID10 and RAID6 storage architectures.

The chapter is organized as follows. Section 3.2presents the traditional CTMC-based modeling approach for storage system and clarifies the modeling errors criticized by researchers. To overcome the issue of memoryless property assumed in traditional models, MRGP models are developed for storage systems with RAID level 6 and 10 and described in

Section 3.3. Section 3.4 gives the results of numerical studies on the presented MRGP models and shows that the data availability is insensitive to memoryless property assumed in CTMC models. I also conduct the performability comparison between RAID6 with RAID10 by combining the numerical results and performance benchmarks on the real storage system. In Section 3.5 I review the related work for reliability and performance analysis of RAID storage systems. Finally Section3.6 summarizes the findings and concludes.

3.2 RAID Markov Models

This section reviews the CTMC-based reliability models for RAID storage systems. Consider a disk array with N disks each of which can fail at a constant rate λ. When a disk fails, a rebuild operation is started to recover the affected data on a spare disk at a constant rate of μ. I assume that the failed disk is replaced with a new spare disk so that the total number of operational disks within the RAID array does not change. The state transitions of RAID system can be represented by a CTMC in which states are labeled by the number of failed disks.

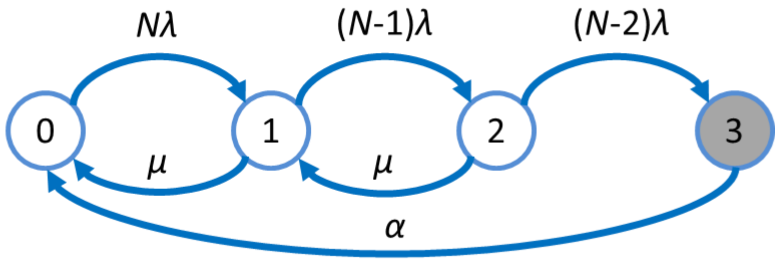

$$\begin{tikzcd}
0 \arrow[r, bend left, "N\lambda"] & 1 \arrow[l, bend left, "\mu"] \arrow[r, bend left, "(N-1)\lambda"] & 2 \arrow[l, bend left, "\mu"] \arrow[r, bend left, "(N-2)\lambda"] & 3 \arrow[lll, bend left=40, "\alpha"]
\end{tikzcd}$$

Figure 3.1. CTMC Model for RAID6 Storage System

Figure 3.1 shows an instance of CTMC model for RAID6 configuration where I further assume that the entire disk array should be replaced (at a constant rate α) after encountering triple disk failures. Note that in state 0 there are *N* available disks which results in λ times *N* being the state transition rate to state 1. State 1 and state 2 have similar state transition rates to state 2 and state 3 that depend on the numbers of available disks (as presented by (N-1)λ and (N-2)λ). The RAID6 CTMC model yields a closed form solution for steady-state availability that is the sum of steady-state probabilities of states 0, 1 and 2:

$$\begin{aligned} A_{RAID6} &= \sum_{i=0,1,2} \pi_i \\ &= \frac{\alpha\{2(N-1)^2\lambda^2 + (N\lambda+\mu)(\mu+(N-2)\lambda)\}}{\alpha\{2(N-1)^2\lambda^2 + (N\lambda+\mu)(\mu+(N-2)\lambda)\} + N(N-1)(N-2)\lambda^3} \end{aligned} \tag{3.1}$$

Although the CTMC model provides a reasonably simple representation of state transitions of a RAID system, several drawbacks have been discussed in the literature. The first criticism is directed at the unreality of the assumption of exponential distribution. Empirical studies on disk failure trends show that the times to disk failure do not follow exponential distributions and two parameter distributions such as Weibull are more appropriate [1][2]. Since disk failures can occur due to a variety of causes, it is unrealistic to model a single state transition with a constant failure rate. Failure distributions can be mixtures of multiple distributions because of several failure modes and distinct production vintages. The second criticism is related to the memoryless property of the sojourn times in a homogeneous CTMC that results in modeling errors. Under the memoryless assumption, the state transition rates do not depend on the amount of time spent in a state. The disk failure rate does not change according to the age and all the intermediate results of rebuild process are cleared if another disk fails during the rebuild operation. However, in reality, disk failure rates do depend on disk ages and disk rebuild progress may not be cleared by another disk failure.

In fact, these two issues are connected because memoryless property is equivalent to the time-independent transition rate (i.e., exponential distribution). In other words, the memoryless property holds only when exponential distributions are assigned to all the state transitions. Extending the model by replacing disk failure distributions with more general ones is certainly possible. However, the difference in time to failure distribution might not to have a huge impact on the steady-state measure as Venkatesan and Iliadis's study on the impact of failure distributions on MTTDL [7] shows. The assumption of exponential distributions for disk failures could be acceptable in the early phases of system design where the users do not have much empirical data on disk failure times.

The memoryless assumption of disk rebuild times is more problematic. According to the common implementation of disk rebuild operations in RAID, the rebuild process is cumulative and the operation can continue execution even after another disk failure. The CTMC does not account for this non-exponential behavior. In contrast with the uncertainty of disk failure distributions, disk rebuild times can be controlled by the design and implementation. The modeling error caused by the limitation of CTMC has thus far not been studied for RAID architectures. Thus in the next section I will investigate this issue with comprehensive Markov regenerative chain models.

3.3 RAID Markov Regenerative Chain Models

3.3.1 RAID6 MRGP model

In a RAID storage system which has the property of Double Disk Failure (DDF) tolerance, the rebuild operation of a failed disk continues even after another disk failure. The system state change induced by the additional failure does not erase the memory of the time elapsed since the beginning of the rebuild operation. Such time dependent behavior across states can be modeled by an MRGP. Figure 3.2 shows the MRGP RAID model for RAID6 configuration where Z_{ij}, $i, j \in \Phi$, denotes the random variable for the transition time from state i to state j.

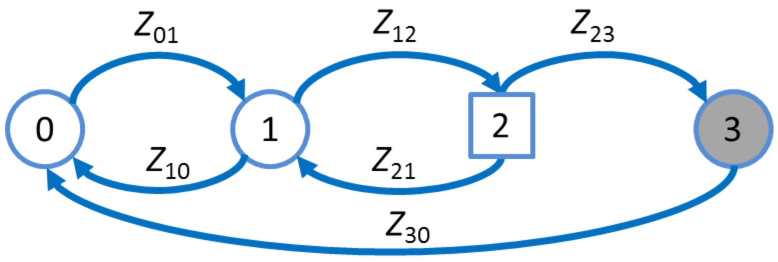


Figure 3.2. MRGP Model for RAID6 Storage System

A major difference between the MRGP model and the CTMC model (shown in Section 3.2) is the introduction of a state with time-dependent exit rates denoted by a rectangle (i.e., state 2). The entrance into such a state by the process does not regenerate the latter, unlike the other states (denoted as circles) and all states in a CTMC model. More specifically, in state 2, the distribution for the time to rebuild a failed disk depends on the time spent in state 1, hence it does not have the Markov property and is thus a non-regenerative state.

I assume that the time to a disk failure follows an exponential distribution with rate λ whereas the time to rebuild a disk and reconstruct RAID system follows the general distributions $G_1(t)$ and $G_2(t)$, respectively. The sequence of visits to the regeneration states in $\Omega = \{0,1,3\}$ form the embedded Markov chain of the MRGP. The global kernel of the RAID6 MRGP model is:

$$\boldsymbol{K}(t) = \begin{bmatrix} 0 & K_{01}(t) & 0 \\ K_{10}(t) & K_{11}(t) & K_{13}(t) \\ K_{30}(t) & 0 & 0 \end{bmatrix} \tag{3.2}$$

Note that the subscripts ij in $K_{ij}(t)$ denote the actual state labels in $\Omega = \{0,1,3\}$. From the definition of kernel distributions, $K_{10}(t)$ is the conditional probability that the process has regenerated into state 0 by time t given the prior regeneration occurred in state 1 at time 0. In this particular context, it is the probability that the rebuild operation for the failed disk completes before the RAID system encounters another disk failure. Thus:

$$K_{10}(t) = Pr\{Z_{10} \le t, Z_{12} > Z_{10}\} = \int_0^t \left(1 - F_{Z_{12}}(x)\right) dF_{Z_{10}}(x) = \int_0^t e^{-(N-1)\lambda x} dG_1(x) \tag{3.3}$$

Next, $K_{11}(t)$ is derived as the probability that the process starting from state 1 has regenerated again into state 1 by time t after visiting state 2. Note that the process is not regenerated at the entrance of state 2 and holds the memory of sojourn time in state 1. A disk failure occurs while the rebuild operation is ongoing, but the rebuild operation does complete before a triple disk failure which would have caused the RAID to fail (i.e. entering state 3). Let R be the random variable for time to rebuild a failed disk whose distribution is $G_1(t)$, and Z_{13}be the random variable representing the time to triple disk failures from a single disk failure. Note that Z_{13} is the sum of the two variables Z_{12} and Z_{23} in Figure 3.2. $K_{11}(t)$ is then given by:

$$\begin{aligned} K_{11}(t) &= Pr\{R \le t, Z_{13} > R\} – Pr\{R \le t, Z_{12} > R\} \\ &= \int_0^t \left(1 - F_{Z_{13}}(x)\right) dF_R(x) - \int_0^t \left(1 - F_{Z_{12}}(x)\right) dF_R(x) \\ &= \int_0^t \{(N-1)e^{-(N-2)\lambda x} - (N-2)e^{-(N-1)\lambda x}\} dG_1(x) - \int_0^t e^{-(N-1)\lambda x} dG_1(x) \\ &= (N-1)\int_0^t \{e^{-(N-2)\lambda x} - e^{-(N-1)\lambda x}\} dG_1(x) \end{aligned} \tag{3.4}$$

In a similar fashion, $K_{13}(t)$ is obtained as:

$$K_{13}(t) = Pr\{Z_{13} \le t, R > Z_{13}\} = \int_0^t \big(1 - F_R(x)\big) dF_{Z_{13}}(x)$$
$$= (N-1)(N-2)\lambda \int_0^t \big(1 - G_1(x)\big)\big(e^{-(N-2)\lambda x} - e^{-(N-1)\lambda x}\big)\, dx \tag{3.5}$$

Denote $K_{10}(\infty)$, $K_{13}(\infty)$ and $K_{11}(\infty)$ by *a*, *b* and 1-*a*-*b*, respectively. Solving the linear system with constraint:

$$\boldsymbol{v} = \boldsymbol{v} \cdot \boldsymbol{K}(\infty)\ subject\ to \sum_{i \in \Omega} v_i = 1$$

and the steady-state probabilities of the embedded Markov chain are given by:

$$v_0 = \frac{a+b}{a+2b+1}, \quad v_1 = \frac{1}{a+2b+1}, \quad v_3 = \frac{b}{a+2b+1} \tag{3.6}$$

Next I derive the local kernel of the RAID6 MRGP model. The local kernel describes the process dynamics between two successive regeneration points and is given by a 3×4 matrix:

$$\boldsymbol{E}(t) = \begin{bmatrix} E_{00}(t) & 0 & 0 & 0 \\ 0 & E_{11}(t) & E_{12}(t) & 0 \\ 0 & 0 & 0 & E_{33}(t) \end{bmatrix} \tag{3.7}$$

E_{ii}(t), $i \in \{0,1,3\}$ represent the probabilities that the process starting in state *i* stays in the same state till time *t*. Therefore:

$$E_{00}(t) = 1 - F_{Z_{01}}(t) = e^{-N\lambda t} \tag{3.8}$$

$$E_{11}(t) = \Big(1 - F_{Z_{10}}(t)\Big)\Big(1 - F_{Z_{12}}(t)\Big) = \big(1 - G_1(t)\big)e^{-(N-1)\lambda t} \tag{3.9}$$

$$E_{33}(t) = 1 - F_{Z_{30}}(t) = 1 - G_2(t) \tag{3.10}$$

Meanwhile, $E_{12}(t)$ is the probability that the process starting from state 1 is in state 2 at time *t* without any regeneration occurring:

$$\begin{aligned} E_{12}(t) &= Pr\{R > t, Z_{13} > t \geq Z_{12}\} \\ &= Pr\{R > t, Z_{13} > t\} - Pr\{R > t, Z_{12} > t\} \\ &= \left(1 - F_R(t)\right)\left(1 - F_{Z_{13}}(t)\right) - \left(1 - F_R(t)\right)\left(1 - F_{Z_{12}}(t)\right) \\ &= \left(1 - F_R(t)\right)\left(F_{Z_{12}}(t) - F_{Z_{13}}(t)\right) \\ &= \left(1 - G_1(t)\right)(N-1)\left(e^{-(N-2)\lambda t} - e^{-(N-1)\lambda t}\right) \end{aligned} \tag{3.11}$$

From the local kernel distributions, the mean sojourn times are computed as:

$$\alpha_{00} = 1/N\lambda \tag{3.12}$$

$$\alpha_{11} = \int_0^\infty \left(1 - G_1(t)\right) e^{-(N-1)\lambda t} dt \tag{3.13}$$

$$\alpha_{33} = \int_0^\infty \left(1 - G_2(t)\right) dt \tag{3.14}$$

$$\alpha_{12} = (N-1)\int_0^\infty \left(1 - G_1(t)\right)\left(e^{-(N-2)\lambda t} - e^{-(N-1)\lambda t}\right) dt \tag{3.15}$$

The data on the storage system is accessible when the storage system is in state 0, 1, or 2. Hence the expected data availability is computed by:

$$\begin{aligned} A_{RAID6} &= \sum_{i=0,1,2} \pi_i = \sum_{i=0,1,2} \frac{\sum_{k\in\Omega} v_k \alpha_{ki}}{\sum_{k\in\Omega} v_k \sum_{l\in\Omega} \alpha_{kl}} \\ &= \frac{(a+b)\alpha_{00} + \alpha_{11} + \alpha_{12}}{(a+b)\alpha_{00} + \alpha_{11} + \alpha_{12} + b\alpha_{33}} \end{aligned} \tag{3.16}$$

and performability is given by the following, where $\{r_j\}$ is the performance measure in state *j*:

$$P_{RAID6} = \sum_{j\in\Phi} \pi_j \cdot r_j = \frac{(a+b)\alpha_{00} r_0 + \alpha_{11} r_1 + \alpha_{12} r_2}{(a+b)\alpha_{00} + \alpha_{11} + \alpha_{12} + b\alpha_{33}} \tag{3.17}$$

Since A_{RAID6} has the following relationship with the traditional notion of mean time to data loss:

$$A_{RAID6} = \frac{MTTDL_{RAID6}}{MTTDL_{RAID6} + \alpha_{33}} \tag{3.18}$$

MTTDL$_{\mathrm{RAID6}}$is computed by:

$$MTTDL_{RAID6} = \frac{A_{RAID6} \cdot \alpha_{33}}{1 - A_{RAID6}} = \frac{(a+b)\alpha_{00} + \alpha_{11} + \alpha_{12}}{b} \tag{3.19}$$

As an example, if we assume that the rebuild and reconstruction times are deterministic with the distribution functions given by $G_1(t) = u(t - \tau_1)$ and $G_2(t) = u(t - \tau_2)$ where $u(\cdot)$ is the unit step function, we would have:

$$a = e^{-(N-1)\lambda\tau_1} \tag{3.20}$$

$$b = 1 - (N-1)e^{-(N-2)\lambda\tau_1} - (N-2)e^{-(N-1)\lambda\tau_1} \tag{3.21}$$

$$\alpha_{11} = \frac{1}{(N-1)\lambda}\left[1 - e^{-(N-1)\lambda\tau_1}\right] \tag{3.22}$$

$$\alpha_{33} = \tau_2 \tag{3.23}$$

$$\alpha_{12} = \frac{1 - (N-1)e^{-(N-2)\lambda\tau_1} + (N-2)e^{-(N-1)\lambda\tau_1}}{(N-2)\lambda} \tag{3.24}$$

3.3.2 RAID10 MRGP model

In this section I present the MRGP model for the RAID10 configuration with N disks ($N \geq 6$). Like the RAID6 MRGP model, the non-exponentially distributed rebuild time can be modeled by introducing states with time-dependent exit rates. Since the probability of more than four disk failures is negligibly small, I elect to neglect the storage failures caused by more than four disk failures. With this approximation, the RAID10 MRGP model is shown in Figure 3.3.

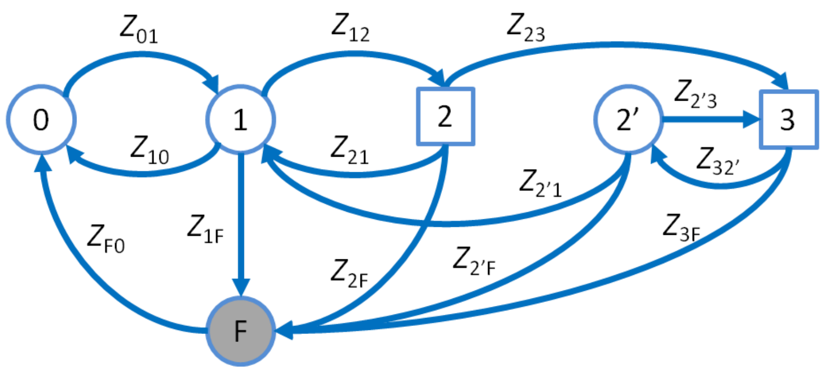


Figure 3.3. MRGP Model for RAID10 Storage System

The state labels 0, 1, 2, 3 represent the number of failed disks in the RAID system, whereas state F is the storage failure state. A RAID10 storage system can fail even by two disk failures depending on the combination of failed disks. If the second failure occurs at the mirroring pair of the first failed disk, the storage loses data (the model enters state F). That is, when the system is in state 1, another disk failure causes storage failure with probability 1/(*N*-1). The storage state in which two disks have failed is divided into two states; state 2 caused by a disk failure during the rebuild operation to the prior failed disk and state 2' resulting from the completion of the rebuild operation to one of the three failed disks. The stochastic process is not regenerated at the entrance of state 2, while it regenerates at the entrance of state 2'. From state 2 or state 2’, the storage system can fail with probability 2/(*N*-2), which is the probability of failure occurring to one of the disks that pair with the two already failed (hence breaking the mirroring protection). There is a risk that a third disk fails before completing a rebuild operation to the first failed disk, resulting in state 3. Since the rebuild operation continues execution even after entering into state 3, the latter is a non-regenerative state. The embedded Markov chain is formed by the sequence of state transitions among the regenerative states Ω = {0,1,2',F}.

I assume the disk failure time is exponentially distributed with rate λ and define $G_1(t)$ and $G_2(t)$ as the distribution functions for disk rebuild time and RAID reconstruction time, respectively. The global kernel of the RAID10 MRGP model is given by:

$$K(t) = \begin{bmatrix} 0 & K_{01}(t) & 0 & 0 \\ K_{10}(t) & K_{11}(t) & K_{12'}(t) & K_{1F}(t) \\ 0 & K_{2'1}(t) & K_{2'2'}(t) & K_{2'F}(t) \\ K_{F0}(t) & 0 & 0 & 0 \end{bmatrix} \tag{3.25}$$

$K_{10}(t)$ is the conditional probability that the process has regenerated into state 0 by time t given the prior regeneration occurred in state 1 at time zero. This corresponds to the event that a rebuild process completes before encountering another disk failure. Thus:

$$\begin{aligned} K_{10}(t) &= Pr\{Z_{10} \le t, Z_{1F} > Z_{10}, Z_{12} > Z_{10}\} \\ &= \int_0^t \left(1 - F_{Z_{1F}}(t)\right)\left(1 - F_{Z_{12}}(t)\right) dF_{Z_{10}}(x) \\ &= \int_0^t e^{-(N-1)\lambda x} dG_1(x) \end{aligned} \tag{3.26}$$

Next,$K_{11}(t)$ is the probability that the process starting from state 1 has regenerated again into state 1 by time t after visiting state 2. This corresponds to the situation that during the data rebuild process one more disk (but not the one pairing with the failed one) fails. Conditioning on $Z_{12} = \delta < Z_{10}, Z_{1F}$:

$$
\begin{aligned}
&Pr\{Z_{12}+Z_{21}\le t, Z_{21}<Z_{2F}, Z_{21}<Z_{23}|Z_{12}=\delta<Z_{10},Z_{1F}\}\\
&\quad = Pr\{Z_{21}\le t-\delta, Z_{21}<Z_{2F}, Z_{21}<Z_{23}|Z_{12}=\delta<Z_{10},Z_{1F}\}\\
&\quad = \int_{\delta}^{t} e^{-(N-2)\lambda(x-\delta)}dG_1(x|x>\delta)
\end{aligned}
\tag{3.27}
$$

Unconditioning on Z_{12}, $K_{11}(t)$ is given by:

$$
\begin{aligned}
K_{11}(t) &= Pr\{Z_{12}+Z_{21}\le t, Z_{21}<Z_{2F}, Z_{21}<Z_{23}, Z_{12}<Z_{10}, Z_{12}<Z_{1F}\}\\
&= \int_0^t Pr\{Z_{21}\le t-\delta, Z_{21}<Z_{2F}, Z_{23}|Z_{12}=\delta<Z_{10},Z_{1F}\}\\
&\quad \cdot\left(1-F_{Z_{10}}(\delta)\right)\left(1-F_{Z_{1F}}(\delta)\right)dF_{Z_{12}}(\delta)\\
&= \int_0^t \left\{\int_{\delta}^{t} e^{-(N-2)\lambda(x-\delta)}dG_1(x|x>\delta)\right\}\cdot\left(1-G_1(\delta)\right)\cdot(N-2)\lambda e^{-(N-1)\lambda\delta}\,d\delta
\end{aligned}
\tag{3.28}
$$

In a similar manner, I derive $K_{12'}(t)$ by first considering the probability conditioned on the time to reach state 3 from state 1 through state 2, $Z_{12}+Z_{23}=\delta_2$,

$$
\begin{aligned}
&Pr\left\{\begin{matrix} Z_{12}+Z_{23}+Z_{32'}\le t, Z_{32'}<Z_{3F}|Z_{12}+Z_{23}=\delta_2\\ ,Z_{12}<Z_{10},Z_{12}<Z_{1F},Z_{23}<Z_{21},Z_{23}<Z_{2F}\end{matrix}\right\}\\
&\quad = Pr\left\{\begin{matrix} Z_{32'}\le t-\delta_2, Z_{32'}<Z_{3F}|Z_{12}+Z_{23}=\delta_2\\ ,Z_{12}<Z_{10},Z_{12}<Z_{1F},Z_{23}<Z_{21},Z_{23}<Z_{2F}\end{matrix}\right\}\\
&\quad = \int_{\delta_2}^{t} e^{-3\lambda(x-\delta_2)}dG_1(x|x>\delta_2)
\end{aligned}
\tag{3.29}
$$

Further conditioning on $Z_{12}=\delta_1<Z_{10}$, Z_{1F} while unconditioning on $Z_{12}+Z_{23}=\delta_2$:

$$
\begin{aligned}
&Pr\{Z_{12}+Z_{23}+Z_{32'}\le t, Z_{23}<Z_{21}, Z_{23}<Z_{2F}, Z_{32'}<Z_{3F}|Z_{12}=\delta_1<Z_{10},Z_{1F}\}\\
&\quad = \int_{\delta_1}^{t} Pr\left\{\begin{matrix} Z_{12}+Z_{23}+Z_{32'}\le t, Z_{32'}<Z_{3F}|Z_{12}+Z_{23}=\delta_2\\ ,Z_{12}<Z_{10},Z_{12}<Z_{1F},Z_{23}<Z_{21},Z_{23}<Z_{2F}\end{matrix}\right\}\cdot
\end{aligned}
\tag{3.30}
$$

$$\left(1 - F_{Z_{21}}(\delta_2 - \delta_1 | Z_{12} = \delta_1 < Z_{10}, Z_{1F})\right)$$

$$\cdot \left(1 - F_{Z_{2F}}(\delta_2 - \delta_1 | Z_{12} = \delta_1 < Z_{10}, Z_{1F})\right) \cdot dF_{Z_{123}}(\delta_2 | Z_{12} = \delta_1 < Z_{10}, Z_{1F})$$

$$= \int_{\delta_1}^{t} \left\{ \int_{\delta_2}^{t} e^{-3\lambda(x-\delta_2)} dG_1(x | x > \delta_2) \right\} \cdot \left(1 - G_1(\delta_2 | \delta_2 > \delta_1)\right) \cdot e^{-2\lambda(\delta_2 - \delta_1)} \cdot$$

$$(N-4)\lambda e^{-(N-4)\lambda(\delta_2-\delta_1)} d\delta_2$$

Unconditioning on Z_{12}:

$$K_{12'}(t) = Pr\left\{ \begin{matrix} Z_{12} + Z_{23} + Z_{32'} \leq t, Z_{12} < Z_{10}, Z_{12} < Z_{1F} \\ , Z_{23} < Z_{21}, Z_{23} < Z_{2F}, Z_{32'} < Z_{3F} \end{matrix} \right\}$$

$$= (N-2)(N-4)\lambda^2 \int_0^t \left\{ \int_{\delta_1}^t \left\{ \int_{\delta_2}^t e^{-3\lambda(x-\delta_2)} dG_1(x | x > \delta_2) \right\} \right. \quad (3.31)$$

$$\left. \cdot \left(1 - G_1(\delta_2 | \delta_2 > \delta_1)\right) \cdot e^{-(N-2)\lambda(\delta_2-\delta_1)} d\delta_2 \right\}$$

$$\cdot \left(1 - G_1(\delta_1)\right) \cdot e^{-(N-1)\lambda\delta_1} d\delta_1$$

Considering the rebuild operation started from state 2', $K_{2'1}(t)$ is given by:

$$K_{2'1}(t) = Pr\{Z_{2'1} \leq t, Z_{2'1} < Z_{2'3}, Z_{2'1} < Z_{2'F}\}$$

$$= \int_0^t \left(1 - F_{Z_{2'3}}(x)\right)\left(1 - F_{Z_{2'F}}(x)\right) dF_{Z_{2'1}}(x) \quad (3.32)$$

$$= \int_0^t e^{-(N-2)\lambda x} dG_1(x)$$

$K_{2'2'}(t)$ is the probability that the process starting from state 2' regenerates again into state 2' at time t after visiting state 3. This corresponds to the case of one additional disk failure during data rebuild without causing data loss. By conditioning on $Z_{2'3} = \delta < Z_{2'1}, Z_{2'F}$:

$$\begin{aligned}&Pr\{Z_{2'3}+Z_{32'}\le t, Z_{32'}<Z_{3F}|Z_{2'3}=\delta<Z_{2'1},Z_{2'F}\}\\&\quad=Pr\{Z_{32'}\le t-\delta, Z_{32'}<Z_{3F}|Z_{2'3}=\delta<Z_{2'1},Z_{2'F}\}\\&\quad=\int_{\delta}^{t}e^{-3\lambda(x-\delta)}dG_1(x|x>\delta)\end{aligned}\tag{3.33}$$

Unconditioning on $Z_{2'3}$, $K_{2'2'}(t)$ is:

$$\begin{aligned}K_{2'2'}(t)&=Pr\{Z_{2'3}+Z_{32'}\le t, Z_{2'3}<Z_{2'1},Z_{2'3}<Z_{2'F}\}\\&=\int_0^t\left\{\int_\delta^t e^{-3\lambda(x-\delta)}dG_1(x|x>\delta)\right\}\cdot\left(1-G_1(\delta)\right)\\&\quad\cdot(N-4)\lambda e^{-(N-2)\lambda\delta}d\delta\end{aligned}\tag{3.34}$$

Denote $K_{10}(\infty)$, $K_{11}(\infty)$, $K_{12'}(\infty)$, $K_{2'1}(\infty)$ and $K_{2'2'}(\infty)$ as *a*, *b*, *c*, *d* and *e*, respectively. The transition probability matrix of the embedded Markov chain is then given by:

$$K(\infty)=\begin{bmatrix}0&1&0&0\\a&b&c&1-(a+b+c)\\0&d&e&1-(d+e)\\1&0&0&0\end{bmatrix}\tag{3.35}$$

Again solve the linear system with constraints:

$$v=v\cdot\boldsymbol{K}(\infty)\ subject\ to\sum_{i\in\Omega}v_i=1$$

and the steady-state probabilities of the embedded Markov chain are:

$$(v_0,v_1,v_{2'},v_F)=\left(\frac{1-b-\frac{cd}{1-e}}{w},\quad\frac{1}{w},\quad\frac{\frac{c}{1-e}}{w},\quad\frac{1-a-b-\frac{c}{1-e}}{w}\right)\tag{3.36}$$

where

$$w = 3 + 2b - a + \frac{d}{1-e}(1-2c) \tag{3.37}$$

Next, the local kernel of the RAID10 MRGP model is derived as a 4×6 matrix:

$$E(t) = \begin{bmatrix} E_{00}(t) & 0 & 0 & 0 & 0 & 0 \\ 0 & E_{11}(t) & E_{12}(t) & E_{13}(t) & 0 & 0 \\ 0 & 0 & 0 & E_{2'3}(t) & E_{2'2'}(t) & 0 \\ 0 & 0 & 0 & 0 & 0 & E_{FF}(t) \end{bmatrix} \tag{3.38}$$

$E_{ii}(t)$, $i \in \{ 0, 1, 2', \mathrm{F} \}$ are given by:

$$E_{00}(t) = 1 - F_{Z_{01}}(t) = e^{-N\lambda t} \tag{3.39}$$

$$E_{11}(t) = Pr\{Z_{10} > t, Z_{1F} > t, Z_{12} > t\} = \big(1 - G_1(t)\big)e^{-(N-1)\lambda t} \tag{3.40}$$

$$E_{2'2'}(t) = Pr\{Z_{2'1} > t, Z_{2'F} > t, Z_{2'3} > t\} = \big(1 - G_1(t)\big)e^{-(N-2)\lambda t} \tag{3.41}$$

$$E_{FF}(t) = 1 - F_{Z_{F0}}(t) = 1 - G_2(t) \tag{3.42}$$

$E_{12}(t)$ is the probability that the process starting from state 1 is in state 2 at time t. To capture this probability, I consider the failure of disk D, which is the mirror of the failed disk in state 1, separately from other disk failures. Let Z_{1F}^* denote the random variable for the time to failure of disk D. Such a failure results in RAID entering the state F, either directly from state 1 or through state 2 or state 3. Let R and U be the random variables for the time to rebuild a failed disk and the time to two more failures of disks other than D (with the process entering either state 3 or state F via state 2) from state 1, respectively. $E_{12}(t)$ is then given by:

$$\begin{aligned} E_{12}(t) &= Pr\{R > t, {Z_{1F}}^* > t, U > t \geq Z_{12}\} \\ &= Pr\{R > t, {Z_{1F}}^* > t, U > t\} - Pr\{R > t, {Z_{1F}}^* > t, Z_{12} > t\} \\ &= \big(1 - G_1(t)\big) \cdot e^{-\lambda t} \cdot \Big(F_{Z_{12}}(t) - F_U(t)\Big) \end{aligned} \tag{3.43}$$

The failure rate to state 2 from state 1 is (N - 2)λ, since individual disk failure time is exponentially distributed with rate λ and a failure of the mirrored disk paired with the previously failed disk needs to be excluded from this case (it is accounted for in the transition from state 2 to state F). Similarly, the failure rate to state 3 from state 2 is given by (N - 3)λ. Therefore, $F_U(t)$ follows a 2-stage hypoexponential distribution with parameters (N - 2)λ and (N - 3)λ:

$$F_{\mathrm{U}}(t) = 1 - (N-2)e^{-(N-3)\lambda t} + (N-3)e^{-(N-2)\lambda t} \tag{3.44}$$

Applying (3.44) in (3.43) we get:

$$E_{12}(t) = (N-2)\big(1 - G_1(t)\big)\big(e^{-(N-2)\lambda t} - e^{-(N-1)\lambda t}\big) \tag{3.45}$$

Next, $E_{13}(t)$ is the probability that the process starting from state 1 stays in state 3 at time t. Similar to the case of$E_{12}(t)$, I separate the failures of the disk D and D_2, which are disks paired with the ones failed at the entrance of state 1 and state 2 respectively, from other disk failures. Let $Z_{2F}{}^{*}$and $Z_{23F}{}^{*}$be the random variables for the time to fail the paired disk D_2 resulting in RAID failure from state 2 and the time to RAID failure due to quadruple disk failures caused by disks other than D and D_2 from state 2. Note that $Z_{2F}{}^{*}$is exponentially distributed with parameter λ and $Z_{23F}{}^{*}$ follows a two-stage hypo-exponential distribution with parameter (N - 4)λ,which does not include a failure of mirrored disk, and λ that corresponds to the transition from state 3 to state F.$E_{13}(t)$ is given by:

$$\begin{aligned} E_{13}(t) &= Pr\left\{\begin{matrix} R > t, Z_{1F}{}^{*} > t, Z_{12} + Z_{2F}{}^{*} > t, \\ Z_{12} + Z_{23F}{}^{*} > t \geq Z_{123} \end{matrix}\right\} \\ &= Pr\{R > t, Z_{1F}{}^{*} > t, Z_{12} + Z_{2F}{}^{*} > t, Z_{12} + Z_{23F}{}^{*} > t\} \\ &\quad - Pr\{R > t, Z_{1F}{}^{*} > t, Z_{12} + Z_{2F}{}^{*} > t, Z_{12} + Z_{23} > t\} \end{aligned} \tag{3.46}$$

$$= \left(1 - G_1(t)\right) e^{-\lambda t} \int_{\delta=0}^{t} \left(1 - F_{Z_{2F}^*}(t-\delta)\right)$$

$$\cdot \left(F_{Z_{23}}(t-\delta) - F_{Z_{23F}^*}(t-\delta)\right) dF_{Z_{12}}(\delta)$$

$$= \left(1 - G_1(t)\right) e^{-\lambda t} \int_{\delta=0}^{t} \left(1 - F_{Z_{2F}^*}(t-\delta)\right)$$

$$\cdot \left(F_{Z_{23}}(t-\delta) - F_{Z_{23F}^*}(t-\delta)\right) (N-2)\lambda e^{-(N-2)\lambda\delta} d\delta$$

$$= \left(1 - G_1(t)\right)(N-2)\lambda e^{-(N-1)\lambda t} \int_{x=0}^{t} \left(1 - F_{Z_{2F}^*}(x)\right)$$

$$\cdot \left(F_{Z_{23}}(x) - F_{Z_{23F}^*}(x)\right) e^{(N-2)\lambda x} dx$$

$$= \frac{N-2}{2(N-5)} \left(1 - G_1(t)\right)\left(2e^{-3\lambda t} + (N-6)e^{-(N-1)\lambda t} - (N-4)e^{-(N-3)\lambda t}\right)$$

$E_{2'3}(t)$ is the probability that the process starting from state 2' is in state 3 at time t. Let $Z_{2'F}^*$ be the random variable for the time to RAID failure from state 2' due to disk failures whose mirrors have failed in state 2', and $Z_{2'3F}^*$ be the random variable for the time to enter F via state 3 (i.e. due to two subsequent failures beyond those in state 2'). $Z_{2'F}^*$is exponentially distributed with parameter 2λ and Z_{23F}^* follows a two-stage hypoexponential distribution with parameter $(N - 4)\lambda$ and λ. $E_{2'3}(t)$ is then given by:

$$\begin{aligned} E_{2'3}(t) &= Pr\{R > t, Z_{2'F} > t, Z_{2'3F}^* > t \geq Z_{2'3}\} \\ &= Pr\{R > t, Z_{2'F}^* > t, Z_{2'3F}^* > t\} - Pr\{R > t, Z_{2'F}^* > t, Z_{2'3} > t\} \\ &= \left(1 - G_1(t)\right) \cdot e^{-2\lambda t} \cdot \left(F_{Z_{2'3}}(t) - F_{Z_{2'3F}^*}(t)\right) \\ &= \frac{N-4}{N-5}\left(1 - G_1(t)\right)\left(e^{-3\lambda t} - e^{-(N-2)\lambda t}\right) \end{aligned} \qquad (3.47)$$

From the local kernel distributions, the mean sojourn times are obtained as:

$$\alpha_{00} = 1/N\lambda \tag{3.48}$$

$$\alpha_{11} = \int_0^{\infty} \big(1 - G_1(t)\big) e^{-(N-1)\lambda t} dt \tag{3.49}$$

$$\alpha_{2'2'} = \int_0^{\infty} \big(1 - G_1(t)\big) e^{-(N-2)\lambda t} dt \tag{3.50}$$

$$\alpha_{FF} = \int_0^{\infty} \big(1 - G_2(t)\big) dt \tag{3.51}$$

$$\alpha_{12} = (N-2) \int_0^{\infty} \big(1 - G_1(t)\big)\big(e^{-(N-2)\lambda t} - e^{-(N-1)\lambda t}\big) dt \tag{3.52}$$

$$\begin{aligned}\alpha_{13} &= \frac{N-2}{2(N-5)} \int_0^{\infty} \big(1 - G_1(t)\big) \\ &\quad \cdot \big(2e^{-3\lambda t} + (N-6)e^{-(N-1)\lambda t} - (N-4)e^{-(N-3)\lambda t}\big) dt\end{aligned} \tag{3.53}$$

$$\alpha_{2'3} = \frac{N-4}{N-5} \int_0^{\infty} \big(1 - G_1(t)\big)\big(e^{-3\lambda t} - e^{-(N-2)\lambda t}\big) dt \tag{3.54}$$

Since the storage system is available in state 0, 1, 2, 2′ and 3, the expected data availability is computed by:

$$\begin{aligned}A_{RAID10} &= \sum_{i=0,1,2,2',3} \pi_i = \sum_{i=0,1,2,2',3} \frac{\sum_{k\in\Omega} v_k \alpha_{ki}}{\sum_{k\in\Omega} v_k \sum_{l\in\Omega} \alpha_{kl}} \\ &= \frac{v_0\alpha_{00} + v_1(\alpha_{11} + \alpha_{12} + \alpha_{13}) + v_{2'}(\alpha_{2'2'} + \alpha_{2'3})}{v_0\alpha_{00} + v_1(\alpha_{11} + \alpha_{12} + \alpha_{13}) + v_{2'}(\alpha_{2'2'} + \alpha_{2'3}) + v_F\alpha_{FF}}\end{aligned} \tag{3.55}$$

Assigning reward rates {r_j} to each state, the performability is:

$$P_{RAID10} = \sum_{j \in \Phi} \pi_j \cdot r_j$$
$$= \frac{v_0\alpha_{00}r_0 + v_1\alpha_{11}r_1 + (v_1\alpha_{12} + v_{2'}\alpha_{2'2'})r_2 + (v_1\alpha_{13} + v_{2'}\alpha_{2'3})r_3}{v_0\alpha_{00} + v_1(\alpha_{11} + \alpha_{12} + \alpha_{13}) + v_{2'}(\alpha_{2'2'} + \alpha_{2'3}) + v_F\alpha_{FF}} \tag{3.56}$$

From the availability, the mean time to data loss can be obtained as:

$$MTTDL_{RAID10} = \frac{A_{RAID10} \cdot \alpha_{FF}}{1 - A_{RAID10}}$$
$$= \frac{v_0\alpha_{00} + v_1(\alpha_{11} + \alpha_{12} + \alpha_{13}) + v_{2'}(\alpha_{2'2'} + \alpha_{2'3})}{v_F} \tag{3.57}$$

As an example, assume deterministic times for disk rebuild and RAID reconstruction processes, then the transition probabilities and the mean sojourn times are rewritten using $G_1(t) = u(t - \tau_1)$ and $G_2(t) = u(t - \tau_2)$:

$$a = e^{-(N-1)\lambda\tau_1} \tag{3.58}$$

$$b = (N-2)e^{-(N-2)\lambda\tau_1}\left(1 - e^{-\lambda\tau_1}\right) \tag{3.59}$$

$$c = \frac{(N-2)(N-4)}{N-5}\left\{\frac{e^{-3\lambda\tau_1}}{N-4}\left(1 - e^{-(N-4)\lambda\tau_1}\right) - e^{-(N-2)\lambda\tau_1}\left(1 - e^{-\lambda\tau_1}\right)\right\} \tag{3.60}$$

$$d = e^{-(N-2)\lambda\tau_1} \tag{3.61}$$

$$e = \frac{N-4}{N-5}e^{-3\lambda\tau_1}\left(1 - e^{-(N-5)\lambda\tau_1}\right) \tag{3.62}$$

$$\alpha_{11} = \frac{1}{(N-1)\lambda}\left(1 - e^{-(N-1)\lambda\tau_1}\right) \tag{3.63}$$

$$\alpha_{12} = \frac{1}{\lambda}\left(1 - e^{-(N-2)\lambda\tau_1}\right) - \frac{(N-2)}{(N-1)\lambda}\left(1 - e^{-(N-1)\lambda\tau_1}\right) \tag{3.64}$$

$$\alpha_{13} = \frac{N-2}{2(N-5)\lambda}\left\{\frac{2}{3}\cdot\left(1-e^{-3\lambda\tau_1}\right)+\frac{N-6}{N-1}\cdot\left[1-e^{-(N-1)\lambda\tau_1}\right]-\frac{N-4}{N-3}\cdot\left[1-e^{-(N-3)\lambda\tau_1}\right]\right\} \tag{3.65}$$

$$\alpha_{2'2'} = \frac{1}{(N-2)\lambda}\left(1-e^{-(N-2)\lambda\tau_1}\right) \tag{3.66}$$

$$\alpha_{2'3} = \frac{N-4}{(N-5)\lambda}\left\{\frac{1}{3}\left(1-e^{-3\lambda\tau_1}\right)-\frac{1}{(N-2)}\left(1-e^{-(N-2)\lambda\tau_1}\right)\right\} \tag{3.67}$$

$$\alpha_{FF} = \tau_2 \tag{3.68}$$

Alternatively, if we assume the disk rebuild time and the RAID reconstruction times are exponentially distributed with rate α and μ, respectively, the MRGP model becomes CTMC and the memoryless property holds. Substituting $G_1(t) = 1 - e^{-\mu t}$ and $G_2(t) = G_1(t) = 1 - e^{-\alpha t}$ yields the fully symbolic expressions for availability and performability for RAID10 storage system, which are consistent with those derived in the previous paper [28].

3.4 Numerical Results

In this section I present the results of my numerical study on the proposed MRGP models. First, storage reliability is evaluated by the mean years to RAID storage failure using the MRGP model. Next I focus on data availability perspectives where I compare the results of CTMC model and MRGP model and show that the difference between the estimated downtimes computed by CTMC and that by MRGP is negligibly small. Moreover, I compare the performability of RAID6 and RAID10 storage systems based on the proposed models with storage benchmark results. Finally, I conduct the sensitivity analysis to rebuild time distribution using gamma distribution with different parameter values.

3.4.1 RAID storage reliability

Using the MRGP models, first I compute the mean years to RAID storage failure as the reliability measure from (3.19) and (3.57). The parameter values used are shown in Table 3.1. The number of disks *N* is set to six because the experimental storage system used in the performability study consists of six disks. I assume the mean time to individual disk failure ranges from 10^4 hours to 10^6 hours according to specifications provided by disk vendors and experience of bad batches by users. Earlier studies of disk failure statistics also support this range of values [19][29]. The disk rebuild time observed in the test system varies from an hour to a few hours. Considering the rebuild time might prolong due to workload conditions and/or data volume, I vary the values in the range from one hour to 24 hours. When a storage failure occurs, manual operation is needed to reconstruct the storage system and recover the data from backup. I assume it takes one day for these manual operations.

Table 3.1. Parameter Values

Parameters	***Values***	***Description***
N	6	Number of disks in the array
$1/\lambda$	10^4– 10^6 [hours]	Mean time to disk failure
$1/\mu$ (=τ_1)	1 –24 [hours]	Mean time to disk rebuild
$1/\alpha$(=τ_2)	24 [hours]	Mean time to storage reconstruction

Figure 3.4shows the comparison of RAID storage reliabilities by varying the mean time to disk failure ($1/\lambda$) in the range [10^4– 10^6] with different disk rebuild rates (1/2 or 1/24). In the model, I assume deterministic transitions for the disk rebuild time (two hours or 24 hours). As the figure shows, the RAID6 architecture achieves higher reliability compared to the RAID10 architecture with the same number of disks, regardless of the MTTF of a single disk. Since the computation of mean time to storage failure uses the MRGP model, the non-exponential

distributions of disk rebuild times are accounted for. The derivation of MTTDL under general rebuild times is studied in [25][30].Although the derivation process in this chapter is different from those, I confirm that the findings here about the impact of variance on MTTDL agree with their result through a sensitivity study using gamma distribution in Section 4.4.

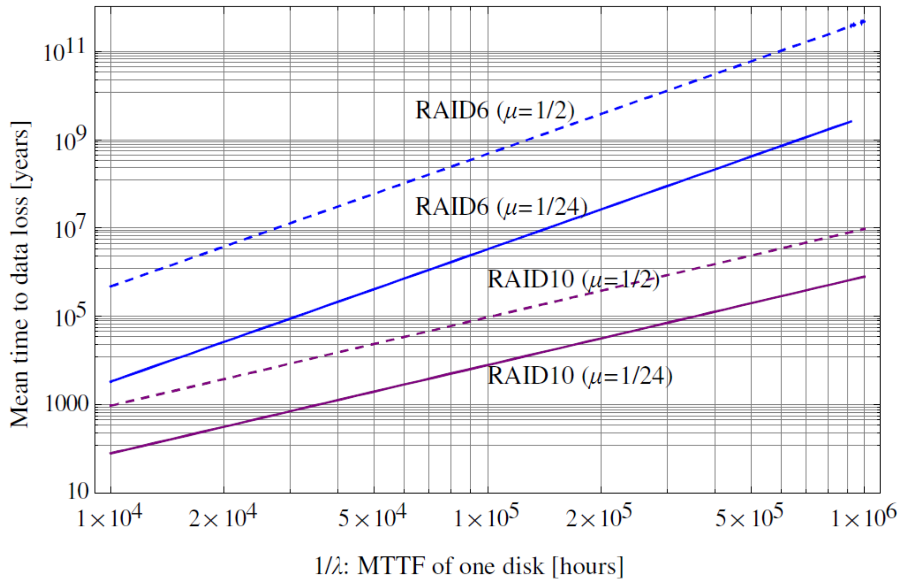


Figure 3.4. RAID Storage Reliability by Years to Storage Failure

3.4.2 Data Availability

Provided that the data is restorable from backup once the storage fails with data loss, data availability becomes an important dependability measure that predicts how long the user will be unable to access the data on the storage, i.e. the storage system downtime. In this section this metric is computed from the data availability point of view for RAID6 and RAID10 storage systems using (3.16) and (3.55). Table 3.2 shows the downtime in seconds per year computed by the RAID6 and RAID10 MRGP models and the downtime of a disk array with no redundancy (i.e., RAID0 with six disks). The estimated downtime of RAID6 storage system is several orders of magnitude smaller than that of RAID10 regardless of the disk failure time and rebuild time. This is because RAID10 can fail with two disk failures, and imply that given the same number of disks, RAID6 configuration is preferable in terms of data availability.

Table 3.2. Data Availability Comparison: RAID6 vs. RAID10

$1/\mu$ [hours]	$1/\lambda$[hours]	Downtime per year [seconds]		
		RAID6	RAID10	RAID0
2	10^4	0.18150	90.8143	447671.9
	10^5	0.00018	0.90823	45346.54
	10^6	$<10^{-6}$	0.00908	4540.53
24	10^4	25.9041	1088.40	447671.9
	10^5	0.02613	10.8975	45346.54
	10^6	0.00003	0.10899	4540.53

Next I evaluate the impact of the exponential distributional assumption of the rebuild operation times on the downtime through sensitivity analyses. First, I fix the mean time to disk rebuild to two hours and vary the mean time to disk failure from ten thousand hours to a million hours. Figure 3.5 shows the downtimes from CTMC and MRGP models of RAID6 and RAID10.

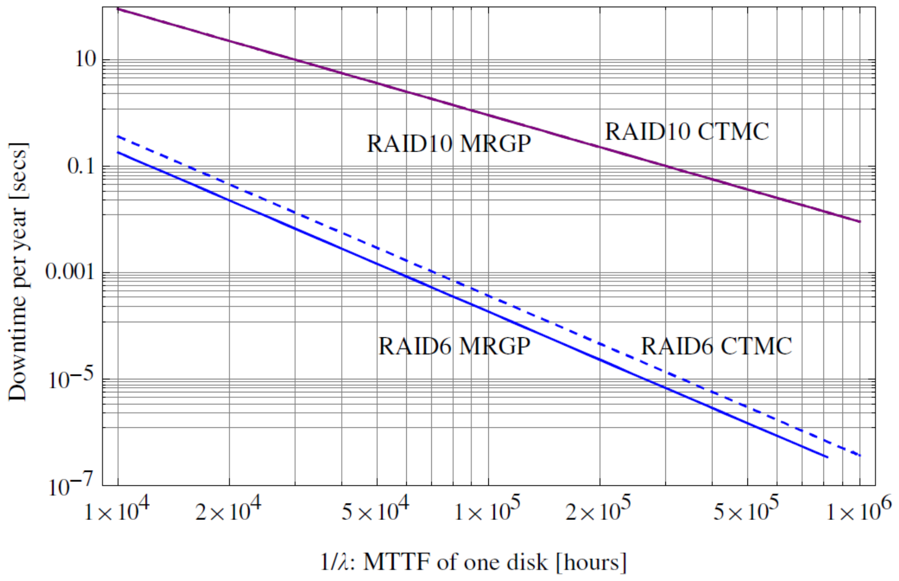


Figure 3.5. Comparison of CMTC and MRGP Downtime by Varying Single Disk MTTF

As the RAID6 results show, CTMC tends to overestimate the downtime due to the memoryless assumption of rebuild times which erroneously increases the latter. However, the difference is negligibly small (<0.2 seconds) especially in the practical range of disk failure rates (i.e., less than 10^{-4} hour^{-1}). For RAID10, although both MRGP and CTMC results are plotted, the difference between the two curves is too small to be visible. The difference between the CTMC and MRGP results is less than 10^{-9} in the whole range of the sensitivity results.

Next I look into the sensitivity of the downtime to mean rebuild times by fixing the disk failure rate to 10^{-6} hour^{-1}. Figure 3.6 shows the comparison results obtained from CTMC and MRGP models for RAID6 and RAID10, and indicates that the CTMC overestimates the downtime regardless of disk rebuild times. Although the difference becomes larger as the disk rebuild time increases, it is marginal (<0.1 second) in both cases.

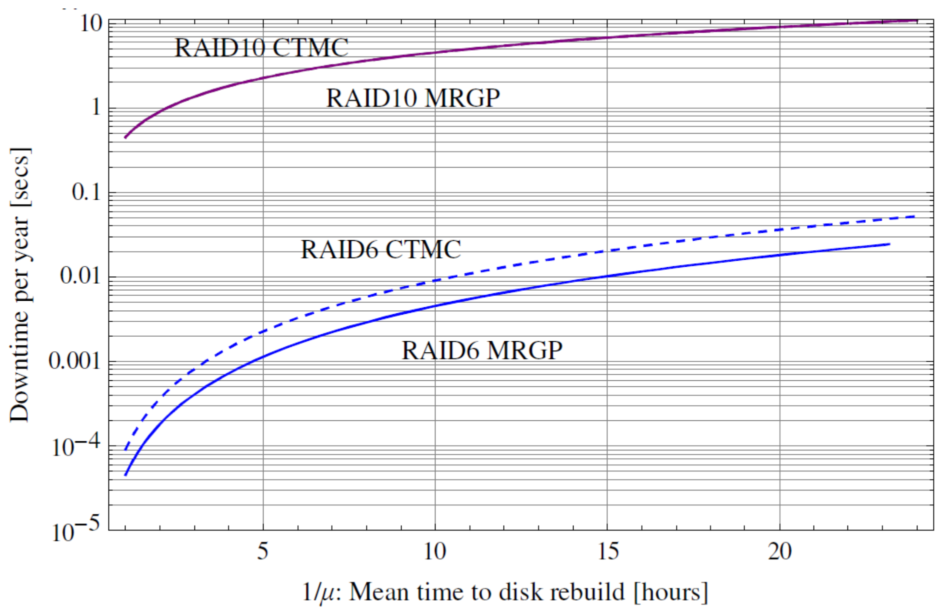


Figure 3.6. Comparison of CMTC and MRGP Downtimes by Varying Disk Rebuild Time

Finally, I conduct the sensitivity analysis of the number of disks by setting λ = 10^{-6} and μ = 0.5. In general, as the number of disk increases, the storage reliability worsens because of the increased failure rate. Figure shows the results of the estimated downtimes for RAID6 and RAID10. As the number of disks constituting RAID6 increases, the difference between the computed downtimes becomes larger. However, the difference is very small (<0.01 second) even when the number of disks is 20. For the case of RAID10, the difference is less than 0.01 second in the whole range of the graph.

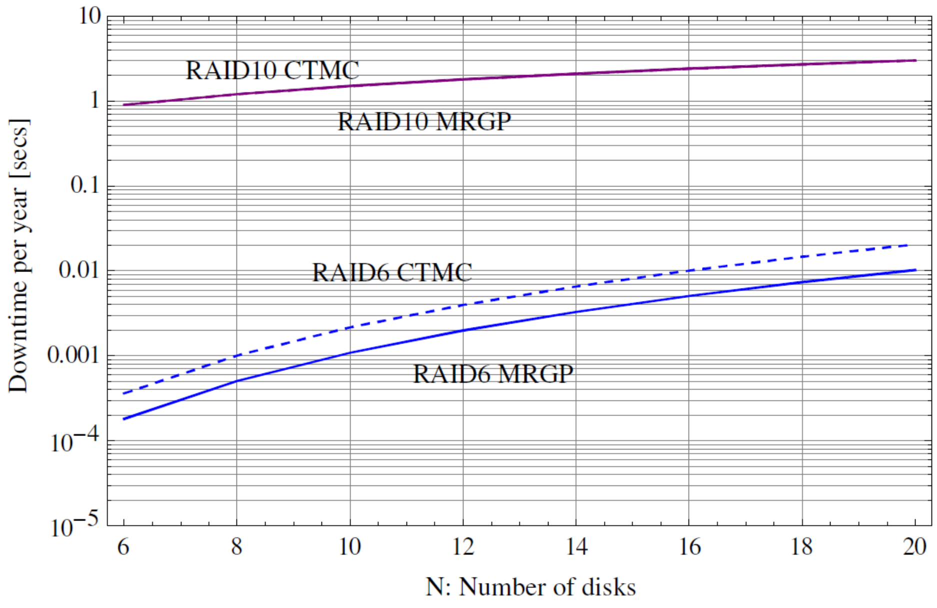


Figure 3.7. Comparison of CMTC and MRGP Downtimes by Varying Disk Number

From the above observations, the CTMC model tends to overestimate the downtime due to its assumption of memoryless property for the disk rebuild time. However, the difference is generally negligible in practical ranges of disk failure rates and disk rebuild times.

3.4.3 Performability comparison

The second part of the experiments focuses on the performability of RAID storage systems. In addition to data availability, the performance degradation caused by disk failures should be considered in the design of storage configurations. To quantify the performance degradation, firstly I conduct disk benchmarks on the experimental RAID storage systems.

The test system comprises two physical servers equipped with the same hardware components including hardware RAID controller. One server is configured as a RAID6 system with six disks and another one configured as a RAID10 with the same number of disks. All the disks used in the test system have the same specification produced by the same vendor. The disk benchmark is performed with fio [31], a tool for disk benchmarking. In my experiments, fio creates four jobs that continuously issue I/O requests of 1MB block with specified access pattern; either sequential read/write or random read/write. In order to apply the benchmark results to the degraded RAID storage system, I emulate disk failures by manually ejecting disks from the servers.

Figure 3.8 summarizes the benchmark results that show the average bandwidth measured by fio with different access types (i.e., sequential read/write or random read/write) for each degradation level of RAID6 and RAID10 storage systems. The degradation level corresponds to the number of failed disks in the storage system. Note that RAID6 tolerates any two disk failures, while RAID10 might tolerate up to triple disk failures. An interesting observation is that the read performance of RAID6 decreases considerably after disk failures (in both sequential and random accesses). In particular, sequential read performance of RAID6 in state 0 is 44% higher than that of RAID10, but the performance advantage is overtaken by RAID10 after one disk failure. The considerable performance degradation is caused by the decrease in striping level due to disk failures in RAID6; meanwhile the striping level of RAID10 is not reduced as long as the RAID as a whole survives. On the other hand, the write performance of RAID6 does not decrease significantly upon disk failures because the level of striping has relatively small impacts on the write overhead including parity generation.

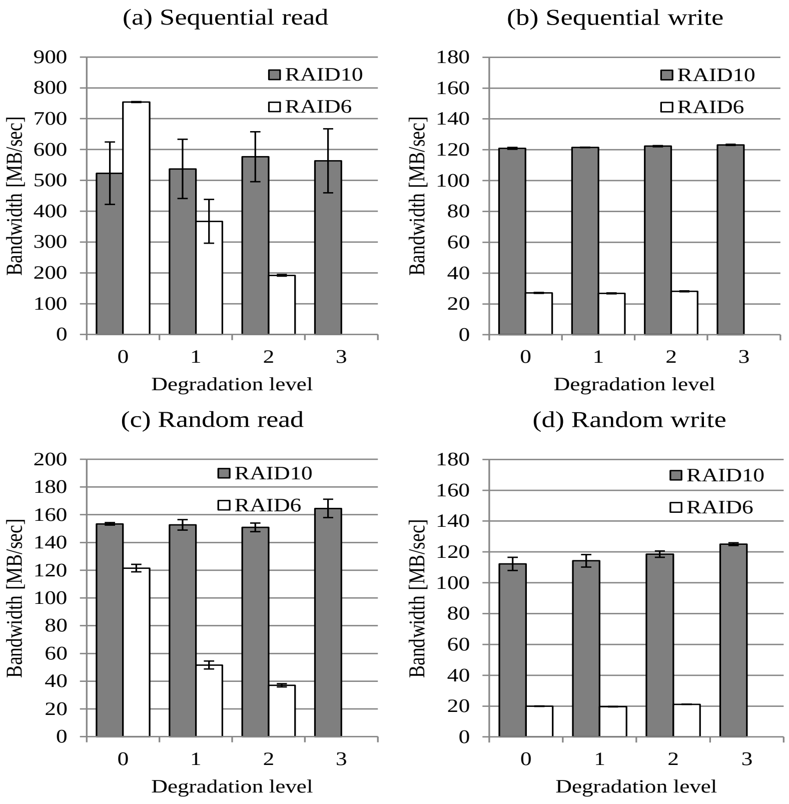


Figure 3.8. Benchmark Results using fio; (a) Bandwidth for Sequential Read, (b) for Random Read, (c) Sequential Write and (d) Random Write

The benchmark results are then used for reward assignment in the MRRM models based on the MRGP. Specifically, I assign the read access throughput values in MB per second at different degradation stages to the corresponding MRGP states in RAID6 and RAID10 models. I compute the performability of read accesses only, because the write performance is not much influenced by disk failures as presented in the benchmark results. I set the mean disk rebuild time to two hours and 24 hours and vary the mean time to disk failure from a thousand hours to a million hours. Figure 3.9 and Figure 3.10 show the computed performability for sequential read access and random read access, respectively. Despite the significant performance degradation after disk failures, the sequential read performance of RAID6 is still superior. Meanwhile, the random read performance of RAID6 is apparently worse than that of RAID10.

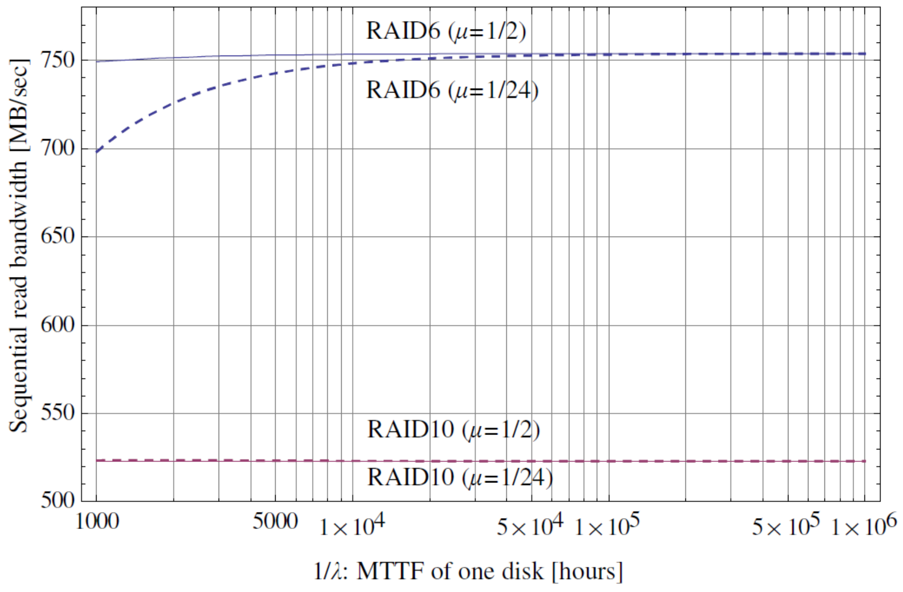


Figure 3.9. Performability Comparison of Sequential Read Access

Next I compare the results with the performability predicted by the CTMC models. Using the expressions presented in [28], I compute the difference ΔP_i ($i \in$ {RAID6, RAID10}) between the performability values predicted by MRGP and by CTMC. Figure 3.11 and Figure 3.12 show the performability differences in sequential read access and random read access, respectively.

As Figure 3.12 shows, there is a change point around λ = 1120in ΔP_{RAID6} at μ = 1/24. The difference ΔP_{RAID6} increases for $1/\lambda$< 1119, while it starts decreasing in $1/\lambda$> 1120.

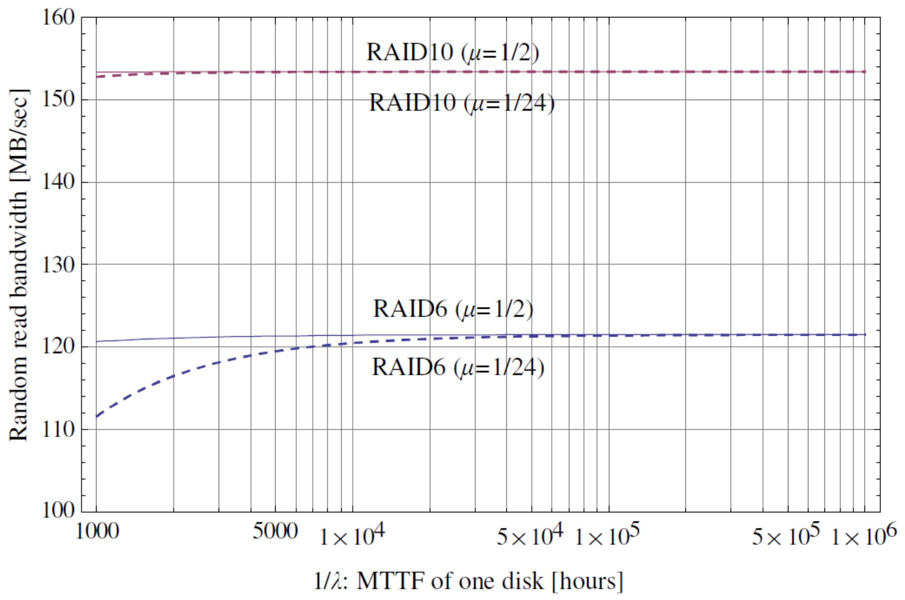


Figure 3.10. Performability Comparison of Random Read Access

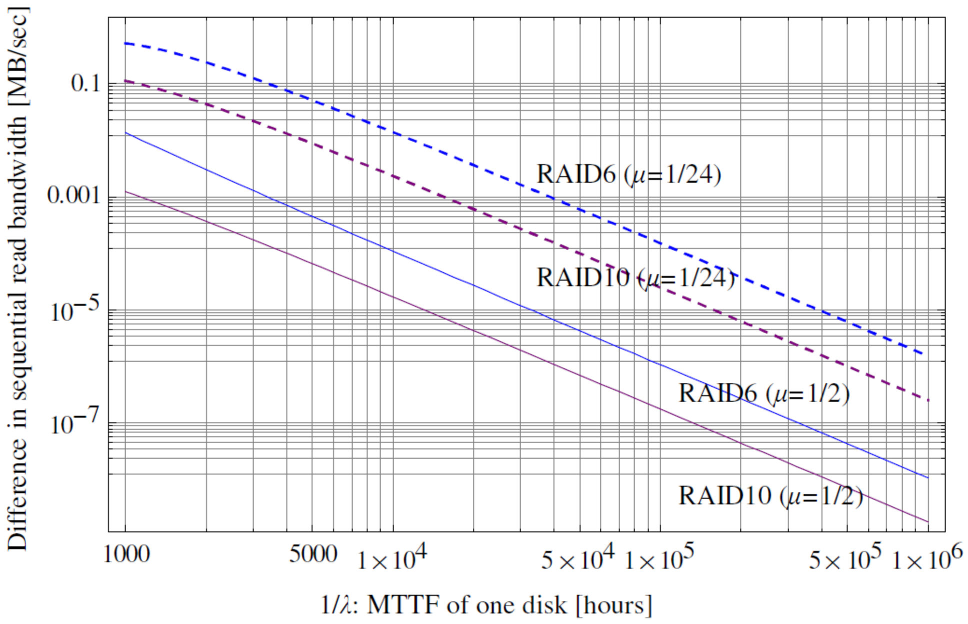


Figure 3.11. Difference in Performability Prediction of Sequential Read Access

In general, the difference becomes small as the mean time to disk failure increases. For example, if we assume the mean time to disk failure to be 10^6 hours, the performability differences among the RAID models (CTMC vs. MRGP) are less than 10^{-5} MB/sec in sequential read access and less than 10^{-7} MB/sec in random read access, regardless of the rebuild rate and RAID architecture (i.e., RAID6 or RAID10). In both sequential and random access cases, the difference observed among the RAID10 MRGP and CTMC is generally smaller than that among the RAID6 MRGP and CTMC. Also it can be observed that the smaller rebuild rate (μ = 1/24) generally causes larger difference between the results of CTMC and MRGP especially when the mean time to disk failure is larger than 2000 hours.

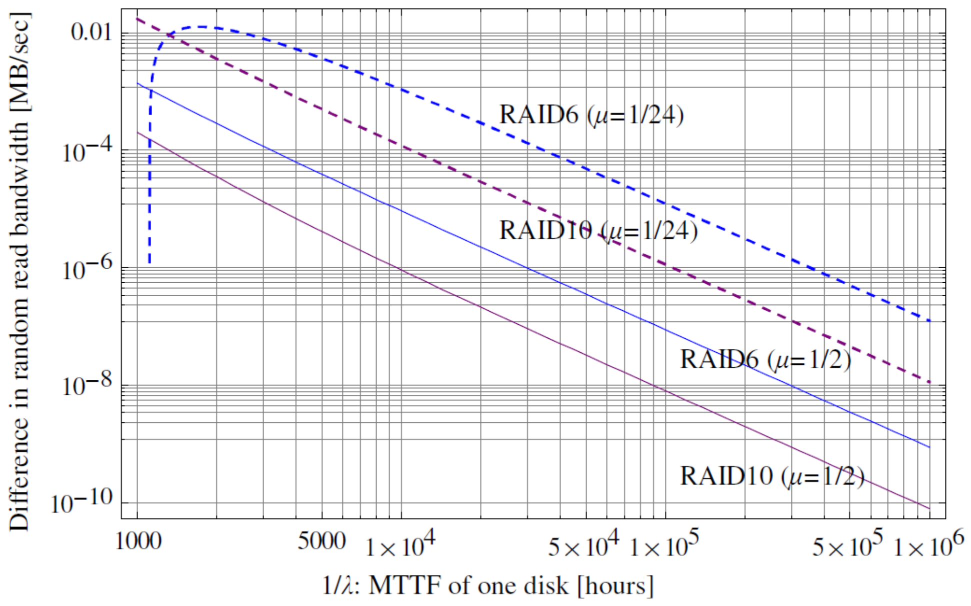


Figure 3.12. Difference in Performability Prediction of Random Read Access

The relative difference, computed from the difference ΔP_i divided by the performance in the failure-free state (r_0), is less than 0.01 in the whole range of the mean time to disk failure (not presented in the graphs). In contrast to the data availability prediction studied in

Section3.4.2, the approximation error in performability prediction from CTMC may not be negligible especially when the performance drastically changes in degraded states. The MRGP models in this chapter provide more accurate prediction of performability in such cases.

3.4.4 Sensitivity to rebuild time distribution

In the above comparative study, I used deterministic rebuild times in MRGP models. Although not much variance is observed in disk rebuild time in the experimental system, rebuild times in reality may vary due to workload changes or other external factors. To look into the impact of variance in rebuild time, I conduct another sensitivity analysis using gamma distribution for rebuild time with same mean ($1/\mu$ = 2). The probability density function of gamma distribution with mean value ($1/\mu$) is defined by:

$$g(x) = (\beta\mu)^{\beta} x^{\beta-1} \frac{e^{-\beta\mu x}}{\Gamma(\beta)}, \quad \text{where } \Gamma(\beta) = \int_0^{\infty} x^{\beta-1} e^{-x}\, dx \tag{3.69}$$

β (>0) is the shape parameter of the distribution and if β is an integer value the distribution represents an Erlang distribution. Note that the exponential distribution is the special case of the gamma distribution with β = 1.

I perform a sensitivity analysis on the RAID6 MRGP model by using the following parameter values for β: [0.1, 0.5, 1, 2, 10]. Figure 3.13 shows the mean time to storage failure under varying disk failure rate, which indicates that the mean time to storage failure becomes longer than the CTMC case (β = 1) if β > 1, while it becomes smaller if β < 1. Since the second moment of the above gamma distribution is $1/(\beta\,\mu^2)$, the smaller βincreases the variance that leads to lower mean time to storage failure, which agrees with the findings from [25]. Similarly, Figure 3.14 shows the results of the sensitivity analysis in terms of storage downtime.

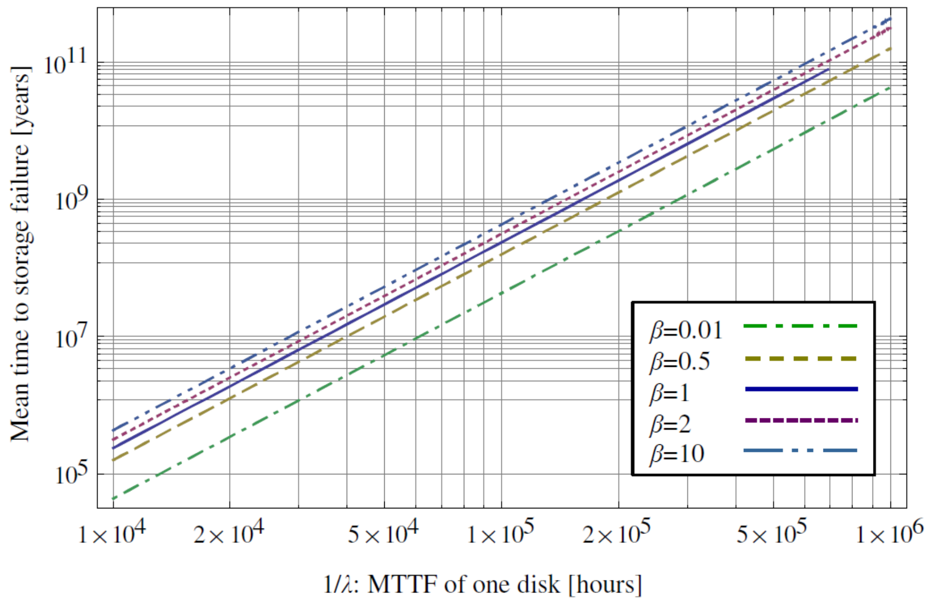


Figure 3.13. Sensitivity of Mean Time to Storage Failure to Rebuild Time Distribution

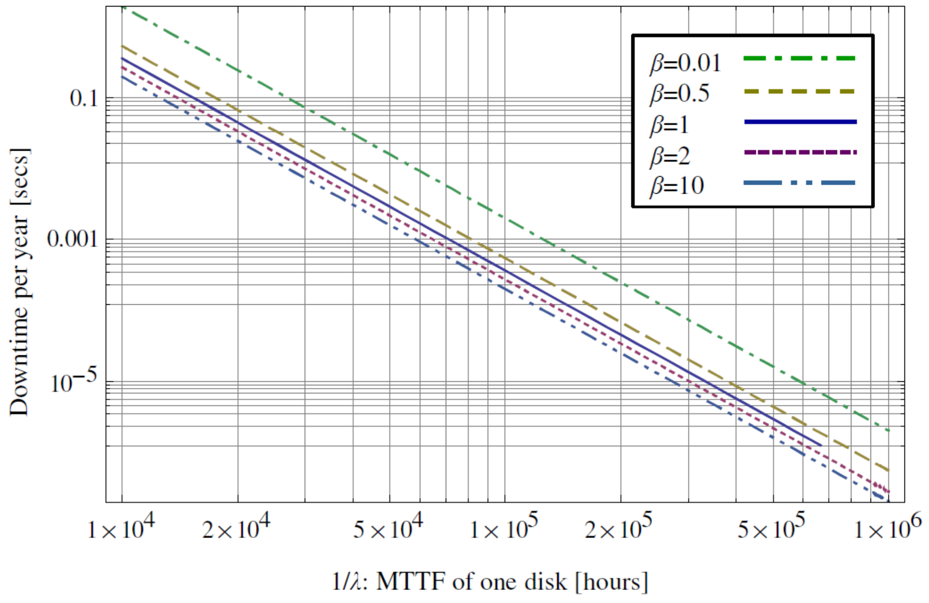


Figure 3.14. Sensitivity of Storage Downtime to Rebuild Time Distribution

The downtime becomes larger than the CTMC case (β = 1) if β < 1, while smaller if β > 1. Considering that the rebuild operation typically requires fixed amount of time at the minimum, the shape parameter β in reality tends to be large. In the extreme case, the distribution is unit step function (i.e., deterministic), as assumed in the previous sections. In conclusion, although there is some impacts of rebuild time distributions on reliability and availability, the difference is generally negligible in the practical range of the parameter values.

3.5 Related Works

Performability is defined as a composite measure of reliability (or availability) and performance of degradable systems [32]. In the late 70s, Beaudry presented performance-related reliability measures to take into account the different performance levels of degradable systems [33]. Huslende considered performance reliability by assuming a minimum performance threshold and presented a threshold-based performability measure [34]. Smith et al. evaluated the performability of multiprocessor system by complementary distribution of time-averaged accumulated performance measure [35]. Some other related papers about the performability study are summarized in [36]. In this chapter, I adopt MRRM based performability analysis, which was first studied by Logothetis et al [37], since it is necessary to capture the non-exponential nature of RAID rebuild times.

An approach for assessing the performability of RAID storage system is presented by Sun et al [38]. They construct a CTMC model which captures the expected behavior of a storage system and combine the model with performance benchmark results for performability assessment. A new performability metric called P-Graph is used to visualize performability of storage systems. Although their availability model is comprehensive as it captures failures of several components like RAID controller, all the state transition times are assumed to be exponentially distributed. The performability analysis in this chapter extends their work with non-exponential distributions of the rebuild times.

Thomasian et al. presented analytical studies on reliability and performance of storage systems with different RAID configurations [39][40][41]. Several different RAID1 organizations are compared with each other [39] and their performances are compared against that of RAID5 systems [40]. While MTTDL has been used as a reliability measure in their papers, in a recent study, the authors compute the reliability, in terms of MTTDL, without considering repair operations [41]. My performability study differs from their work as we compute the performability as a combination of availability and performance, focusing on a simple RAID10 configuration evaluate the performability with real benchmark results.

It is well-known that the reliability of disk-based storage systems is highly influenced by latent sector errors (LSE), i.e., errors that go undetected until the corresponding disk sectors are accessed. A large-scale field study on LSE revealed the high degree of temporal locality between successive LSE occurrences [42]. The practical approach to cope with LSEs is disk scrubbing that continually scans the disk in order to detect LSEs proactively [43][44]. Intra-disk redundancy is proposed as another mechanism to protect against LSEs [45]. Comparative study of these two mechanisms is found in [46][47] and further comprehensive analysis and corrected arguments are presented by Iliadis et al 0. The reliability model incorporating the impacts of LSEs is presented by Wu et al. [48] and the effectiveness of disk scrubbing is first studied by Schwarz et al [44]. Besides LSE, undetected disk errors (UDE) are another type of problem in storage systems. UDEs are silent data corruption events and have potential to corrupt data being delivered to user applications. Rozier et al. presented a reliability model of RAID system with UDE and presented a hybrid solution method which combines discrete event simulation and analytic-numerical solution [49]. In this paper, I do not incorporate the impacts of LSEs, scrubbing or UDEs in the models. It is challenging to extend the MRGP model to incorporate these features, since I believe analytical-numerical solution may become ineffective in this case. As a result, discrete-event simulation or hybrid solution approach as in [20][49] may be needed.

3.6 Conclusions and Future Directions

In this chapter I presented MRGP models for RAID storage systems. MRGP can express the time-dependent rates behavior of disk rebuild and storage reconstruction times and hence the memoryless assumption of the conventional RAID reliability models is relaxed. Nevertheless, the numerical study shows that the difference between the computed downtime by CTMC and that by MRGP is insignificant if disk rebuild time is a few hours and the mean time to disk failure is in the rage of 10^4-10^6 hours. This implies that the memoryless assumption for disk rebuild time as in the CTMC models for RAID storage system is acceptable for practical use. On top of the MRGP models, I also presented the performability comparison between RAID10 and RAID6. Disk benchmark results are used to assign reward rates for the MRGP models and the sensitivity to the mean time to disk failure is analyzed. Although RAID10 generally achieves better performance than RAID6, RAID6 has an advantage in performability of sequential read access.

Compared with CTMC models, the MRGP model provide more accurate performability prediction.

4. Cloud Storage Provisioning Modeling

4.1 Overview

With the increasing prevalence of cloud computing paradigms, more IT functionalities other than pure computing are being made available through the cloud service model. Among these, cloud storage plays a prominent part in an age of explosive data growth. Storage provided in the form of a cloud service enjoys the benefits commonly associated with the cloud computing paradigm, such as flexibility in investment and use and reduced cost to the user due to the economy of scale available to the service providers. Furthermore, cloud storage synergizes well with the cloud computing services, and the popularity of the latter that is already in place means consumers are likely to increase their use of cloud storage as they rely more on cloud computing to provide and/or improve their core businesses. All of these factors combined together make cloud storage a natural focus for many major cloud providers, and the current market has seen the emergence of many competing cloud storage products, with both commercial ones such as Amazon S3 [50], Microsoft Azure Storage [51] and ECM Atmos [52] as well as open source ones such as Openstack Swift [53].

Similar to other forms of cloud computing services, cloud storage is also regulated by Service Level Agreements (SLA) between the provider and the customer, which specifies the level of service (in terms of measures such as storage availability [54]) that should be provided and what compensation is needed if the level of service is not met. In the storage context, availability and durability are to a large extent provided through data replication and backup. To make such techniques possible, it is important for a cloud storage provider to ensure that sufficient storage capacity is in place to support provisioning of data storage with adequate data protection. This is a capacity planning problem in the cloud context, and to effectively solve this problem it is helpful to have a quantitative approach for predicting the result of storage provisioning under specific storage capacity and request workloads.

In [1], an evaluation framework for predicting the performance of cloud system resource provisioning was proposed, which could be applied to the storage provisioning scenario as well. The framework was based on stochastic models that capture the interactions among different system components involved in the process of resource provisioning for customer requests. Specifically, the framework focused on a system consisting of a provisioning module and different resource pools into which arriving requests are provisioned. The overall levels of request losses, either due to inadequate buffer space at the provisioning module or insufficient resource at the pools, are obtained as the primary measures of interest. Compared to alternative evaluation approaches such as simulation, the stochastic model enjoys lower cost of development and faster solution. Nevertheless, model scalability could present an issue as a monolithic model is likely to suffer the curse of dimensionality, especially when applied to a large system such as a cloud with thousands of resource instances. The work of [1] addresses this concern by using a decomposition modeling approach in which various system sub-models are solved individually and their interdependencies are then resolved through fixed-point iterations [55]. More specifically, the system is captured in two sets of submodels, one for the provisioning module and another for the resource pools. For a resource pool, the behavior of each resource instance in a pool is captured using the same submodel, and then the output of the submodel is related to the behavior of the whole pool using a closed-form mathematical expression. Regarding the dependencies among the submodels, the output of the provisioning module affects the request arrival rates at the resource pools, while the resource availabilities of the pools in turn affect the provisioning results of requests inside the provisioning module. These two submodels (decision module and the resource pools) are solved iteratively by successive substitution, with the former's output fed into the latter and vice versa, until both sets of output converge within a given tolerance.

Although the decomposition approach offers good scalability, its accuracy turns out to be inadequate in many cases. In this chapter I present an alternative set of models for the cloud storage provisioning scenario, with the goal of achieving both model scalability and accuracy. The chapter consists of the following parts:

1) Development of a set of stochastic models that has good accuracy and captures important characteristics of the cloud storage provisioning process. While the approach uses the same stochastic formalisms as [1], it utilizes a novel observation regarding the system to establish a different modeling approach. In solving part of the models this chapter also utilizes matrix geometric method to achieve better efficiency. Finally, a more complex (non-Poisson) request arrival process is also introduced and solution methods for part of system submodels under the new arrival process are discussed.
2) Development of a simulation-numerical hybrid solution method that further improves model scalability while reducing overall solution time compared to a pure

simulation approach. Specifically, the hybrid method follows a hierarchical pattern in which the lower level model is solved using simulation, whose results are then feed into the upper level model which is then solved by numerical methods.

3) Development of a Markov Chain Monte Carlo (MCMC)-based solution method as an alternative to the simulation in the hybrid model. Correctness proof of the method is presented, and its accuracy and efficiency are investigated.

This Chapter is organized as follows. Section 4.2 provides a brief description of the cloud storage system under consideration and the existing models in [1]. Section 4.3 describes the new stochastic model and present numerical results concerning its accuracy and scalability. In Section 4.4, I describe the hybrid solution approach of the new decomposed model and present results regarding its improvement in both scalability and solution efficiency. Section 4.5 describes the MCMC method, establishes its validity in the context of this chapter and investigates its utility. In Section 4.6, I review some existing works on cloud system evaluation. Finally, Section 4.7 concludes this chapter with directions for future work.

4.2 System Scenario and Existing Models

4.2.1 System Scenario

The cloud storage system considered in this chapter consists of a Resource Provisioning Decision Engine (RPDE) and a number of physical storage devices (such as sets of RAID installations) organized into one or more device pools. The storage space on each device is divided into a number of partitions that can be assigned to customers for use based on their requests. The RPDE is responsible for making decisions regarding the customer requests for storage space, and tries to provision each request with a storage partition while holding waiting requests in a queue. Typically, such a provisioning decision involves multiple factors, such as the size of the requested storage provision and the current usage of the available storage capacity. In this chapter I make the assumptions that all requests require the same amount of storage space (one partition), occupy the provisioned storage resource for the same duration on average, and that the lengths of all inter-event times are exponentially distributed. Some discussions regarding these assumptions are provided below.

1) While in reality requests for storage space differ in their requested sizes and storage occupation duration lengths, restricting attention to a uniform workload allows me to focus on the issues from the modeling side. When using the models in practice, the actual workload can be classified into groups (as is done in [56]) based on their requested size and resource usage duration, and then a separate model can be used to analyze each group.
2) Regarding the exponential inter-event time distributions, this choice is mainly made to simplify the modeling process. However, when using the model in situations where the exponential assumption does not hold, the assumption can be relaxed by approximating the actual distributions with a Coxian [57] distribution, which is known for its capability to approximate most distributions.

In processing a customer request for storage space, the RPDE checks the devices for unassigned partitions. If one is located, the RPDE assigns the partition to the customer; otherwise the request is dropped. Such a process takes some time to complete, and since the

RPDE has a finite buffer for temporarily holding requests it is possible for a newly arriving request to find a full RPDE buffer and get blocked from entering. The measure of interest for the provisioning system is the combined probability of rejecting a user request, either due to request dropping or blocking. Since request rejection constitutes system unavailability from the users' perspective, its quantification is important for the design of an adequate capacity plan for the storage cloud.

4.2.2 Existing Modeling Approach

As mentioned earlier, [1] developed a set of stochastic models for cloud resource provisioning. While the resource in that case is for computation (virtual machines (VM) hosted on physical machines (PM)) purposes, the logical components of the provisioning process, which are the targets in both that case and the scenario in this chapter, are the same. Therefore the method developed in [1] should also be applicable in this scenario, and it is for this reason that I briefly describe their scenario and method below while highlighting the connection to the scenario in this chapter along the way.

In [1], the PMs (analogous to the storage devices) are organized into pools with different degrees of readiness. The degree of readiness determines how long it takes for a VM (analogous to a partition) from a particular PM pool to become ready for running a customer request. A system setup consisting of three pools is considered: a hot pool, a warm pool and a cold pool. The hot pool PMs are kept at a running state and can be quickly utilized to provision incoming requests using pre-instantiated VM images. By contrast, a PM in the warm pool is kept at an energy-conservation state and thus takes additional time to become fully operational (i.e., "hot") before it can be used to provision requests. Finally, PMs in the cold pool are turned off normally, and would take the longest time to enter the running state and start provisioning customer requests. Based on the pool organization, the provisioning decision then follows a simple policy where the RPDE first tries to randomly assign a request to a hot pool PM with capacity left. Failing that, the RPDE then tries to provision the request on a warm pool PM. If this is also unsuccessful, the RPDE again retries provisioning the request in the cold pool. If still no PM has capacity, the request is dropped, i.e., rejected by the system due to insufficient resource. If a PM with capacity is found, the provisioning procedure is considered successful and the request is deposited into a waiting queue on the PM which proceeds to set up relevant resources and start execution for that request. Another form of request rejection may also occur if the request buffer at the RPDE is filled up when it is making the decision for a request, thus blocking incoming requests. Similar to the case considered in this chapter, such request rejections contribute to the user-perceived unavailability of the system.

To capture the dynamics of the system, in [1] both a monolithic model and a set of decomposed models are developed. The monolithic model explicitly captures the status of every system component, i.e., the RPDE, the status of every PM at all three pools, and the execution of every VM, simultaneously. While the monolithic model covers the system behavior in great detail, it suffers from a large underlying state space which makes it unable to scale beyond a few PMs in each pool. To address this issue, a set of decomposed models are also developed. In the decomposition approach, the status of the RPDE, as well as that of a PM in a given pool, is captured in a separate homogeneous continuous-time Markov chain model. The RPDE model yields the rates of request blocking and dropping, while a PM model gives the probability that the PM has no capacity left (i.e., PM unavailability). Assuming independence among the PMs in a given pool, the unavailability of the pool can be computed from the unavailability of a member PM. The unavailabilities of all three pools are then used for computing the rate of request dropping. Clearly, there exists a cyclic dependency among the decomposed models: the RPDE needs the unavailabilities of the pools to compute request blocking and dropping rates, while the pool unavailabilities must be computed from the unavailabilities of PMs which in turn need the request blocking rate to determine the arrival rate at each PM. Such a cyclic dependency is resolved using fixed-point iteration [55].

The decomposition approach described above offers significantly improved scalability over the monolithic model. However, the accuracy of the decomposed models turns out to be inadequate, as Table 4.1 show. The values for other system parameters are given in Table 4.2.

Table 4.1. Accuracy Results of Existing Decomposed Models

	Arrival Rate (hour^{-1})	Blocking Rate (hour^{-1})	Dropping Rate (hour^{-1})	P_h	P_W	P_c
Monolithic	2.5	2.531e-16	1.389e-4	0.9605	1.0000	1.0000
Decomposed		0	0	0.9913	1.0000	1.0000
Monolithic	5	7.667e-14	8.064e-2	0.6867	0.9762	0.9993
Decomposed		0	7.699e-7	0.8783	1.0000	1.0000
Monolithic	7.5	1.897e-12	9.807e-1	0.4813	0.8173	0.9779
Decomposed		1.201e-12	8.195e-3	0.7193	0.9961	1.0000
Monolithic	10	1.683e-11	2.918	0.3648	0.6132	0.8774
Decomposed		1.165e-11	2.810e-1	0.5917	0.9312	1.0000
Monolithic	15	3.388e-10	7.701	0.2442	0.3675	0.5586
Decomposed		2.601e-10	2.931	0.4279	0.6585	0.9758
Monolithic	20	2.768e-09	12.67	0.1832	0.2534	0.3606
Decomposed		2.241e-09	7.049	0.3329	0.4717	0.7401
Monolithic	30	5.252e-08	22.66	0.1220	0.1531	0.1960
Decomposed		4.467e-08	16.36	0.2297	0.2922	0.3968

Table 4.2. Parameter Values for Accuracy Results

Parameter	Meaning	Value
m	The number of PMs in each Pool	2
L_h	The buffer size at each hot pool PM	2
L_w	The buffer size at each warm pool PM	2
L_c	The buffer size at each cold pool PM	2
N	The maximum number of requests in RPDE (buffer size + 1)	6
n_h	The number of VMs on each hot pool PM	2
n_w	The number of VMs on each warm pool PM	2
n_c	The number of VMs on each cold pool PM	2
$1/\delta_h$	Mean decision time of provisioning on a hot pool PM	3 seconds
$1/\delta_w$	Mean decision time of provisioning on a warm pool PM	3 seconds
$1/\delta_c$	Mean decision time of provisioning on a cold pool PM	3 seconds
$1/\beta_h$	Mean provisioning time of a request on a hot pool PM	5 minutes
$1/\beta_w$	Mean provisioning time of a request on a warm pool PM	10 minutes
$1/\beta_c$	Mean provisioning time of a request on a cold pool PM	20 minutes
$1/\gamma_w$	Mean warm-up time of a warm pool PM	1 minute
$1/\gamma_c$	Mean warm-up time of a cold pool PM	8 minutes
$1/\mu$	Mean request execution time	1 hour

In Table 4.1 the values of several system measures from both the monolithic and decomposed models are compared over a range of request arrival rates at the system. The "Blocking Rate" and "Dropping Rate" stand for the rate of request blocking (due to RPDE buffer full) and the rate of request dropping (due to no PM capacity at any pool), respectively. "P_h" "P_w" and "P_c" stand for the availability, i.e., the probability that there is at least one PM with capacity in the pool, of the hot, warm and cold pools respectively. "Monolithic" indicates the results from the monolithic model, while "Decomposed" refers to those from the decomposed models. As the results in Table 4.1 show, when there are multiple PMs in each pool the accuracy of the decomposed modeling approach becomes inadequate. This problem quickly becomes more severe as the number of PMs in each pool further increases. As such, while intended for a scenario that closely mirrors the one of interest in this chapter, the existing modeling approach is not useful due to its inaccuracy. In the next section, I will first discuss the potential causes of the inaccuracy, and then develop a new set of decomposed models that offers much improved accuracy while retaining better scalability over the monolithic model.

4.3 New Decomposed Models

As previously discussed, the system components of both the provisioning scenario in [1] and the one in this chapter are essentially the same. The only difference is that, while in the context of [1] the resource pools are likely to be visited in sequence, in the context of storage provisioning it is more likely that incoming requests will be divided into classes based on, e.g., their performance requirements and sent to corresponding pools. As such the different pools operate in parallel and can be modeled separately from each other. On the other hand, the modeling of a single device pool corresponds exactly to a VM provisioning scenario with one PM pool. Thus it would be straightforward to consider details of the previous modeling approach when applied to the context of cloud storage provisioning, and this section follows this perspective and begins with discussion on the causes of inaccuracy in the previous approach.

4.3.1 Analysis about causes of inaccuracy

The scalability improvement of the modeling approach in [1], if applied to the storage provisioning context, would rely on the following two major assumptions:

1) The request arrival processes at the RPDE and the device pools are Poisson.
2) Each storage device in a particular pool behaves independently from its peers within that pool, i.e., each device statistically handles *1/n* of the requests arriving at the pool, where *n* denotes the number of devices in the pool.

The first assumption would allow the device pool to be modeled independently from RPDE. The second assumption replaces the explicit modeling of all the devices in a pool with that of a single device, and computes the availability of the pool from that of the single device using a closed-form expression. Although the two assumptions permit a significant reduction in model complexity and consequently improve scalability, they do not reflect accurately some aspects of system dynamics. Firstly, the input processes to the RPDE and to the device pool are not Poisson in general. Secondly, the devices in a given pool do not behave independently from each other, since once a device exhausts its capacity, the request arrival rates to the other devices will increase (as the requests originally going to that device will be diverted to its peers), resulting in the other devices exhausting their capacities more quickly. Consequently these two assumptions, although fundamental to the scalability improvement in the previous approach, have too much impact on its accuracy. Instead, to find a balance between a monolithic model (which would be detailed but not scalable) and the previous model decomposition approach (which permits efficient solution but does not have sufficient accuracy), it is necessary to modify the two assumptions above to develop a new set of models that are both accurate and scalable.

The first assumption can be relaxed by replacing the Poisson request arrival process at the system modules with an on-off arrival process. An on-off arrival process is a special case of the Markov Modulated Poisson Process (MMPP) [58], where an arrival source alternates between an "On" state and an "Off" state. In the "On" state requests arrive with exponentially distributed inter-arrival times, while in the "Off" state no request arrival occurs. The sojourn times of the arrival source in both states are assumed to follow exponential distributions with possibly different rates. This choice of the arrival processes serves two purposes. Firstly, it captures the input processes of system components (i.e., from external to the RPDE and from the RPDE to the storage device pool) more accurately than the Poisson arrival process employed in the previous models, while retaining much of the simplicity. Secondly, it serves as the basis for analyzing more complex arrival processes at the system, which in reality can be non-Poisson.

The second assumption can be relaxed based on the following important observation: the behavior of a device in a pool is affected (via changes to its request arrival rate) by how many other devices have exhausted their capacities, but not by exactly which devices are out of capacity. In other words, it is possible to capture the behavior of the whole pool accurately so long as the model keeps track of the number of devices in a given operational status (in terms of how many devices are running and how many requests are waiting), instead of the status of each individual device. This observation offers a chance of complexity reduction whose power sits between the full details of the monolithic model and the single device model of the previous decomposition approach.

Based on the modified model assumptions above, a new set of decomposed models are developed that will be described next. This new set of models still adopt a separation of system components, i.e., the RPDE and the device pool, as well as assuming exponential distributions for RPDE decision, request provisioning and partition occupation times. The main differences, as described previously, are the inclusion of on-off arrival processes between system components and tracking the number of devices in a pool that are in a particular operational status (in terms of the numbers of their unassigned partitions and waiting requests). Due to the (near-

)independence among different storage device pools, this section considers only one device pool and presents its model alongside that of the RPDE. After the validity of the new approach is established, extension to include more pools would be straightforward.

4.3.2 The New Models for Cloud Storage Provisioning

4.3.2.1 The RPDE Model

In the model of Figure 4.1, each state of the RPDE is labeled as (k_{Status}) where *"k"* is the number of requests awaiting decision and *"Status"* = {On, Off} is the status of the arrival source. The other terms are summarized in Table 4.3.

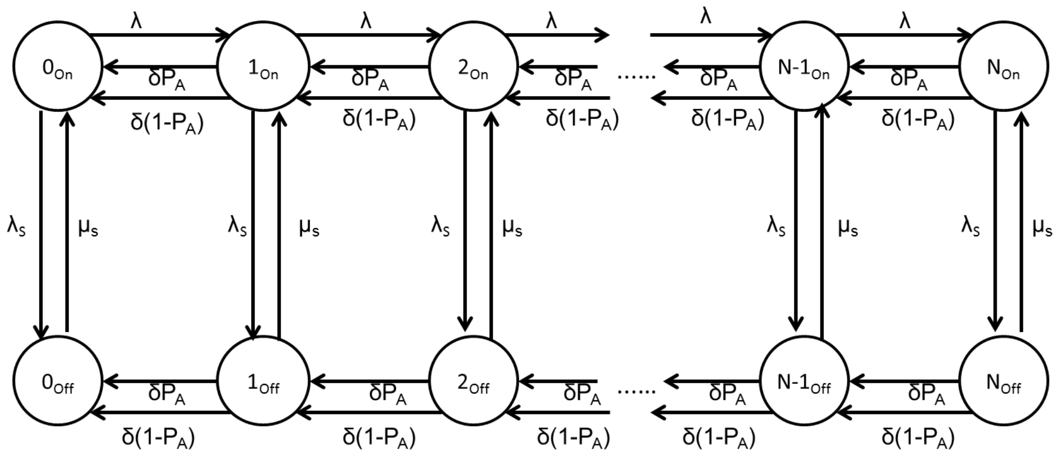


Figure 4.1. The New RPDE model

Table 4.3. Notations of the RPDE model

Symbol	Meaning
N	Maximum number of requests in RPDE (buffer size + 1)
λ	Job arrival rate at the system, during the "On" period of the arrival source
$1/\delta$	Mean time of decision on provisioning a request at the device pool
P_A	Probability that the device pool has capacity left (i.e. is available)
$1/\lambda_S$	Mean time for the arrival source to stay in the "On" state
$1/\beta_S$	Mean time for the arrival source to stay in the "Off" state

Output measures of interest from the RPDE model include the rate of request blocking (i.e., rejection rate of requests due to RPDE buffer being full), the rate of request dropping (i.e., request rejection rate as a result of insufficient resources in the device pool), and the mean number of requests in the RPDE buffer. The three measures can be computed using the following equations.

$$R_{block} = \pi(N_{On}) \cdot \lambda \tag{4.1}$$

$$R_{drop} = \sum_{i=1}^{N} [\pi(i_{On}) + \pi(i_{Off})] \cdot \delta \cdot (1 - P_A) \tag{4.2}$$

$$E[N_{RPDE}] = \sum_{i=1}^{N} [\pi(i_{On}) + \pi(i_{Off})] \cdot i \tag{4.3}$$

In Equations (4.1)(4.2) and (4.3), $\pi(i_{status})$ refers to the steady-state probability of the RPDE being in the state (i_{status}), where *i* refers to the number of requests inside the RPDE and *status* refers to the state of the arrival source ("On" or "Off"). Aside from being an overall measure of interest, the blocking rate is also used to compute the arrival rate at the device pool model, as will be described in the next section.

When the value of *N* is small (e.g., $N \leq 10$), the steady-state balance equation of the Markov chain in Figure 4.1 can be solved via well-known numerical approaches such as Gauss-Seidel method and successive over-relaxation. However, as *N* grows larger it is desirable to find a more efficient solution method. There are several candidates. Firstly, observe that the above RPDE model is a special case of the "many-server queue with service interruption" model in [59] with the number of servers set to one. The generating-function-based closed-form solution for the number of requests in the RPDE in [59] assumes the buffer is infinite, but if the value of *N* is sufficiently large it could still be used as an approximation. Alternatively, by utilizing the block structure of the generator matrix underlying the RPDE model, methods such as matrix-geometric solution [60] and spectral expansion method [61] can be applied. Here I illustrate application of the matrix-geometric method by writing out the generator matrix in Equation (4.4). Note that all omitted entries are zero.

$$\begin{array}{c|cccccccccccc|}
 & 0_{On} & 0_{Off} & 1_{On} & 1_{Off} & 2_{On} & 2_{Off} & \cdots & \cdots & N\text{-}1_{On} & N\text{-}1_{Off} & N_{On} & N_{Off} \\
\hline
0_{On} & *_0 & \lambda_s & \lambda & 0 & & & & & & & & \\
0_{Off} & \mu_s & *_1 & 0 & 0 & & & & & & & & \\
1_{On} & \delta & 0 & *_2 & \lambda_s & \lambda & 0 & & & & & & \\
1_{Off} & 0 & \delta & \mu_s & *_3 & 0 & 0 & \ddots & & & & & \\
2_{On} & 0 & 0 & \delta & 0 & *_2 & \lambda_s & \ddots & \ddots & & & & \\
2_{Off} & & & 0 & \delta & \mu_s & *_3 & \ddots & \ddots & \ddots & & & \\
\vdots & & & & & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & & \\
\vdots & & & & & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & & \\
N\text{-}1_{On} & & & & & & \ddots & \ddots & \ddots & *_2 & \lambda_s & \lambda & 0 \\
N\text{-}1_{Off} & & & & & & & \ddots & \ddots & \mu_s & *_3 & 0 & 0 \\
N_{On} & & & & & & & & & \delta & 0 & *_4 & \lambda_s \\
N_{Off} & & & & & & & & & 0 & \delta & \mu_s & *_3 \\
\hline
\end{array} \qquad (4.4)$$

In the above matrix, for the sake of brevity the following shorthand notations are used: $*_0 = -\lambda_s - \lambda$, $*_1 = -\mu_s$, $*_2 = -\delta - \lambda_s - \lambda$, $*_3 = -\delta - \mu_s$, $*_4 = -\delta - \lambda_s$. Each state ($k_{Status}$), $k = 0, 1, \ldots, N$, *Status* = {"*On*", "*Off*"}, stands for the RPDE state with k requests waiting in the RPDE (those in the queue plus the one being processed) and the arrival process being "on" (e.g., "k_{On}") or "off" (e.g., "k_{Off}"). Then it becomes clear that the generator matrix can be rewritten in a block form as in Equation (4.5),

$$\begin{array}{c|cccccc} & \overline{0} & \overline{1} & \overline{2} & \dots & \overline{N-1} & \overline{N} \\ \hline \overline{0} & B_0 & A_0 & & & & \\ \overline{1} & A_2 & A_1 & A_0 & & & \\ \overline{2} & & A_2 & A_1 & \ddots & & \\ \vdots & & & \ddots & \ddots & A_0 & \\ \overline{N-1} & & & & \ddots & A_1 & A_0 \\ \overline{N} & & & & & A_2 & B_N \end{array} \tag{4.5}$$

with the block matrices defined as:

$$\boldsymbol{A_0} = \begin{bmatrix} \boldsymbol{\lambda} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}, \boldsymbol{A_1} = \begin{bmatrix} -\boldsymbol{\lambda_s} - \boldsymbol{\lambda} - \boldsymbol{\delta} & \boldsymbol{\lambda_s} \\ \boldsymbol{\mu_s} & -\boldsymbol{\mu_s} - \boldsymbol{\delta} \end{bmatrix}, \boldsymbol{A_2} = \begin{bmatrix} \boldsymbol{\delta} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{\delta} \end{bmatrix},$$

$$\boldsymbol{B_0} = \begin{bmatrix} -\boldsymbol{\lambda_s} - \boldsymbol{\lambda} & \boldsymbol{\lambda_s} \\ \boldsymbol{\mu_s} & -\boldsymbol{\mu_s} \end{bmatrix}, \boldsymbol{B_N} = \begin{bmatrix} -\boldsymbol{\lambda_s} - \boldsymbol{\delta} & \boldsymbol{\lambda_s} \\ \boldsymbol{\mu_s} & -\boldsymbol{\mu_s} - \boldsymbol{\delta} \end{bmatrix}$$

In Equation (4.5), all state vectors $\overline{k} = (k_{on}, k_{off})$, *k* = 0, 1, ..., *N,* are of size 2, and all matrix blocks are of size 2×2. Thus it is possible to apply the matrix geometric method [60]. Specifically, there exist two 2×2 matrices R_1 and R_2 and two vectors of size 2, $\overline{v_1}$ and $\overline{v_2}$, that can be used to obtain the steady-state probabilities of all the state vectors $\{\overline{\pi_k}\}$ through Equation (4.6) [62].

$$\overline{\pi_k} = \overline{v_1} R_1^k + \overline{v_2} R_2^{N-k}, k = 0, 1, 2, \dots, N \tag{4.6}$$

where R_1 and R_2 satisfy

$$A_0 + R_1 A_1 + R_1^2 A_2 = 0 \tag{4.7}$$

$$A_2 + R_2 A_1 + R_2^2 A_0 = 0$$

and the vectors $\overline{v_1}$, $\overline{v_2}$ satisfy

$$[\overline{v_1}, \overline{v_2}] = [\overline{v_1}, \overline{v_2}] M \tag{4.8}$$

and the matrix *M* is defined as

$$M = \begin{bmatrix} B_0 + R_1 A_2 & R_1^{N-1}(A_0 + R_1 B_N - R_1) \\ R_2^{N-1}(R_2 B_0 + A_2 - R_2) & B_N + R_2 A_0 \end{bmatrix} \tag{4.9}$$

Notice that the two expressions of Equation (4.7) give an iterative approach to compute *R₁* and *R₂* with the following expression, where successive substitution is performed until the matrices converge.

$$\begin{aligned} R_1 &= -(A_0 + R_1^2 A_2) A_1 \\ R_2 &= -(A_2 + R_2^2 A_0) A_1 \end{aligned} \tag{4.10}$$

The *R_1* and *R_2* matrices obtained from Equation (4.10) can then be combined with Equations (4.6)(4.8)(4.9) and the following normalization equation, where $(1, 1)^T$ stands for the transpose of the row vector (1, 1), to yield all the steady-state probabilities.

$$\left(\sum_{k=0}^{N} \overline{\pi_k}\right) \cdot (1,1)^T = 1 \tag{4.11}$$

Finally, if the rate of the arrival source changing state is much smaller than those of request arrival and reaching provisioning decisions, then an approximation approach following the idea of [63] can also be applied, which allows a more efficient solution.

4.3.2.2 The Storage Pool Model

As discussed earlier, the pool model needs to keep track of the states of different devices in the pool. To facilitate model construction, this section makes use of the Stochastic Reward Net (SRN) [7] formalism instead of Markov chain as used for the RPDE model. Software packages such as SPNP [64] or SHARPE [15] can be used to automatically convert the SRN model into the underlying Markov reward model and solve for the output measures. The device pool model is presented in Figure 4.2.

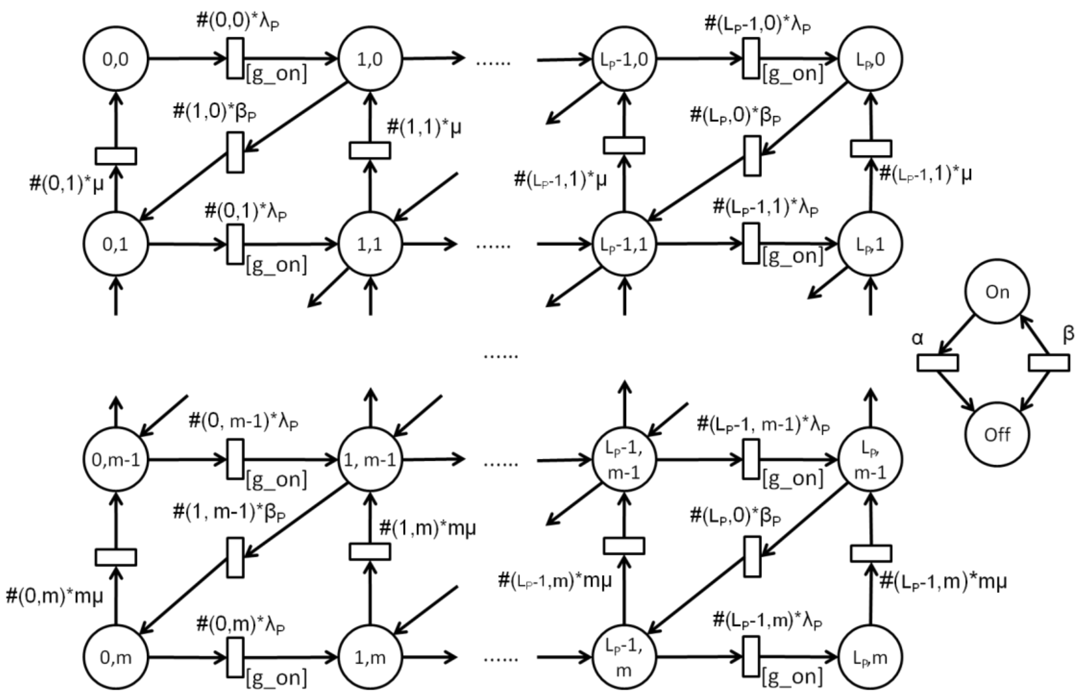


Figure 4.2. The Device Pool Model

In the SRN model of Figure 4.2, a place labeled as (*x, y*) represents the device state in which *x* requests are waiting for provisioning on that device and *y* partitions from it have been allocated. Each of these places may hold multiple tokens, with a token representing one device in that particular state. The sum of tokens across all the places in the model equals the total number of devices in the device pool. Consequently, the model in Figure 4.2 summarizes the statuses of all the devices in the device pool without explicitly specifying that for any particular device. In addition, the two places "On" and "Off" represent the states of the (hypothetical) arrival source. The pool and the arrival source are connected via guard functions which will be described shortly.

The transitions of the SRN model capture the change of device states as requests arrive, get provisioned and eventually release the assigned partitions. Because of the exponential distribution assumption for the times of individual request provisioning, partition occupation and request inter-arrival (during the “On” periods of the arrival source), and because each place in the SRN may hold multiple tokens (i.e., multiple devices in the same operation state), all the transitions have place-dependent rate values. Regarding how to determine the transition rates:

1) The upward vertical transitions, i.e., those with a rate $\#(x, y)\times(y\times\mu)$, represent the release of assigned storage partitions by the customers. For a given transition of this type, the term “$\#(x, y)$” denotes the number of tokens in its input place, while “μ” is the completion rate of each individual request. The multiplication by “y” stems from the assumption that the partition occupation durations of the requests, even for those hosted on the same device, are independent from each other. Hence the time to the next partition release on a given device follows an exponential distribution whose rate equals the sum of the rates to release the assigned partition from all its guest requests. Note that this independence assumption among requests is distinct from the type of independence assumption of the earlier decomposition method, which in this context would be among devices.
2) The diagonal transitions, i.e., those with a rate $\#(x, y)\times\beta_P$, represent the provisioning of requests. The term “β_P” is the provisioning rate of a request. Note that there is no additional “y” term since the provisioning of requests on a device proceeds in a first-come-first-serve fashion. Also notice that each transition of this type, when firing,

removes a token from a place (x, y) and deposits one in the place (x-1, y+1) and corresponds to the completion of provisioning for a waiting request on the device and the start of storage partition usage for that request.

3) Horizontal transitions, i.e., those with a rate $\#(x, y)\times\lambda_P$, represent the arrival of requests. Because arrivals occur only when the arrival source is in the “On” state, these transitions are only enabled when there is a token in the “On” place. These conditions are enforced by the guard functions “g_on”. The term “λ_P” represents the request arrival rate at each device, and its value requires some careful consideration. There are two factors to consider: a) out of all the requests that arrive at the system (during the “On” period of the source), some of them are blocked due to RPDE buffer full; b) for all those accepted by RPDE, they are randomly assigned to all devices that have capacity in a uniform way. Taking both factors into account, the value of λ_P (during the “On” period of the source; it is zero during the “Off” period) can be computed using the following equation.

$$\lambda_P = (\lambda - R_{block})/(n_P - \sum_{i=1}^{m} \#(L_P, i)) \tag{4.12}$$

In Equation (**Error! Reference source not found.**), λ stands for the total request arrival rate (at RPDE) when the arrival source is in the “on” state, while R_{block} denotes the blocking rate of the RPDE (as computed by Equation (4.1)). “n_P” denotes the total number of devices in the pool, and the summation yields the number of devices that have no capacity left. Note that it is unnecessary to consider the case where the number of devices with no capacity equals n_P, since in that case no device will accept any request, and all arriving requests will be dropped as a result of resource exhaustion.

4) The remaining two transitions capture the state changes of the arrival source, and they each follow an exponential distribution with rates "α" and "β" respectively.

The output measure from the device pool model is the pool availability, i.e., the probability that there is at least one device with some capacity left, conditioned on the event that the source is in the "On" state. It can be computed by the following equation.

$$P_A = \left[1 - \Pr(\left(\sum_{i=1}^{m} \#(L_P, i) = n_P\right) \& (SrcOn))\right] / \Pr(SrcOn) \tag{4.13}$$

In Equation (4.13), the term "*SrcOn*" denotes the event that the arrival source is in the "On" state. "Pr(E)", where "E" is an event, then gives the probability of that event occurring. The pool availability computed in this way is then used in the RPDE model in Figure 4.1 to compute the request dropping rate through Equation (4.2). It is also important to note that, unlike the case of [1], fixed-point iteration is no longer needed for solving the new decomposed models. This is because in the new models the pool model only depends on the RPDE model through the request blocking rate which is not affected by the value of P_A and is computed directly from the RPDE model. In other words, although some of the final output measures (i.e., the dropping rate) are still computed from the RPDE model with the value of P_A from the pool model, the state probability vector of the RPDE model does not depend on P_A. Thus the solution of the new decomposed models proceeds in three steps:

1) Solve the RPDE model to obtain its state probabilities. These probabilities are then used to compute the request blocking rate and the mean number of requests in RPDE, following Equations (4.1) and (4.3).
2) Input the request blocking rate to the pool model and obtain the pool availability P_A.
3) Use the value of P_A and relevant state probabilities from the RPDE model to compute the request dropping rate, another output measure, via Equation (4.2).

Finally, I present numerical results that provide insights into the accuracy and scalability of the new models. I first compare the output measure values from the monolithic model and the new decomposed models over a range of arrival rates to investigate the accuracy of the latter. Other model parameter values are summarized in Table 4.4.

Table 4.4. Model Parameters

Parameter	Value	Parameter	Value	Parameter	Value
n_P	5	m	4	L_P	2
$1/\beta_P$	5 minutes	$1/\mu$	5 hours	$1/\delta$	3 seconds
N	6	$1/\alpha$	1 hour	$1/\beta$	1 hour

The rate values remain the same as in Table 4.2, wherever applicable. The number of devices (“n_P”), device buffer size (“L_P”) and number of partitions per device (“m”) are chosen so that it is still possible to solve the monolithic model and obtain the baseline results to compare the decomposed model results against. Based on the parameter values in Table 4.4, I compare the results from both models in terms of request blocking rate, request dropping rate and the mean number of requests in the RPDE. I also compare the time it takes to solve the two models, where the solution is performed using the SHARPE software package [15] on a computer with 2.53GHz 8M E5540 Xeon Processor and 12GB RAM. The results are presented in Table 4.5. Note that "Overall Arrival Rate" denotes the total arrival rate as seen by the system, which takes into account the state changes of the arrival source.

Table 4.5. Numerical Results from Monolithic and Decomposed Models

	Overall Arrival Rate (hour^{-1})	Blocking Rate (hour^{-1})	Dropping Rate (hour^{-1})	Mean Number of Requests in RPDE	Solution Time (sec)
Monolithic	5	1.652e-12	1.448	4.202e-3	247.3
Decomposed		1.652e-12	1.448	4.202e-3	1.822
Monolithic	7.5	2.811e-11	3.774	6.329e-3	220.4
Decomposed		2.811e-11	3.774	6.329e-3	1.528
Monolithic	10	2.097e-10	6.221	8.474e-3	212.8
Decomposed		2.097e-10	6.221	8.474e-3	1.702
Monolithic	12.5	9.957e-10	8.697	1.064e-2	210.7
Decomposed		9.957e-10	8.697	1.064e-2	1.668
Monolithic	15	3.552e-09	11.184	1.282e-2	211.3
Decomposed		3.552e-09	11.184	1.282e-2	1.882
Monolithic	17.5	1.041e-08	13.675	1.502e-2	210.9
Decomposed		1.041e-08	13.675	1.502e-2	2.637
Monolithic	20	2.639e-08	16.170	1.724e-2	213.4
Decomposed		2.639e-08	16.170	1.724e-2	2.863
Monolithic	22.5	5.992e-08	18.665	1.948e-2	212.9
Decomposed		5.992e-08	18.665	1.948e-2	3.427
Monolithic	25	1.247e-07	21.162	2.174e-2	217.2
Decomposed		1.247e-07	21.162	2.174e-2	2.411

As Table 4.5 shows, the results from the new decomposed models match well with those from the monolithic model. Moreover, it also takes significantly less time to solve the decomposed models.

Next I look at the scalability of the new decomposed model. To this end, I increase the number of devices in the pool and look at the solution time, the number of states in the underlying Markov chain and the number of non-zero entries in the generator matrix, while keeping the values of *m* and L_P at four and two respectively and the value of λ at 10 hour^{-1}. All the executions are still carried out on the same Xeon computer described earlier. Note that I expect the improvement in scalability to be less significant than that of the previous decomposed models, since the new models capture more information about the pool, i.e., the distribution of devices across all possible device states. The results are summarized in Table 4.6.

Table 4.6. Scalability of the New Decomposed Model

# of devices	Solution time (sec)		# of states		# of non-zero entries	
	mono.	decomp.	mono.	decomp.	mono.	decomp.
1	0.265	0.544	210	44	530	112
2	0.243	0.550	3150	254	10650	1022
3	0.813	0.611	47250	1374	203250	7382
4	12.62	0.861	708750	6134	1063125	40142
5	212.8	1.880	10631250	23270	65831250	176278
6	*o.m.*	3.852	*o.m.*	77534	*o.m.*	658942
7	*o.m.*	12.80	*o.m.*	232574	*o.m.*	2170582
8	*o.m.*	39.35	*o.m.*	639554	*o.m.*	6453562
9	*o.m.*	153. 8	*o.m.*	1634394	*o.m.*	17622902
10	*o.m.*	395.6	*o.m.*	3922526	*o.m.*	44782034
11	*o.m.*	817.3	*o.m.*	8914814	*o.m.*	106977622
12	*o.m.*	2315.7	*o.m.*	19315414	*o.m.*	242185422
13	*o.m.*	5260.5	*o.m.*	40116614	*o.m.*	523001622
14	*o.m.*	*o.m.*	*o.m.*	*o.m.*	*o.m.*	*o.m.*

In Table 4.6, '*o.m.*' means 'out of memory', which indicates the model solution takes more memory than is available and is thus forcibly terminated. As the results show, the new decomposed model has better scalability than the monolithic model by taking advantage of the fact that the status of each individual device does not need to be explicitly tracked. However, the improvement is still limited as the models do not scale beyond thirteen devices. This deficiency prompts further investigation into the alternative approach of simulative solution, as described in the next section.

4.4 Hybrid Solution Approach

As the previous section shows, the decomposed model, although fairly accurate, does not scale to large numbers of storage devices when a numerical solution technique is applied. As an alternative, simulative solutions can be used to obtain quantitative results, and in this section I present simulation results for both the monolithic and decomposed models. I focus on dropping rate in this section, both for the sake of brevity and for the fact that it is the only output measure (out of all the measures considered in the preceding sections) whose accuracy is negatively affected by the decomposition approach. Two scenarios of simulation are considered: the first is a simulative solution of the monolithic model which directly yields the dropping rate values; the second is a hybrid solution of the decomposed models where the lower level device pool model is simulated to produce the value for P_A (the probability that the pool has some capacity). This value is then used in the RPDE model, which is solved numerically, to obtain the dropping rate following Equation (4.2).

I use the batch-means method to obtain the steady-state dropping rate with a batch length of 3000 hours, and the number of batches is set to 50. Due to the inherent stochastic nature of the model, it is reasonable to expect the dropping rate values obtained from different simulation runs differ from each other to some degree. This fact makes it important to quantify the trust one can put on the sample mean of the dropping rate values (treating the result from each simulation run as a sample) with a confidence interval. I conjecture that one of the advantages of the decomposed model over the monolithic model, in the context of simulation, is that the former (using the hybrid approach) would require fewer samples to produce the same confidence interval width. I make this conjecture based on the observation that when the device pool model is solved using simulation, its confidence interval is propagated to the RPDE model through Equation (4.2). Since the only term in Equation (4.2) that has uncertainty is P_A, the decomposed models in essence avoid sampling events occurring within the RPDE model as is done when simulating the monolithic model.

In order to validate this conjecture, I compare the confidence interval widths from pure simulation on the monolithic model and those from the hybrid approach on the decomposed models. To compute confidence interval widths for the decomposed models, I adopt the central ideas of [65], and observe that the case in this section is simpler: instead of multiple parameters, the RPDE model only needs as its input one output measure (i.e., P_A) from the simulation on the device pool model. Also the value of the dropping rate is a monotonic function with respect to the value of P_A (as in Equation (4.2)). This observation obviates the need to use the relatively complex posterior distribution formulas and Monte Carlo procedure employed in [65], and use the following reasoning to obtain the confidence interval of the dropping rate with a given confidence level L: suppose we have a confidence interval, e.g., W, with a given confidence level L for the value of P_A obtained from simulation on the device pool model. Based on the concept of confidence interval, the interval W would contain the true value of P_A $100\times L\%$ percent of times (if the simulation is repeated multiple times). Assume the lower/upper bounds of W are LB_W and UB_W respectively, then plugging the values of LB_W and UB_W into (4.2) produces the corresponding confidence interval [LB_R, UB_R], where LB_R and UB_R are obtained by replacing the term P_A in Equation (4.2) with UB_W and LB_W respectively. Because Equation (4.2) is monotonically decreasing with respect to P_A, the interval [LB_R, UB_R] also contains the true value of the dropping rate 100×L% percent of time (since Equation (4.2) yields the true value if and only if the true value of P_A is used). This fact then makes the interval [LB_R, UB_R] a confidence interval for the value of the dropping rate with the given confidence level L.

I assume a 95% confidence level in both cases of simulation and present the results in Figure 4.3, and compare the results from the monolithic model (using both analytical-numerical solution and simulation) and from the decomposed models (using the hybrid solution approach) under different numbers of devices in the pool. It is clear from Figure 4.3 that the simulation results from both models are very close to each other and both match those from the analytical-numerical solution well (recall that the decomposed models yield analytical-numerical results that are essentially identical to those from the monolithic model, so I only show one set of analytical-numerical results). This observation validates the accuracy of the hybrid approach. The major difference between the two set of simulation results is the widths of the confidence intervals, which are not shown in Figure 4.3 for the sake of clarity. In Figure 4.4 I illustrate this difference by zooming into one region of the figure and focus on the confidence interval widths of the results from the two solution approaches (simulation and hybrid solution) with different request arrival rates. In this figure the inner pairs of bars around each data point come from the decomposed model (hybrid solution) while the outer ones come from the monolithic model (pure simulation). Since the focus of Figure 4.4 is on the interval widths, I do not include the results from the analytical-numerical solution.

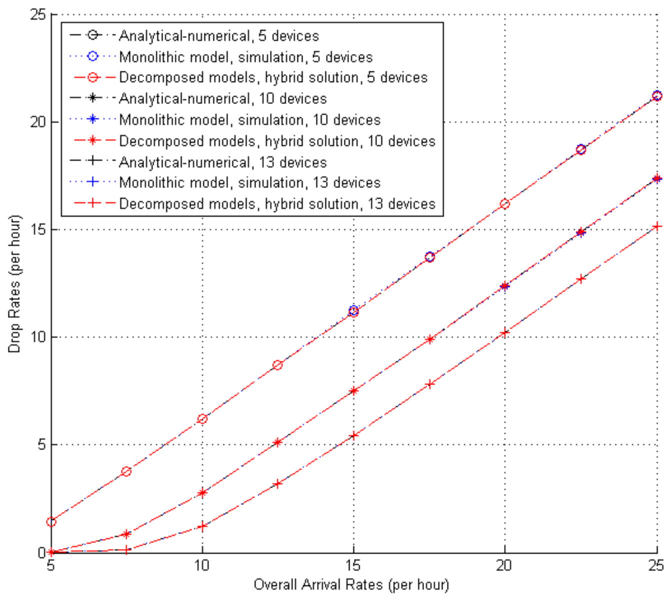


Figure 4.3. Results Comparison for Monolithic and New Decomposed Models

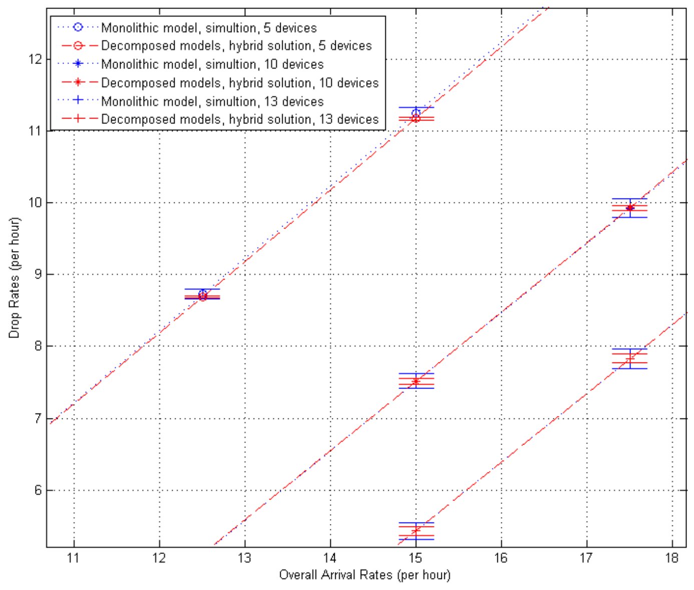


Figure 4.4. Zoomed-in View on the Width of Confidence Intervals

As Figure 4.4 shows, there are major differences between the confidence interval widths from the two models, where the hybrid solution produces narrower intervals. The following results in Figure 4.5, which presents the difference between confidence interval widths from the hybrid approach and the pure simulation approach as request arrival rate increases, offer another perspective on this issue.

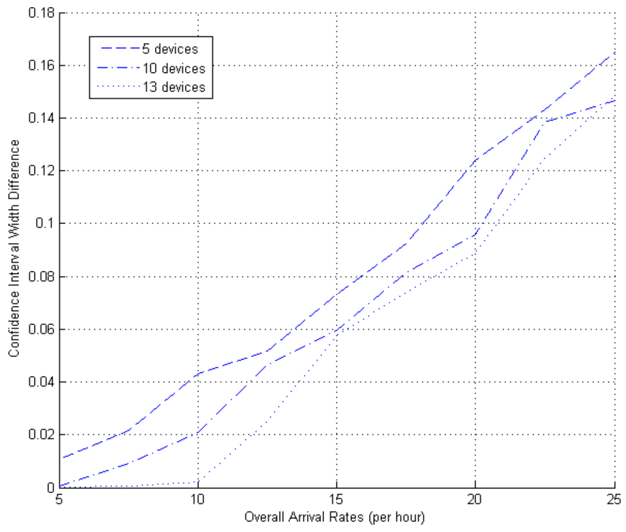


Figure 4.5. Comparison of Confidence Interval Widths

As the results in Figure 4.5 show, when the value of request arrival rate increases (which leads to a greater number of events occurring), both approaches yield wider confidence intervals. However, the interval width from the hybrid solution increases much slower than that of the pure simulation. To further illustrate the benefit of the hybrid solution, I perform pure simulation (on the monolithic model) to obtain the number of batches needed to achieve the same confidence interval width as the hybrid approach. Figure 4.6 presents the result.

As Figure 4.6 shows, with the arrival rate increasing, the number of batches needed for the pure simulation approach to produce the same confidence interval width as the hybrid approach increases fairly rapidly (especially when compared to the constant number of batches, 50, used by the hybrid approach). This observation indicates that using simulation to solve part of the decomposed models can be an effective approach to yield accurate results.

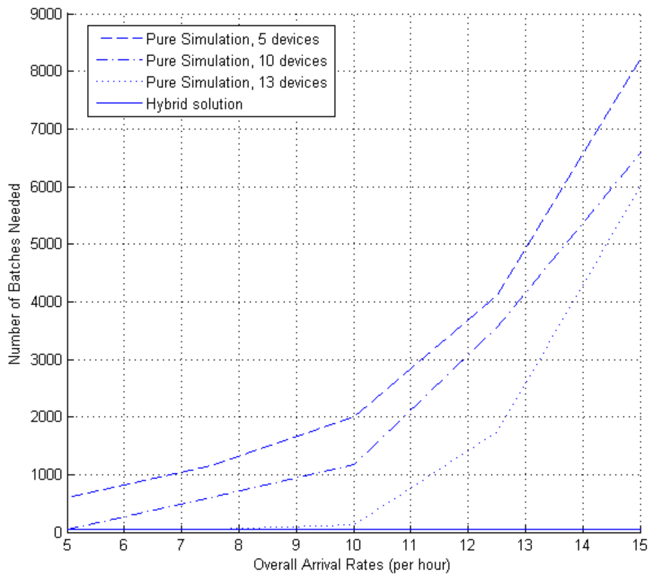


Figure 4.6. Number of Batches Needed

Finally, I make a scalability comparison between the monolithic and decomposed models in the contexts of both analytical-numerical solution and simulation. The comparison is illustrated in Figure 4.7, where the X-axis is the number of devices and Y-axis is the solution times (both are in log scale to better accommodate all the results). To compare pure simulation (on the monolithic model) with the hybrid approach, 50 batches of simulation are performed for the hybrid approach, and then as many batches as needed are executed for pure simulation to produce the same confidence interval width. As Figure 4.7 shows, although the analytical-numerical approach typically runs much faster, it suffers from scalability issue. With this solution approach, the monolithic model is unable to go beyond five devices, while the decomposed model doesn't scale beyond thirteen devices. On the other hand, the simulation approach handles large numbers of devices better, with the solution times of both the pure simulation and the hybrid approach exhibiting slight increase as the number of devices rises to large values, although the pure simulation is still at a disadvantage as it takes more time to produce the same confidence interval widths.

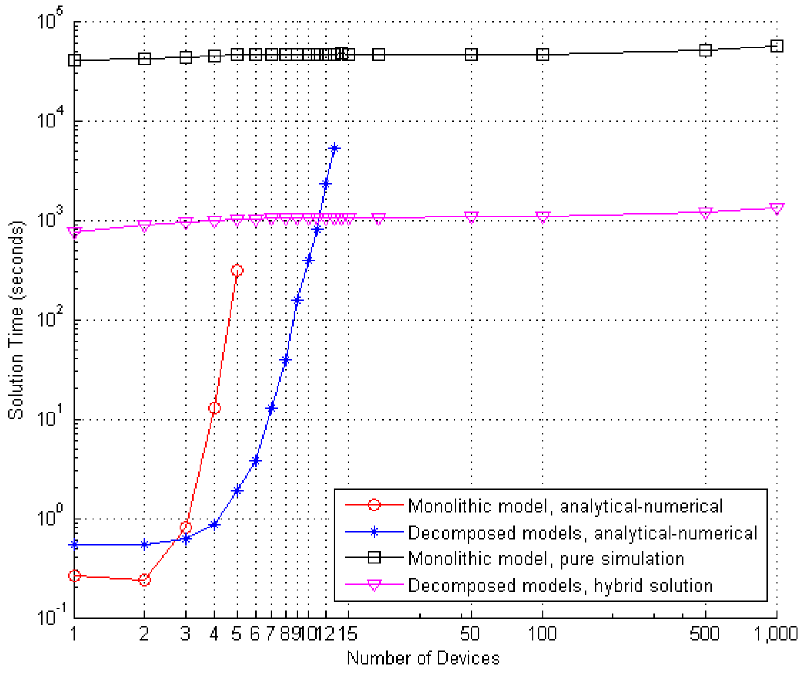


Figure 4.7. Comparison of Solution Times

4.5 MCMC Solution Method

The previous sections show that a model based on the combined behavior of the device pool yields very good accuracy. On the other hand, an interesting question remains as to whether it is possible to analyze the pool dynamics based on models corresponding to each individual device, while accounting for the complex interdependencies among the devices. In this section I present such an approach based on Gibbs sampling, a method from the family of Markov Chain Monte Carlo (MCMC) sampling techniques.

4.5.1 Overview of the method

The sampling-based method uses the single device model (analogous to the single PM model in [1]) given in Figure 4.8.

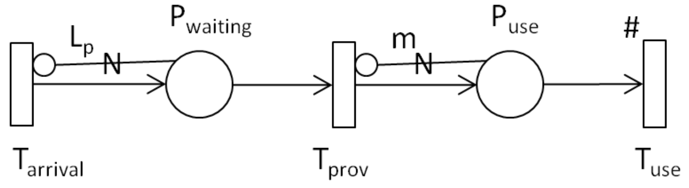


Figure 4.8. Single Device Model

In the SRN model of Figure 4.8, the places "$P_{waiting}$" and "P_{use}" correspond to the request status of waiting in the buffer of the storage device and using a partition assigned to the request issuer from the device respectively. The numbers of tokens in these two places represent the numbers of requests waiting for provisioning and using assigned partitions respectively. Similarly, the transitions "$T_{arrival}$" "T_{prov}" and "T_{use}" captures the arrival of the next request, the completion of provisioning for the next request and the end of partition use (and release of the partition) of the next request respectively. The "#" sign on the usage transition indicates that the rate of the transition is proportional to the number of tokens in "P_{use}", which represent the fact that the partition usages of different requests are independent. Finally, the two inhibitor arcs and their cardinalities (L_p and m respectively, where L_p is the size of device request buffer and m is the number of partitions the device can accommodate) enforces the capacity constraints of the device in question.

The single device model is much simplified compared to the pool model. However the individual devices have interactions among them in that the arrival rate (i.e. the rate of "$T_{arrival}$")

to a device depends on how many other devices are at the limit of their capacities. As previously shown, this dependence makes it nontrivial to generate correct values for pool-level metrics (such as pool availability) using some combination of device-level metrics (e.g. device state probability distributions). A straightforward application of the device-level metrics can produce inaccurate results, as those in Section 4.2 indicate. The device pool model in Section 4.3 resolves such dependency by explicitly accounting for it and adjusting the arrival rates to the devices accordingly. However its scalability to large systems relies on simulation (or the hybrid approach in Section 4.4), which impacts its efficiency to some extent.

In this section I again start from the basic single device model and try to devise a new approach to account for the dependencies among devices. The approach is based on the following reasoning:

1) Consider the state of an arbitrary device; this state is described by a random variable. If the states of all the other devices are known, then the request arrival rate to this device becomes fixed and the distribution of its state variable can be evaluated accurately and efficiently, given the simplicity of the single device model.
2) Given that the ultimate goal of analyzing the device/pool is to produce the pool unavailability, we would like to know the joint distribution of all the device state variables (or alternatively, be able to evaluate the expectation of a function with respect to the joint distribution; the pool unavailability can be cast as such an expectation). However, due to the dependence among the devices it is difficult to do this directly.

3) So I would like to have an approach which uses the easy-to-evaluate single device state distribution (when conditioned on other devices' states) to produce estimates about the joint distribution (or the expectation of a function with respect to it).

Essentially, the above reasoning simply recasts the original problem in the light of joint and conditional distributions. However, this change of perspective would allow application of additional techniques to the problem. Before proceeding further, however, it is worthwhile to note that the single device state distribution is a conditional distribution, not a marginal one, as it is produced by conditioning (instead of marginalizing) the joint distribution on the states of other devices.

The problem of estimating the joint distribution of many random variables, while knowing only their individual distribution conditioned on all their peers, has been addressed in the context of statistical physics via sampling methods, notably by the Gibbs sampling technique [66]. The steps of Gibbs sampling procedure are summarized below:

1) Starts with some random initial values for all the random variables, denoted as (X_1(0), X_2(0), ..., X_n(0)), where "0" indicates the initial time index of the random walk and the subscripts 1~n indicate the variables.
2) Going from 1 to n, replace the value of X_i, i∈{1~n} with a new sample drawn from its distribution conditioned on the current values of all other variables, i.e.,

$$X_i(k+1) = P(X_i|X_1(k+1), X_2(k+1), \cdots, X_{i-1}(k+1), X_{i+1}(k), \cdots X_n(k))$$

This procedure generates a new sample for the joint distribution. Typically this assumes that all the distributions are time-homogeneous, i.e. X_i(k+1) = X_i(k) for any k, although it does not have to be the case.

3) Repeat step 2) until either a predetermined number of samples have been collected, or some metrics computed from the samples (for example the sampled frequency of specific states) converge.

Upon termination, the distribution of collected samples (if the sample set is sufficiently large) would approximate the joint distribution over the variables. The Gibbs sampling technique is a special case of Markov Chain Monte Carlo (MCMC) methods, and it performs random walk on a (discrete time) Markov chain whose states are defined by the possible values of all the variables combined and whose transitions are defined by the distributions of variable values when conditioned on the states of all their peers.

Some of the most important issues regarding the Gibbs sampling technique, or indeed all MCMC methods, are discussed below.

1) There must be guarantees that the sampling procedure does indeed samples from the desired joint distribution. In cases where the individual conditional distributions are derived from the joint one then this requirement will automatically be satisfied [67][68]. However in other cases, e.g. when there is no closed-form expression for the joint distribution available (as is the case in this chapter), satisfaction of this requirement must be proved explicitly.

2) Gibbs sampling will have problems converging or approximating the real distribution efficiently if the underlying Markov chain is reducible (i.e. having multiple communicating state classes) or if certain sets of variables have strong correlation so that producing samples with values against the correlation becomes difficult.
3) The final samples produced need to be independent. It should be noted that successive samples generated by Gibbs sampling (or other MCMC methods) are correlated as they correspond to successive states visited during a random walk on the underlying Markov chain.

In the next section I discuss how to apply Gibbs sampling to solve the storage provisioning model and address the above issues in this particular context.

4.5.2 Solve the Provisioning Model with Gibbs sampling

The basic idea of applying Gibbs sampling to this context is to treat the state of each storage device in the pool as a random variable, and then sample each of them in turn from the single device model stationary state probability distribution with the state of all the others fixed (thus fixing the request arrival rate to the one being sampled). The idea is straightforward, but it is necessary to address the aforementioned issues in this context. The independent issue can be addressed by subsampling the original samples with large enough intervals and only include the subsamples in the final results. The other two issues are discussed in more details below.

4.5.2.1 Ensuring sampling from the desired joint distribution

Here I prove that the sampling from the individual device model, following the Gibbs procedure, would be adequate in approximating the desired joint distribution. To this end I start with the following observations.

Consider the model in Figure 4.8. The SRN model defines an underlying CTMC, and it is easy to see this CTMC is both finite and irreducible. Also, there are a subset of states corresponding to the storage device being full, that is, the state where the number of tokens in "P_{waiting}" equals L_p. Because a device only ceases to accept request after it enters these states, and because the model is concerned about the (un)availability of this device, it suffices to rewrite the underlying CTMC into one with only two states, one corresponding to the set of device full states (S_{full}) and one to all the others states combined (S_{nonfull}). It is straightforward to modify the rates in the new CTMC so that the stationary probability of S_{full} remains the same as the sum of the stationary probabilities of its component states in the original CTMC [6]. The method is sketched below.

1) Solve the model in Figure 4.8 (under a fixed value for the arrival rate) and obtain the state probabilities, $\{\pi_i\}$.
2) Divide the states into the two sets, Set_{full} and $Set_{nonfull}$. They correspond to S_{full} and S_{nonfull} respectively.
3) Compute the rates between the two states as:

$$R_{S_{full}-S_{nonfull}} = \frac{\sum_{s\in Set_{full}} \sum_{k\in Set_{nonfull}} \pi_s R_{sk}}{\sum_{s\in Set_{full}} \pi_s R_{sk}} \tag{4.14}$$

$$R_{S_{nonfull}-S_{full}} = \frac{\sum_{s\in Set_{nonfull}} \sum_{k\in Set_{full}} \pi_s R_{sk}}{\sum_{s\in Set_{nonfull}} \pi_s R_{sk}}$$

Note that because the state probabilities are computed based on a given arrival rate value, and hence a given number of storage devices that are full, the two rates computed are also functions of the latter number. While the above transformation does not impact the stationary probabilities (after the appropriate state mappings) of the model, it facilitates the proof of the applicability of Gibbs sampling that follows. I now give some definition and lemmas.

Definition 5.1. Define this new two-state CTMC as the *reduced device model*.

Lemma 5.1. The samples produced from the stationary distribution of reduced device model will be the same as those from the original CTMC in terms of whether the device is out of capacity.

Proof: Per the construction of the reduced device model, it has only two states and they communicate (otherwise the original model would not be irreducible). Hence the reduced model is also irreducible and finite, thus having a unique stationary distribution. Because it is constructed in a way that the stationary probability of S_{full} is the same as the sum of probabilities of the states in Set_{full} in the original model, and because those are the only states where the device is out of capacity, the samples produced from the reduced device model would be the same as those from the original CTMC in this regard.

Now consider the whole pool. It is easy to see that a global CTMC can be constructed from the Cartesian product of the state spaces of all device variables, and this CTMC is finite and irreducible. Based on a similar reasoning as above, this CTMC can also be rewritten into a version where the possible values of a variable are collapsed into two, one for the corresponding device being full and one for otherwise.

Definition 5.2. Define this reduced global CTMC as the *reduced global model.*

Lemma 5.2. The samples produced from the stationary distribution of reduced global model will be the same as those from the original global CTMC in terms of whether any given device is out of capacity.

Proof: essentially identical to that of Lemma 5.1.

With the definitions established, there are two steps to follow:

1) Prove that sampling from the single/reduced device model will converge to a limiting distribution regardless of the initial device states.
2) Prove that this limiting distribution is the same as the joint distribution of the pool.

To prove the first part, it suffices to show that the Markov chain defined by the device state variables is ergodic.

Proposition 5.1.The Markov chain defined by the device state variables as used in the Gibbs sampling procedure is ergodic.

Proof: The proof is carried out through the following steps:

- The Markov chain is finite since any variable can take only a finite number of values.
- The Markov chain is irreducible because a state variable can assume any value it can take regardless of its previous value (as the conditional distribution does not depend on the latter). This means a new sample (over all the variables) can take any value regardless of the old one (though the probability may be small). As each value

corresponds to a state, all states in the chain communicate and the chain is irreducible.

- Because the chain is both finite and irreducible, all states are positive recurrent.
- All states are aperiodic. As reasoned above a new sample can take the same value as the old one, hence any state of the Markov chain has a period of one.
- Since all chain states are both positive recurrent and aperiodic, the chain is ergodic.

The proof for the second part is slightly more complex. To do this, observe that the stationary distributions of the reduced device model and the reduced global model correspond to the conditional state distribution of a single device and the joint distribution of all device states respectively. Based on Lemma 5.1 and 5.2, to prove that sampling from the single device model converges to that from the pool model it suffices to prove that the reduced device model stationary distribution is the same as the conditional distribution of a single device variable derived from that of the reduced global model, since sampling from the latter conditional distribution, when converging, is guaranteed to follow the desired joint distribution [67][68]. To achieve this, the following lemma is needed.

Lemma 5.3. The reduced global model satisfies detailed balance (or is reversible).

Proof: To prove the lemma it suffices to prove the reduced global model satisfies Kolmogorov's criteria, which is equivalent to proving that for an arbitrary state cycle $\{s_1, s_2, ..., s_n, s_1\}$ in the reduced global model the products of transition rates along the two directions are identical, i.e. $q_{12}q_{23}...q_{(n-1)n}q_{n1} = q_{1n}q_{n(n-1)}...q_{32}q_{21}$. This can be proved based on the following reasoning:

- Each variable takes one of two values (full and not-full. I shall label them as 1 and 0 for simplicity). Thus any transition either increases or decreases the value of one variable by one. Call these the *increase* and *decrease* transitions.

- It is easy to see that for any s_i and s_{i+1}, there will be transitions between them in both directions. Moreover, they consist of one increase and one decrease transition for the same variable.
- It is also easy to see that the number of increase and decrease transitions for any variable along the cycle must be the same. Thus there will be a total of n/2 increase and decrease transitions respectively.
- Because all requests complete at the same rate and all devices can hold the same maximum number of requests, the rate of a variable decreasing from 1 to 0 (as computed from (4.14)) would be the same across all variables. Thus the product of decrease transition rates would be the same along both directions.
- The rate of an increase transition depends on how many variables are currently in state 1. Look at an arbitrary increase transition that brings the number of variables in state 1 from x to x+1. This increase must be canceled out somewhere in the same direction along the cycle by a decrease transition from x+1 to x. Due to the second point above, that decrease transition would be paired with an increase transition, which goes along the other direction, from x to x+1. Hence the same rate term would appear in the products of both directions. Thus the products of increase transition rates are also identical on both directions.
- Combined the above two points, the rate products are the same along both

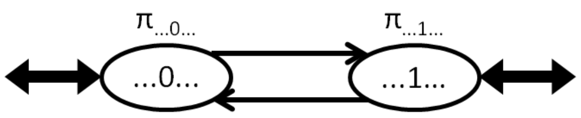
directions for any state cycle. Hence Kolmogorov's criterion is satisfied and the reduced global model satisfies detailed balance.

Because the reduced global model satisfies detailed balance, for any two state s_1 and s_2, $\pi_1 q_{12} = \pi_2 q_{21}$ holds, where π_1 and π_2 are stationary state distribution values. This paves the way to prove the second part.

Proposition 5.2.The reduced device model stationary distribution is the same as the conditional distribution of a single device variable derived from the stationary distribution of the reduced global model.

Proof: The proof consists of three steps.

- Consider the following two states corresponding to the distribution of one variable, with all other variables taking some known values (but without any conditioning), in the Markov chain representing the joint distribution:



  The thick transitions represent all the transitions between the two states and other states. For the state labels and probability terms only the values of this variable is shown. Clearly, the conditional distribution of this variable can be obtained by normalizing $\pi_{...0...}$and$\pi_{...1...}$with their sum.

- Detailed balance ensures that $\pi_{...0...}q_{(...0...)(...1...)} = \pi_{...1...}q_{(...1...)(...0...)}$. Thus it means that if these two terms are normalized they would also be a stationary distribution of the reduced device model because the latter is obtained by keeping only these two states and the transitions between without modifying the rates of the transitions, i.e.,

$$\begin{tikzcd} \pi_0=c*\pi_{\ldots0\ldots} & \pi_1=c*\pi_{\ldots1\ldots} \\ 0 \arrow[r, shift left] & 1 \arrow[l, shift left] \end{tikzcd}$$

where “c” is the normalization factor.

- Clearly the reduced device model above has only one stationary distribution. Thus this stationary distribution is the same as the conditional distribution from the global joint distribution.

With the above proof, I know that sampling from the distribution of the single device model is the same as sampling from the conditional distribution of the global joint distribution. This then [67][68] leads to the conclusion that the former sampling would converge to the global joint distribution. Hence the second part is proved.

Finally, with both parts proved, I have reached the conclusion that performing Gibbs sampling using the model in Figure 4.8 would sample correctly from the joint device state distribution over the pool.

4.5.2.2 Achieving sampling efficiency

Regarding the efficiency, as previously mentioned two of the main issues are chain reducibility and correlation among variables. Per the proof of proposition 5.1, the sampling Markov chain is irreducible and so the first issue does not arise. Regarding the correlations among variables, because the conditional distribution of a variable depends on the total number of (not which) other variables that take value one, when the system is reasonably large the correlation among any small set of variables become small to the point that the second issue is no longer a major concern.

4.5.3 Numerical Results

In this section I present numerical results concerning the accuracy of the sampling approach and its efficiency compared to the simulation model (since the approach is also used to solve the pool part of the provisioning system, it is compared to the simulation procedure, not the hybrid solution which is for the entire model). All of the results are produced with 50 batch runs, where each batch generates 2000 samples, for both simulation and Gibbs sampling. For the first set of results, the arrival rate is set at 50 hour^{-1}. All other parameters take the same values as in Table 4.4.

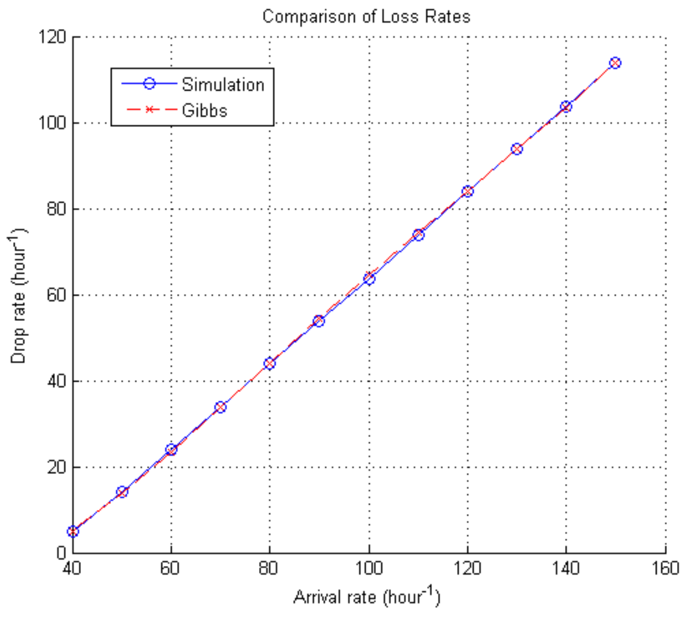


Figure 4.9. Accuracy Comparison

Figure 4.9 presents the accuracy comparison between Gibbs-sampling results and those from simulation. As the figure shows, the sampling method is very accurate across the range of arrival rates that are compared. Figure 4.10 presents the execution time comparisons of simulation and Gibbs-sampling.

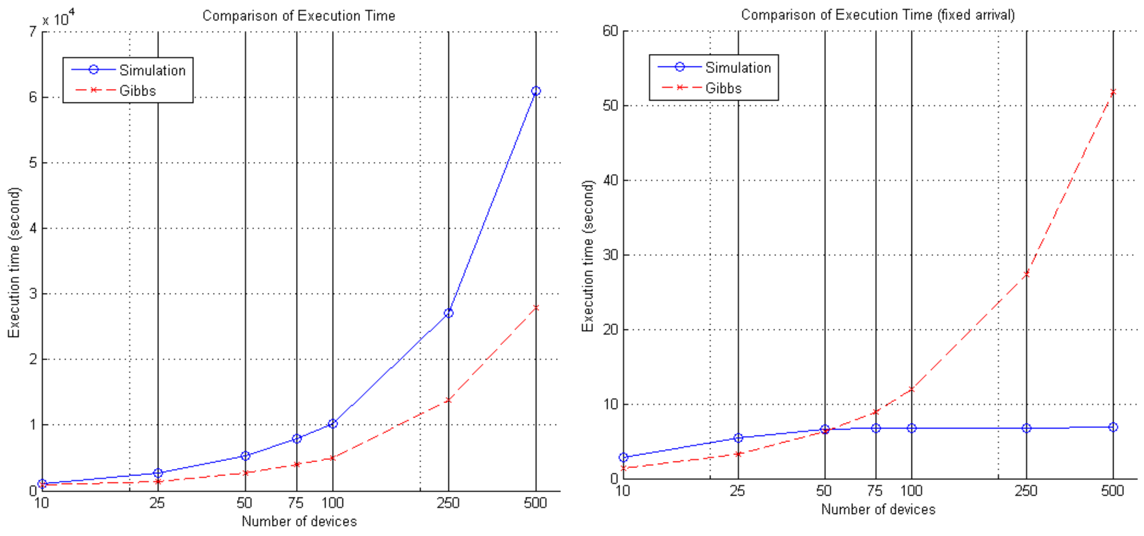


Figure 4.10. Execution Time Comparison

In Figure 4.10, the left figure shows the results when the arrival rate is kept at a value ten times that of the number of devices. This setup means the devices are relatively heavily loaded, and in this case the results show that the Gibbs-sampling method has better efficiency compared to simulation. This is because, since the ratio of arrival rate and device number does not change, the Gibbs sampling procedure does not have to deal with increasing complexity from the arrival rate increase, while the simulation approach will need to handle these additional events. In fact as the right figure of Figure 4.10 shows, when the arrival rate is fixed at 50 hour^{-1} while the number of devices is increased the sampling procedure is actually less efficient than the simulation approach. This suggests that it may be beneficial to combine these two methods for good analysis efficiency in different regions of the system parameter space.

4.6 Existing Work on Cloud System Performance Evaluation

Due to the rapid development and high potential of cloud systems, much research effort has been devoted to their analysis. Research work has been carried out on the performance of different requests in cloud systems, including scientific workloads [69][70] and web applications [71][72][73]. Performance comparison and investigation of specific cloud systems, open-source [74] or proprietary [75] have also been conducted. There are also other papers that focus on the performance of specific aspects of the cloud system itself, such as live VM migration [76][77], storage access [78] and resource provisioning [79]. While most of these papers are empirical and use measurement or simulation as the primary approaches, the work in this chapter adopts stochastic modeling with solution provided by a hybrid approach combining analytical-numerical solution and simulation/sampling to improve solution efficiency while maintaining accuracy.

Research on cloud system analysis by means of stochastic modeling and simulation has also been conducted. For stochastic modeling, the work in [80] centers on automated learning of linear performance models through a combination of queuing theory and data mining. Also using queuing network formalisms, [81] develops analytical models for cloud server farms. In the context of cloud system simulation, several simulation toolkits are available (e.g., [82]. A review of tools and methods is provided in [83]), and modeling work has been done using these tools. Compared to these papers, which mainly relies on one method or the other (analytical-numerical solution of analytical models or simulation), the development of the decomposed models in this chapter permits greater flexibility in covering system details, while the hybrid numerical-simulation-sampling approach provides good balance between model scalability and solution efficiency.

4.7 Conclusion and Future Work

This chapter presents a set of improved decomposed models that capture several performance measures of a cloud storage provisioning system. The models account for the presence of non-Poisson arrival processes through an on-off arrival process, and the results from analytical-numerical solution indicate that the models are very accurate. In order to further improve model scalability, I combine simulation and MCMC sampling methods with analytical-numerical solution and show that the hybrid solution approach offers higher efficiency, in terms of the number of simulation/sampling runs needed, to achieve the same level of confidence in the results as compared to a pure simulative approach.

For future work, I will seek to extend the current on-off arrival process to more general non-Poisson arrival processes so as to make the model better suited to represent actual cloud system dynamics. Another direction is to extend the modeling and solution approach to other important aspects of the cloud system, such as computation task execution, and include more system details, such as multiple pools of resources, different types of requests, management overhead, and the potential impacts from the distributed nature of the cloud during the execution of different request types. It is also an interesting direction to try to identify more details in the current decomposed models to enable analytical-numerical solution with larger number of devices, as this type of solution technique is much faster than simulation when it is applicable.

5. Stochastic Modeling for Data Backup[1]

5.1 Overview

Data backup plays an important role in the protection of IT system data. By storing copies of system data, backup defends the data against possible losses caused by hardware and software failures, human errors and natural disasters. While data backup is essential for data protection, its deployment can negatively affect system performance and availability. For example, backup execution may bring the system offline ('offline backup') and incur additional downtime. Even if the system is accessible during a backup, i.e., by using 'online backup' techniques, system performance is still likely to be affected. This is because the backup process may compete with normal service processes for system resources and can be intensive on some types of resources, e.g., disk I/O.

In order to reduce the impact of the backup process on system performance and availability, a suitable backup policy needs to be devised which specifies the technique, scope and schedule of the backup. For example, a commonly adopted backup approach is to combine full backups, where all the data is copied to the backup storage, with partial backups [84] which only copy data change that has occurred since an earlier backup. Partial backups can be classified as differential and incremental backups. The former processes data changes since the last full backup and the latter processes changes since the most recent backup (full or

[1] This chapter is based on the following publication:
Performance and Availability Modeling of IT Systems with Data Backup and Restore. Ruofan Xia, Xiaoyan Yin, Javier Alonso, Fumio Machida and Kishor Trivedi. IEEE Transaction on Dependable and Secure Computing (TDSC), issue 04, volume 11.

incremental). The tradeoff here is that an incremental backup takes the least time to perform, but restoration tends to be slower as it requires all the partial backups since the last full backup. On the other end of the spectrum, the full backup is slow to perform but restores the system faster. Finally, the differential backup sits in between. Thus, depending on system failure characteristics, a mixed backup strategy may reduce the time spent on backup/recovery and improve system availability and performance. Such a strategy is coupled with a backup schedule that defines the frequencies of full and partial backups. The backup frequencies affect the system recovery point objective (RPO), the maximum time period in which data could be lost due to a system failure. Hence, a tradeoff exists where higher backup frequencies achieve lower RPOs but potentially decreases system availability and/or performance.

To devise a suitable backup policy, it is important to properly evaluate system performance and availability under different policies. A previous paper [85] focused on automated model composition and availability of an offline backup system. In this paper, I extend the previous paper by: 1) considering online backup and workload priorities, and 2) deriving new availability and performance metrics based on more detailed stochastic models. The models are developed with a variety of model types including Markov chains, queuing networks and Stochastic Reward Nets (SRN), and capture the details of both the normal system operation and the backup process. As a case study, I investigate a file service (e.g., file repository) system which provides access to local file data and employs periodic data backup to preserve user updates and defend against data loss. I am able to obtain five important system availability and performance metrics for my example: the file server availability, the rejection rate and ratio of user requests, and the loss rate and ratio of system data. The first three metrics concern the availability aspect, while the remaining two are related to system performance. Notably, the rejection rate and ratio capture the system availability as perceived by the users. I then investigate these metric values under several backup policies. My results provide insights into the interactions between the backup process and the applications running on the system, and the tradeoffs needed to provide data protection with data backup while minimizing the impact on system availability and performance.

The rest of this chapter is organized as follows: Section 5.2 describes the system and scenarios under study. Section 5.3 details the modeling techniques and system metrics of interest. In Section 5.4, the parameter values and numerical results are presented. Section 5.5 reviews related work and discusses the novelty of this paper. Finally, Section 5.6 summarizes the conclusions and future work directions.

5.2 System Backup Scenario

5.2.1 System Description

In this section I first describe the system scenarios in which I will investigate the backup policies. As shown in Figure 5.1, the system is a web service system providing access to files stored on a local file server. The system also contains a backup server to perform backup operations. The two servers are connected using a dedicated backup network. Outside users access the file server where user requests of file read/write are handled by an Apache web server [86. The system periodically backs up data from the file server disk to the backup server using the ‘rsync’ tool [87.

In Figure 5.1, 'Network I/O', 'Disk', 'CPU' and 'Backup Network' are physical resources of the system, while 'Apache Server', 'Sender (rsync)', 'Generator (rsync)' and 'Receiver (rsync)' are software processes running on different CPUs. As the names suggest, the 'Sender (rsync)', 'Generator (rsync)' and 'Receiver (rsync)' processes are functional components of the 'rsync' tool. The dashed and solid arcs correspond to the data flows of user accesses and the backup process respectively.

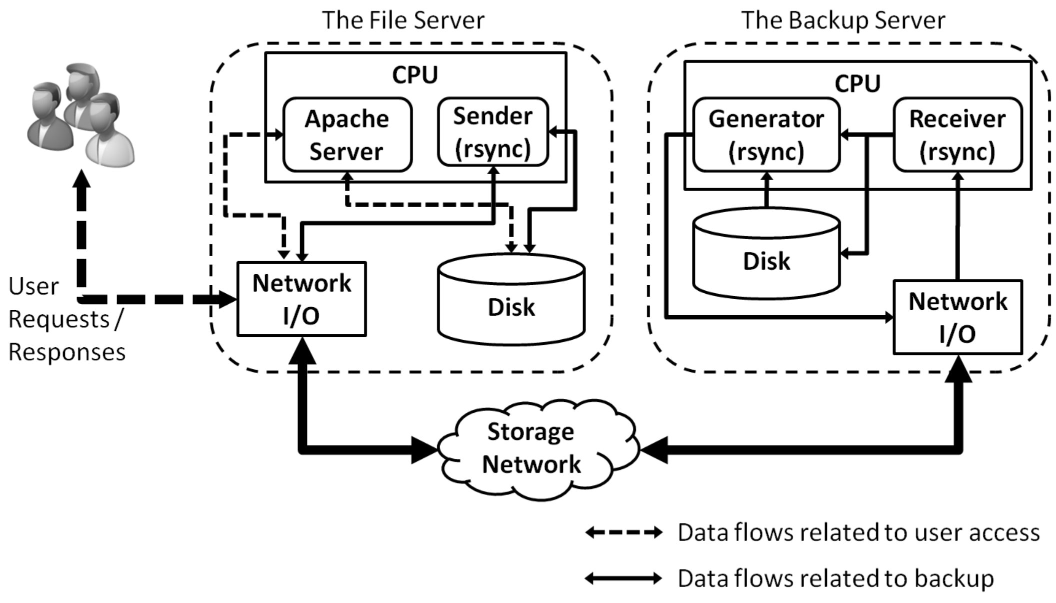


Figure 5.1. System Diagram

To facilitate understanding of the system data flows, I briefly describe the Apache web server and the 'rsync' tool. The Apache web server runs a main process that listens for incoming user requests (using non-persistent HTTP 1.0 connections). On the arrival of a new request, the main process creates a service process which will handle that request until its completion. Based on the Apache architecture, I assume that:

1) A service process handles one request at a time;

2) There is an upper limit on the number of concurrent service processes;

3) There is no timeout for user requests, so no accepted request is ever dropped.

These assumptions mean that the Apache server can only handle a finite number of user requests simultaneously, and user requests that arrive when all service processes are busy will be rejected. Such system specifications have important implications for system performance and availability, as later sections will show. For each user request, I assume a simple data flow model as in Figure 5.2, where the filled circle represents the arrival of a user request and the half-filled ones are the processing results of this request. The rectangles represent the handling of the request at a particular system component, e.g., the network I/O, the disk I/O, or the service process (which runs in the CPU). Finally, the rhombuses represent conditional branches whose outcomes depend on certain system status. The specific steps are explained below:

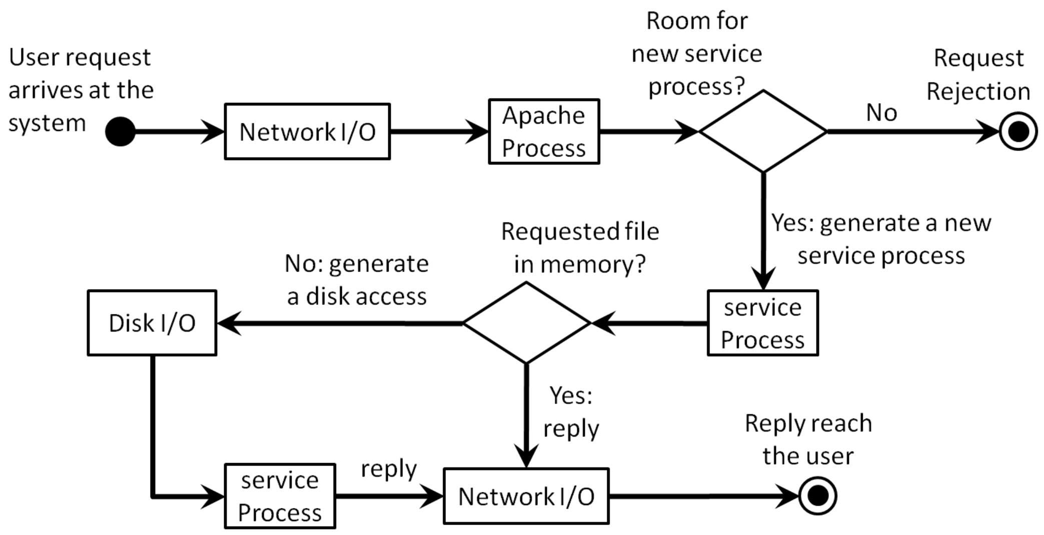

User request arrives at the system
Network I/O
Apache Process
Room for new service process?
No
Request Rejection
Yes: generate a new service process
service Process
Requested file in memory?
No: generate a disk access
Disk I/O
Yes: reply
service Process
reply
Network I/O
Reply reach the user

Figure 5.2. Apache Process Data Flow

1) The request passes the file server Network I/O.

2) The request is handed to the main Apache process.

3) If there are fewer active service processes than the maximum amount allowed, the main Apache process generates a new service process to handle the request; otherwise, the request is rejected.
4) The request may be seeking file items that are within the file server memory. In this case, the service process directly forms a response and replies to the user through the file server Network I/O.
5) If the requested file is not found within the file server memory, an access to the file server disk is initiated by the service process.
6) After obtaining the file data from the disk, the service process applies the action specified by the request (e.g., read or write) and then sends the response, through the Network I/O, to the user.

The backup functionality of the 'rsync' tool consists of three processes running on the servers: 'Sender' on the file server, and 'Generator' and 'Receiver' on the backup server as in Figure 5.1. The backup proceeds as follows:

1) At the beginning of a 'rsync' backup process, the 'Sender' builds a list of all the files (except for those ruled out by, e.g., user-defined backup configurations) on the file server, and sends it to the 'Generator' on the backup server.
2) The 'Generator' compares the file list with its local contents and decides which files are necessary to obtain from the file server based on file metadata, e.g., 'file size' or 'time of last modification'. Note that the exact file versions used in this decision process depends on whether the backup is a full backup, a differential one, or an

incremental one. After this decision process, the 'Generator' walks the list of files to be transferred and sends each file name with its block checksums to the 'Sender'.

3) If the backup being performed is a full backup, the 'Sender' will directly send the whole requested file to the 'Receiver'; otherwise, it will perform more complicated file block checksum computations and only send file blocks whose checksum values are different from those provided by the 'Generator'.

4) The file (or some of its blocks) from 'Sender' is accepted by the 'Receiver' and wrote to the backup server disk. 'Receiver' then notifies 'Generator' to send the name of the next file with its checksums.

Steps 3) and 4) in the above process will repeat until the 'Generator' finishes walking the file list, which also signals the completion of the backup procedure.

5.2.2 Backup Scenarios

This chapter partially overlaps with the direction of a previous paper [88] by investigating system availability metrics under different policies composed of mixtures of full and differential offline backups under a variety of backup schedules. However, this chapter makes significantly extension to the previous one by considering incremental backups and online backups. I also investigate several new issues related to system metrics under the new backup policies.

1) *The effects of incremental partial backup.* A differential backup always backs up all changed data since the last full backup. In contrast, an incremental backup only processes data change that has occurred since the last backup, which may be either

a full backup or an incremental one. An incremental backup typically completes faster due to the reduced amount of data involved, but may lead to longer restoration time. The cause of a longer restoration is that the system in this case must first be restored to the last full backup, then restoration based on every incremental backup since the last full backup up to the one just before system failure must be performed in sequence. The effect of such a trade-off on system metrics is an interesting issue.

2) *The effects of online backup*. For the offline backup scenario, user accesses are not allowed during the backup. In contrast, online backup permits user access while backup is underway. This feature has several implications for both the Apache and the backup process. As Figure 5.1 shows, the Apache and the backup process share a set of resources: 'CPU', 'Network I/O' and 'Disk' on the file server (I assume the file server has a single Network I/O which handles both backup data and user accesses). Naturally, this raises the question that, since both processes are running simultaneously during an online backup, how much do they affect each other? To investigate this issue, I consider the following two system configurations:

   a) Firstly, the two processes compete for shared resources if neither holds priority. In this case, it may happen that the backup process takes longer to complete and user requests are processed more slowly, leading to exhaustion of service processes and request rejection.

b) On the other hand, if the backup process has a higher priority it can be expected to complete as scheduled, but it may consume more system resources compared to the first case. This can potentially cause further performance degradation of the Apache process. However, in practice backups usually execute in periods of reduced user activities (e.g., in the nights), and the degradation may not be a significant issue.

This chapter is concerned with whether and to what extent such situations can arise.

3) *The effects of a limited number of service processes*. As previously described, I consider the number of concurrent Apache service processes to be limited. Exhaustion of the service processes, which occurs when they are all handling user requests, leads to rejection of newly arriving ones. Understandably, the rejection could become more severe when the backup process is running and taking away shared resources. However, even in a normal system operation period without backup process some requests may be rejected due to occasional exhaustion of service processes. Based on the models in the next section, I will look into this issue.

I develop several new backup and user access models which greatly extend the models in [88] and allow me to tap into these issues and evaluate important system availability and performance metrics under various backup policies, system configurations and workload scenarios.

5.3 Models and System Metrics

In this section, I first discuss the motivation behind developing the models and how to apply them. Second, I present the stochastic models developed for the system described in the previous section.

Based on the description earlier, the failure and recovery of the storage system affects how likely it is for data loss to occur, and more frequent data backups help reduce data loss. On the other hand, more frequent backup potentially impacts system availability and performance by either taking the system offline (e.g., in an offline backup) or by reducing application performance (e.g., in an online backup). Consequently models are needed that capture the availability and performance impact of backups to study the tradeoffs between performance, availability and data loss and determine backup frequency.

Motivated by the above considerations, I have developed a set of availability models using Stochastic Reward Nets (SRN) and a set of performance models with Markov chains and product-form queuing networks. The availability models capture system failure/restore/backup behavior and yield system availability and data loss metrics, while the performance models capture resource contention during different system operation periods and provide the corresponding task rejection probabilities. Finally, I also derive new formulas that take the availability and rejection probability values to compute the rate/ratio of task loss and rejection, including the effect of failure/recovery/contention, during system operation.

5.3.1 Stochastic Models

5.3.1.1 Availability Models

For availability, I concentrate on the file server and assume the other devices, e.g., the network and the backup server, do not fail. I further assume that the file server availability is determined by two of its components: the Apache process and the underlying storage, i.e., the file server disk. Infrequent failures of the network card and CPU are ignored for simplicity. For file server availability, I mainly adopt the SRN models from [88] and introduce new parameters so that it also applies to the online backup scenario investigated in this chapter. These models are then used to compute the availability of the file server subject to different backup policies. All these availability models can be generated from high-level system descriptions (e.g., SysML) via tools such as Candy [85] in an automated fashion. The models are briefly described below.

The set of SRN models consists of one model for the storage system behavior, one for backup execution, and two for full and partial backup schedules respectively. The storage system SRN captures the operation, failure and recovery of the storage, the backup execution SRN captures the steps of backup execution (both full and partial), and the schedule SRNs capture scheduling of backups (full and partial). There are dependencies among the models: the execution and schedule SRNs depend on the storage SRN (backups may execute only if the system is running) as well as on each other (backups are initiated based on schedules and schedule timers reset after corresponding backups). Such dependencies among apparently unconnected models are captured via guard functions.

Figure 5.3 shows the Stochastic Reward Net (SRN) model of the storage. The 'P_{UP}' place represents the normal operation of the storage system. A failure in the system may or may not result in data loss (corresponding to the two transitions '$T_{dataloss}$' and 'T_{pfail}' respectively). If data loss occurs, after an automatic detection (transition '$T_{detect1}$'), the system administrator is summoned (whose arrival is captured by the transition '$T_{arrival}$'). The administrator will first restore the data from the last backup ('$T_{restore}$') and then restart the storage system ('$T_{restart1}$'). On the other hand, if no data is lost, the system will be automatically restarted ('$T_{restart2}$') after detection of the failure ('$T_{detect2}$').

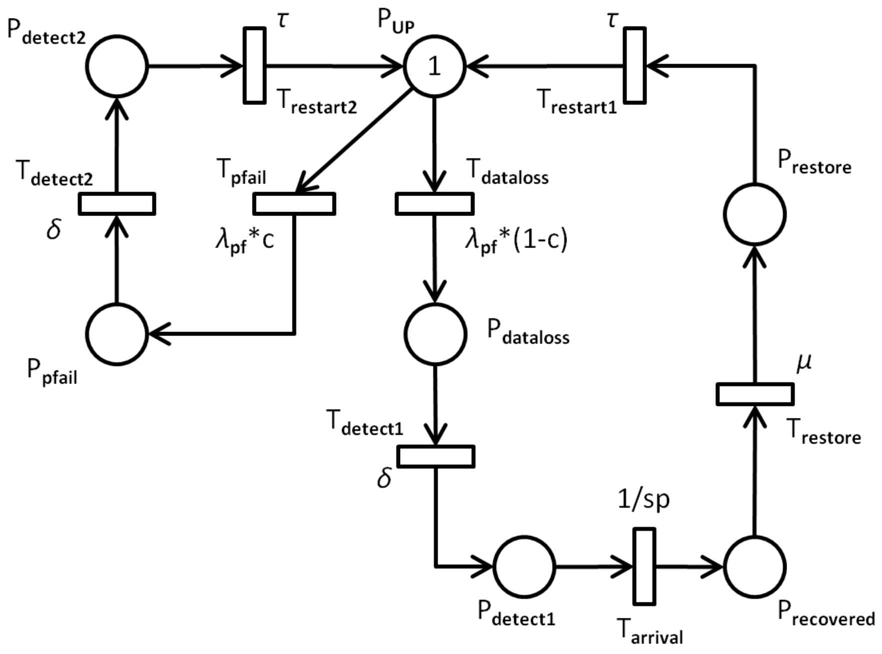


Figure 5.3. SRN for Storage System Failure/Recovery

Figure 5.4 presents the full and partial (incremental or differential) backup operations. The place '$P_{internal\_UP}$' represents the system state when no backup is underway. Different sets of transitions and places capture the cases of full and partial backup: when a full backup occurs the transition 'T_{in_SFBkp}' fires, and 'T_{in_SPBkp}' fires when a partial backup occurs. The model components leading from the two transitions capture the failure ('T_{FBFail}' and 'T_{PBFail}'), recovery ('T_{FBRec}' and 'T_{PBRec}') and success ('T_{FBkp}' and 'T_{PBkp}') of the two backup types and the system restart after a backup completes ('$T_{STRstart1}$' and '$T_{STRstart2}$'). Finally, 'T_{out_SPBkp}' and 'T_{out_SFBkp}' provide synchronization with the schedule models. All the transitions are guarded by enabling functions summarized in Table 5.1. Specifically, the first eight functions, with dependency only on the number of tokens in place 'P_{UP}' in Figure 5.3, capture the fact that a backup can execute only if the system is not running. Similarly, the other functions capture the effect of different backup schedules through additional dependency on places 'P_{in_CSFBkp}' etc. (discussed in detail later).

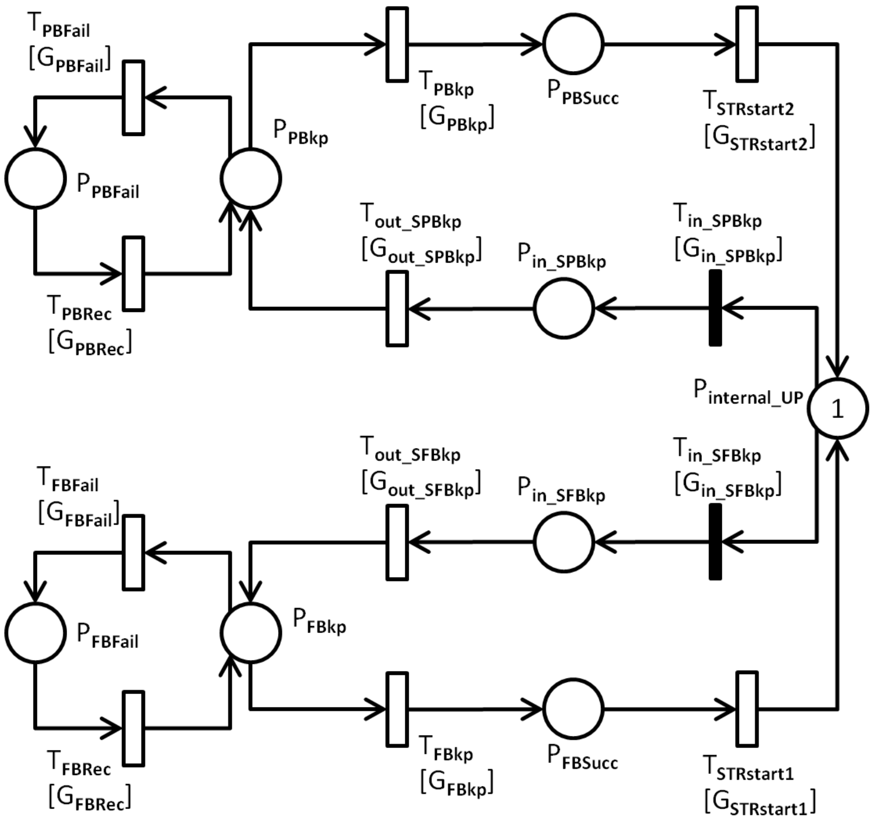


Figure 5.4. SRN for Storage System Backup Operations

Table 5.1. Backup SRN Guard Functions

Function	***Definition***
G_{FBkp},G_{PBkp}, $G_{STRstart1}$, $G_{STRstart2}$, G_{FBFail}, G_{PBFail}, G_{FBRec}, G_{PBRec}	if(#('P_{UP}')==1) 1 else 0
Function	***Definition***
G_{in_SFBkp}	if(#('P_{UP}')==1 and #('P_{in_CSFBkp}')==1) 1 else 0
G_{out_SFBkp}	if(#('P_{UP}')==1 and #('P_{out_CSFBkp}')==1) 1 else 0
G_{in_SPBkp}	if(#('P_{UP}')==1 and #('P_{in_CSPBkp}')==1) 1 else 0
G_{out_SPBkp}	if(#('P_{UP}')==1 and #('P_{out_CSPBkp}')==1) 1 else 0

Backups are performed according to specific schedules, e.g., one full backup per week with three partial backups in between. Figure 5.5 and Figure 5.6 show the schedule models for both backup types (i.e., full and partial). The schedule is captured by the places 'P_{in_clock1}' and 'P_{out_clock1}', and the transitions '$T_{clockFB}$', 'T_{reset1}' and 'T_{wait1}' in Figure 5.5. Similar places/transitions are present in Figure 5.6. Notably, '$T_{clockFB}$' and '$T_{clockPB}$' are deterministic transitions and capture the time between two backups of the same type (e.g., seven days between two full backups). In this way the two models achieve synchronization and enforce the backup schedule. The other model components capture the dependency on the other SRN models (Figure 5.3 and Figure 5.4) via a series of control places and transitions with enabling functions. The functions are summarized in Table 5.2.

The guard functions in Table 5.2 capture the connection between backup schedules/execution and the status of the storage system, as well as that between the two types of backup. Specifically, the four functions that depend on the place 'P_{UP}', i.e., 'G_{out1_decn1}' etc., account for the fact that backups are only performed if the storage system is running. Functions 'G_{in_CSFBkp}' and 'G_{out_CSFBkp}' provide synchronization with the backup execution model in Figure 5.4: the former is enabled when the full backup starts, i.e. when a token is in 'P_{in_SFBkp}' in

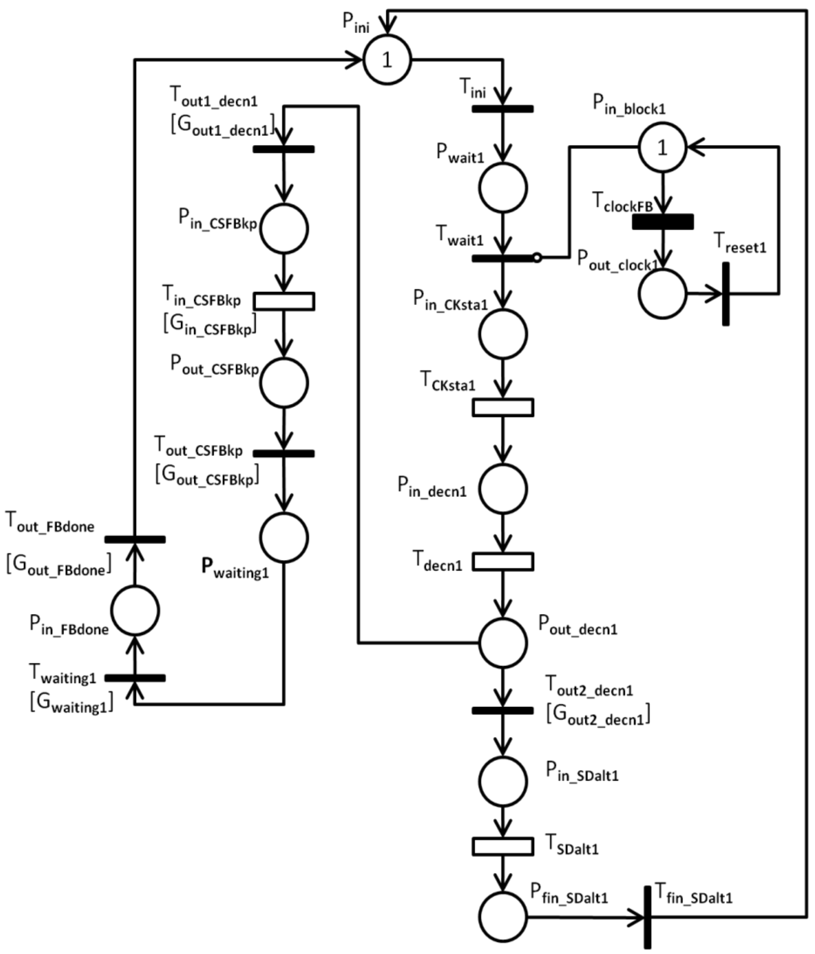


Figure 5.5. SRN for Full Backup Schedule

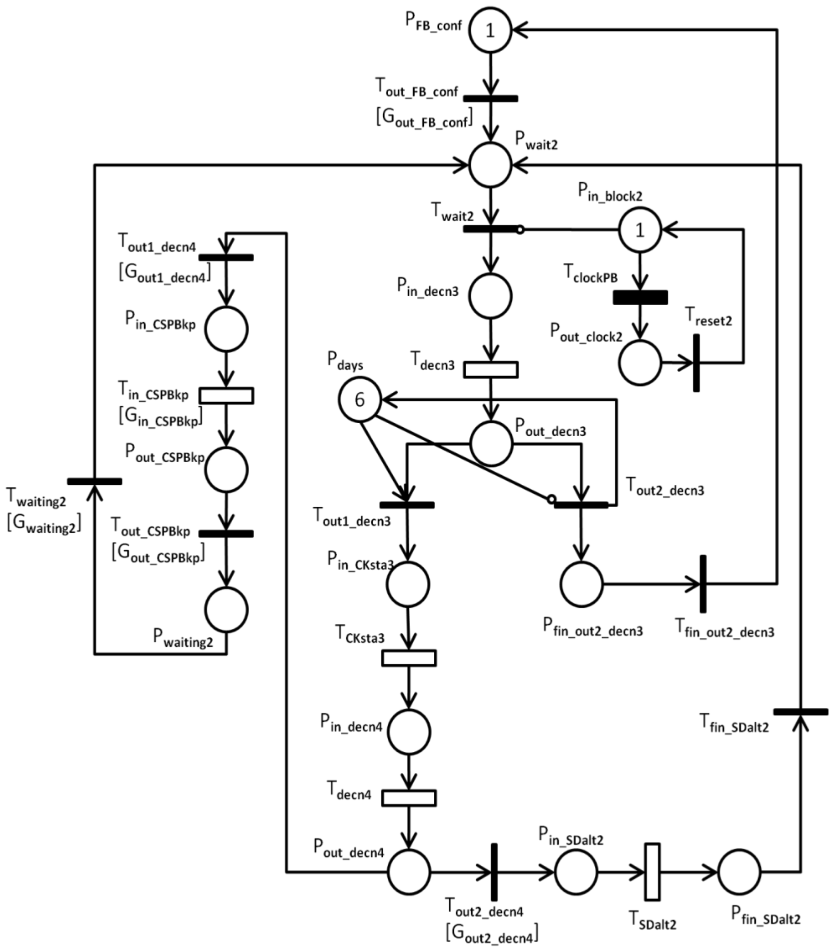


Figure 5.6. SRN for Partial Backup Schedule

Figure 5.4, while the latter puts the schedule model into a waiting state (‘$P_{waiting1}$’) while the backup is underway. After full backup completion, ‘$T_{waiting1}$’ is enabled allowing the full backup timer to restart, unless a partial backup is underway (captured by ‘G_{out_FBdone}’). The partial backup process is also controlled by functions in Table 5.2in a similar fashion.

Table 5.2. Backup Schedule SRN Guard Functions

Function	Definition
Full Backup Schedule SRN (Figure 5.5)	
G_{out1_decn1}	if (#(‘P_{UP}’)==1) 1 else 0
G_{out2_decn1}	if (#(‘P_{UP}’)==1) 0 else 1
G_{in_CSFBkp}	if(#(‘P_{in_SFBkp}’)==1) 1 else 0
G_{out_CSFBkp}	if(#(‘P_{in_SFBkp}’)==0) 1 else 0
$G_{waiting1}$	if (#(‘P_{FBSucc}’)==1) 1 else 0
G_{out_FBdone}	if(#(‘P_{FB_conf}’)==0) 1 else 0
Partial Backup Schedule SRN (Figure 5.6)	
$G_{out\_FB\_conf}$	if(#(‘P_{in_FBdone}’)==1) 1 else 0
G_{out1_decn4}	if (#(‘P_{UP}’)==1) 1 else 0
G_{out2_decn4}	if (#(‘P_{UP}’)==1) 0 else 1
G_{in_CSPBkp}	if(#(‘P_{in_SPBkp}’)==1) 1 else 0
G_{out_CSPBkp}	if(#(‘P_{in_SPBkp}’)==0) 1 else 0
$G_{waiting2}$	if (#(‘P_{PBSucc}’)==1) 1 else 0

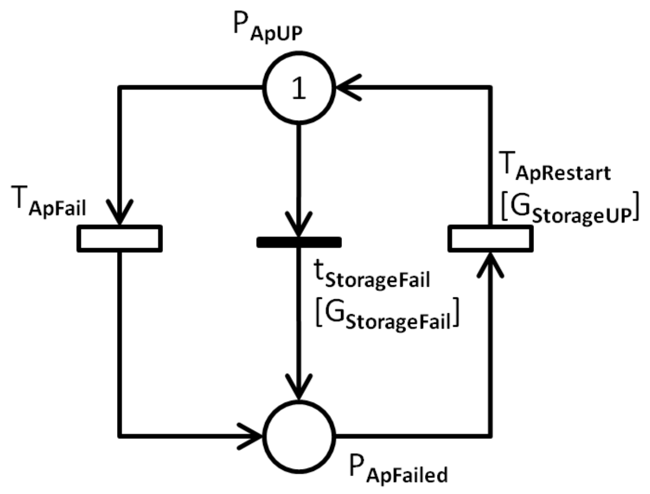


Figure 5.7. SRN for Apache Process Availability

The Apache process running on the storage system is modeled as in Figure 5.7. A token in the place ‘P_{ApUP}’ indicates the Apache process is working properly. When the process fails, the transition ‘T_{ApFail}’ fires and a token is deposited in the place ‘$P_{ApFailed}$’. Since the process has a hosted dependency to the storage system, the immediate transition ‘$t_{StorageFail}$’ fires whenever there is no token in the place ‘P_{UP}’ in Figure 5.3 or the place ‘$P_{internal\_UP}$’ in Figure 5.4.The guard functions $G_{storageFail}$ and $G_{storageUp}$ assume value zero/one when there is a token in the place ‘P_{UP}’ in Figure 5.3, and one/zero otherwise. The availability of the Apache process can be obtained as the probability that the ‘P_{ApUP}’ place is not empty. To capture complex failure and recovery behaviors, this simple model can be replaced by more sophisticated ones without modifying the other availability and/or performance models presented in this chapter.

Note that the four SRN models described above constitute a single overall SRN model that is solved to obtain availability and data loss metrics. In this overall model, the firing time distributions of transitions fall into three types: deterministic, immediate and exponential. The deterministic transitions (filled rectangles) are used to capture the backup schedules in Figure 5.5 and Figure 5.6. An SRN model with a deterministic transition (i.e., a DSPN) can be solved either through the method of subordinated CTMC [89] or with Erlang approximation (here I use the latter). The immediate transitions (by solid, narrow stripes) are used in the models to capture conditions and branches in conjunction with guard functions. I assume exponential distributions for all other transition (represented by hollow rectangles) firing times so that the models can be solved analytical-numerically. Although this assumption potentially affects the result values, I would like to point out that the main purpose of the modeling procedure presented in this chapter is to facilitate comparison of backup techniques and schedules. Thus my approach would still capture the relative merits of backup policies and facilitate meaningful comparisons. On the other hand, it is also possible to incorporate non-exponential distributions using the device of stages at the expense of a larger state space [6]. Furthermore, it would be straightforward to switch to discrete-event simulation with the same models and incorporate general firing time distributions, using packages such as SPNP [64].

In addition to the availability models shown above, I also develop detailed performance models to capture the system behavior during online backups. Specifically, I develop three stochastic models: 1) for the backup process with priority, 2) for backup without priority, and 3) for the normal operation of the Apache process when no backup is taking place. Each of the three models consists of a top-level continuous time Markov chain and an underlying closed queuing network. The Markov chain remains the same in all cases while the queuing network models differ. These models are described now.

5.3.1.2 Online Backup Performance (with Priority)

The performance model for online backups with priority consists of two models: a closed queuing network sub-model (Figure 5.8), and a top-level Markov chain (Figure 5.9).

The model in Figure 5.8 captures the behavior of each process (backup and Apache) in a separate queuing chain, with the two chains overlapping at certain queuing stations corresponding to shared system resources. The solid arcs represent the data flows at the file server, while the dashed arcs stand for those at the backup server.

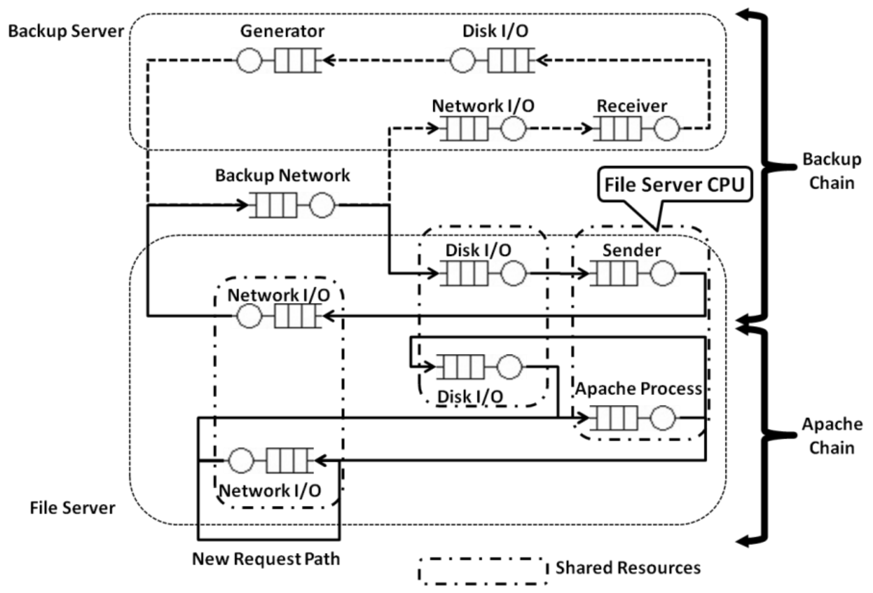


Figure 5.8. Closed Queuing Network Sub-model

$$\begin{tikzcd}
0 \arrow[r, shift left, "\lambda"] & 1 \arrow[l, shift left, "\mu_1"] \arrow[r, shift left, "\lambda"] & 2 \arrow[l, shift left, "\mu_2"] \arrow[r, shift left, "\lambda"] & 3 \arrow[l, shift left, "\mu_3"] \arrow[r, shift left, "\lambda"] & \cdots\cdots \arrow[l, shift left, "\mu_4"] \arrow[r, shift left, "\lambda"] & L\text{-}1 \arrow[l, shift left, "\mu_{L-1}"] \arrow[r, shift left, "\lambda"] & L \arrow[l, shift left, "\mu_L"]
\end{tikzcd}$$

Figure 5.9. Main Apache Model

The ‘Backup Chain’ captures the system component (hardware resources and software processes running on CPUs) behaviors in the backup procedure. I make a few simplifications when translating the backup procedure in Figure 5.1 into this model. These simplifications and the reasoning behind are provided as follows:

1) In the backup server, I assume that the work flow goes through the components in sequence. In reality (as in Figure 5.1), there is some communication between ‘Receiver’ and ‘Generator’, and ‘Generator’ also sends requests through ‘Network I/O’. I simplify the first case because ‘Receiver’ only sends short notification messages to ‘Generator’ whose size is small compared to other data transfer. For the second case, the simplification comes from the fact that, compared to the major

source of traffic at 'Network I/O' which is the file block data from the file server, the size of new file request sent by 'Generator' is small and can be ignored.

2) In the file server, I assume that the work flows from 'Disk I/O' to 'Sender' and then to 'Network I/O'. In reality, the work flow consists of both requests from 'Generator' to 'Sender' via 'Network I/O' and I/O requests from 'Sender' to the disk. However, both types of requests are much smaller in size than the major traffic at the respective system resources (the file blocks at 'Network I/O', and the content of the files to be transferred at 'Disk'), and thus can be ignored.

The 'Apache Chain' (see Figure 5.8) captures request execution by the Apache process during the backup. Note that such a backup is an online backup since user accesses are allowed. A request enters the system through the network interface (represented by the station 'Network I/O'), and requires one or more CPU processing (at the 'Apache Process' station, which represents execution of the service process responsible for this request in the file server CPU) and zero or more disk accesses (at the 'Disk I/O' station) before leaving through the network interface. I assume that the queuing stations corresponding to network interfaces and disks use the First-Come-First-Serve (FCFS) scheduling policy, while those corresponding to CPU processing, i.e., 'Receiver', 'Generator', 'Sender' and 'Apache Process', use the Process Sharing (PS) scheduling policy. I also assume the 'Backup Network' is an Infinite Server (IS) station. This is based on the assumption that the main delay in the network is the transmission delay instead of the queuing at networking devices.

In this model I assume the backup process holds priority over the Apache process. Such a priority can usually be implemented at the CPU, while at the network interface and the disk the two processes compete for accesses. To capture the priority, I use the 'shadow server' approach by slowing down the CPU processing rate for the Apache process (i.e., the rate at the station 'Apache Process') [90]. The effect can be captured by scaling the said rate with one minus the CPU utilization of the backup process, which is the utilization of the station 'Sender'. However, the fact that priority is only present at the CPU prevents a direct application of the approach since it requires priorities at all queuing stations. To deal with this issue, I adopt the following iterative approach:

1) Start with a situation without priority between the two chains. Solve for the utilization of the 'Sender'.
2) Scale down the rate of the station 'Apache Process' by multiplying it with one minus the utilization obtained in step 1). Update the utilization of the 'Sender' by solving the new model.
3) Repeat 2), until the utilization of the 'Sender' no longer changes between two successive iterations, or when the change falls below a threshold.

Following this approach, it is possible obtain the throughput of the 'Network I/O' station in both chains under different numbers of concurrent requests, with the backup process holding priority. The Apache process throughput can then be obtained as the throughput of the 'Network I/O' station multiplied by 0.5 in the Apache chain. The 0.5 factor captures the fact that, since the queuing station processes each request and its reply once, in the steady-state half of the tasks processed by the station will be replies.

I use the closed queuing network to model the scenario for two reasons: 1) it captures the fact that a limited number of user requests can be executed concurrently in the system, and2) it admits a product-form solution [6]. However, the assumption of closed queuing networks dictates that when a request leaves the system another immediately enters to keep the number of requests in the system constant. Consequently, the model deviates from reality in that the number of user requests in a real system can vary from zero to the maximum number of concurrent service processes. To address this issue, I use the flow-equivalent server approach [91] as described below:

1) Vary the number of user requests in the sub-model from one to L, where L is the maximum number of concurrent requests the system can handle.
2) For every number of user requests, compute the throughput of 'Network I/O' in the Apache chain. This value, after multiplied by 0.5 to exclude incoming requests, gives the throughput (i.e., rate of providing responses to the users) of the Apache server under the given number of requests.
3) Finally, construct an overall continuous-time Markov chain model (as in Figure 5.9) whose states reflect the number of user requests being concurrently processed by the system (which varies from zero to L). The rate λ equals the arrival rate of new user requests, and the service rates μ_i, i from 1 to L, equal the throughput obtained from the sub-model with the corresponding numbers of concurrent user requests, as described in step 2 above.

Following the above steps, I can obtain the probability of requests rejection, which is the probability that the model is in state L, from the Markov chain in Figure 5.9.

5.3.1.3 Online Backup Performance (without Priority)

The models for the backup without priority case have the same structure as the models given in Figure 5.8 and Figure 5.9. Specifically, as in Figure 5.8, the two processes are each modeled as a chain with the two chains overlapping at queuing stations corresponding to shared resources. Similar as before, all the stations are assumed to have FCFS queuing policies, with the exceptions of 'Sender' and 'Apache Process' (which reside within the file server CPU) which are PS, and the 'Backup Network' which is IS. No priority is assigned in this case and the two processes (backup and Apache) contend for resources. The absence of priority means that the 'shadow server' approach is no longer needed, resulting in an ordinary multi-chain network. I can obtain the throughput of the Apache process (as the throughput of the 'Network I/O' station, multiplied by 0.5, in the Apache process chain) under different numbers of concurrent requests. Such values are then used in the main Markov model in Figure 5.9to obtain request rejection probability. I also obtain the throughput of the backup process from the model, which may be smaller than in the priority case. This throughput is used to adjust the backup completion time, as will be described later.

5.3.1.4 Performance Model for Normal Operation

Finally, I have the performance models for normal system operations, i.e., when the Apache process is running without backup taking place. The models are similar to those in previous cases, consisting of a closed queuing network sub-model and a top-level Markov chain. The Markov chain remains the same as in Figure 5.9, while the queuing network model is given in Figure 5.10.

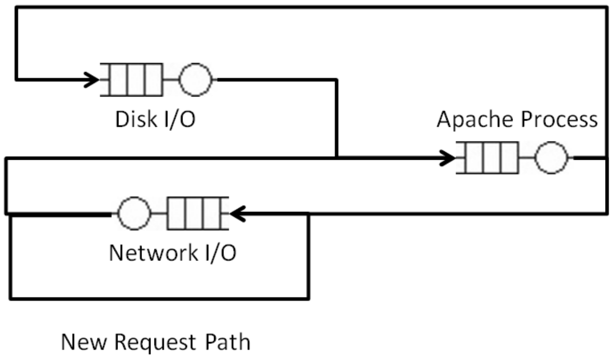


Figure 5.10. Queuing Model for Normal Operation

In Figure 5.10, the backup chain is no longer present, and throughput of the Apache process equals the throughput of the station ‘Network I/O’ multiplied by 0.5. This value is then used in the Markov chain to compute request rejection probability during normal operation.

5.3.2 System Metrics

With the SRN models developed in Section 0, I obtain the file service availability as the probability that no token is in ‘P_{UP}’ in Figure 5.3, ‘$P_{\_internal\_UP}$’ in Figure 5.4 or ‘P_{ApUP}’ in Figure 5.7. I also introduce four other metrics: ‘data loss rate’, ‘data loss ratio’, ‘request rejection rate’ and ‘request rejection ratio’. The first two metrics are related to system performance, while the last two represent system availability perceived by users. I develop new formulas for these metrics based on the outputs from the models presented in Section 0, including a major revision on how the first two metrics are computed as compared to [88]. A summary of the metrics is presented in Table 5.3.

As described in Section 5.2, I will investigate these system metrics under combinations of incremental backup, online backup with different priority process settings and user workloads. In the next sub-section, I derive the metrics in Table 5.3in these new scenarios.

Table 5.3. System Metrics of Interest

Metrics	Descriptions
File Service Availability	The probability that the file service is available in steady-state
Data Loss Rate	The rate at which user updates to local data are lost
Data Loss Ratio	The fraction of user updates lost out of all accepted updates/writes
Request Rejection Rate	The rate at which incoming user requests are rejected
Request Rejection Ratio	The ratio of rejected user requests out of all incoming requests

5.3.2.1 File Service Availability

The file service availability is obtained directly from the SRN availability models described earlier. New completion and restoration rates corresponding to different backup techniques (i.e., full, differential, and incremental) are put into the models to compute the availability metric.

5.3.2.2 Data Loss Rate and Loss Ratio

These two metrics correspond to the number of user updates to local data that are lost due to storage failure, before the system has a chance to back it up. To understand how I derive their formulas, it helps to first introduce the backup scenarios in greater detail.

1) First, I use a more detailed request arrival model in which a day is divided into two periods: the office period and the after-hours period. I assume an office period lasts for sixteen hours and an after-hours period lasts for eight hours. I further assume that the request arrival processes within the two periods are Poisson with different parameters, and ignore more complex daily workload patterns. I also assume the

office period sees a higher rate than the after-hours period, and ignore the transient arrival rate variation at the boundary of the two periods.

2) Second, I classify user requests into ‘write’ and ‘read’ types, and distinguish between request rejection and data loss. Request rejection occurs when anew request finds the system down, undergoing backup (in offline backups), or the number of existing requests has reached the allowed limit. In contrast, data loss occurs when a system failure that is not covered occurs before the data updates from some ‘write’ requests is backed up. Thus only ‘write’ requests may contribute to data loss, while both request types contribute to request rejection.

3) Third, I assume that a backup always starts at the beginning of an after-hours period, and takes some time (the ‘backup period’) to complete. Each backup covers all the data updates accumulated in a certain time period before the beginning of the backup. For offline backup, this period starts from the end of the last backup. For online backup, it starts from the beginning of the last backup, since new requests may be processed while the previous backup is underway, and I assume the data updates from these new requests are not processed by the previous backup. A diagram illustrating the different periods in my scenario is given in Figure 5.11.

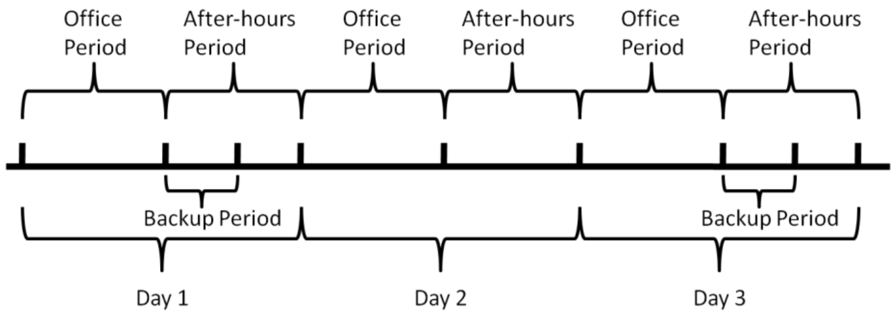


Figure 5.11. System Operation Periods

4) Lastly, I assume that the backup server does not fail and that once a data undergoes backup it will not be affected by any failure of the system in the future. Such an assumption can be easily relaxed by introducing another server availability model for the backup server.

Based on the scenarios described above, I derive two formulas for the expected data loss per file server failure for both the offline and online backup cases. Based on these formulas, I can then derive other system metrics: data loss rate, data loss ratio, request rejection rate and request rejection ratio. To do this, first I define the symbols used in the formulas in Table 5.4.

Table 5.4. Formula Symbols

Symbols	Descriptions
N	The number of full days between two backups
T_O, T_A	The length of an office/after-hours period
T_{B1}, T_{B2}	The lengths of the two adjacent backup periods
T	The time between the starting points of two backups, i.e.,$T=(N+1)(T_O+T_A)$.
λ_O, λ_A	Overall arrival rates of user requests during the office periods and the after-hours periods
λ_{OA}, λ_{AA}, λ_{BA}	Arrival rates of accepted requests during the office, the after-hours and backup periods
λ_{OW}, λ_{AW}, λ_{BW}	Arrival rates of accepted write requests during the office, the after-hours and backup periods
P_{rej_O}, P_{rej_A}, P_{rej_B}	Request rejection probabilities of the office, the after-hours and the backup periods due to system resource exhaustion
P_{write}	The fraction of write requests out of all requests
A_F	File service availability without backup

P_{rej_O}, P_{rej_A} and P_{rej_B} can be obtained from the performance models in Section 0. λ_O and λ_A are the total arrival rates of requests (accepted or not) in the office and after-hours periods respectively. Given these two rates λ_{OA}, λ_{AA}, λ_{BA}, λ_{OW}, λ_{AW} and λ_{BW} can be obtained following (5.1):

$$\begin{aligned} \lambda_{OA} &= \lambda_O A_F (1 - P_{rej\_O}) & \lambda_{OW} &= \lambda_{OA} P_{write} \\ \lambda_{AA} &= \lambda_A A_F (1 - P_{rej\_A}) & \lambda_{AW} &= \lambda_{AA} P_{write} \end{aligned} \tag{5.1}$$

$\lambda_{BA} = \lambda_A A_F (1 - P_{rej\_B})$ $\lambda_{BW} = \lambda_{BA} P_{write}$

Notice that in (5.1) I use A_F, the file service availability without backup, which is computed as the probability that there is no token in 'P_{UP}' in Figure 5.3 or in 'P_{ApUP}' in Figure 5.7. I use this metric instead of file service availability because, as will be described in subsequent sections, my formulas for computing data loss and request rejection explicitly account for the effect of backups. Therefore, the arrival rates in (5.1) should exclude the backup part to avoid double-counting its effect. Also note that since the backup is performed in after-hours periods, it's execution may affect the number of accepted write requests, thus making the arrival rates of accepted write requests during the backup periods (i.e., λ_{BW}) different from those for the rest of the after-hours periods (i.e., λ_{AW}).

The formulas for the expected data loss per storage failure, conditioned on one failure within the period *T*, are derived next. I only consider storage failures since Apache process failures do not lead to loss of data update already processed.

The offline case:

$$\begin{aligned}
E = &\int_0^{T_A - T_{B1}} \lambda_{AW} \frac{t}{T} dt \\
&+ \sum_{n=0}^{N} \{ \int_0^{T_O} [\lambda_{AW}(T_A - T_{B1}) + n\lambda_{OW}T_O + n\lambda_{AW}T_A + \lambda_{OW}t] \frac{1}{T} dt \} \\
&+ \sum_{n=0}^{N-1} \{ \int_0^{T_A} [\lambda_{AW}(T_A - T_{B1}) + (n+1)\lambda_{OW}T_O + n\lambda_{AW}T_A + \lambda_{AW}t] \frac{1}{T} dt \} \\
&+ \int_0^{T_{B2}} [\lambda_{AW}(T_A - T_{B1}) + (N+1)\lambda_{OW}T_O + N\lambda_{AW}T_A] \frac{1}{T} dt \\
&= \frac{\lambda_{AW}(T_A - T_{B1})[2(N+1)T_O + (2N+1)T_A - T_{B1} + 2T_{B2}]}{2T}
\end{aligned} \tag{5.2}$$

$$+\frac{[(N+1)T_O + NT_A - T_{B1} + 2T_{B2}][(N+1)\lambda_{OW}T_O + N\lambda_{AW}T_A]}{2T}$$

The online case:

$$\begin{aligned}
E &= \int_0^{T_A - T_{B1}} (\lambda_{BW}T_{B1} + \lambda_{AW}t)\frac{1}{T}dt \\
&+ \sum_{n=0}^{N}\left\{\int_0^{T_O}[\lambda_{BW}T_{B1} + \lambda_{AW}(T_A - T_{B1}) + n(\lambda_{OW}T_O + \lambda_{AW}T_A) + \lambda_{OW}t]\frac{1}{T}dt\right\} \\
&+ \sum_{n=0}^{N-1}\left\{\int_0^{T_A}[\lambda_{BW}T_{B1} + \lambda_{AW}(T_A - T_{B1}) + (n+1)\lambda_{OW}T_O + n\lambda_{AW}T_A + \lambda_{AW}t]\frac{1}{T}dt\right\} \\
&+ \int_0^{T_{B2}}[\lambda_{BW}T_{B1} + \lambda_{AW}(T_A - T_{B1}) + (N+1)\lambda_{OW}T_O + N\lambda_{AW}T_A + \lambda_{BW}t]\frac{1}{T}dt \qquad (5.3) \\
&= \left[\lambda_{BW}T_{B1} - \frac{\lambda_{BW}T_{B1}^2}{T} + \frac{\lambda_{BW}T_{B1}T_{B2}}{T} + \frac{\lambda_{BW}T_{B2}^2}{2T}\right] \\
&+ \frac{\lambda_{AW}(T_A - T_{B1})[2(N+1)T_O + (2N+1)T_A - T_{B1} + 2T_{B2}]}{2T} \\
&+ \frac{[(N+1)T_O + NT_A + 2T_{B2}][(N+1)\lambda_{OW}T_O + N\lambda_{AW}T_A]}{2T}
\end{aligned}$$

Equations (5.2) and ((5.3) focus on the occurrence of a storage failure between two neighboring backups, since once a backup is done the data will not be affected by future failures (as per my assumption). In (5.2), the first line has four terms. Since I assume that the occurrence of file server failures follows a Poisson distribution, in all the four terms the *1/T* factor is present. This factor is the density of a failure occurring in the time between two neighboring backups, conditioned on that only one failure occurs in this period. Because the storage failure rate is usually very small, I believe the assumption of one failure is reasonable so long as *T* is small compared to the MTTF of the storage. The first term computes the amount of user write requests from the completion of the previous backup to the end of that after-hours period; these write requests will be lost if a failure occurs (with probability 1/*T*) within the after-hours period. The second and third terms correspond to the writes losses if a failure occurs during one of the office periods or after-hours periods, respectively, before the next backup. Taking the second term as an example, $\lambda_{AW}(T_A\text{-}T_{B1})$ is the expected number of updates received from the after-hours period after the last backup; $n\lambda_{OW}T_O$ and $n\lambda_{AW}T_A$ are the expected numbers of updates received from all the office and after-hours periods before the failure, with *n* being the number of whole days that passed from the last backup to the moment of failure; $\lambda_{OW}t$ is the expected number of updates in the office period where the failure occurs. The sum of these components is weighted by *1/T*, the probability that a failure occurs at that given moment, to give the expected loss if a failure occurs in an office period. Similarly, the third term captures the expected loss if a failure occurs in an after-hours period. Finally, the last term represents a failure before the next backup is completed, causing the loss of all the updates since the completion of the previous backup. The terms in ((5.3) have similar interpretations.

One difference between (5.2) and ((5.3) is that in the offline case, the number of write requests is accumulated from the end of the last backup to the beginning of the current one, while in the online case it also includes write requests accepted during the two backup periods. This is a result of the user access during online backup, and is reflected by the differences in the first and last terms at the first lines of the two formulas: for the online case, the term $\lambda_{BW}T_{B1}$stands for the amount of updates received during the previous or the current backup.

Based on (5.1), (5.2) and ((5.3), I obtain the data loss rate and data loss ratio using the following equations.

$$DataLossRate = \frac{\lambda_f T e^{-\lambda_f T}(1-c)E}{T} = \lambda_f e^{-\lambda_f T}(1-c)E \tag{5.4}$$

$$OfflineDataLossRatio = \frac{OfflineDataLossRate}{[(N+1)T_O\lambda_{OW} + \lambda_{AW}(NT_A + T_A - T_{B1})]/T} \quad (5.5)$$

$$OnlineDataLossRatio = \frac{OnlineDataLossRate}{[(N+1)T_O\lambda_{OW} + \lambda_{AW}(NT_A + T_A - T_{B1}) + \lambda_{BW}(T_{B1} + T_{B2})]/T} \quad (5.6)$$

In the above equations, the term λ_f is the file server failure rate, *c* is the coverage factor of file server failures, and *T* is the time length between two successive backups. The first three items in the numerator of (5.4) gives the probability that only one failure occurs between two backups. This is then multiplied with (*1*-*c*) and *E* to obtain the expected loss within the period *T.* Finally, loss rate is obtained by dividing by *T* which gives the final form of (5.4). Note that there are two values for *E*, one for the offline case and one for the online case, which leads to two different *DataLossRate* in (5.5) and (5.6). The terms in the denominators of (5.5) and (5.6) correspond to the time-averaged arrival rates of 'write' requests accepted by the system, and some of their values also depend on the backup type under consideration, e.g., the value of λ_{BW} for the offline case would be zero since no user access is accepted.

From (5.1)-(5.6), it is clear that in order to compute the data loss rate/ratio, I need to obtain the following quantities from the models: the file service availability (from the SRN model) and the request rejection rates of different periods (from the queuing network and Markov chain models under different arrival rate parameter values). Different values of parameters lead to different results for the system metrics, as will be shown in Section 5.4.

5.3.2.3 Data Rejection Rate and Rejection Ratio

These two metrics represent how often the system turns down incoming user requests, either due to system failure or exhaustion of resources (i.e., no available service processes). Since a user would consider the system unavailable if the request is rejected, these two metrics capture the system availability as perceived by the users. The metrics are computed using the following equations.

$$Accepted = (N+1)T_O\lambda_{OA} + \lambda_{AA}(NT_O + T_O - T_{B1})\text{offline} \tag{5.7}$$

$$Accepted = (N+1)T_O\lambda_{OA} + \lambda_{AA}(NT_A + T_A - T_{B1}) + \lambda_{BA}(T_{B1} + T_{B2})\text{online} \tag{5.8}$$

$$Total = (N+1)T_O\lambda_O + T_A\lambda_A \tag{5.9}$$

$$RejectionRate = (Total - Accepted)/T \tag{5.10}$$

$$RejectionRatio = (Total - Accepted)/Total \tag{5.11}$$

Here *Accepted* stands for the number of requests accepted by the system, while *Total* is the total number of arrived requests. The arrival rate terms are the same as explained in (5.1), and the others are defined in Table 5.4.

5.4 Parameters and Numerical Results

In this section, I describe the parameter values used in this chapter, and present the numerical results for system output metrics based on these values.

5.4.1 Availability Parameters

For the SRN availability model, the full backup parameters largely remain the same as in [88]. I introduce two new sets of partial backup and restoration rate values to capture the characteristics of differential and incremental backups and set parameter values for the Apache process availability model. For incremental partial backups, I assume the average times for a backup and restoration from a failure are captured by the following expressions. Here *d1* is the number of days between two successive partial backups and *d2* is the number of days passed since the last full backup. $E[T_{comp\_incre}]$ denotes the expected completion time of an incremental partial backup, while $E[T_{restore\_incre}]$ denotes the expected restoration time when incremental backups are used alongside full backups.

$$E[T_{comp\_incre}]=60 + 10(d1 - 1)\text{minutes} \tag{5.12}$$

$$E[T_{restore\_incre}]=40 + 20(d2 - 1) \text{ minutes} \tag{5.13}$$

For differential partial backups, I use similar expressions for the average completion and restoration times.

$$E[T_{comp\_diff}] = 60 + 10(d2 - 1) \text{ minutes} \tag{5.14}$$

$$E[T_{restore\_diff}]= 40 +1\ 0(d2 - 1) \text{ minutes} \tag{5.15}$$

Notice that the position exchange of *d1* and *d2* in (5.12)-(5.15) accounts for the fact that incremental partial backups are faster to perform but makes restoration slower as compared to differential partial backups. As explained earlier, an incremental backup only processes data updates since the last backup (whether incremental or full), hence *d1* in (5.12)). However, when restoration is needed all the incremental backups since the last full backup must be restored, resulting in *d2* in (5.13). Similar reasons exist for (5.14) and (5.15). Based on (5.12)-(5.15), I obtain the storage availability from the SRN models as the steady-state probability that a token is in place 'P_{UP}' in Figure 5.3.

For the Apache process availability model, I assume the failure rate of each Apache process (rate for transition 'T_{Fail}' in Figure 5.7) to be 1/96 hour^{-1} and the restart rate (for '$T_{Restart}$') to be 0.25 minute^{-1}. These values are roughly based on the estimations in [92]. Such values, combined with the parameters described earlier, allow me to obtain the file service availability from the SRN models, which is the probability that there is no token in place 'P_{UP}' in Figure 5.3, place '$P_{internal\_UP}$' in Figure 5.4, or place 'P_{ApUP}' in Figure 5.7.

5.4.2 Data Loss/Request Rejection Parameters

For the models described in Section 5.0, it is necessary to establish the following parameter values: the maximum number of concurrent service processes; the probability that a user request can be handled within memory; the processing rates of various queuing network stations; and finally the arrival rates during different periods of a day.

I assume that the Apache server maintains a maximum of 256 concurrent service processes which is usually the default value. I also assume that a user request has a probability of 1/3 to be completed within memory and not require a disk access. Note that I use this probability to capture all the possible scenarios where a request is completed without disk access, e.g., file cache for read requests, batch-write to disk for write requests, etc.

For the arrival rates, I establish the values based on [93]. Specifically, I use the 'engineering trace' in the reference and obtain the quantities presented in Table 5.5. The total number of requests in Table 5.5 (the first three rows) spans a period of about 97 days.

Table 5.5. User Access Characteristics

Total write/read requests	76032000
Total read requests	53152000
Total write requests	22880000
Overall request arrival rate	≈ 9.0722 sec^{-1}
Read request arrival rate	≈ 6.3421 sec^{-1}
Write request arrival rate	≈ 2.7300 sec^{-1}
Fraction of write requests	30.09%
Total data read	723.4GB
Total data wrote	364.4GB
Avg. read size	723.4GB/53152000 ≈ 13.61KB
Avg. write size	364.4GB/22880000 ≈ 15.93KB

I assume the processing rates of the backup server are three times those of the file server. The reason of this assumption are: first, a backup server tends to be better optimized than the file server for backup-related tasks; second, in practice it is common for the backup server to be a high-end, specifically designed appliance while the file server is a generic, commodity machine. Based on this assumption and the quantities in Table 5.5, I assume the following processing rates for server resources in Table 5.6.

Table 5.6. System Component Processing Rates

File Server disk rate	52.73 sec^{-1}
File Server network I/O rate	46.88 sec^{-1}
Backup Server disk rate	158.20 sec^{-1}
Backup Server network I/O rate	140.63 sec^{-1}
Apache processing rate	1000 sec^{-1}
Sender processing rate	500 sec^{-1}
Generator processing rate	1000 sec^{-1}
Receiver processing rate	1500 sec^{-1}
Backup Network transmission rate	50 sec^{-1}

I assume the processing rates of the backup server are three times those of the file server. The reason of this assumption are: first, a backup server tends to be better optimized than the file server for backup-related tasks; second, in practice it is common for the backup server to be a high-end, specifically designed appliance while the file server is a generic, commodity machine. Based on this assumption and the quantities in Table 5.5, I assume the following processing rates for server resources in Table 5.6.Note that I assume the ‘Sender’ has a lower rate than the ‘Generator’ and ‘Receiver’, based on the fact that the ‘Sender’ needs to conduct advanced block checksum (i.e., rolling checksum) computation which is CPU-intensive.

The arrival rates given earlier are the daily average values. As described earlier, I divide a day into two periods and each with different arrival rates. Specifically, I assume request arrival rate during the after-hours periods is 80% of that for the office periods. The arrival rate values for the two periods are given in Table 5.7, where ‘Office’ and ‘After’ refer to the office period and the after-hours period, respectively. The last two columns give the arrival rates of read and write requests.

Table 5.7. Arrival Rate Values

	Request Arrival Rate (sec^{-1})	Read Arrival Rate (sec^{-1})	Write Arrival Rate (sec^{-1})
Office	9.7202	6.7954	2.9248
After	7.7762	5.4364	2.3398

In order to use the formulas derived in Section 0 and compute the system metrics, the following parameters are also needed: the number of full days between two backups, the arrival rates, and period lengths. The number of days depends on the backup policy, and can range from zero (one backup per day) to six (one backup per week). The arrival rates can be established based on (5.1) and the new models. Finally, I assume the office period to be sixteen hours and the after-hours period to be eight. The backup period lengths (T_{B1} and T_{B2}) by default are computed as the reciprocals of the backup rates depending on the backup types (full, differential or incremental) used in the previous and the current backup periods, but may increase in the non-priority scenario as some resources are consumed by the Apache process. Specifically, the duration of an online backup is as follows:

$$T_{B\_online} = T_B \cdot \frac{Throughput_{priority}}{Throughput_{non-priority}} \tag{5.16}$$

Here $Throughput_{priority}$ and $Throughput_{non\text{-}priority}$ stand for the throughput values of the backup process with and without priority over the Apache process.

The backup policies considered cover a variety of configurations. Specifically, the following factors are varied:

1) Whether partial backup is used;

2) Type of partial backup (incremental/differential);

3) Offline/online backup;

4) In the online case, whether the backup process has priority over the Apache process.

Aside from the techniques used, the backup schedule is also specified. I consider the following schedules:

1) If only full backup is used:
   - Once per week;
   - Once per day.
2) If both full back and partial backup are used:
   - One full per month, one partial each week;
   - One full every two weeks, one partial every two days;
   - One full per week, one partial every two days;
   - One full each week, one partial every day.

5.4.3 Numerical Results

Based on the models and parameters in the previous section, I obtain various numerical values for system metrics. In this section I present file service availability and performance metrics under different backup policies. First I consider policies that only use full backups.

Table 5.8. Offline full backup only

Schedule	File Service Availability	Data Loss Rate (sec^{-1})	Data Loss Ratio	Request Rejection Rate (sec^{-1})	Request Rejection Ratio
Once per week	0.9631	0.0013	4.8632e-4	0.2842	0.0313
Once per day	0.7567	1.8078e-4	8.4347e-5	1.9493	0.2149

In Table 5.8, the results for two backup policies using only offline full backup are given. The results show the effect of different backup frequencies. If we compare the second column from the left, it can be observed that performing full backup once per day is excessive and leads to significantly lower file service availability and heavier request rejection. It does lower data loss, but as the system rejects about one fifth of user requests the price seems too high.

Table 5.9 and Table 5.10 provide comparison between two policies that both use online full backup only. The difference lies in whether the backup process has priority over the Apache process. Comparing the last four columns, we can see that priority for the backup process is not really beneficial, causing heavy request rejection while providing marginal reduction in data loss. Therefore, if only full backups are used, backup priority is not necessary. We may also observe from a comparison between Table 5.8 and Table 5.10 that the introduction of online backup largely removes the negative effect of excessive backups, especially when no priority is given. This is understandable since user accesses are allowed during online backups.

Table 5.9. Online full backup only with priority

Schedule	File Service Availability	Data Loss Rate (sec^{-1})	Data Loss Ratio	Request Rejection Rate (sec^{-1})	Request Rejection Ratio
Once per week	0.9993	0.0013	4.7937e-4	0.1569	0.0173
Once per day	0.9993	1.8078e-4	7.4964e-5	1.0578	0.1166

Table 5.10. Online full backup only without priority

Schedule	File Service Availability	Data Loss Rate (sec^{-1})	Data Loss Ratio	Request Rejection Rate (sec^{-1})	Request Rejection Ratio
Once per week	0.9993	0.0013	4.7257e-4	0.0067	7.3959e-4
Once per day	0.9993	1.7769e-4	6.5139e-5	0.0067	7.3959e-4

Before presenting further results concerning partial backups, it is instructive to give some results regarding the accuracy of the model. The following Table 5.11 shows the comparison of model outputs with simulation results for the full-backup-only scenarios shown in previous tables. The values in the brackets define the 90% confidence intervals.

Table 5.11. Accuracy Results

	Schedule	File Service Availability		Data Loss Rate (sec^{-1})		Rejection Rate (sec^{-1})	
		Model	Simu.	Model	Simu.	Model	Simu.
Offline	Once per week	0.9631	0.9612 [0.9252, 0.9987]	0.0013	0.0013 [0.0012, 0.0013]	0.2842	0.3084 [0.2832, 0.3210]
	Once per day	0.7567	0.7495 [0.6923, 0.8042]	1.81E-04	1.79E-04 [1.76E-04, 1.82E-04]	1.9493	1.9620 [1.7205, 2.1472]
Online, priority	Once per week	0.9993	0.9883 [0.9602, 1.0108]	0.0013	0.0013 [0.0012, 0.0013]	0.1569	0.1824 [0.1602, 0.2035]
	Once per day	0.9993	0.9950 [0.9647, 1.0221]	1.81E-04	1.79E-04 [1.76E-04, 1.81E-04]	1.0578	1.2280 [1.1326, 1.3040]
Online, no priority	Once per week	0.9993	0.9882 [0.9602, 1.0107]	0.0013	0.0012 [0.0011, 0.0013]	0.0067	0.0075 [0.0068, 0.0083]
	Once per day	0.9993	0.9961 [0.9643, 1.0218]	1.78E-04	1.65E-04 [1.59E-04, 1.71E-04]	0.0067	0.0071 [0.0063, 0.0078]

As the accuracy results show, most results are reasonably accurate, especially the availability parts. When no backup execution priority is present, the model results generally agree with those from the simulation. On the other hand, once backup priority is in place, the results differ more significantly, especially for the rejection rate which is (relatively) heavily affected by the resource contention between backup and normal workloads. This indicates the approximation of the priority queueing network is not fully satisfactory. However it may remain useful as a rough estimate.

Next I consider the case where partial backups are also used. In order to keep the presentation compact, I do not show the results for every policy, but first compare the schedules while fixing the configurations. Then I pick one of the best policies to show the effect of varying configurations. The fix configuration I use is as follows:

- full backup with incremental partial backup
- online backup
- backup has priority over the Apache process

With the above configuration, I compare the results of different backup schedules in Table 5.12. From Table 5.12 it can be observed that the frequency of full backups has the greatest effect on file service availability due to their long durations. By varying the frequency of partial backups, a trend of tradeoff between data loss and request rejection can be observed. More frequent backups reduce the chance that a storage failure destroys data updates, but also increase request rejection due to the consumption of additional system resources. However, the tradeoff is not significant in either direction, and which policy is best will probably be decided on a case-by-case basis. If we compare the last row of Table 5.12 with that of Table 5.8, we can see the shorter durations of partial backups allow them to be applied more frequently without incurring the cost of high user request rejection.

Table 5.12. Schedule comparison with partial backup

Schedule	File Service Availability	Data Loss Rate (sec^{-1})	Loss Ratio	Request Rejection Rate (sec^{-1})	Rejection Ratio
1 full every two weeks, 1 partial every two days	0.9662	0.0073	0.0027	0.3385	0.0187
1 full and 3 partial per week	0.9450	0.0066	0.0025	0.2443	0.0269
1 full and 6 partial per week	0.9373	0.0036	0.0014	0.3069	0.0338

Next I investigate the effect of different backup techniques. For brevity, I fix the schedule to be one full backup per week with six partial backups in between (i.e., one per day). The results are given in Table 5.13.

Table 5.13. Configuration comparison with partial backup

Configuration	File Service Availability	Data Loss Rate (sec^{-1})	Loss Ratio	Request Rejection Rate (sec^{-1})	Rejection Ratio
Differential, offline	0.9135	0.0036	0.0014	0.6772	0.0746
Differential, online, no priority		0.0037	0.0013	0.0066	7.2288e-4
Differential, online, priority		0.0036	0.0014	0.3694	0.0407
Incremental, offline	0.9373	0.0036	0.0014	0.5616	0.0619
Incremental, online, no priority		0.0037	0.0013	0.0066	7.2421e-4
Incremental, online, priority		0.0036	0.0014	0.3069	0.0338

The first three columns in Table 5.13 indicate that with a fixed schedule the file service availabilities and data loss are comparable under different configurations. However, the request rejection metrics are affected to a larger extent by the choice of online and offline backups and whether the backup process has priority. Online backup without priority is clearly superior according to the results. On the other hand, the effect of differential and incremental backup is relatively minor. An interesting observation is that the incremental backup seems to be affected less by priority, although the non-priority case is still superior.

Based on the results presented in the above tables, the following conclusions be drawn:

1) A mixed backup policy combining full and partial backups is more effective than an all-full backup policy. This corresponds to the observations in [88].

2) Online backups are better than offline backups in terms of request rejection. This is easy to understand since user accesses are allowed in an online backup. Online backups do have higher data loss, but the amount is small and may have been caused by the fact that, since more requests are accepted there are more to lose in

the first place. Whether online backup is worthwhile should probably be determined case-by-case, but from the results presented it would be the general choice.

3) Priority for the backup process is not effective. It usually raises data rejection heavily without significant decrease in data loss. This implies that under the current system parameters the backup process is short enough and completes relatively quickly even with resource contention from the Apache process. However, problems could arise if the competition from the Apache process is heavy enough so that the backup process cannot be completed within the after-hours period and thus extends into the office period. In that case the presence of the delayed backup process could lead to greater rejection of user requests since the Apache process may have insufficient resource to handle the heavier workload in the office period.
4) It does not produce significant differences whether the partial backup is performed using differential backup or incremental backup, so the relative merit of the two techniques would probably need to be determined by the requirement of specific applications. This may also be a result of the parameter setting, where the backup and restoration rates of the two approaches do not differ significantly.

5.5 Related Works

There exist many commercial solutions for data backup [94][95][96][97], signifying the importance of this issue in practice. Meanwhile, research has been done [98][99][100][101] on designing a backup system to minimize the side effects of backup while providing adequate data protection. In [98], the authors present a backup scheduling approach, based on integer programming, to minimize the overall backup time. In [99], the authors propose a framework for automating the design of disaster protection, where dependability goals are translated into monetary terms based on which an optimal protection technique is selected. This framework is further extended in [100], where techniques are developed to evaluate the dependability impact from multiple design choices and their combined effect on a storage system. In [101], further extension is made by considering the presence of multiple applications that may have different optimal dependability design choices.

The large amount of data in today's IT environment may consume much storage as well as make backup inefficient. One solution is data de-duplication which identifies and eliminates redundant data content. [102] applies this technique to reduce the volume of backup data and the backup time needed. [103] looks into the effect of de-duplication on data reliability, since de-duplication may amplify the effect of individual block loss.

Data backup may also be conducted in a distributed fashion by spreading the data onto multiple storage systems. Following this approach, [104] presents a distributed backup strategy to improve backup efficiency by considering the network bandwidth between storages systems. Approaching the issue from another angle, the authors of [105] propose a new architecture to develop distributed fault-tolerant backup system. The paper focuses on formally establishing the fault-tolerance of the system.

In contrast to the aforementioned papers, this chapter focuses on stochastic modeling of a system with periodic data backups. While some earlier papers (e.g., [99][100][101]) provide frameworks to analyze storage system dependability and design optimization schema, my study instead focuses on the impacts of different data backup policies and extends the research scope from system availability to performance. The analytic models presented in this chapter are more detailed and comprehensive, and account for both backup and application processes. In addition, I propose five availability and performance metrics that describe the key aspects of the application service as well as reflect the effect of backup operations. As demonstrated by the investigation of system metrics under different backup policies, system configurations and user workloads, I provide an effective way to evaluate data backup policies under different environments. The models developed can form the basis of a design and management tool for IT systems with periodic data backup.

5.6 Conclusion and Future Work

In this chapter I presented a stochastic modeling framework to evaluate the availability and performance of a storage system with periodic data backup. The extended models capture the effects of different backup policies on system metrics under a variety of system configurations and workloads. The models and formulas that I developed extend existing work on storage backup modeling [88] and provide a basis for future work on more detailed and comprehensive models for storage system operations.

The current models were developed based on a variety of assumptions on system operation scenarios and parameter settings. Although I believe most of the assumptions adopted are reasonable, I will seek to further improve confidence in model accuracy. For this purpose, I would strive to perform model validation using either simulation and/or experimental data in the future, and obtain information about configurations and operational patterns of systems deployed in practice.

6. A Scalable Optimization Framework for Storage Backup Operation Planning[1]

6.1 Overview

The ubiquity of modern information technologies produces increasingly large amount of data, and companies often need to handle and store voluminous data from both its own operation and from users of its service. Businesses are also capitalizing on the abundance of data and increasingly relying on data analytics to provide insight and guide decision-making. The growth in data volume and its importance drives businesses to quickly expand storage infrastructure and seek effective ways to manage the larger and more complex storage system. However, it remains an unfortunate fact that important data can be rendered unusable in many ways [106]. Files may be corrupted through unintentional modification or malicious behaviors. Data may become inaccessible as a result of storage system component failures. And natural disasters, while less common, have the potential to outright destroy the physical infrastructure which stores the data. Consequences from such data loss range from reduced productivity and capability to conduct normal operation, to significant financial losses, to damaged business reputation and loss of customer confidence [107]. Thus the importance of data in today's business world mandates greater effort in protecting against such undesired events.

The importance of effective data protection techniques is well recognized, and businesses today can choose from many data protection technologies. While new methods such as data replication across geographically distributed sites are gaining popularity, the classic data

[1] This chapter is based on the following publication:
Optimizing Replicated Storage Backup Operation using Markov Decision Process. Ruofan Xia, Fumio Machida, and Kishor Trivedi. Accepted by the 21st IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2015).

backup techniques still have a significant role to play. Data backup works by creating data copies that no longer change with and are (typically) separated from the original copy and stored securely. It consequently provides a frozen record of the data at a given time, which is important for the purpose of data archiving. Moreover, since data replication itself does not tolerate data loss caused by operational errors or malicious activities, it is important to take data backup periodically. The appropriate frequency of data backup depends on the users' business requirements that specify the acceptable damage due to data loss. Two frequently used requirements are Recovery Point Objective (RPO) and Recovery Time Objective (RTO) [106]. For a given dataset (e.g., a collection of database records), RPO defines a time point in system history so that a recovery procedure must be able to recover the system status up to that point. On the other hand, RTO specifies the maximum length of the recovery period that is acceptable. These requirements serve as a major part of the design and operational guidelines for the data sets. To meet such requirements, execution of backup operations requires careful planning which can be a challenging task. The reason for this difficulty is two-fold. First, data backups typically have associated cost in terms of downtime and/or performance overhead. Backup execution may affect normal system operation, by suspending data access (for instance to maintain data consistency) or contending with production workload execution. Second, data backup must take into consideration various other factors such as data priorities and the available system resources for backup, which may be limited.

In this chapter I investigate the aforementioned issue of effective data backup operation and seek specifically to address the following question: how should data backups be performed so that the protection requirements (in terms of RPO and RTO) are satisfied while backup operations yield minimal impact on normal storage system operation? Specifically I start by constructing an optimization framework based on Markov decision processes [16], which casts the question as an instance of constrained optimization problem. For a given data storage system, the framework takes into account the protection requirements and failure/recovery behaviors of different data sets, as well as the available system resource for backup execution. It provides an optimal backup operation plan that minimizes the system downtime caused by backup execution and data loss recovery while satisfying the protection requirements. However, the basic framework is prone to a scalability issue that is common to many practical applications of MDP. To deal with this issue, I devise a simple yet quite effective approximation method. The approximation method decomposes the overall constrained optimization problem into smaller ones that can be solved independently, yielding significant saving in terms of solution complexity and allowing much larger problems to be investigated. To demonstrate the utility of the optimization framework and the effectiveness of the decomposition method, I also provide comprehensive numerical investigation results comparing the planning produced by the framework to multiple heuristics in this chapter.

This chapter is organized as follows. Section 6.2 details the system scenario under investigation. Section 6.3introduces the basic MDP-based framework in details, while Section 6.4 discusses the scalability issue of the basic approach and proposes the approximation method to handle the issue. Section 6.5 provides results of numerical investigation and relevant discussions. Section 6.6 discusses related work. Finally, Section 6.7 concludes this chapter with possible directions for future work.

6.2 System Scenario

Consider a storage system maintaining multiple data sets (e.g., collections of files that are managed together, databases, etc.). In normal operation, the data sets accept changes to the data resulting from user operations. The system is expected to store such data updates securely. However, data sets may experience failures (caused by, e.g., software bugs in the program managing the data or failure of hardware components hosting data) during the operation which can lead to loss of the accumulated user updates to the data. In addition to such losses, data set failures and the subsequent recovery also introduce additional downtime for data access. In order to protect against data loss, there are periodic backup points during system operation when each data set may perform backup. I assume the data sets accumulate data, fail and recover independently, and that there is no common failure mode that affects multiple data sets at the same time. I also assume that a data set is unavailable for access during backup or recovery executions.

I assume that a decision must be made at each backup point regarding whether to perform backup operation for a given data set, and thus I refer to these points as 'decision points', while the period between two neighboring decision points is the 'decision interval'. Thus at each decision point, a data set may execute backup or skip this backup opportunity and continue normal operation. It may also happen that a data set is recovering from a previous failure at the beginning of the backup window. In that case I assume: 1) the data set can still be scheduled for backup; 2) the data set will recover before the next decision point, and 3) if the data set is scheduled for a backup, the backup will execute after the recovery and complete before the next decision point. As a simplification I assume backup operations do not fail, and when a backup completes all the data processed is no longer subject to loss. An illustration of the system with two data sets is given in Figure 6.1, where the texts above the arrows describe the backup choice (whether to backup and what type of backup to perform) for each data set at the decision points.

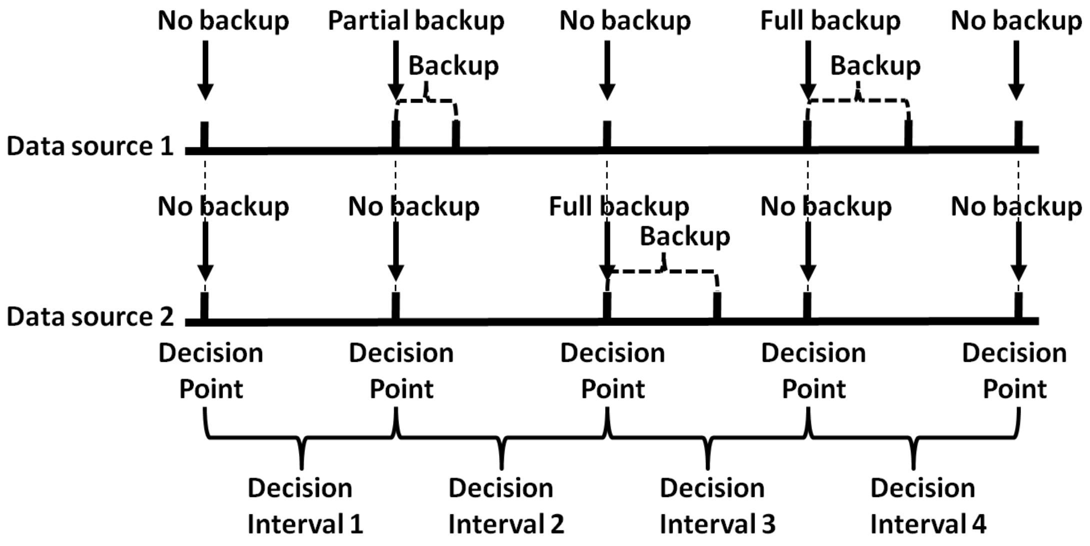


Figure 6.1. Example of Data Backup Schedule for a System with Two Data Sets

If a data set executes backup, it can be either a full backup or a partial (incremental) backup. A full backup processes all the data and produces a copy from which alone the data set can be recovered (to the status when the backup was made), while a partial backup covers the data changes since the last backup and would need the nearest full backup in the past, in addition to all the partial backups in between, to restore data to the moment when the partial backup was made. Thus in my context a completed full backup removes all the accumulated partial backups since the latter are no longer needed for recovery. Execution of a backup also requires some system resource (e.g., network bandwidth to transfer the processed data), and there may not be enough resource to accommodate concurrent backup execution for all data sets. This fact may require some data sets to backup more/less frequently, and serves as both the major connection among the data sets (recall that I assume they behave largely independently of each other) and the primary factor that makes backup planning nontrivial. In this chapter I consider a single type of shared resource, the backup network bandwidth, but other shared resources can also be easily incorporated by adding similar constraints in the formulation.

I assume that a full backup takes more time than a partial backup. If a partial backup is performed after several intervals since the last backup, there will be more data changes to process, and the backup time would also be longer. On the other hand, increased data change accumulation also increases the chance that some data changes would overwrite earlier changes and reduce the net amount of changes to process. To account for such effect, I assume the expected time to perform a partial backup can be described with the following functional form:

$$ET_{PB} = ET_{FB}[1 - e^{-(\alpha n_{skipped})}] \tag{6.1}$$

In the above form, ET_{PB} and ET_{FB} are the expected time to carry out a partial and full backup respectively. The term $n_{skipped}$ is the number of backup epochs a data set has skipped since the last full backup execution, and α is a positive data set-dependent parameter that describes how likely later data changes overwrite earlier ones. In this chapter I use this form for its simplicity, but would also like to note that more complex forms can be easily incorporated.

For recovery, I assume each recovery of a data set takes firstly a certain amount of time to restore the previous full backup copy, and then some additional time to restore the copies of partial backups that have been taken since that full backup, up to the moment of the current failure. The second time quantity is assumed to be proportional to the number of partial backups involved, and may be zero if the closest backup in the past was a full backup.

As described earlier, each data set has its requirements for protection in terms of RPO and RTO. In my context, the RPO defines the maximum number of backups that a data set may consecutively skip, since if a data loss occurs the system cannot recover the status of data at a skipped backup point. As for RTO, it defines how many partial backups a data set may accumulate before a full backup should take place, as each additional partial backup to process during recovery increases the overall recovery time. I would like to note that these two definitions used here are specializations of those given earlier, in that RPO specifies a point in time to recover to while RTO specifies how long the recovery process is allowed to be. I assume that the operational goal of the system is to minimize expected system downtime while satisfying the RPO and RTO requirements of all the data sets.

Because data sets may fail, the backup planning should take into account their failure and recovery behavior. Since the focus of this chapter is on the MDP-based framework, I assume a simple availability model (for each data set, since I consider them to fail and recover independently). The model is a semi-Markov chain with an up state ("U"), a backup execution state ("B") and a down state ("D") as shown in Figure 6.2.

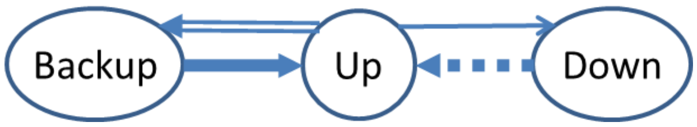


Figure 6.2. Availability Model

In Figure 6.2I assume the following firing time distributions for the transitions:

1) The firing time of the failure transition (the thin arrow from "Up" to "Down") is exponentially distributed.

2) The firing time of the recovery transition (the dashed arrow from “Down” to “Up”) is generally distributed with bounded supports, and the upper bound is smaller than the time between two backup windows.
3) The firing time of the data backup initiation transition (the hollow arrow from “Up” to “Backup”) is deterministic to model data backup schedules.
4) The firing time of the backup completion transition (the thick arrow from “Backup” to “Up”) is generally distributed with bounded supports, and the upper bound is smaller than the time between two backup windows.

I assume bounded supports for the backup and recovery completion times because in practice it is generally a priority for the system administration to complete backup and recovery within a given timeframe. The data set in question is considered unavailable in both the down and backup states. The failure rate is given and shared by the availability model and the MDP model, while the parameter values for the non-exponential distributions are determined by the backup plan generated by the MDP. More specifically, the time of backup initiation is given by the expected time between two backup executions (full or partial), and that of the backup completion is given by the expected completion time of a backup. Since in the model of Figure 6.2 I do not distinguish between full and partial backup states, the two parameters mentioned above will be computed from the weighted average across all backup execution times, full or partial, with the weights given as the stationary probability of the states where a full or partial backup is performed. Similarly, the completion time of a recovery operation is also obtained from the average of all recovery times with respect to the stationary state probabilities.

6.3 MDP-based Backup Planning Framework

In this section I first introduce the basic MDP-based planning framework and show how a storage system of the type discussed in the previous section is cast as an instance of MDP. Then I will discuss the scalability limitation of this basic framework, and present an approximate decomposition scheme to address this issue.

I will start by considering the MDP elements (described in Chapter 2) and discuss their construction in turn. The state space of the MDP is constructed from the status of the data sets. I denote the status of data set *i* as (A_i, X_i, Y_i) where A_i represents the failure status of data set *i*, X_i is the number of decision intervals since the last backup of data set *i*, and Y_i is the number of partial backups taken since the last full backup. The variable A_i is either *U* or *D* according to the availability model in Figure 6.2. Note that this variable will not take a value of *B*, to indicate backup, since that is captured by the actions of MDP which will be described shortly. The second variable is the number of successive backup points that the data set has skipped. Thus the assumption is that a data set accumulates data changes at a uniform rate so long as it is operational. The maximum values of X_i and Y_i are specified by the values of RPO_i and RTO_i respectively, which are the RPO and RTO requirements of data set *i*.

The action of the MDP is constructed from the backup choices of each data set at a backup point. Each data set normally has three choices: skip this backup and continue normal operation, take a partial backup, or take a full backup. The RPO and RTO requirements dictate whether a given choice is possible under the current state for that data set. For example, if a data set has reached its maximum allowed number of skipped backups then it must perform a backup, and if it has also reached its maximum number of partial backups then a full backup is the only choice. A small MDP example, composed of one data set with an RPO of 2 and RTO of 2, is shown in Figure 6.3to illustrate the above points. Notice that while in general three actions are available in a state, in some states the set of actions becomes more restricted. For example, in (U, 2, 0) a backup operation must be taken (so skipping is not an option), while in (U, 2, 2) only full backup is possible.

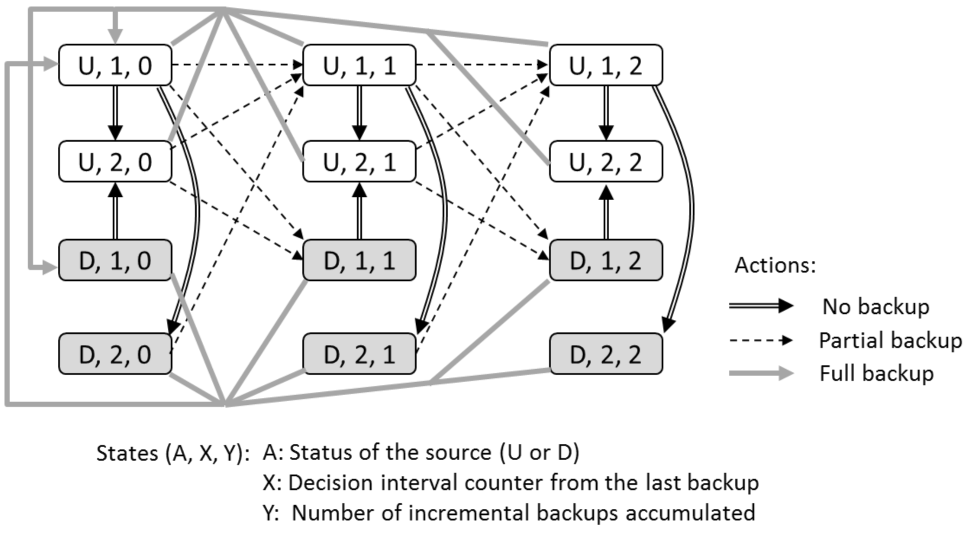

U, 1, 0
U, 1, 1
U, 1, 2
U, 2, 0
U, 2, 1
U, 2, 2
D, 1, 0
D, 1, 1
D, 1, 2
D, 2, 0
D, 2, 1
D, 2, 2
Actions:
No backup
Partial backup
Full backup
States (A, X, Y): A: Status of the source (U or D)
X: Decision interval counter from the last backup
Y: Number of incremental backups accumulated

Figure 6.3. A Simple MDP Example

Since an MDP state is composed of the states of all the data sets, the possible actions in a given MDP state are also constructed as the combination of individual data set actions. For example, if a system consists of two data sets, Src_1 and Src_2, which take action A_1 and A_2 in state S_1 and S_2 respectively, then the corresponding MDP action in state (S_1, S_2) would be (A_1, A_2). One additional constraint put on the MDP actions is that the number of data sets executing backup in one decision point cannot exceed the capacity of system backup resource. To simplify discussion, I assume a backup operation consumes one unit of resource regardless of the set executing it. This can be easily relaxed by allowing some backup executions to consume more.

In a similar fashion, the transition probabilities and rewards of MDP actions are also constructed from those of each data set. The transition probabilities are obtained by multiplying the probabilities of entering corresponding new states from all data sets, while the reward is the average of those from all data sets, which is the discounted downtime that a given data set can expect to incur during its operation. This is a result of the assumption that the data sets behave independently (aside from the resource constraint). In the previous simple example, if Src_1 goes into state S_{1a} and S_{1b} with probabilities P_{1a} and P_{1b} and rewards R_{1a} and R_{1b} under action A_1, while Src_2 goes into state S_{2a} and S_{2b} with probabilities P_{2a} and P_{2b} and rewards R_{2a} and R_{2b} under action A_2, then the transition probabilities and rewards of the MDP by choosing action (A_1, A_2) at state (S_1, S_2) would be:

$$P(S_{1a}, S_{2a}|S_1, S_2) = P_{1a}P_{2a} \qquad R(S_{1a}, S_{2a}|S_1, S_2)=(R_{1a}+R_{2a})/2$$

$$P(S_{1b}, S_{2a}|S_1, S_2) = P_{1b}P_{2a} \qquad R(S_{1b}, S_{2a}|S_1, S_2)=(R_{1b}+R_{2a})/2$$

$$P(S_{1a}, S_{2b}|S_1, S_2) = P_{1a}P_{2b} \qquad R(S_{1a}, S_{2b}|S_1, S_2)=(R_{1a}+R_{2b})/2$$

$$P(S_{1b}, S_{2b}|S_1, S_2) = P_{1b}P_{2b} \qquad R(S_{1b}, S_{2b}|S_1, S_2)=(R_{1b}+R_{2b})/2$$

The exact formulas for computing the transition probabilities and rewards for each action of a data set are given next. In the following formulas, X is a positive integer, Y is a nonnegative integer, and the terms t, γ and τ are the time variables of different probability density functions. T stands for the length of the decision interval, Re stands for reward, and E(-) stands for expectation. $R(t)$ is the reliability of the data source, i.e. the probability that the data source does not fail within time t. The terms $f(t)$ and $g(t)$ stand for the density functions of time-to-failure and time-to-recover, while U_r and L_r represent the upper and lower bound of the time-to-recover which is assumed to be a uniformly distribution random variable to give a concrete example. For each case I will give the expressions and derivations for the transition probability and expected reward of that transition. The expected reward of the action can then be obtained simply by summing up all the transition expected reward values.

1) If the current status is (*U*, *X*, *Y*) and no backup is performed, the possible new status are (*D*, *X+1*, *Y*) and (*U*, *X+1*, *Y*), which correspond, respectively, to the cases of the data source being down or up at the next decision point. The former happens if the data source does not recover from a failure that occurs within the current decision interval, while the latter happens if either there is no failure in the interval, or if there is one failure but the system recovers before the next decision point. The respective probabilities are computed via an exponential distribution with the same rate parameter as in the availability model (e.g., Figure 6.2). For the reward, the downtime is the expected time to recovery from a failure. The exact formulas are presented below.

   - New status: (U, X+1, Y), if no failure occurs or when the source recovers from one before the next decision point.

$$Prob = R(T) + Prob1$$

$$= R(T) + \{\int_0^{T-U_r} f(t) \int_{L_r}^{U_r} g(\tau) R(T-t-\tau) d\tau\, dt$$

$$+ \int_{T-U_r}^{T-L_r} f(t) \int_{L_r}^{T-t} g(\tau) R(T-t-\tau) d\tau\, dt\} \qquad (6.2)$$

$$= e^{-\lambda T} + \int_0^{T-U_r} \lambda e^{-\lambda t} \int_{L_r}^{U_r} \frac{1}{U_r - L_r} e^{-\lambda(T-t-\tau)} d\tau\, dt$$

$$+ \int_{T-U_r}^{T-L_r} \lambda e^{-\lambda t} \int_{L_r}^{T-t} \frac{1}{U_r - L_r} e^{-\lambda(T-t-\tau)} d\tau\, dt$$

$$= e^{-\lambda T} + \int_0^{T-U_r} \frac{e^{-\lambda T}}{U_r - L_r}\left[e^{\lambda U_r} - e^{\lambda L_r}\right]dt$$

$$+ \int_{T-U_r}^{T-L_r} \frac{e^{-\lambda T}}{U_r - L_r}[e^{\lambda(T-t)} - e^{\lambda L_r}]dt$$

$$= e^{-\lambda T} + \frac{(\lambda T - \lambda U_r + 1)e^{-\lambda T}}{\lambda(U_r - L_r)}\left[e^{\lambda U_r} - e^{\lambda L_r}\right] - e^{-\lambda(T-L_r)}$$

$$E[Re] = E[g(t)] \times Prob1 = \frac{U_r + L_r}{2} \times Prob1$$

- New status: (*D*, *X+1*, *Y*), if a failure occurs and the source has not recovered by the next decision point.

$$Prob = \int_{T-U_r}^{T-L_r} f(t) \int_{T-t}^{U_r} g(\tau)d\tau\, dt + \int_{T-L_r}^{T} f(t)dt$$

$$= \int_{T-U_r}^{T-L_r} \lambda e^{-\lambda t} \int_{T-t}^{U_r} \frac{1}{U_r - L_r} d\tau\, dt + \int_{T-L_r}^{T} \lambda e^{-\lambda t} dt$$

$$= \int_{T-U_r}^{T-L_r} \lambda e^{-\lambda t} \frac{U_r - T + t}{U_r - L_r} dt + e^{-\lambda T}(e^{\lambda L_r} - 1)$$

$$= \frac{U_r - T}{U_r - L_r} \int_{T-U_r}^{T-L_r} \lambda e^{-\lambda t} dt + \int_{T-U_r}^{T-L_r} \frac{\lambda t e^{-\lambda t}}{U_r - L_r} dt + e^{-\lambda T}(e^{\lambda L_r} - 1) \tag{6.3}$$

$$= \frac{U_r - T}{U_r - L_r} e^{-\lambda T}\left[e^{\lambda U_r} - e^{\lambda L_r}\right] + \frac{e^{-\lambda T}}{U_r - L_r}[\left(T - U_r + \frac{1}{\lambda}\right) e^{\lambda U_r}$$

$$-(T - L_r + \frac{1}{\lambda})e^{\lambda L_r}] + e^{-\lambda T}(e^{\lambda L_r} - 1)$$

$$E[Re] = E[g(t)] \times Prob = \frac{U_r + L_r}{2} \times Prob$$

The term *Prob1* is the probability that the data source recovers from a failure within *T*, and the terms within the two integrations in (6.2) are to the probabilities that the source recovers before the next decision point conditioned on different instant of failure occurrence. The integrations then uncondition to yield the total probability of source recovering timely. Equation (6.3) uses a similar approach, changing the terms within the integrations to be the conditional probabilities that the source does not recover. Finally, the *Prob* terms in (6.2) and (6.3) are normalized by their sum to reflect the assumption that at most one failure can occur in an interval.

2) If the current status is (*D*, *X*, *Y*) and no backup is performed, due to the assumption of at most one failure within a decision interval, there will be no other failure before the next decision point. Thus the only possible new status is (*U*, 1, *Y*) with transition probability one, while the expected downtime will both be zero (the amount of time required to recover has been included in the downtime of the backup choice that the data source selected at the previous decision point).

3) If the current status is (*U, X, Y*) and a backup (full or partial) is scheduled, there are four potential new statuses, (*U, 1, Y+1*), (*D, 1, Y+1*), (*U, 1, 0*) and (*D, 1, 0*). The first two correspond to the cases where partial backup is performed, while the latter two occurs when full backup is executed. The first and third cover the cases where the data source is operational at the next decision point, regardless of whether a failure occurs. The other two cases correspond to the situation that the data source has not recovered from a failure by the next decision point. The transition probabilities and expected reward terms are defined as below. Most of the other terms in (6.4) and (6.5) have identical meanings to those in (6.2) and (6.3) earlier. Of the new ones, *b(t)* stands for the density function of time-to-backup-completion, which is assumed to

be uniform, while U_b and L_b stand for its upper and lower bounds. The method of derivation is also similar to those for (6.2) and (6.3), in that (where applicable) I first condition on the duration of backup, then on the occurrence time of failure, and finally on the time of recovery execution.

- New status: (*U, 1, Y+1*) or (*U, 1, 0*), a partial/full backup is executed and the source is up at the next decision point.

$$Prob = Prob1 + Prob2 + Prob3$$

$$Prob1 = \int_{L_b}^{U_b} b(t)R(T-t)dt = \int_{L_b}^{U_b} \frac{1}{U_b - L_b} e^{-\lambda(T-t)}dt$$

$$= \frac{e^{-\lambda T}}{\lambda(U_b - L_b)}\left(e^{\lambda U_b} - e^{\lambda L_b}\right)$$

$$Prob2 = \int_{L_b}^{U_b} b(t)\left[\int_0^{T-t-U_r} f(\tau)\int_{L_r}^{U_r} g(\gamma)R(T-t-\tau-\gamma)d\gamma\, d\tau\right]dt$$

$$= \int_{L_b}^{U_b} \frac{1}{U_b - L_b}\int_0^{T-t-U_r} \lambda e^{-\lambda\tau}\int_{L_r}^{U_r} \frac{1}{U_r - L_r} e^{-\lambda(T-t-\tau-\gamma)}d\gamma\, d\tau\, dt$$

$$= \int_{L_b}^{U_b} \frac{1}{U_b - L_b}\frac{e^{-\lambda(T-t)}}{U_r - L_r}\int_0^{T-t-U_r}\left(e^{\lambda U_r} - e^{\lambda L_r}\right)d\tau\, dt \tag{6.4}$$

$$= \frac{e^{-\lambda T}(e^{\lambda U_r} - e^{\lambda L_r})}{(U_b - L_b)(U_r - L_r)}\int_{L_b}^{U_b} e^{\lambda t}(T - t - U_r)dt$$

$$= \frac{e^{-\lambda T}\left(e^{\lambda U_r} - e^{\lambda L_r}\right)}{\lambda(U_b - L_b)(U_r - L_r)}\cdot$$

$$\left[\left(L_b e^{\lambda L_b} - U_b e^{\lambda U_b}\right) + \left(T - U_r + \frac{1}{\lambda}\right)(e^{\lambda U_b} - e^{\lambda L_b})\right]$$

$$Prob3 = \int_{L_b}^{U_b} b(t)\int_{T-t-U_r}^{T-t-L_r} f(\tau)\int_{L_r}^{T-t-\tau} g(\gamma)R(T-t-\tau-\gamma)d\gamma\, d\tau\, dt$$

$$= \int_{L_b}^{U_b} \frac{1}{U_b - L_b} \int_{T-t-U_r}^{T-t-L_r} \lambda e^{-\lambda\tau} \int_{L_r}^{T-t-\tau} \frac{1}{U_r - L_r} e^{-\lambda(T-t-\tau-\gamma)} d\gamma\, d\tau\, dt$$

$$= \int_{L_b}^{U_b} \frac{1}{U_b - L_b} \frac{e^{-\lambda(T-t)}}{U_r - L_r} \int_{T-t-U_r}^{T-t-L_r} \int_{L_r}^{T-t-\tau} \lambda e^{\lambda\gamma} d\gamma\, d\tau\, dt$$

$$= \int_{L_b}^{U_b} \frac{1}{U_b - L_b} \frac{e^{-\lambda(T-t)}}{U_r - L_r} \int_{T-t-U_r}^{T-t-L_r} \left[e^{\lambda(T-t-\tau)} - e^{\lambda L_r}\right] d\tau\, dt$$

$$= \frac{e^{-\lambda T}}{\lambda(U_b - L_b)(U_r - L_r)} \cdot$$

$$\left[\frac{1}{\lambda}\left(e^{\lambda U_r} - e^{\lambda L_r}\right) - e^{\lambda L_r}(U_r - L_r)\right] \int_{L_b}^{U_b} \lambda e^{\lambda t} dt$$

$$= \frac{e^{-\lambda T}\left(e^{\lambda U_b} - e^{\lambda L_b}\right)}{\lambda(U_b - L_b)(U_r - L_r)} \left[\frac{1}{\lambda}\left(e^{\lambda U_r} - e^{\lambda L_r}\right) - e^{\lambda L_r}(U_r - L_r)\right]$$

$$E[Re] = E[b(t)] \times Prob + E[g(t)] \times (Prob2 + Prob3)$$

$$= \frac{U_b + L_b}{2} \times Prob + \frac{U_r + L_r}{2} \times (Prob2 + Prob3)$$

- New status: (*D, 1, Y+1*) or (*D, 1, 0*), a partial/full backup is executed and the source is down at the next decision point.

$$Prob = \int_{L_b}^{U_b} b(t)[\int_{T-t-U_r}^{T-t-L_r} f(\tau) \int_{T-t-\tau}^{U_r} g(\gamma) d\gamma\, d\tau + \int_{T-t-L_r}^{T-t} f(\tau) d\tau]\, dt$$

$$= \int_{L_b}^{U_b} \frac{1}{U_b - L_b} [\int_{T-t-U_r}^{T-t-L_r} \lambda e^{-\lambda\tau} \int_{T-t-\tau}^{U_r} \frac{1}{U_r - L_r} d\gamma\, d\tau$$

$$+ \int_{T-t-L_r}^{T-t} \lambda e^{-\lambda\tau} d\tau]\, dt \qquad (6.5)$$

$$= \int_{L_b}^{U_b} \frac{1}{U_b - L_b} [\int_{T-t-U_r}^{T-t-L_r} \lambda e^{-\lambda\tau} \frac{U_r - T + t + \tau}{U_r - L_r} d\tau$$

$$+ e^{-\lambda(T-t)}(e^{\lambda L_r} - 1)] dt$$

$$= \frac{1}{U_b - L_b} \int_{L_b}^{U_b} [\frac{U_r - T + t}{U_r - L_r} \int_{T-t-U_r}^{T-t-L_r} \lambda e^{-\lambda\tau} d\tau$$

$$+ \frac{1}{U_r - L_r} \int_{T-t-U_r}^{T-t-L_r} \lambda\tau e^{-\lambda\tau} d\tau + e^{-\lambda(T-t)}(e^{\lambda L_r} - 1)]dt$$

$$= \frac{e^{-\lambda T}}{U_b - L_b} \int_{L_b}^{U_b} e^{\lambda t} \{ \frac{U_r - T + t}{U_r - L_r} \left(e^{\lambda U_r} - e^{\lambda L_r}\right) + \left(e^{\lambda L_r} - 1\right)$$

$$+ \frac{1}{U_r - L_r} \left[\left(T - t - U_r + \frac{1}{\lambda}\right) e^{\lambda U_r} - \left(T - t - L_r + \frac{1}{\lambda}\right) e^{\lambda L_r} \right] \} dt$$

$$= \frac{e^{-\lambda T}}{U_b - L_b} \int_{L_b}^{U_b} e^{\lambda t} [- \frac{U_r - T + t}{U_r - L_r} e^{\lambda L_r} + \frac{e^{\lambda U_r}}{\lambda(U_r - L_r)}$$

$$- \frac{T - t - L_r}{U_r - L_r} e^{\lambda L_r} - \frac{e^{\lambda L_r}}{\lambda(U_r - L_r)} + \left(e^{\lambda L_r} - 1\right)]dt$$

$$= \frac{e^{-\lambda T}}{U_b - L_b} \int_{L_b}^{U_b} [- \frac{U_r - L_r}{U_r - L_r} e^{\lambda L_r} + \frac{e^{\lambda U_r} - e^{\lambda L_r}}{\lambda(U_r - L_r)} + \left(e^{\lambda L_r} - 1\right)] e^{\lambda t} dt$$

$$= \frac{e^{-\lambda T}}{U_b - L_b} \int_{L_b}^{U_b} [\frac{e^{\lambda U_r} - e^{\lambda L_r}}{\lambda(U_r - L_r)} - 1] e^{\lambda t} dt$$

$$= \frac{e^{-\lambda T}}{U_b - L_b} [\frac{e^{\lambda U_r} - e^{\lambda L_r}}{\lambda(U_r - L_r)} - 1] \frac{1}{\lambda} (e^{\lambda U_b} - e^{\lambda L_b})$$

$$= \frac{e^{-\lambda T}}{\lambda(U_b - L_b)} [\frac{e^{\lambda U_r} - e^{\lambda L_r}}{\lambda(U_r - L_r)} - 1] (e^{\lambda U_b} - e^{\lambda L_b})$$

$$E[Re] = \{E[b(t)] + E[g(t)]\} \times Prob = \left[\frac{U_b + L_b + U_r + L_r}{2}\right] \times Prob$$

In (6.4), *Prob1*, *Prob2* and *Prob3* stand for the probability of no failure after backup, one failure with the source re-covering timely, and one failure with the source not recovering. The first probability term covers the case of no failure after the backup, while the other two together express the probability of a failure after backup and the later successful recovery of the source. Similarly, (6.5) computes the probability that a failure occurs in the following decision interval from which the source does not recover by the next decision point.

4) If the current status is (*D, X, Y*) and a backup (full or partial) is scheduled, then the new status would be either (*U, 1, Y+1*) or (*U, 1, 0*) depending on the type of backup. Due to the assumption of at most one failure, the second variable cannot be zero at the next decision point. In both cases, the probability would be one and expected downtime is the expected backup execution time, as below.

   - New status: (*U, 1, Y+1*) or (*U, 1, 0*), a partial/full backup is executed after the source recovers from the failure.

$$Prob = 1$$

$$E[Re] = E[b(t)] = \frac{U_b + L_b}{2} \tag{6.6}$$

The transition probabilities and expected rewards of the MDP actions are constructed from those of the data sources, the former through multiplication and the latter through averaging. The state-space and actions, with the rewards and transition probabilities, complete the specification of the MDP instance. Based on these elements, an MDP can be constructed following the procedure below.

1) Construct the MDP state-space from the combination of states of all data sets.
2) Generate a reachability graph from the states, where a state can reach another state if there is a valid action (i.e., one that does not exceed the resource constraint) that causes the system to transition from the former state to the latter in the case of no failure occurring.

3) Prune the state space: do a depth-first search through the reachability graph. Any state that cannot reach itself is a state on a path leading to RPO and/or RTO violation. All such states are removed from the MDP state-space.

4) Construct the transition and reward matrices of the MDP. Check each state for possible actions and decide the next-states, corresponding transition probabilities and expected action rewards.

The final part of the framework is to specify how the expected system downtime is computed. To compute this quantity from the availability model it is necessary to obtain the values for the expected time between backup executions, the expected time to complete a backup execution as well as the expected time to complete a recovery. These can all be obtained from the MDP solution. Specifically, because the MDP reduces to a Markov chain under a given plan (e.g. the optimal one), from the MDP plan it is possible to obtain the expected number of skipped backups and partial backup copies by solving the resulting Markov chain. These values can be used to determine the completion time parameters (e.g., using (6.1)) of recovery and backup. Similarly, the expected time to the next backup execution can also be obtained from the expected number of skipped backups. Figure 6.4 shows the flow of such computation. Aggregation of the expected down times from all data sets yields the storage system downtime, which is computed as the average of the sum of data set downtimes.

6.4 Scalable Planning Framework

The MDP framework described so far is relatively simple to implement. However, it suffers from a scalability issue that significantly limits its utility. The issue arises because the MDP state and action spaces are constructed from the Cartesian products of all data set state and action spaces respectively. This results in an exponential growth in the MDP state and action space sizes with respect to the number of data sets, and thus restricts application of the framework to fairly small systems.

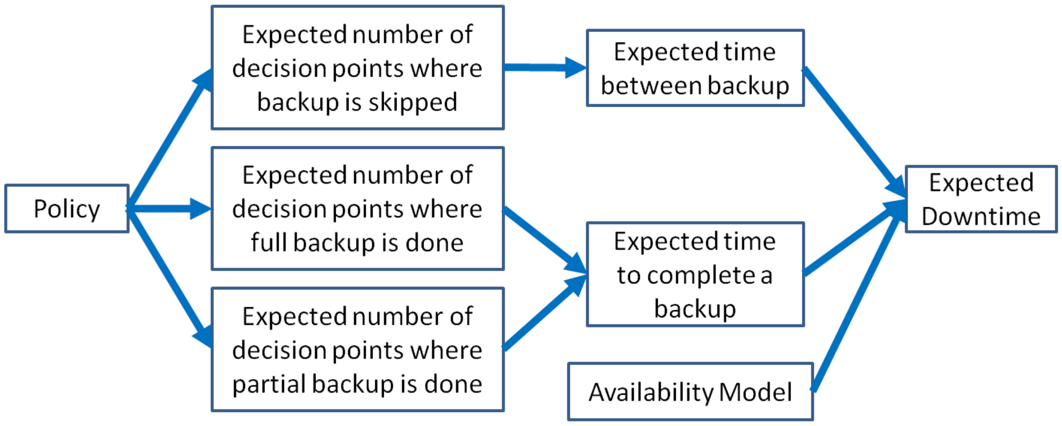

Policy
Expected number of decision points where backup is skipped
Expected number of decision points where full backup is done
Expected number of decision points where partial backup is done
Expected time between backup
Expected time to complete a backup
Availability Model
Expected Downtime

Figure 6.4. Data Set Downtime Computation

Recall that I assume that during normal operation the data sets behave independently. The data sets would only become coupled, in the sense that their behavior become dependent, at the moments of making backup decisions, and the coupling arises solely due to the limit in system backup resource, which potentially forces a data set to account for the states of its peers when making its decision. This is an instance of the classic problem of “weakly coupled MDP” [108], which arises frequently in practice but is in general difficult to solve exactly. Hence most existing research in dealing with this type of problem relies on approximation or heuristic methods that exploit problem structure specific to the context being handled. Some examples include [109] and [110]. Unfortunately, the techniques described therein are not easily applicable to the scenario in this chapter as I will briefly discuss in Section 6. Here I instead describe a simpler alternative approach that also relies on approximation.

Similar to some existing methods, the central idea of my approach is to decompose the MDP into (much) smaller ones for each data set. This would transform the original exponential problem size into a linear one (in terms of the number of data sets) and improve scalability. The key issue is how to account for the coupling among individual data set backup plans. In my context, this amounts to removing the resource constraints so that the backup of each data set can be planned by a separate MDP, while making sure the resource constraint would still be satisfied if each data set acts according to its own plan.

To achieve this I first consider the consequence of such coupling not being accounted for. If data sets perform backups while disregarding each other's action, then it may happen that the number of simultaneous backup operations exceed system resource limit, and some data sets would have to postpone their backup execution to the next opportunity. This has the potential consequence of breaching the RPO requirements of some data sets. Intuitively, the likelihood for such a situation to occur increases when more data sets have skipped multiple backups and/or are closer to their RPO limit. Thus if we consider the "undesirability" of a state in the MDP of a given data set, intuitively speaking a state closer to the RPO limit is more "undesirable". It is possible to express such "undesirability" by introducing additional penalty terms into each state of the decomposed MDP. The penalty should satisfy the following principles:

1) It should induce the tendency for the individual data set backup plans to stay away from states that are close to the RPO limits.
2) It should have minimal changes on states that are far from the RPO limit.

There could be many ways to introduce the penalty that would satisfy the above principles, and it is not the focus of this chapter to investigate their relative merits. Thus here I introduce the following simple penalty function ρ_i:

$$\rho_i = V_{\text{base}} \cdot \left(\frac{n_{\text{skipped}}}{RPO_i}\right)^3 \tag{6.7}$$

Here V_{base} is a tunable parameter, n_{skipped} is the number of backup points skipped in the current MDP state, and RPO_i is the RPO requirement of the data set i. The formula is simple and does introduce much smaller penalty if the value of n_{skipped} is small. It also accounts for the varying strictness of different data set RPO values. I do not claim this is the best choice for the penalty values, but leave the investigation of more complex formula choices as future work.

By decomposing the full system MDP into smaller ones for each data set with added penalty, it is then possible to solve the latter independently and obtain their respective optimal plans and value function entries. It remains an issue, however, as to how to assemble an overall plan for the whole system which ensures satisfaction of the resource constraint, something that is not guaranteed by the simple combination of individual plans. To achieve this goal, I use the value function entries, instead of the individual plans, from the MDP solution of each data set. The idea is to use the value function entries to perform a one-step planning for the whole system whenever it is needed, i.e.,

$$V'(s) = \min_{a \in A_{s,R}} P(s'|s,a)[R(s'|s,a) + \beta V'(s')] \tag{6.8}$$

In (6.8) the terms again denote those in the overall MDP, e.g., s denotes the global state governing all data set states. The action set $A_{s,R}$ indicates all actions that are allowed in state s (with respect to RPOs and RTOs) while also satisfying the system resource constraint. It is thus a subset of all the actions allowed in state s. The term $V'(s')$ is the sum of appropriate individual data set value function entries (for individual data set states that correspond to the global one indicated by *s'*). These entries are obtained from the individual MDP solutions.

The optimality equation in (6.8) can be solved using various means. Here I present an integer programming-based approach in (6.9). Here the term *res* is the capacity of system resource for concurrent backup execution and I_{ia} indicates which action is chosen for a given data set (indexed with *i*). The first and second constraints then together enforce the requirements that only one action is chosen for a data set, and that the total number of simultaneous backup operations does not exceed which is allowed by the system resource.

$$\begin{aligned} & min \sum_{i \in Src} \sum_{a \in A_{is}} I_{is} P(s'|s,a)[R(s'|s,a) + \beta V'(s')] \\ & s.t.\ \forall i, \sum_{a \in A_{is}} I_{ia} = 1 \\ & \sum_{i \in Src} \sum_{a \in \{partial\ backup,\ full\ backup\}} I_{ia} \leq res \\ & I_{ia} \in \{0,1\}\ \forall i, a \end{aligned} \tag{6.9}$$

Assuming the state space sizes of individual data set MDPs are bounded by |S| and the number of data set is N, then the objective function in (6.9) involves on the order of 3N|S| summations. 3N|S| terms are also needed to specify the first type of constraints, and 3N terms for the second type of constraint. Thus the total size of the formulation in (6.9) is linear in the number of data sets. This result means the problem can be solved with reasonable efficiency, and thus can be invoked conveniently at each backup point to determine the backup operations for each data set.

There remains one piece of detail to discuss regarding the proposed approximation. As described earlier, I introduce extra penalty into data set MDPs to produce individual plans that are likely to satisfy RPO/RTO requirements as well as the limit on system resource. To achieve this, a suitable value of the base penalty term (V_{base}) must be selected in (6.1). However, I did not find this to be a major issue in my experiments. While a predefined base penalty value may not work for a given system configuration, it is straightforward to utilize binary search to find one that does.

The difference in solution steps between the full MDP and the decomposed version can be summarized in Figure 6.5.

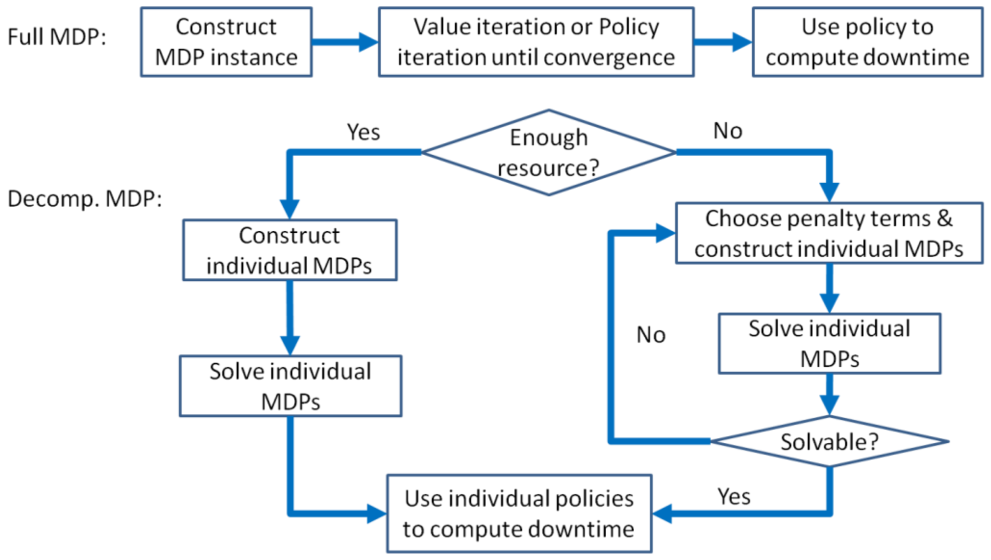

Full MDP:
Construct MDP instance
Value iteration or Policy iteration until convergence
Use policy to compute downtime
Decomp. MDP:
Enough resource?
Yes
No
Construct individual MDPs
Solve individual MDPs
Choose penalty terms & construct individual MDPs
Solve individual MDPs
Solvable?
No
Yes
Use individual policies to compute downtime

Figure 6.5. Summary of Solution Steps

6.5 Quantitative Investigation

In this section I present the numerical results of my investigation into the effectiveness of both the original MDP framework and my new decomposition approach. Specifically I simulate system evolution under the plans from the overall MDP and the decomposed one, as well as several heuristic data backup plans that may be used in practice. I demonstrate the effectiveness of the optimization framework by comparing the expected system downtime values from the simulation runs under different plans. One thing I would like to note is that the simulation is for validation purpose as the MDP solutions and subsequent computation of downtime do not themselves require simulation. All the solutions and simulations are performed on a computer with an Intel(R) Core(TM) i5-2540M CPU at 2.6GHz and 4GB memory.

6.5.1 Base Comparison

I consider the following base system setup. The system consists of four data sets. These data sets share the availability model in Figure 6.2 but have different failure, recover and backup parameter values which are presented in Table 6.1. In the table U(a, *b*) represents a uniform distribution between *a* and *b*. All the time quantities in the table are in the unit of hours. These values are chosen based on the assumptions made earlier regarding the nature and relative magnitudes of various time quantities, e.g., as in Section 6.2.

Table 6.1. Data Set Parameters

	Set 1	Set 2	Set 3	Set 4
Time to complete a full backup (t_{FB})	U(6.5,7.5)	U(5.5,6.5)	U(7.5,8.5)	U(8.0,9.0)
Ratio of effective data change (α)	0.5	0.2	0.4	0.3
Time to recover a full backup (t_{FR})	U(6.5,7.5)	U(5.5,6.5)	U(7.5,8.5)	U(8.0,9.0)
Time to recover a partial backup (t_{PR})	U(0.7,0.8)	U(0.55,0.65)	U(0.6,0.7)	U(0.8,0.9)
Mean time to failure (10^4) (λ)	2	0.1	1.3333	1.6667

As indicated earlier, the primary comparison in this chapter is between several heuristic backup plans and the policies from the MDP formulation, both the monolithic and the decomposed versions. The heuristic plans are summarized in Table 6.2.

I perform comparison using two sets of RPO/RTO requirements for the data sets, and compare the expected annual downtimes. The first comparison takes place in a situation with sufficient system resources for the concurrent backup execution of all data sets. Thus in this case the data sets are indeed decomposed by default. Hence the results, presented in Table 6.3

in the units of hours, are mainly to showcase the optimality of the MDP approaches compared to most of the heuristic planning policies.

Table 6.2. Heuristic Backup Policies

h_1	One full backup per day
h_2	One partial backup daily until reaching RPO, then a full backup
h_3	Skip backup until reaching RPO, then perform a partial backup. Continue until reaching RTO, then perform a full backup.
h_4	First determine the smallest RPO value, v_1, among all data sets. Then perform a backup for each data set every v_1 backup points. The exact type of backup to perform is determined by a greedy comparison between the full and partial options.

Table 6.3. Expected Downtime with Concurrent Backup

	RPO/RTO requirements (P1,T1), (P2,T2), (P3,T3), (P4,T4)	
	Configuration. 1: *(5, 3), (4, 2), (5, 5), (3, 1)*	Configuration. 2: *(4, 3), (2, 4), (6, 2), (3, 2)*
Full MDP	30.78 [30.51, 31.06]	31.47 [31.16, 31.78]
Decomp. MDP	30.79 [30.49, 31.09]	31.47 [31.16, 31.78]
h_1	140.46 [140.28, 140.64]	140.36 [140.18, 140.53]
h_2	74.31 [74.10, 74.51]	70.60 [70.33, 70.86]
h_3	30.80 [30.54, 31.06]	31.48 [31.17, 31.75]
h_4	35.79 [35.24, 36.34]	46.89 [46.57, 47.21]

In Table 6.3, each row lists the downtimes produced by different plans under a particular set of data RPO and RTO requirements, and the brackets contain 98% confidence intervals. As the results shows, the MDP-based approaches can generate better plans, in terms of downtime, than the common heuristics. Also note that with sufficient system resource the data sets effectively become decoupled, and there is no need to introduce the extra penalty to the decomposed version. It then becomes clear (and validated by the results) that the decomposed method produces the same results, subject to simulation variations, as the original MDP. In this particular example, the heuristic 3 also performs well. This is mainly because the data sets have small failure rate values, and so performing backups in a lazy fashion, as heuristic

3 does, is actually a good way to operate the system. On the other hand, heuristics 1 and 2 are performing data backups too aggressively resulting in unnecessary system downtime from these executions. Finally, heuristic 4 stands somewhere in the middle. It relies on synchronizing all data set backup operations to avoid running into RPO/RTO violation. This means it sometimes does make backups unnecessarily. So long as the variation in RPO values is not large, however, it still performs reasonably well.

Next I consider the situation where the system resource is insufficient to allow simultaneous backup executions for all data sets. In this case heuristic 3 no longer works, since it is guaranteed to run into a decision point where all data sets need backup execution, e.g., within a period defined by the least common multiple of all data set RPO values. Hence I only compare the other heuristics with the MDP approaches. The results are presented in Table 6.4. The confidence intervals are narrow and thus not shown in the table.

Table 6.4. Expected Downtime without Concurrent Backup

	RPO/RTO requirements (P1,T1), (P2,T2), (P3,T3), (P4,T4)			
	(5, 3), (4, 2), (5, 5), (3, 1)		(4, 3), (2, 4), (6, 2), (3, 2)	
	res 3	res 2	res 3	res 2
Full MDP	29.81	30.17	31.19	31.23
Decomp. MDP	32.95	33.35	34.20	35.23
h_1	104.83	69.88	103.60	65.76
h_2	63.31	45.33	54.86	45.13
h_4	38.14	37.22	47.74	41.01

In Table 6.4, "res x" indicates the system can support *x* concurrent backup executions simultaneously. Again the results show that the MDP-based approaches outperform the heuristics by some fair margin. On the other hand it is worth noting that the result from the decomposed MDP is different from that from the full MDP. This is a result of introducing the penalty terms to satisfy the various system constraints. Even so, the decomposed MDP still performs noticeably better than even heuristic 4. It is also interesting to observe that, while both MDP-based methods become slightly worse off when resource is reduced, the heuristics actually perform better in this setting. This is due to the fact that originally the heuristics were already performing backups unnecessarily frequent. Therefore as they are forced by resource limit to decrease backup frequency the situation actually improves for them. However, this does not suggest the heuristics would eventually surpass the performance of the MDP approaches.

At this moment it is also worthwhile to discuss the rationale of designing heuristic 4, which would also shed some light on the difficulty of designing a valid backup plan when system resource is limited. In such situations, allowing each data set to backup at its will can easily lead to RPO violation. To see an example, consider a system of three data sets with RPO values 4, 5 and 6 and only one backup execution allowed. Suppose all the data set chooses to only carry out backups when skipped ones equal the RPO values, then starting from any initial data set statuses and within a period that equals the least common multiple of the RPO values (120 in this case) there is guaranteed to be at least one backup point when all the data sets are at their maximum skipped values and attempting a backup simultaneously. Then the RPO violation of one or more data sets ensues. While this example may seem a bit extreme, it is also very difficult to come up with a valid backup plan (or to prove none exists) for a given system with a specific starting state. One compromise is to try to avoid this type of problem by enforcing the same backup period for all data sets. Clearly, the period length is limited by the smallest RPO among data sets, and the approach pays the cost of potentially performing more backups than is necessary for some data sets. This is exactly what heuristic 4 does. More complex backup plans are certainly possible, but would in general involve complicated timing of the backup executions for different data sets, and to design such a plan manually would thus be difficult. This is the place where the MDP-based approach can be of much help, by providing an optimal plan (or at least a good one in the case of the approximation) that satisfies the various constraints.

Another thing I compare in the base case is the memory consumption and execution times of the two MDP versions. Since the purpose of the decomposition is to reduce the cost of the MDP solution (at the expense of some optimality), it is worthwhile to know how effective it is in this respect. These two quantities for the base case are presented in Table 6.5.

Table 6.5. Complexity Comparison of MDP Versions

	Memory consumption (MB)		Solution time (second)	
	configuration 1	configuration 2	configuration 1	configuration 2
Full MDP	1250	892	23.56	18.72
Decomp. MDP	35	28	31.08	21.31

In Table 6.5, “configuration 1” and “configuration 2” refer to the two RPO/RTO settings used in the base scenario (as in Table 6.3). The results illustrate the much larger size (and hence larger memory consumption) of the full MDP formulation compared to the decomposed version. This result is unsurprising given the exponential and linear size growth of the two approaches. On the other hand, the full MDP enjoys some minor advantage in terms of simulation time. This apparent discrepancy arises mainly from the cost of solving the integer programming (IP) in (6.9), during the simulation using the solution of the decomposed approach. Even though individual IP instances are quick to solve (due to the small instance size), one must be solved at each decision point hence increasing the simulation time. The combined solution time of all the data set MDPs, even without parallelization, is actually much faster than solving the full MDP. Nevertheless, even with this extra time cost the decomposed approach is still fairly fast to solve (which validates its use in actual system operation), and the saving in problem size is significant:

as additional data sets (with the same RPO/RTO requirements across data sets) are introduced, the full MDP approach breaks down at around five data sets due to heavy memory usage, while the decomposed version handles up to sixty data sets without any sign of problem.

6.5.2 Effect of Varying Parameters

I then look at the effect of varying the system parameters on the performance of the different approaches. Specifically, I vary the mean values of the times to complete/recover a full backup (t_{FB}, t_{FR}), the times to recover a partial backup (t_{PR}), as well as the mean times to failure (λ) of the data sets (refer back to Table 6.1). I do this by multiplying a factor to a particular parameter across all data sets and compare the performance (expected downtime) from different plans. The results are presented in Figure 6.6. In all cases the system has sufficient resource for concurrent backup executions for all data sets.

In the figures on the left column, all the plans yield increased expected downtimes as the time cost to complete a full backup or to recover one increases. The decomposed method yields the same performance as the full MDP does in all cases and thus compare favorably versus the heuristics. On the middle column, the increase in partial backup recovery times do not change the values much. This is mainly the result of the small mean-time-to-failure values that I assume. While I could (and did) increase the partial recovery times to very large values and observe the resulting increase in the expected downtime values, those parameter values become less realistic and the results less meaningful. Thus I do not present those results here, and just note that the decomposition results are better than those of heuristics. Finally on the right column the increase in mean-times-to-failure leads to decrease in the expected downtime.

In this case the MDP-based approaches actually switches from doing full backups to partial backups at some states to take advantage of the decreased expected recovery cost.

In all the above results the decomposed approach coincides with the full MDP and outperforms the heuristics. Similar situations exist in the cases where the system does not have sufficient resource to allow simultaneous backup executions for all data sets. I show one example in Figure 6.7, where I assume the system can accommodate at most two concurrent backup executions. Note that as heuristic 3 no longer works it is not present in the figure. Again as I increase mean-time-to-failure values the expected downtimes decrease. Notice that due to the insufficiency of resource and introduction of extra penalty into the decomposed MDPs, its solution does not coincide with that of the full MDP. Nevertheless it compares quite favorably against the heuristics.

6.5.3 Large System Scenarios

In this section I investigate the performance of the decomposition in large system scenarios where the full MDP approach is not applicable. Again I first consider the case where system resource is sufficient. To set up the large system, I randomly generate new data sets, with the parameter values sampled uniformly from those in Table 6.6. All time quantities are in the units of hours, and the uniform distributions have bounds 10% away from their means.

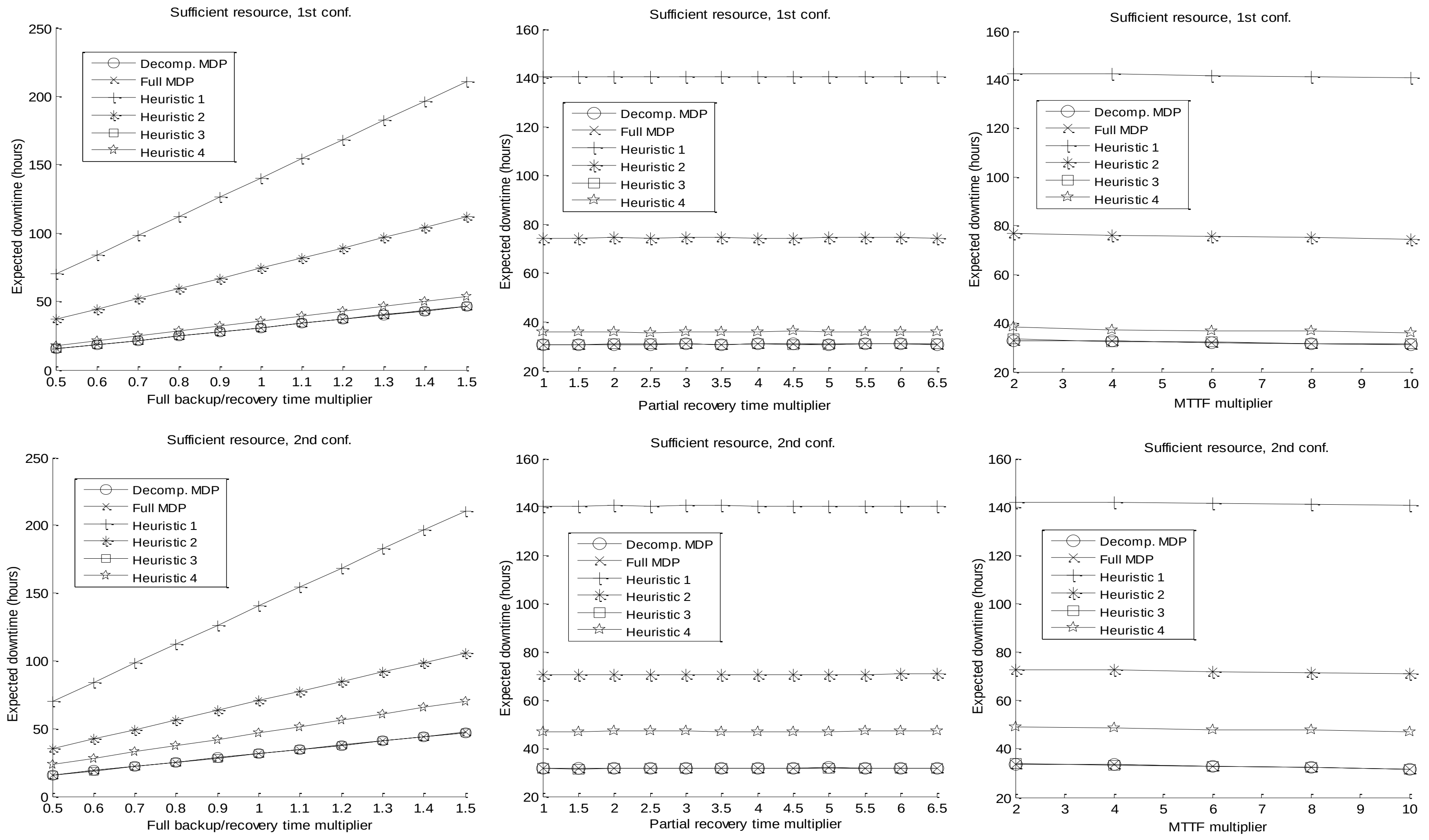
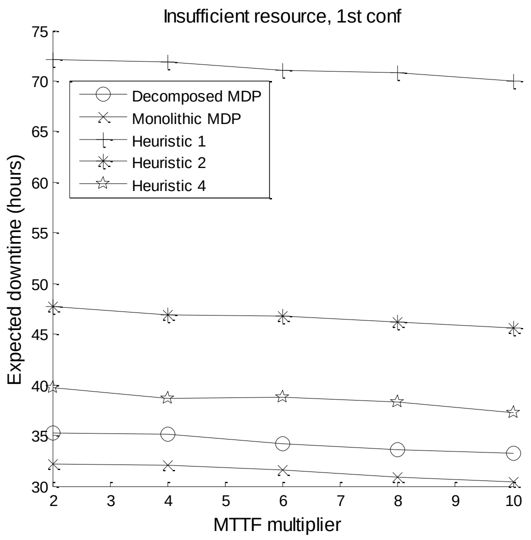


Figure 6.6. Effect of Varying Data Set Parameters.
Left column: Increasing the time to complete a full backup or to recover one; **Middle column**: Increasing the time to recover a partial backup; **Right column**: Increasing mean-time-to-failure.



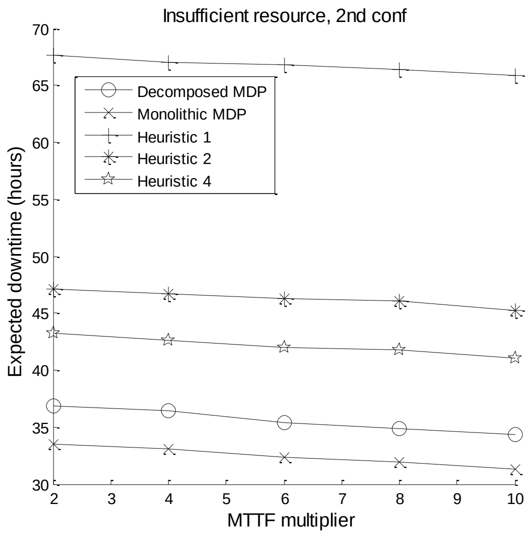


Figure 6.7. Effect of Varying MTTF under Insufficient Resource.

Table 6.6. Random Data Set Parameters

Parameters	Sample Range
RPO	2~7
RTO	2~6
Mean Time to complete/recover a full backup	9~20
Mean Time to recover a partial backup	0.1~0.6
Mean time to failure (10^4) ($1/\lambda$)	2, 1, 0.6667, 0.5, 0.4

Using the values in Table 6.6 I generate from five to sixty data sets randomly. Since the full MDP method no longer works, I compare the performance of the decomposition approach versus that of the heuristics. The results are presented in Figure 6.8. From the results it is clear that the decomposition approach again compares favorably against most other heuristics, including the relatively complex heuristic 4. On the other hand, the "lazy" heuristic 3 still performs quite well, to a level on par with the MDP approach, due to the parameter settings in use. It does not, however, work in general cases of limited resource, and as a fixed heuristic it cannot respond to changes in data set parameters either. Thus the formal MDP-based approach still has advantage in these respects.

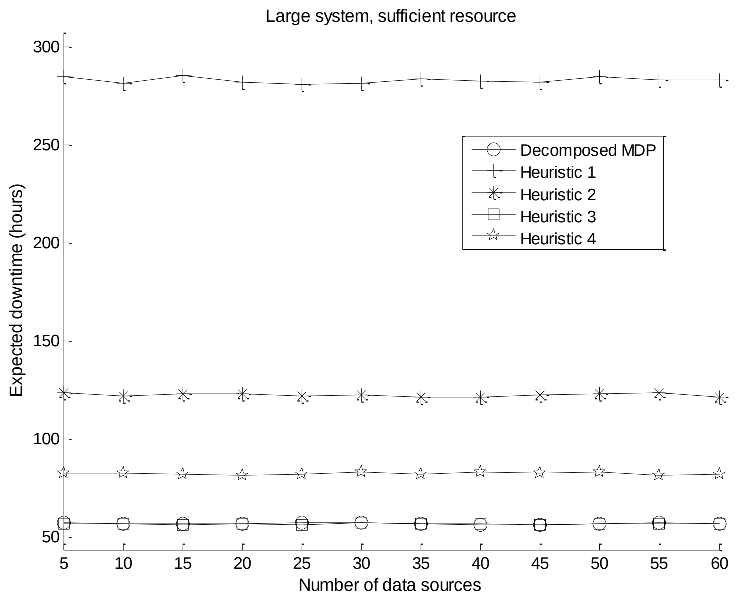


Figure 6.8. Large Systems with Sufficient Resource.

The advantage in solution complexity of the decomposed MDP is shown in Figure 6.9.

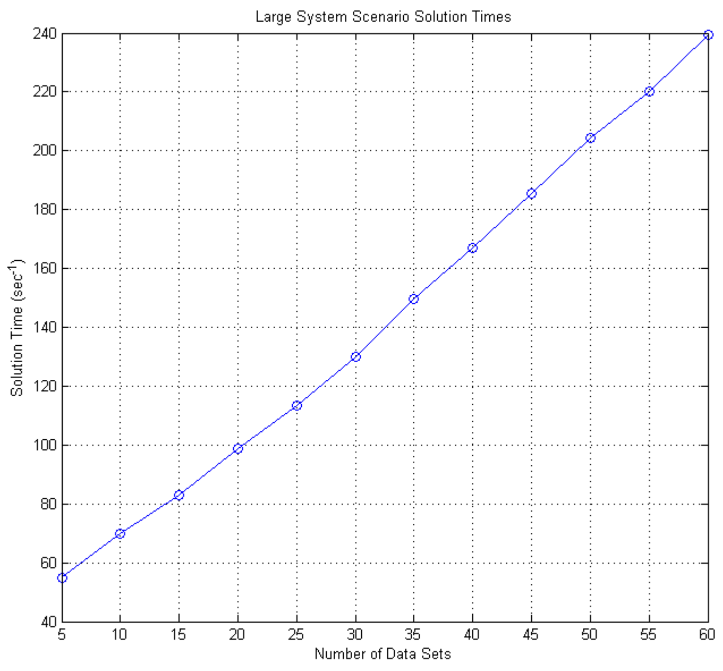


Figure 6.9. Solution Times of Decomposed MDP in Large System Scenarios

As Figure 6.9 shows, with the increase in the number of data sets the solution time increases in a linear fashion, validating the earlier theoretic results on the decomposition solution complexity and showcases its significant improvement over the monolithic MDP.

Next I look at the case of large system where the resource is limited. I pick one randomly generated system instance of twenty data sets and vary the number of allowed concurrent backups from five to eighteen. The results are shown in Figure 6.10.

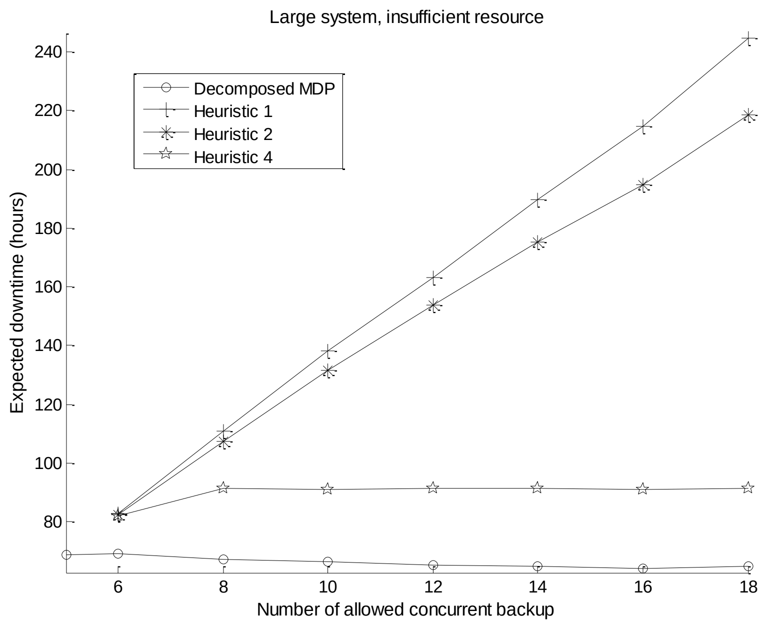


Figure 6.10. Large Systems with Insufficient Resource.

The results in Figure 6.10 again demonstrate the benefit of the MDP-based approach over the heuristics, even when the latter has seemingly improved performance as resource reduces as a result of their inherent over-aggressiveness in performing backups. Another thing to note is that when only five backups can be performed simultaneously, it is possible, depending on the initial data set statuses, for heuristic 1 and 2 to fail to find a valid plan. This is because executing a backup earlier would accelerate the arrival of the next moment when a backup must be performed. Thus if a plan executes backups too aggressively then it is possible for it to introduce too many additional backups within a given period to overload the system capacity. Hence in some cases too frequent backup executions would not only be inefficient but also unsafe (in terms of satisfying the RPO/RTO requirements). This further demonstrates the utility of the MDP-based framework in locating an optimal (or very good) backup plan.

6.6 Related Work

Due to the importance of data protection, there has been much work devoted to this topic. Some research work focuses on design and implementation of new techniques to improve effectiveness of data backup and other protection techniques, for example [111] and [112] focus on the technique of data de-duplication, while [113] and [114] investigate online backups. Some other works provide overview on general issues and techniques, perhaps in specific contexts, such as [115]. There are also works that focus on modeling and evaluation of specific data backup techniques and strategies, such as [116][100] and [117]. My work here differs from these examples in that I focus on the designing of a generic framework to optimize data backup plans, instead of focusing on specific techniques.

There are also works on effective design and/or management of data backup operations. Many of these works utilize some form of optimization. For example, [118] focuses optimization of data backup intervals, [119] considers optimal data placement and level of replication, while [120] investigates the design of a storage solution for a specific context. By comparison, my work focuses on backup planning, with the application of MDP which allows a large set of practical scenarios to be modeled and optimized for.

The effective solution of a “weakly coupled MDP” that is present in this paper is a classic topic in fields such as automatic control, operation research and artificial intelligence. Some examples of such work include [109] and [110]. While these works present some effective techniques, they are not applicable in my case. The method in [109] does not deal with the issue of a large action space or the restriction of action options in some states, which is unavoidable in my scenario if their approach is to be applied. The work in [110] has quite general applicability, but the relaxation approaches presented there only satisfies constraints in expectation, not in probability as in my context. As these approaches are inadequate for my purpose, I opted to develop a simpler method that (as the evaluation section demonstrates) is still effective in this context.

6.7 Conclusions and Future Work

In this chapter I presented an optimization framework for data backup planning based on Markov Decision Process (MDP). I provided details on how to construct an MDP instance from system specifications and interpret system operation constraints in the context of the MDP formalism. To tackle the scalability issue common in applied instances of MDP, I proposed a simple approximation technique based on decomposing the original MDP to address the scalability issue of the original framework. The subsequent numerical investigation and discussion sections establish the effectiveness and importance of the MDP-based approach and the decomposition method in optimizing and planning data backup operations. In future work I would seek to introduce more complex models for data set/system availability as well as more realistic relationships between different backup types and backup/recovery operations. Incorporating these additional elements into the existing framework would then further improve the practical utility of proposed method.

7. Modeling Replicated Storage Systems

7.1 Overview

IT operations today are faced with rapidly increasing amount of data generated by daily operation, and the complexity of storing such data in a reliable and cost-effective fashion is also escalating. With increasing public awareness about data security and privacy come more stringent requirements on data protection, and the consequence of losing important business/client data can be devastating to an organization. On the other hand, the sheer amount of data that has to be handled also creates a problem when the limited IT budget is taken into account, as it becomes increasingly difficult to satisfy the data storage need with highly-reliable but also costly high-end data storage solutions. While there is still a significant market for such solutions, people are looking into other techniques that may provide a better cost-effectiveness trade-off when it comes to storing data.

With such demands in place, replicated storage has been gaining much attention and adoption recently. While the idea of replication is by no means unfamiliar to people working in the IT industry, and indeed it has been playing fairly central roles in data storage technologies such as RAID, and featured in solutions such as IBM General Parallel File System (GPFS) [121], the onset of the so-called "Big Data" era has greatly increased its practical relevance. In newer generations of replicated storage systems, as exemplified by instances such as Hadoop File System (HDFS) [122] and Lustre File System [123], much attention is paid to scalability and modularity, so that the system can be separated relatively easily into multiple logical components, each of which can effectively make use of technologies from different vendors/sources. Such features allow the system to easily expand as the amount of data requires by adding new components. In such a setup, while higher performance can certainly still be achieved by upgrading existing components, the primary focus is shifted towards harnessing the collective power of large numbers of commodity storage devices. Such an approach helps to lower system cost through economy of scale and provides the needed storage space and data access performance through horizontal scaling. To achieve data reliability and availability, such systems employ replication by maintaining multiple copies of data items across their storage components. In a scenario where some components have failed and made the data replicas hosted on them inaccessible (or outright lost), such items still have other accessible replicas so the component failures are effectively masked from users. In addition, replication also serves to improve performance by, e.g. distributing data access workload across different storage nodes.

While replicated storage systems have proven to be effective in meeting the need of efficient data storage in the age of data explosion, it comes with its own challenges in implementation and operation. It needs to deal with the same issues as traditional storage systems, such as data reliability, availability, consistency and access performance, but the fact that replication is employed adds complexity to many of these issues. To assist in the design and management of replicated storage systems, it would be helpful to have a set of stochastic models that can generate accurate estimates on system metrics of interest. Such models would then help provide guidelines on how to make tradeoffs in system design and evaluate potential implementation choices, as well as provide estimates on how the system would behave under given configurations during operation. The models should also be easy to solve, so as to facilitate its use in comparing multiple design options and operational configurations. The stochastic models do not have to capture every detail of the system, but should cover those having important influence on the metrics of interest to ensure enough accuracy to be useful.

In this chapter I will describe a set of stochastic models designed for the purposes discussed above. I will first discuss the issues of concern in replicated storages and give an example system that I am going to model (Section 7.2). Then I will describe the modeling details and the necessary assumptions in the modeling process (Section 7.3). The accuracy of a key approximation is investigated in Section 7.4, and Section 7.5 provides the numerical results from the main models. Application scenarios of the model, as well as possible extensions are discussed in Section 7.6. Related works are reviewed in Section 7.7. Finally Section 7.8 concludes this chapter.

7.2 Important Issues in Replicated Storage

As with more traditional forms of data storage systems, replicated storage system must also deal with a range of issues in its design and operation to provide effective data storage functionality. Some of the most important issues in this category are how to achieve data availability and reliability in the face of inevitable component failures (especially in situations where commodity components are employed in scale). Furthermore, replicated storage systems are almost by definition distributed, and thus they also need to handle some of the important issues in distributed systems, a prominent one being data consistency and its impact on system access performance. To further complicate things these issues are also interrelated, as improving data availability and reliability through replication means multiple copies of the data coexist, which then requires appropriate techniques to maintain data consistency without hurting performance, especially if the replication is done across geographically separated locations. In this section I will first discuss the notions of data reliability, availability, access performance and data consistency in the specific context of replicated storage, and then discuss some details on the different aspects of the issues.

7.2.1 Data Reliability

Data reliability indicates the capability of the storage system to maintain the data as originally input by the users, ensuring the data is not subject to loss or corruption. It concerns mainly the property of the data itself (specifically its integrity) instead of the status of the whole storage system, but its level of fulfillment (or violation) will depend on how the system is structured and operated. For example, failures of storage nodes/devices are obvious ways in which data can be lost, and data corruption can occur either through disk access errors, silent background corruption or error during transmission. All of these occurrences are tied to properties of the system structure and its components, and prevention of such incidents or mitigation of their effects are thus important goals in replicated storage design and operation.

7.2.2 Data availability

Data availability indicates the capability of the storage system to provide specific data to the users as required. Compared to data reliability, data availability is more about a property of the system as a whole, although it certainly is related to data reliability. To ensure data availability, not only the data itself must be present in the system, but there must also be a means to transfer user access request and the requested data back and forth. The former is largely the same as the requirement on data reliability, while the latter requires an access path, e.g. a data network, to be operational between the entrances of user access to the data storage node(s). Due to the greater number of devices and other system components involved in an access process, it may be argued that achieving good data availability is more complicated.

It is worthwhile to note the connection and difference between data availability and reliability. These two properties can have overlap in the scenarios they cover. For example, should the system lose data (hence violating its reliability promise), such data is also unavailable for the user to access (thus availability also suffers). On the other hand these notions still remain distinct in many situations. For example there may be cases where corrupted data can still be useful (for example if the user-side employs high-level error correction mechanisms), which means a decrease in data reliability (in this case the onset of data corruption) does not necessarily cause the same level of impact on availability. Similarly, the data may become inaccessible due to failures of system components that are essential for data access but not storage, e.g. the storage system network. In such cases data reliability is not affected even though the data is not available for user access. Generally speaking though, it is possible to consider the requirement of data availability as containing that of data reliability, especially if the data is expected to be updated relatively frequently such that inaccessible data quickly becomes outdated.

7.2.3 Data access performance

Access performance indicates how fast the users can access data stored in the system. While the notion of “fastness” itself is definite, the ways in which different users accessing the data may place emphasis on different aspects of the access process, and as such the level of performance that a particular access may experience can depend on the type of access. As a result, a system may provide different levels of performance when user access workload pattern (in terms of the mixture of different access types) varies, and to design the “most performing” system typically requires tradeoffs in this respect. As an example, a system may be optimized for data read if that represents the majority of user accesses (an example being a storage serving mostly static content), but data write operations may then take longer to complete (e.g. when the system adopts a large write quorum to allow a small read quorum). Another example also arises in system performance optimization, where access throughput is optimized by combining multiple requests (e.g. so as to reduce IO overhead) at the expense of larger variance in user response times. This may create tension, for example, when a system previously designed and optimized for batch-processing jobs is increasingly used to handle latency-sensitive ones, as was the case of Google File System [124].

7.2.4 Data consistency

Data consistency indicates the capability of the storage system to ensure data content stays in legitimate forms. It is connected partially to data reliability, since corrupted data certainly is not legitimate. However its usual standpoint is on a higher logical level and it is also concerned about other undesired scenarios. In a typical scenario of consistency violation, the data may be free from corruption and completely legible, yet it may correspond to an illegitimate state that cannot or should not have arisen according to the logic governing the data and the data operations permitted. The ways to determine what constitutes “legitimacy” vary in different scenarios and/or for different systems, but typically revolve around the guarantee that, subject to some relaxation or constraints that must be made clear in system specification, the data should reflect correctly the combined results of all the previous user operations on the data when it is accessed later, irrespective of from where the access is launched. In addition, no other change should occur besides those resulted from the said operations.

One notable distinction of data consistency as compared to data availability and reliability is that it is less clear-cut in its definition. As mentioned above, the very notion of "legitimate" data is in general scenario-dependent. Depending on the nature and usage of the data, a "legitimate" access result in one case may well be intolerable in some others. For example, regarding time sensitiveness it is generally acceptable for a social network application to return different statuses to different users after an update to a topic so long as it doesn't take too long to start replying with the same result, but for a system with more strict requirement, such as a database for stock price fluctuations, the same behavior is no longer acceptable. Another example is in the case of concurrent data operations from multiple users, where the notion of "correctness" of the combined user actions may become situational. For instance, an online chatting program could tolerate slight reordering of users' comments, but a storage system supporting concurrent computation programs modifying the same sets of data would not have such luxury.

Moreover, there are both theoretical results (such as the CAP theorem [125]) and practical considerations (e.g., the need to construct geographically-separated replication systems subject to various physical limits such as the speed of light) that indicate such scenario-dependent nature may be inevitable for data consistency. Hence in addition to more traditional consistency models such as sequential consistency [126], various relaxed form of data consistency models have arisen in practice, a notable example being that of eventual consistency [126][127]. The choice on which consistency model(s) to adopt is then subject to the design goal of the system in question.

7.2.5 How replicated storages address such issues

All the aforementioned issues are important in the design and operation of a replicated storage system, and different systems address these issues in various innovative ways.

- Data reliability is mainly handled by the replication procedure (guard against loss), but is also frequently augmented by error-correction at various system levels (e.g. checksum). The replication procedure itself is usually designed to ensure that the replicas are distributed onto storage nodes that are physically separated, e.g. on different racks or even different datacenters. The purpose of such separation is to ensure (or at least make it very likely) that the different replicas would not be affected within a short time frame by failures, so that the system has an adequate time window to react to replica losses by additional replication of the data. Note that depending on the consistency model, the extent to which such physical separation is done may require compromise from the performance aspect as greater separation potentially leads to longer time for synchronized data accesses. Therefore while wider separation defends against more severe failure scenarios (e.g. natural disasters) it may not always be possible or desirable given particular performance and consistency requirements.
- Data availability is also mainly realized via replication onto physically separated nodes, and thus shares some of the issues/tradeoffs as described above. In addition, the data access path itself must also have redundancy built in. The effort to achieve

such a goal could benefit from the results of many years' networking research, especially those on datacenter network design.

- Compared to data reliability and availability, the endeavor to provide good data access performance can be more complicated and may require more tradeoffs. For example, to better handle different workload mixes:
  - Data may be hosted on different nodes based on whether its access is read- or write-heavy, and/or whether the user application owning the data is more batch-oriented or interactive. Then optimization may be set up on the nodes based on the characteristics of the data hosted.
  - If justified by the actual need, more algorithmic intelligence can be implemented in the system to dynamically adjust certain aspects of the system, e.g. the data placement strategy, to better handle the workload pattern.

  A major factor in determining the performance enhancement available to a system is the consistency model the system needs to provide, and this is closely tied to the replicated nature of the data storage. The replication itself typically benefits read access performance by providing more sources from which to read data, thus reducing contention on individual hosting nodes. However, write operations will need to involve multiple replicas that are potentially separated by significant physical distances. If a weaker consistency model is acceptable (even if only for some subsets of data), then the replicas may be updated asynchronously, which opens up to a range of options including:

- Replicas may be updated in a totally asynchronous fashion by replying to the user after the first replica is updated, while others are updated opportunistically, e.g. via lazy propagation from the updated replicas.
- The system requires a write operation to update a quorum of the replicas synchronously, while the rest are updated opportunistically. In this case an update-by-read scheme may be employed where both read and write touches a quorum with the requirement that the two quorums overlap. Then the replicas that were not updated during the write can receive the new data when a later read discovers their staleness and forwards the data to them.

Sometimes opportunities arise from the nature of the data and the operations that the users may perform. Some examples include:

- If it is possible to divide data into small blocks and separately store them, then user accesses can be directed onto different host nodes and distributed at a finer granularity, improving concurrency while potentially reducing lock contention.
- Concurrency control, an important aspect of data consistency, may be relaxed by allowing more nodes to perform concurrent write if occasional duplication or out-or-order application of data updates can be tolerated. The former can happen if the updates are idempotent, while the latter can happen if the user applications already adopt high-level mechanisms to identify data items.

Even if a strong consistency model is needed, there are still opportunities to improve performance in system design. Some examples include:

    - The user sends the data to one replica, which in turn transmits the data to its peers. The data write is considered complete only when all replicas have confirmed reception of the data updates, hence strong consistency is maintained. The main benefit is to utilize network bandwidth resources more efficiently by making use of the network between replicas.
    - The control of metadata consistency is separated from that of the actual data. While each aspect is still controlled by a single node (so that strong consistency, or more specifically concurrency control in this case, is still in effect), the controlling node for metadata is not subject to the workload of actual data access which generally dwarfs the volume of metadata operations. This setup also allows the metadata controller to manage the identities of data consistency controlling nodes, permitting control of subsets of data to be assigned dynamically based on system and node conditions.

  The above are just some examples of performance-improving techniques in replicated storage. In practice what methods are employed would heavily depend on the design goal, usage pattern and cost-effectiveness analysis of the system.

- Data consistency is closely tied to the design goal of the storage system and is a primary factor in determining the implementation details of the other aspects, especially regarding system performance. Generally speaking, in system design a given consistency model is established based on the expected usage pattern and access characteristics and the system is then designed and implemented around it. Some consistency models that appear in practice are:

- Sequential consistency, a strong form of data consistency defined as “The result of any execution is the same as if the (read and write) operations by all processes on the data store were executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by its program”[126].
- Causal consistency, a relaxed form of sequential consistency where ordering constraints are only placed on data updates that have causal relationship, e.g. the result of one update would depend on that of an earlier one.
- Eventual consistency, a weak form of data consistency which specifies that multiple data replicas will eventually converge to the same status/value after a sufficiently long time in which no new write operation to the data arrives.

Stronger consistency typically means more limited options for performance improvement, and may also impact data availability in the face of possible node failures. For example, an access protocol using single controller and synchronous propagation of data updates may block (even indefinitely without human intervention) if some nodes fail during execution of the protocol. Therefore, relaxation of the consistency model is generally sought in system design in order to facilitate adoption of performance/availability improvement techniques. In some cases such a relaxation is essentially mandatory, for example when replication is done across multiple datacenters in different geographical regions that must support low-latency user access. In cases where the system needs to support a mixture of user workload patterns with different emphasis on latency and throughput, some more recent storage systems, such as Amazon Dynamo [128], give users the option to specify the level of consistency they would like to use.

To make the study more concrete, I will focus on developing a stochastic model for one influential replicated storage system, the Google File System (GFS), and illustrate how to capture its structural properties with the model. After a thorough discussion about the GFS case I will then describe how some of the other techniques used in replicated storage systems (as described previously) could be modeled following the same approach for modeling GFS.

7.2.6 The Google File System (GFS)

The GFS is a replicated storage system developed by Google and made publicly known around 2004 [124]. As one of the infrastructure pieces behind Google's phenomenal growth, the GFS has garnered much attention and even inspired an opensource version known as Hadoop File System (HDFS) [122] (the HDFS is commonly associated with the Hadoop computation framework, which is an opensource equivalent to the MapReduce framework, the latter also developed by Google [129]). GFS is noted for its (relative) simplicity in design and good scalability and performance. In this subsection I will describe the components of GFS and the measures of interest regarding GFS, to lay the foundation for the modeling to be described subsequently.

The basic structure of GFS contains a master node (henceforth called "master") and a cluster of data hosting nodes (henceforth called "hosts"), although typically the master is supported by several backups that can quickly step in should the active master goes offline. The original design goal of GFS was to provide high-throughput and support batch-processing style applications (although this has been changing as Google embraces more interactive applications, as discussed in [130]). Towards this goal, the hosts store user data in blocks (called “chunks”) as local files, and one user file in general is split (or “striped”) across multiple hosts to achieve better aggregated access throughput. Replication of data is performed at the unit of blocks across multiple hosts. On the other hand, the master is responsible for managing metadata and the file namespace and is involved in any operation that concerns such information, for example file creation/move/deletion. The master also handles assignment of data blocks to the hosts and provides user clients (henceforth called “clients”) with such information so that the latter can contact the correct hosts for data access. Figure 7.1 illustrates the basic structure of GFS and the steps of data access.

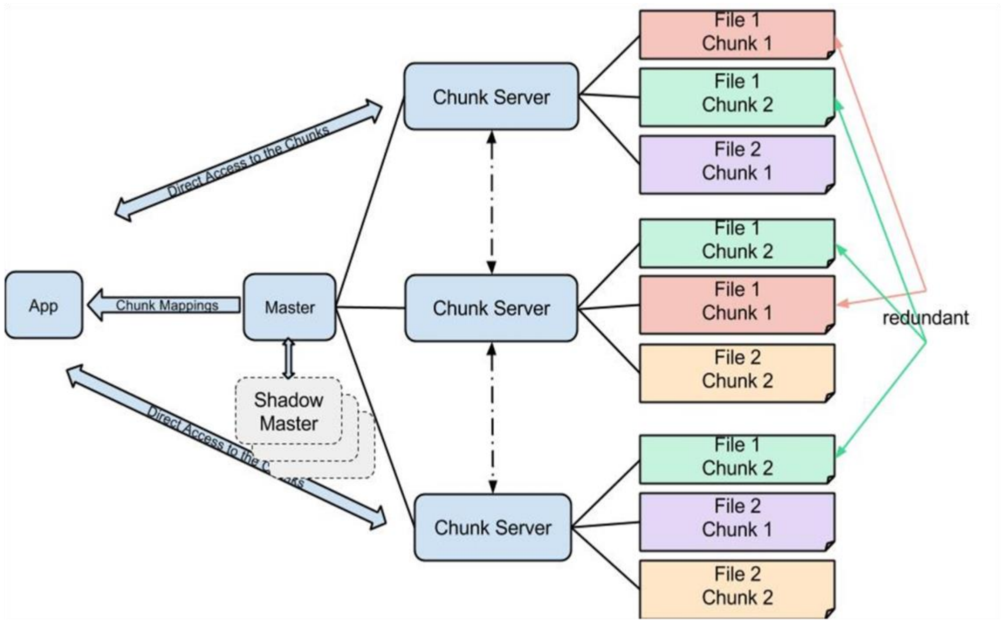


Figure 7.1. GFS Components

In a typical data access flow in GFS, a client first contacts the master about the identities of the hosts that maintain the relevant data blocks. The client caches such info and would not need to contact the master in future accesses unless the identities of the hosts have changed. In the case of a data read operation, the client generally contacts one host, but could optionally contact more to perform parallel accesses. For a write to data, since there is a need for concurrency control among multiple clients writing to the data at the same time, GFS chooses one of the hosts maintaining a replica of the data block as the primary host (henceforth called "primary") for that block (this information is also managed by the master), and the serialization of all the concurrent accesses is performed by the primary. More specifically, the client first pushes data to all the hosts (including the primary) of a block, the latter storing the data in temporary local storage. Then the client contacts the primary to indicate the transfer is done and asks for the write operation to be committed. Since there may be multiple such requests (from multiple clients) pending, the primary picks an order for committing the requests and notifies the other hosts to do the same. After receiving confirmation from all hosts that the commission is done, the primary replies to the client. If any step goes wrong in this process, for example if some hosts become unresponsive, the client (or the primary, depending on which step it is) would retry a few times before considering the said hosts to be down. In such cases the client considers the operation failed, and will need to contact the master again to obtain the identities of replacement hosts. Figure 7.2 illustrates the above access flow.

GFS separates its component nodes into the master and hosts (hosting the data blocks/chunks) to separate the flow of metadata from that of user data. This system structure, combined with caching of metadata info on the client-side, frees the master from the burden of handling user data and allows the metadata management to be concentrated onto one master, significantly simplifying concurrency control and overall system management. In addition, since metadata workload is by comparison much smaller than that of user data, the master can afford to have a stronger consistency model by synchronously replicating operations on the metadata to its backups. Due to the central role the master plays in data access, its high availability is important and is provided by the backups which, due to being synchronized with the master, can quickly step in should the latter fail.

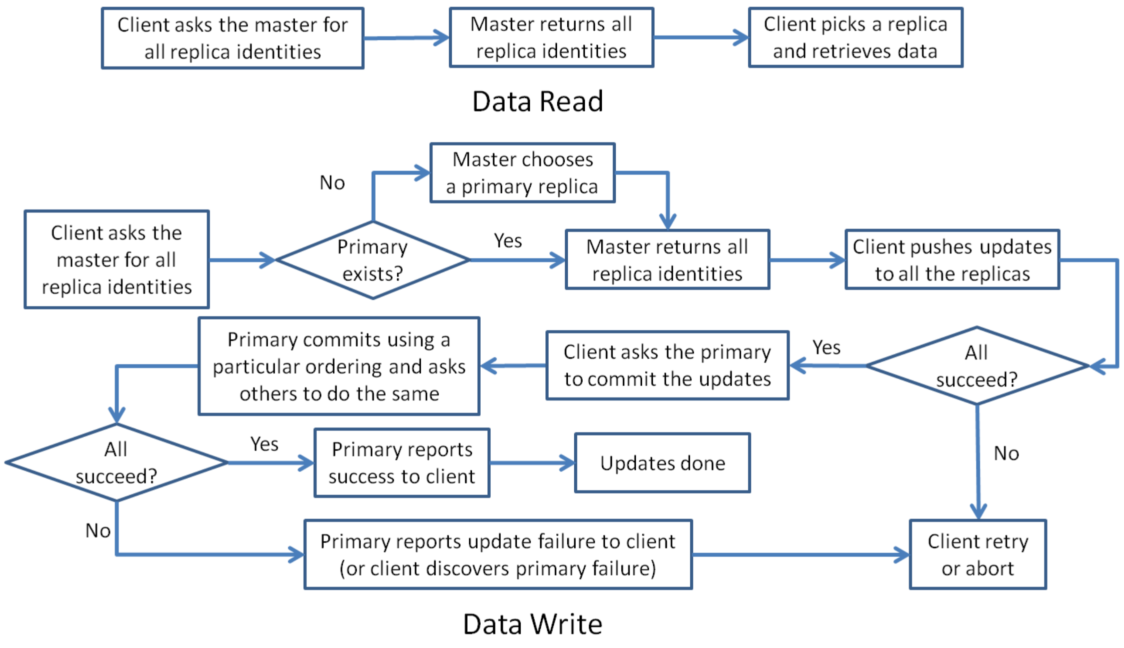

Client asks the master for all replica identities
Master returns all replica identities
Client picks a replica and retrieves data
Data Read
Master chooses a primary replica
No
Client asks the master for all replica identities
Primary exists?
Yes
Master returns all replica identities
Client pushes updates to all the replicas
Primary commits using a particular ordering and asks others to do the same
Client asks the primary to commit the updates
Yes
All succeed?
All succeed?
Yes
Primary reports success to client
Updates done
No
No
Primary reports update failure to client (or client discovers primary failure)
Client retry or abort
Data Write

Figure 7.2. GFS Data Access Flows

On the host side, data availability is provided by the replication of a block onto hosts that are separated in their failure domain (e.g. on different racks). Aside from replication, data reliability is also augmented by checksums on data blocks, which are also needed for consistency purposes (to be discussed shortly). Read performance is improved by the presence of multiple replicas allowing concurrent accesses. Data writes are typically performed by the client first writing to one replica, whose host then starts transferring the data it receives to other replicas at the same time. This pipelined arrangement serves the purpose of better utilizing the inter-host bandwidth (as is stated in the original GFS paper [124] and adopted by HDFS [122]). Regarding data consistency, the primary host performs ordering of concurrent writes. GFS assumes that an overwhelming majority of data writes would be appending to data instead of random write, which allows a relaxed consistency model where each append operation to the data is guaranteed to be performed in its entirety (i.e. without being intermixed with pieces from concurrent updates) at least once. Possible duplicates or mixture of updates are filtered out by checksums at the client side. Finally, host failures are discovered by the master (or by the client notifying the master about such incidents) which then picks additional storage nodes to become new hosts of the data blocks affected by the failure. These new hosts then elicit the data from other old hosts of the blocks to maintain the replication level of the affected data blocks.

As stated before, the original design goal of GFS was to support batch-oriented workload patterns that put a premium on high throughput. Such patterns reflect well the Google workload mixture back then, and as such latency was relegated to a secondary place. However, this assumption has been challenged as Google grows to accommodate more and more user-facing, interactive applications that are latency-sensitive. Consequently many modifications, both within the GFS itself and across the applications that use GFS, have been put in place to better handle this issue. In this thesis I will not look at those modifications (which are typically at a higher logical level, e.g. a multi-home model in the user application) but will focus solely on the GFS itself. With the emphasis of GFS and its challenges in mind, I will mainly focus on investigating two system metrics:

- The mean response time of batch-processing type of workload. The workload would be relatively long-running, and typically will contact multiple hosts concurrently for the many blocks constituting the file(s) being accessed. Their mean response time can be used to predict their throughput values.

- The response time distribution of a request from a latency-sensitive user application. Such an operation is short and involves a small amount of data (e.g. a change of an address book entry in Gmail), and typically would just update one data block.

Aside from these two metrics, some others will also be modeled as they are needed to compute part of the two main ones. These include:

- The availability of the master. More precisely this is the availability of the overall master subsystem. Thus it is the likelihood that any of the master copies are available to handle user metadata requests.
- The availability (and reliability) of a data block. I will look at these measures as hosts fail and recover, and data blocks get replicated to new hosts in this process. These metrics would also affect the main system metrics (the response time in particular) so their effect should be accounted for in the model.

There are other minor metrics that would be computed during modeling the above ones. Such details will be covered in the next section.

7.3 Modeling the GFS

To construct a model for GFS and analyze the measures of interest, I separate the overall system into multiple logical parts and model each of them in turn. This approach allows me to use potentially different modeling formalisms across the parts based on the suitability of particular formalisms in different contexts. Connections among the models are established by passing the output of some models as input to others, wherever appropriate. In this section I will first outline the different models and the relationships among them. Then I will describe the modeling of each part in turn, discussing the possibly different modeling assumptions as I go.

7.3.1 Modeling Overview

This subsection outlines the models for the various GFS parts.

- Data block reliability model. This model looks at an individual data block, and captures the dynamics caused by the losses of block replicas (which may occur due to various system component failure scenarios) and the subsequent system attempt to replicate the block onto new hosts. It accounts for the failures and recoveries of data hosts as well as the racks supporting the hosts. The model uses outputs from the data block access performance model.
- Data block access performance model. This model covers the access performance on the hosts, and it consists of two logical parts: the performance of normal data accesses, and the performance of data replication executions after block replica losses. The main focus is twofold: 1) to analyze the throughput of the replication process so as to provide completion time estimates for the data block reliability model; 2) to analyze the resource contention at the hosts and provide estimates on data access throughput (for batch-processing user operations) and response time (for operations from interactive user applications). These metrics are used to determine the overall data access performance measures of the system.
- Master availability model. This model looks at the failure and recovery behavior of the master subsystem (consisting of the active master and its backups). It provides

estimates on the availability of the master subsystem. This metric will be used in computing the overall data access availability.

- Master access performance model. This model looks at how the master section of GFS performs when handling client metadata requests. Because metadata operations consist of very small amounts of data, the focus of this model is on computing the response time, and as such it is mainly relevant to the experience of latency-sensitive user tasks. This model outputs the response time distribution (via an approximation procedure that will be described in detail later) of a single metadata operation, which is then used to estimate the overall response time for latency-sensitive data access tasks.
- Data access availability model. This model focuses on the overall availability of data access. It takes as input the results from the data block reliability model and the master availability model, and provides the probability that a data access operation can proceed without encountering an instance of system being unavailable.
- Data access performance model. This model focuses on the overall performance of data accesses, in terms of the mean response time for batch operations and the approximate response time distribution for latency-sensitive tasks. The model takes inputs from the master and data block access performance models.

The interactions (input-output relationship) among the models are summarized in an import graph [131] as presented in Figure 7.3.

Next I will describe the details of each model.

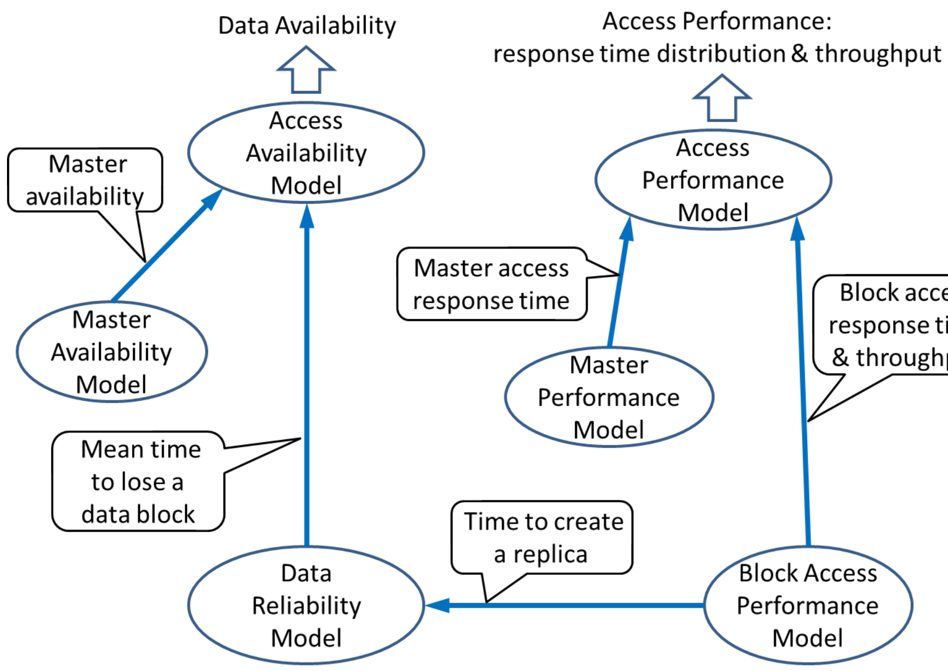

Data Availability
Access Performance: response time distribution & throughput
Access Availability Model
Access Performance Model
Master availability
Master access response time
Block access response time & throughput
Master Availability Model
Master Performance Model
Mean time to lose a data block
Time to create a replica
Data Reliability Model
Block Access Performance Model

Figure 7.3. Import Graph of Models

7.3.2 Data block reliability model

The component availability model describes the failure and recovery behaviors of storage components and their effect to the reliability of a particular data block. The assumptions that I make in this model include:

1) The components (hosts and racks) fail independently and time-to-failure values follow exponential distributions.
2) A rack failure means the hosts installed on that rack also go down from the system perspective, and the data replicas on these hosts are considered lost.
3) New hosts are always available for hosting new replicas of the affected data.
4) A data block is replicated twice, i.e. there are three replicas during normal operation.

5) When replication executes, it creates one new replica at a time. The replication always tries to first establish two replicas on different racks, then a third one on one of the two hosting racks.

Some discussion regarding the assumptions would be helpful for understanding.

- The first assumption contains two parts: the independent failure assumption and the exponential time-to-failure assumption. For the first part, I believe the primary correlation between host failures would be through a common hosting rack (as is also mentioned in [132]), hence I explicitly include the rack in the model. I consider larger scale of correlated failure to be rare in practice and therefore ignore those, but as I describe the details of our model it should become clear that those can also be included if necessary. Regarding the exponential time distribution assumption, it has been shown that (such as in [6]) an exponential time-to-failure distribution is generally appropriate for components that are in the stable range of their life (i.e. not newly made or near the end of their life cycle). Due to this observation and the easy-to-handle mathematical properties of the exponential distribution I adopt this assumption.
- For the second assumption, I believe it is a priority for a replicated storage such as GFS to maintain a sufficient replication level for its data. Since the time it would take to bring the failed host(s) back online (if that is possible at all) has uncertainty associated with it, let alone that the data may already become outdated or corrupted when the host returns online, I believe it is more likely for the master to initiate the replication on other available hosts to ensure adequate replication level.

- For the third assumption, I assume a reasonably large storage system and that the master implements some level of load-balancing. Thus the new replicas should be distributed across multiple existing hosts, and I believe it should produce a behavior similar to the case where new hosts are always available.
- The remaining assumptions simply reflect the typical replication configuration in GFS (as pointed out in [124]), though the modeling approach is easily extended to other choices of replication levels and replication schemes.

With these assumptions in mind, I use a semi-Markov Chain (SMC) model to capture the dynamics of replica loss and recreation. The reason for choosing an SMC instead of a simpler CTMC is that, while I assume exponential distributions for failure transitions, this assumption is less viable for the completion times of replication processes because, e.g. there are likely hard regulation constraints for how much time the replication can take at most. The resulting data block reliability model is given in Figure 7.4, where the circles and arrows are the states and state transitions respectively. The values in the circles indicate how many replicas of the data block still exist, with state “0” denoting loss of the block. The purpose of the model is to compute the expected time until the data block is lost, i.e. by the model entering state “0”. From this perspective it may be considered a survivability model [133]. The SMC model starts in state “1”, and evolves with the onset of component (host and rack) failures and the progress of replication procedures under the possibility of additional failures. As such, state changes (recall that the new state may be the same as the previous one) occur when a failure occurs (for state “3” since there is no replication ongoing in that state) or when a replication execution ends, either successfully and thereby increasing the number of live replicas, or unsuccessfully after being interrupted by additional failures (for states “2”, “2’ ” and “1”).

With the basics of the model given, I next describe the interpretations of the transitions, how their parameters are determined, and how the model is going to be solved to yield the quantities of interest. The meanings of the transitions are described in Table 7.1, where the notation T_{AB} is used to indicate a transition from state *A* (the previous state) to state *B* (the new state). The notation “2’ ” indicates that the two surviving replicas are stored on hosts on the same rack, while “2” indicates the hosts are on different racks.

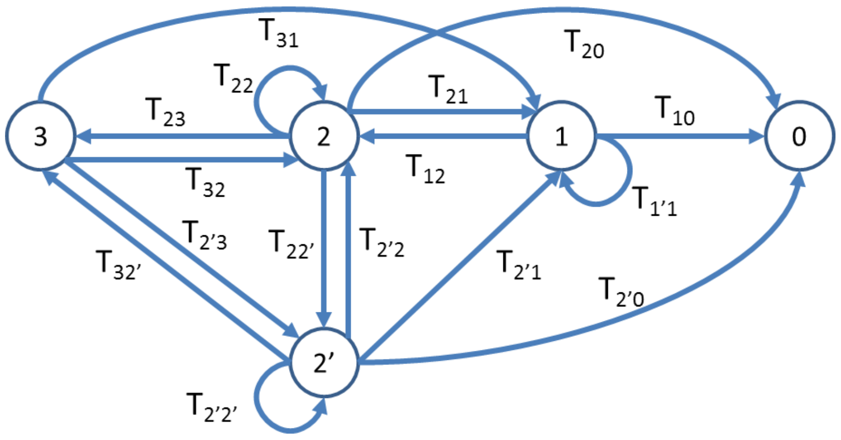


Figure 7.4. Data Block Reliability Model

With the interpretations given, next I present details on how to determine the transition parameters. Note that the metrics of concern in this model are absorbing probabilities and mean time quantities. Thus as described in Chapter 2, the SMP can be parameterized in two ways: 1) parameterize a transition with its time-to-firing distribution; the transition probabilities and mean sojourn time in a state are then computed from such distributions of all the transitions enabled in the current state; 2) parameterize a transition with its probability of firing in the current state as well as the mean time until its firing. The mean sojourn time of the state can then be computed as the weighted sum of transition-conditional mean times. In this part I follow the second approach and present the transition probabilities and conditional mean times in the equations following below. Note that in all these formulas, λ_h stands for the failure rate of a host, λ_r stands for the failure rate of a rack, and the quantity T is the time to complete a replication which is obtained from the data access performance model.

Table 7.1. Block Reliability Model Transition Meanings

Transition	Meaning
T_{32}	One of the replica on the common rack fails.
$T_{32'}$	The replica on the other rack fails, either due to host failure or rack failure.
T_{31}	The rack hosting two replicas (i.e. the common rack) fails causing the loss of two replicas.
T_{23}	The replication succeeds without further failure. The data block recovers the desired replication level.
$T_{22'}$	The replication succeeds but the other existing replica fails.
T_{22}	The replication fails because the new host accepting the replicated data fails.
T_{21}	The replication fails: either because the existing replica performing the replication fails; or because both the other replica and the new host fail.
T_{20}	The replication fails because both existing replicas fail (in sequence). The data block is lost.
$T_{2'3}$	The replication succeeds without further failure. The data block recovers the desired replication level.
$T_{2'2}$	The replication succeeds but the other existing replica fails.
$T_{2'2'}$	The replication fails because the new host accepting the replication fails.
$T_{2'1}$	The replication fails: either because the existing replica performing the replication fails; or because both the other replica and the new host fail.
$T_{2'0}$	The replication fails because both existing replicas fail (either in sequence or due to a rack failure). The data block is lost.
T_{12}	The replication succeeds without further failures.
T_{11}	The replication fails because the new host fails.
T_{10}	The replication fails because the existing replica fails. The data block is lost.

Formulas for transition firing probabilities

$P_{32} = \dfrac{2\lambda_h}{3\lambda_h + 2\lambda_r}$	(7.1)
$P_{32'} = \dfrac{\lambda_h + \lambda_r}{3\lambda_h + 2\lambda_r}$	(7.2)

$$P_{31} = \frac{\lambda_r}{3\lambda_h + 2\lambda_r} \tag{7.3}$$

$$P_{23} = e^{-(3\lambda_h+2\lambda_r)T} \tag{7.4}$$

$$P_{22'} = [1 - e^{-(\lambda_h+\lambda_r)T}]e^{-(2\lambda_h+\lambda_r)T} \tag{7.5}$$

$$P_{22} = \int_0^T \lambda_h e^{-\lambda_h t} e^{-2(\lambda_h+\lambda_r)t} dt = \frac{\lambda_h}{3\lambda_h + 2\lambda_r}[1 - e^{-(3\lambda_h+2\lambda_r)T}] \tag{7.6}$$

$$\begin{aligned}
A1 &= \int_0^T (\lambda_h + \lambda_r) e^{-(\lambda_h+\lambda_r)t} e^{-(2\lambda_h+\lambda_r)t} dt = \frac{\lambda_h + \lambda_r}{3\lambda_h + 2\lambda_r}[1 - e^{-(3\lambda_h+2\lambda_r)T}] \\
A2 &= \int_0^T \lambda_h e^{-(2\lambda_h+\lambda_r)t_1} \int_0^{t_1} (\lambda_h + \lambda_r) e^{-(\lambda_h+\lambda_r)t_2}\, dt_2 dt_1 \\
&= \int_0^T \lambda_h e^{-(2\lambda_h+\lambda_r)t_1}[1 - e^{-(\lambda_h+\lambda_r)t_1}]dt_1 \\
&= \frac{\lambda_h}{2\lambda_h + \lambda_r}\left[1 - e^{-(2\lambda_h+\lambda_r)T}\right] - \frac{\lambda_h}{3\lambda_h + 2\lambda_r}\left[1 - e^{-(3\lambda_h+2\lambda_r)T}\right] \\
P_{21} &= A1 + A2 = \frac{\lambda_h}{2\lambda_h + \lambda_r}\left[1 - e^{-(2\lambda_h+\lambda_r)T}\right] + \frac{\lambda_r}{3\lambda_h + 2\lambda_r}\left[1 - e^{-(3\lambda_h+2\lambda_r)T}\right]
\end{aligned} \tag{7.7}$$

$$\begin{aligned}
P_{20} &= \int_0^T (\lambda_h + \lambda_r) e^{-(2\lambda_h+\lambda_r)t_1} \int_0^{t_1} (\lambda_h + \lambda_r) e^{-(\lambda_h+\lambda_r)t_2}\, dt_2 dt_1 \\
&= \int_0^T (\lambda_h + \lambda_r) e^{-(2\lambda_h+\lambda_r)t_1}[1 - e^{-(\lambda_h+\lambda_r)t_1}]dt_1 \\
&= \frac{\lambda_h + \lambda_r}{2\lambda_h + \lambda_r}\left[1 - e^{-(2\lambda_h+\lambda_r)T}\right] - \frac{\lambda_h + \lambda_r}{3\lambda_h + 2\lambda_r}\left[1 - e^{-(3\lambda_h+2\lambda_r)T}\right]
\end{aligned} \tag{7.8}$$

$$P_{2'3} = e^{-(3\lambda_h+2\lambda_r)T} \tag{7.9}$$

$$P_{2'2} = [1 - e^{-\lambda_h T}]e^{-2(\lambda_h+\lambda_r)T} \tag{7.10}$$

$$P_{2'2'} = \int_0^T (\lambda_h + \lambda_r) e^{-(\lambda_h+\lambda_r)t} e^{-(2\lambda_h+\lambda_r)t} dt = \frac{\lambda_h + \lambda_r}{3\lambda_h + 2\lambda_r}[1 - e^{-(3\lambda_h+2\lambda_r)T}] \tag{7.11}$$

$B1 = \int_0^T \lambda_h e^{-\lambda_h t} e^{-2(\lambda_h+\lambda_r)t} dt = \frac{\lambda_h}{3\lambda_h + 2\lambda_r}[1 - e^{-(3\lambda_h+2\lambda_r)T}]$ $B2 = \int_0^T (\lambda_h + \lambda_r) e^{-2(\lambda_h+\lambda_r)t_1} \int_0^{t_1} \lambda_h e^{-\lambda_h t_2}\, dt_2 dt_1$ $= \int_0^T (\lambda_h + \lambda_r) e^{-2(\lambda_h+\lambda_r)t_1}[1 - e^{-\lambda_h t_1}] dt_1$ $= \frac{1}{2}\left[1 - e^{-2(\lambda_h+\lambda_r)T}\right] - \frac{\lambda_h + \lambda_r}{3\lambda_h + 2\lambda_r}[1 - e^{-(3\lambda_h+2\lambda_r)T}]$ $P_{2'1} = B1 + B2 = \frac{1}{2}\left[1 - e^{-2(\lambda_h+\lambda_r)T}\right] - \frac{\lambda_r}{3\lambda_h + 2\lambda_r}[1 - e^{-(3\lambda_h+2\lambda_r)T}]$	(7.12)
$C1 = \int_0^T \lambda_r e^{-\lambda_r t} e^{-(3\lambda_h+\lambda_r)t}\, dt = \frac{\lambda_r}{3\lambda_h + 2\lambda_r}[1 - e^{-(3\lambda_h+2\lambda_r)T}]$ $C2 = \int_0^T (\lambda_h + \lambda_r) e^{-2(\lambda_h+\lambda_r)t_1} \int_0^{t_1} \lambda_h e^{-\lambda_h t_2}\, dt_2 dt_1$ $= \int_0^T (\lambda_h + \lambda_r) e^{-2(\lambda_h+\lambda_r)t_1}[1 - e^{-\lambda_h t_1}] dt_1$ $= \frac{1}{2}\left[1 - e^{-2(\lambda_h+\lambda_r)T}\right] - \frac{\lambda_h + \lambda_r}{3\lambda_h + 2\lambda_r}[1 - e^{-(3\lambda_h+2\lambda_r)T}]$ $P_{2'0} = C1 + C2 = \frac{1}{2}\left[1 - e^{-2(\lambda_h+\lambda_r)T}\right] - \frac{\lambda_h}{3\lambda_h + 2\lambda_r}[1 - e^{-(3\lambda_h+2\lambda_r)T}]$	(7.13)
$P_{12} = e^{-2(\lambda_h+\lambda_r)T}$	(7.14)
$P_{11} = \int_0^T (\lambda_h + \lambda_r) e^{-2(\lambda_h+\lambda_r)t} dt = \frac{1}{2}[1 - e^{-2(\lambda_h+\lambda_r)T}]$	(7.15)
$P_{10} = \int_0^T (\lambda_h + \lambda_r) e^{-2(\lambda_h+\lambda_r)t} dt = \frac{1}{2}[1 - e^{-2(\lambda_h+\lambda_r)T}]$	(7.16)

Next I present the expected times until the transitions firing. Note that these are NOT conditioned on the respective transition firing: they are computed with respect to the (unconditional) distributions of transition firing (which are in general defective since a transition may not fire, i.e. when preempted by other concurrently enabled ones). The reason I parameterize the transitions with these values, instead of directly specifying the expected sojourn times in each state, is that these quantities can be easily reused in later steps, specifically equation 7.35, to compute several conditional terms; if I specify the expected state sojourn times directly it would be more difficult to perform such conditioning later.

Formulas for expected time to transition firing

$ET_{32} = \frac{2\lambda_h}{(3\lambda_h + 2\lambda_r)^2}$	(7.17)
$ET_{32'} = \frac{\lambda_h + \lambda_r}{(3\lambda_h + 2\lambda_r)^2}$	(7.18)
$ET_{31} = \frac{\lambda_r}{(3\lambda_h + 2\lambda_r)^2}$	(7.19)
$ET_{23} = Te^{-(3\lambda_h+2\lambda_r)T}$	(7.20)
$ET_{22'} = T[1 - e^{-(\lambda_h+\lambda_r)T}]e^{-(2\lambda_h+\lambda_r)T}$	(7.21)
$ET_{22} = \int_0^T \lambda_h t e^{-\lambda_h t} e^{-2(\lambda_h+\lambda_r)t} dt$ $= \frac{\lambda_h}{3\lambda_h + 2\lambda_r}\left\{-Te^{-(3\lambda_h+2\lambda_r)T} + \frac{1}{3\lambda_h + 2\lambda_r}[1 - e^{-(3\lambda_h+2\lambda_r)T}]\right\}$	(7.22)
$D1 = \int_0^T (\lambda_h + \lambda_r) t e^{-(\lambda_h+\lambda_r)t} e^{-(2\lambda_h+\lambda_r)t} dt$ $= \frac{\lambda_h + \lambda_r}{3\lambda_h + 2\lambda_r}\left\{-Te^{-(3\lambda_h+2\lambda_r)T} + \frac{1}{3\lambda_h + 2\lambda_r}[1 - e^{-(3\lambda_h+2\lambda_r)T}]\right\}$ $D2 = \int_0^T \lambda_h t_1 e^{-(2\lambda_h+\lambda_r)t_1} \int_0^{t_1} (\lambda_h + \lambda_r) e^{-(\lambda_h+\lambda_r)t_2}\, dt_2 dt_1$	(7.23)

$$
\begin{aligned}
&= \int_0^T \lambda_h t_1 e^{-(2\lambda_h+\lambda_r)t_1}[1-e^{-(\lambda_h+\lambda_r)t_1}]dt_1 \\
&= \frac{\lambda_h}{2\lambda_h+\lambda_r}\left\{-Te^{-(2\lambda_h+\lambda_r)T}+\frac{1}{2\lambda_h+\lambda_r}[1-e^{-(2\lambda_h+\lambda_r)T}]\right\} \\
&\quad -\frac{\lambda_h}{3\lambda_h+2\lambda_r}\left\{-Te^{-(3\lambda_h+2\lambda_r)T}+\frac{1}{3\lambda_h+2\lambda_r}[1-e^{-(3\lambda_h+2\lambda_r)T}]\right\}
\end{aligned}
$$

$$
\begin{aligned}
ET_{21} = D1 + D2 = &\frac{\lambda_h}{2\lambda_h+\lambda_r}\left\{-Te^{-(2\lambda_h+\lambda_r)T}+\frac{1}{2\lambda_h+\lambda_r}[1-e^{-(2\lambda_h+\lambda_r)T}]\right\} \\
&+\frac{\lambda_r}{3\lambda_h+2\lambda_r}\left\{-Te^{-(3\lambda_h+2\lambda_r)T}+\frac{1}{3\lambda_h+2\lambda_r}[1-e^{-(3\lambda_h+2\lambda_r)T}]\right\}
\end{aligned}
$$

$$
\begin{aligned}
ET_{20} &= \int_0^T (\lambda_h+\lambda_r)t_1 e^{-(2\lambda_h+\lambda_r)t_1}\int_0^{t_1}(\lambda_h+\lambda_r)e^{-(\lambda_h+\lambda_r)t_2}\,dt_2dt_1 \\
&= \int_0^T (\lambda_h+\lambda_r)t_1 e^{-(2\lambda_h+\lambda_r)t_1}[1-e^{-(\lambda_h+\lambda_r)t_1}]dt_1 \\
&= \frac{\lambda_h+\lambda_r}{2\lambda_h+\lambda_r}\left\{-Te^{-(2\lambda_h+\lambda_r)T}+\frac{1}{2\lambda_h+\lambda_r}[1-e^{-(2\lambda_h+\lambda_r)T}]\right\} \\
&\quad -\frac{\lambda_h+\lambda_r}{3\lambda_h+2\lambda_r}\left\{-Te^{-(3\lambda_h+2\lambda_r)T}+\frac{1}{3\lambda_h+2\lambda_r}[1-e^{-(3\lambda_h+2\lambda_r)T}]\right\}
\end{aligned}
\tag{7.24}
$$

$$
ET_{2'3} = Te^{-(3\lambda_h+2\lambda_r)T} \tag{7.25}
$$

$$
ET_{2'2} = T[1-e^{-\lambda_h T}]e^{-2(\lambda_h+\lambda_r)T} \tag{7.26}
$$

$$
\begin{aligned}
ET_{2'2'} &= \int_0^T (\lambda_h+\lambda_r)te^{-(3\lambda_h+2\lambda_r)t}dt \\
&= \frac{\lambda_h+\lambda_r}{3\lambda_h+2\lambda_r}\left\{-Te^{-(3\lambda_h+2\lambda_r)T}+\frac{1}{3\lambda_h+2\lambda_r}[1-e^{-(3\lambda_h+2\lambda_r)T}]\right\}
\end{aligned}
\tag{7.27}
$$

$$E1 = \int_0^T \lambda_h t e^{-(3\lambda_h+2\lambda_r)t} dt]$$

$$= \frac{\lambda_h}{3\lambda_h + 2\lambda_r}\left\{-Te^{-(3\lambda_h+2\lambda_r)T} + \frac{1}{3\lambda_h + 2\lambda_r}\left[1 - e^{-(3\lambda_h+2\lambda_r)T}\right]\right\}$$

$$E2 = \int_0^T (\lambda_h + \lambda_r) t_1 e^{-2(\lambda_h+\lambda_r)t_1} \int_0^{t_1} \lambda_h e^{-\lambda_h t_2}\, dt_2 dt_1$$

$$= \int_0^T (\lambda_h + \lambda_r) t_1 e^{-2(\lambda_h+\lambda_r)t_1}\left[1 - e^{-\lambda_h t_1}\right] dt_1$$

$$= \frac{1}{2} \cdot \left\{-Te^{-2(\lambda_h+\lambda_r)T} + \frac{1}{2(\lambda_h + \lambda_r)}\left[1 - e^{-2(\lambda_h+\lambda_r)T}\right]\right\}$$

$$- \frac{\lambda_h + \lambda_r}{3\lambda_h + 2\lambda_r}\left\{-Te^{-(3\lambda_h+2\lambda_r)T} + \frac{1}{3\lambda_h + 2\lambda_r}\left[1 - e^{-(3\lambda_h+2\lambda_r)T}\right]\right\}$$

$$ET_{2'1} = E1 + E2 = \frac{1}{2} \cdot \left\{-Te^{-2(\lambda_h+\lambda_r)T} + \frac{1}{2(\lambda_h + \lambda_r)}\left[1 - e^{-2(\lambda_h+\lambda_r)T}\right]\right\}$$

$$- \frac{\lambda_r}{3\lambda_h + 2\lambda_r}\left\{-Te^{-(3\lambda_h+2\lambda_r)T} + \frac{1}{3\lambda_h + 2\lambda_r}\left[1 - e^{-(3\lambda_h+2\lambda_r)T}\right]\right\} \tag{7.28}$$

$F1=\int_0^T \lambda_r t e^{-(3\lambda_h+2\lambda_r)t}\,dt$ $=\frac{\lambda_r}{3\lambda_h+2\lambda_r}\left\{-Te^{-(3\lambda_h+2\lambda_r)T}+\frac{1}{3\lambda_h+2\lambda_r}\left[1-e^{-(3\lambda_h+2\lambda_r)T}\right]\right\}$ $F2=\int_0^T(\lambda_h+\lambda_r)t_1e^{-2(\lambda_h+\lambda_r)t_1}\int_0^{t_1}\lambda_h e^{-\lambda_h t_2}\,dt_2dt_1$ $=\int_0^T(\lambda_h+\lambda_r)t_1e^{-2(\lambda_h+\lambda_r)t_1}[1-e^{-\lambda_h t_1}]dt_1$ $=\frac{1}{2}\cdot\left\{-Te^{-2(\lambda_h+\lambda_r)T}+\frac{1}{2(\lambda_h+\lambda_r)}\left[1-e^{-2(\lambda_h+\lambda_r)T}\right]\right\}$ $-\frac{\lambda_h+\lambda_r}{3\lambda_h+2\lambda_r}\left\{-Te^{-(3\lambda_h+2\lambda_r)T}+\frac{1}{3\lambda_h+2\lambda_r}\left[1-e^{-(3\lambda_h+2\lambda_r)T}\right]\right\}$ $ET_{2'0}=F1+F2=\frac{1}{2}\cdot\left\{-Te^{-2(\lambda_h+\lambda_r)T}+\frac{1}{2(\lambda_h+\lambda_r)}\left[1-e^{-2(\lambda_h+\lambda_r)T}\right]\right\}$ $-\frac{\lambda_h}{3\lambda_h+2\lambda_r}\left\{-Te^{-(3\lambda_h+2\lambda_r)T}+\frac{1}{3\lambda_h+2\lambda_r}\left[1-e^{-(3\lambda_h+2\lambda_r)T}\right]\right\}$	(7.29)
$ET_{12}=Te^{-2(\lambda_h+\lambda_r)T}$	(7.30)
$ET_{11}=\int_0^T(\lambda_h+\lambda_r)te^{-2(\lambda_h+\lambda_r)t}dt$ $=\frac{1}{2}\left\{-Te^{-2(\lambda_h+\lambda_r)T}+\frac{1}{2(\lambda_h+\lambda_r)}\left[1-e^{-2(\lambda_h+\lambda_r)T}\right]\right\}$	(7.31)
$ET_{10}=\int_0^T(\lambda_h+\lambda_r)te^{-2(\lambda_h+\lambda_r)t}dt$ $=\frac{1}{2}\left\{-Te^{-2(\lambda_h+\lambda_r)T}+\frac{1}{2(\lambda_h+\lambda_r)}\left[1-e^{-2(\lambda_h+\lambda_r)T}\right]\right\}$	(7.32)

Given these formulas, the SMC can be solved as follows for the probabilities of replica restoration and block loss, as well as the expected times to restoration/loss after a failure:

1) Write down the transition probability matrix of the embedded Markov chain (EMC) of the SMC as follows (the numbers along the matrix borders indicate states):

$$
\boldsymbol{P} = \begin{array}{c|ccccc|}
 & 1 & 2 & 2' & 3 & 0 \\
1 & P_{11} & P_{12} & 0 & 0 & P_{10} \\
2 & P_{21} & P_{22} & P_{22'} & P_{23} & P_{20} \\
2' & P_{2'1} & P_{2'2} & P_{2'2'} & P_{2'3} & P_{2'0} \\
3 & P_{31} & P_{32} & P_{32'} & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1
\end{array}
$$

For later convenience, I also write the matrix ***P*** as:

$$\boldsymbol{P} = \begin{bmatrix} \boldsymbol{Q} & \boldsymbol{C} \\ 0 & 1 \end{bmatrix}$$

where the matrix ***Q*** corresponds to the 4-by-4 submatrix on the left-top corner of ***P***, and ***C*** is a 1-by-4 matrix.

2) The state "0" is the absorbing state, and the initial probability vector of the EMC, i.e. the probabilities of starting in state 1, 2, 2' and 3, is given by:

$$\vec{\boldsymbol{V}} = (0, 0, 0, 1) \tag{7.33}$$

3) To compute the expected time until absorption into state 0, i.e. the mean time to block loss, there are two steps to take. First, it is necessary to compute the expected visit counts of visiting specific non-absorbing states before absorbing into state 0, and this can be obtained by [6]:

$$(EV_1, EV_2, EV_{2'}, EV_3) = \vec{\boldsymbol{V}}(\boldsymbol{I} - \boldsymbol{Q})^{-\mathbf{1}} \tag{7.34}$$

where {(EV_1, EV_2, $EV_{2'}$, EV_3)} are the expected visit counts of state 1, 2, 2' and 3 before the eventual absorption of the EMC into state *i*.

With the expected visit counts computed, it is now possible to compute the expected time until absorption (or block loss), where the previous formulas for $\{ET_{ij}\}$ are utilized in conjunction with the expected visit counts, as:

$$MTDL = \sum_i EV_i ET_i \, , i \in \{1, 2, 2', 3\} \tag{7.35}$$

Before proceeding to the next section, it is worthwhile to discuss how the modeling method described above can be easily extended to different replication levels and replication order. To account for the former, it suffices to expand the model in Figure 7.4 with additional states to indicate the presence of more replicas. To account for a different replication order, e.g. a different strategy that prioritizes creating replicas on identical racks first, the effect can be captured by changes in the derivation of the formulas and the structure of the transition matrix. For example, if a new replica should first be placed on the same rack as the remaining one then P_{12} would be replaced by $P_{12'}$, with the probability and expected time recomputed accordingly.

7.3.3 Data block access performance model

The data block access performance model captures the performance of accessing data at the hosts, potentially under resource contention from other concurrent accesses. It also accounts for the execution of replication, which involves transferring data from/to the host, and therefore provides the mean completion time of the replication process. In constructing this model I make the following assumptions:

1) There are fairness-preserving mechanisms in place on the hosts, so that the data transfer tasks from different sources are treated equally in the amount of resource they are allowed to consume.

2) Tasks vary in the total amount of data they will transfer. Along this dimension, I assume the tasks can be classified into two categories:

   - Small tasks (henceforth termed Type I tasks) from interactive user applications (e.g. Gmail) that contain small amount of data and are latency-sensitive.
   - Large tasks (henceforth termed Type II tasks) from batch-processing applications (e.g. MapReduce) that contain much larger volume of data and require good throughput. They are however not sensitive to latency.

3) The arrival of user tasks to a host form several Poisson streams, i.e. the tasks can be classified into categories (as in the above assumption), with the random time between successive task arrivals in a category following an exponential distribution.

Again I provide some discussion about these assumptions:

- For the first assumption, even though the actual scheduling mechanism for allocating access to data in GFS is not made public, I believe the GFS configuration would try to avoid large task data flows holding resource up at the expense of small tasks. This setup is likely considering the small tasks are much more likely to have come from latency-sensitive users which will be badly affected by such kind of delay. Without going into details on such scheduling issue (which is a field in its own right), I would assume an equal sharing of resources, which is a reasonable scheduling scenario in GFS.
- The second assumption is made to facilitate analysis as the latter can then focus on the response time for the Type I and throughput for the Type II. It is mainly based on

[56] which, while not focusing on the storage aspect, supports bimodality in the task size distribution.

- The third assumption draws its source from the Palm–Khintchine theorem [134] which states that the combined behavior of a large number of (maybe non-Poisson) renewal processes would approach that of a Poisson process. I believe that the “large number” requirement is not a difficult thing to achieve in an environment like GFS, and that in the long run the individual task arrivals (from the many people working at Google, who are likely to routinely use GFS and not changing their project subjects too frequently) can be viewed as renewal processes.

Based on the above assumptions, I choose to model the data block access performance using an open queuing network. By its nature the queuing network formalism is suited for modeling and analyzing the performance aspect of a system subject to resource contention. To model a data host, the main idea is to look at incoming workload at the level of user tasks (instead of individual requests as perhaps more common with queuing network-based modeling), and then consider the pattern of resource contention that arises at the host network IO, CPU and disk IO.

Because the model is established at the level of user tasks, and because of the aforementioned assumption 1), it is reasonable to treat the network IO and disk IO as Processor-sharing (PS) queuing stations. Even though the actual scheduling at these two places, especially the disk IO, could be more complex, the long-run resource-sharing behavior of them could be close to that of a PS queue. Figure 7.5 illustrates the resulting basic model.

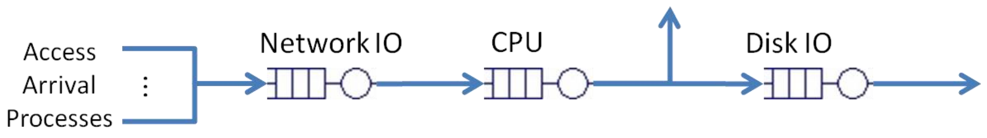


Figure 7.5. Basic Host Performance Model

The above model captures the behaviors of the devices by separate PS queues. The user tasks arrive from the network to the network IO first. Once it goes through it reaches memory and is processed by the CPU. After this stage some of the tasks may not need disk access since the data they seek may be in memory already (e.g. if other tasks are also accessing the same data block(s)). In such cases the tasks would finish and leave (along the direction of the vertical arrow). Note that in reality the network IO should be traversed twice by both the task and the reply to it, but in both read and write accesses this could be modeled by just one traverse (as in the model) because:

1) In read accesses, the first traverse would consists of the request itself thus very small in volume compared to the actual data involved in the second traverse.
2) In write accesses, the opposite happens as the reply would be just a confirmation which should be very small in data volume compared to the data actually written.

Hence based on the above reasoning it suffices to model the traverse just once.

The purpose of this model is to study the response time distribution of Type I tasks and throughput of Type II tasks. However, there are three issues regarding how to compute these:

- The first issue is that, while a product-form queueing network (as is used in this case) can produce the throughput quantity easily, the value is for an infinitely long period of time, over the aggregation of all tasks of that type. In other words, it is not particularly useful for estimating the throughput of a single task of that type.

- The second issue is to account for the impact from data block replication executions. While such executions (caused by component failures) may be relatively infrequent and thus do not have significant impact on the long-running behavior of Type II tasks, they are likely to impact the response time distribution of Type I tasks.
- The third and most important issue is that the mean response time provided by the model is insufficient for studying the behavior of Type I tasks. This is because in real systems typically the mean response time for small tasks is low and the tail/high percentile of the response time distribution is of greater interest. Thus it is necessary to develop a method to obtain, at least approximately, the response time distribution from the queueing model.

To solve the first issue, I note that the model also provides the mean response time of a Type II task, and so it is possible to obtain the throughput of the individual task (instead of that of the whole task class provided by the queueing network directly) by dividing the task size with the mean response time. This idea is similar to the "operational analysis of queueing network" ideas put forward in [135]. Given these thought, I would instead present mean response time results for Type II tasks in the result section.

The following two subsections describe how to solve the other two issues.

7.3.3.1 Modeling the Effect of Replication Execution

To capture the effect of replication execution on the performance of data access tasks, particularly Type I, the model of Figure 7.5 is extended to account for the dynamics of the replication executions as in Figure 7.6.

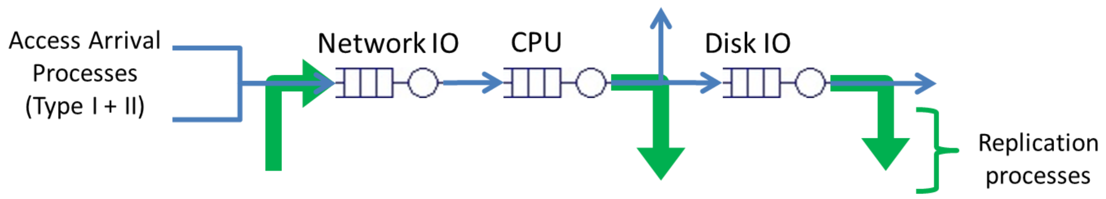


Figure 7.6. Extended Block Access Performance Model

Compared with Figure 7.5, Figure 7.6 defines a model that includes an additional arrival process. This new process captures the progress of the replication processes, and the resulting model provides the approximate response time distribution of the Type I access when subjected to the contention from both the replication processes and Type II tasks. The details on how to obtain the approximate distribution will be covered later. The rate of the replication execution arrival is derived below:

1) Let us first focus on an arbitrary data block that is replicated thrice to three different machines on two different racks. If we look at a specific replica (suppose it is numbered as 1), in a stable condition of the system there are two possible distributions of this replica with its peers, as Figure 7.7 shows:

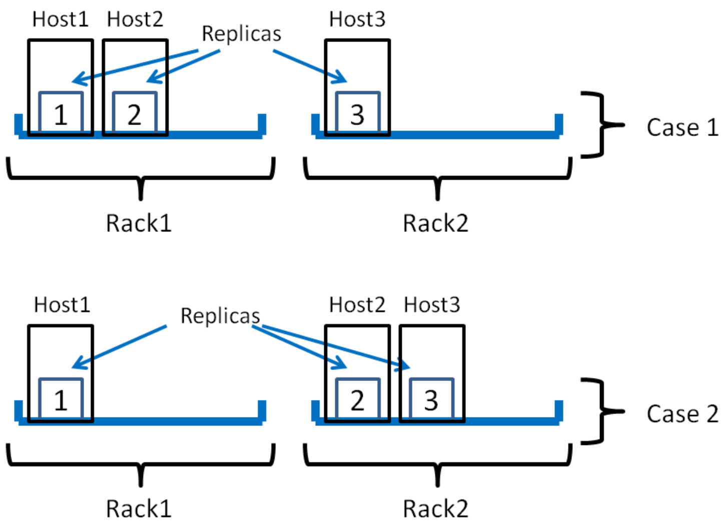


Figure 7.7. Two Cases of Replica Distribution

It is reasonable to assume the probability for either case would be the same, i.e. 0.5.

2) Now I will focus on replica 1 and derive the rate at which request for replication execution arrives at this replica (because some other replica(s) of this block is lost). To do this, consider the two cases in Figure 7.7 separately:

   a) For case 1, replica 2 may be lost due to Host2 failure, while replica 3 may be lost due to Host3 or Rack2 failures. Regardless of which happens, replica 1 would have a 50% chance to be chosen to perform replication. This is because in this case there cannot be a simultaneous loss of two replicas (recall that I am focusing on replica 1 while assuming it is not one of the replicas that are lost), and that replica 1 and the other surviving replica is in symmetric positions. Thus the rate of new replication execution coming to replica 1 would be:

$$Rate_{replicate\_case1} = 0.5(\lambda_h + \lambda_r) + 0.5\lambda_h = \lambda_h + 0.5\lambda_r \tag{7.36}$$

   b) For case 2, if either Host2 or Host3 fails (but not Rack2) then replica 1 is still in the same position as the other surviving one. However if Rack2 fails then replica 1 would, as the sole remaining replica, definitely be called upon to make the replication. Hence the rate in this case would be:

$$Rate_{replicate\_case2} = 0.5 \times 2\lambda_h + 0.5\lambda_r = \lambda_h + \lambda_r \tag{7.37}$$

3) Combining the results from the previous two steps, the rate of replication execution starting on replica 1 caused by this particular data block would be:

$$Rate_{replicate} = 0.5 \times (r_{case1} + r_{case2}) = \lambda_h + 0.75\lambda_r \tag{7.38}$$

4) Next, assuming the total number of data block replicas hosted on the same machine as replica 1 (including itself) is *C*, the total rate of execution starting would simply be:

$$RATE_{replicate} = CRate_{replicate} = C(\lambda_h + 0.75\lambda_r) \tag{7.39}$$

The extended model provides the expected response time of Type II task and the expected completion time of replication execution which is used in the data block reliability model described in the previous section (refer back to equations (7.1)~(7.33)). It also forms the basis for approximating response time distribution for Type I tasks, to be described next.

7.3.3.2 Approximating Type I Response Time Distribution

To solve the third issue, I make three further assumptions:

1) The actual processor-sharing scheduling is implemented by assigning quotas to the tasks waiting to be processed in a round-robin fashion. This means that an internal queue is maintained for the data from each task, and the device (CPU, disk IO, etc.) processes a certain amount (a “slice”) of data from each internal queue before moving to the next queue.
2) The data from a Type I task can be processed in its entirety in one slice by the processing device.
3) The distribution of the processing time for one slice is approximated normal with a small variance.

The first additional assumption is reasonable because it is actually how processor-sharing is typically implemented in practice (as processor-sharing itself is an idealized schedule). I consider the second assumption reasonable because of both the latency-sensitiveness of the task type and its small amount of data. Due to these two reasons I think it is more reasonable to complete such a task in one round to reduce the unnecessary delay the task may experience. Finally, the last assumption is partly for ease of computation. However other distributions can also be used so long as the variance is small, which I believe is reasonable given that during the processing of a small amount of data there should be very little interference from other sources, so that the time should roughly be determined by the ratio between data size (which is small) and the device processing bandwidth.

With these additional assumptions, I proceed to obtain the approximated response time distribution in the following steps:

1) From the open queueing network model, obtain the steady-state probability distribution of the number of tasks existing at a device. This can be done easily from established queueing network analysis methods and is exactly the same distribution an incoming Type I task (recall the arrival follows a Poisson process) would see [136].
2) Consider this incoming Type I task. Because this task can be processed in one round by the device, its response time would consist of two parts:
    a) The processing time of the task itself (its own time slice).
    b) The total time to complete the next slice for each task already at the device. If there is no task waiting this quantity would be zero; if there are one or more, then again because of the Poisson arrival the task would arrive at any point during the processing of the first waiting task with equal probability. Thus the total time to wait before start its processing would equal the sum of slices from all but the first existing tasks, plus a random quantity from the first one.

    Figure 7.8 illustrates the above points. Notice that the second and third pictures illustrate how the new arrival may find some other task execution at various stages.

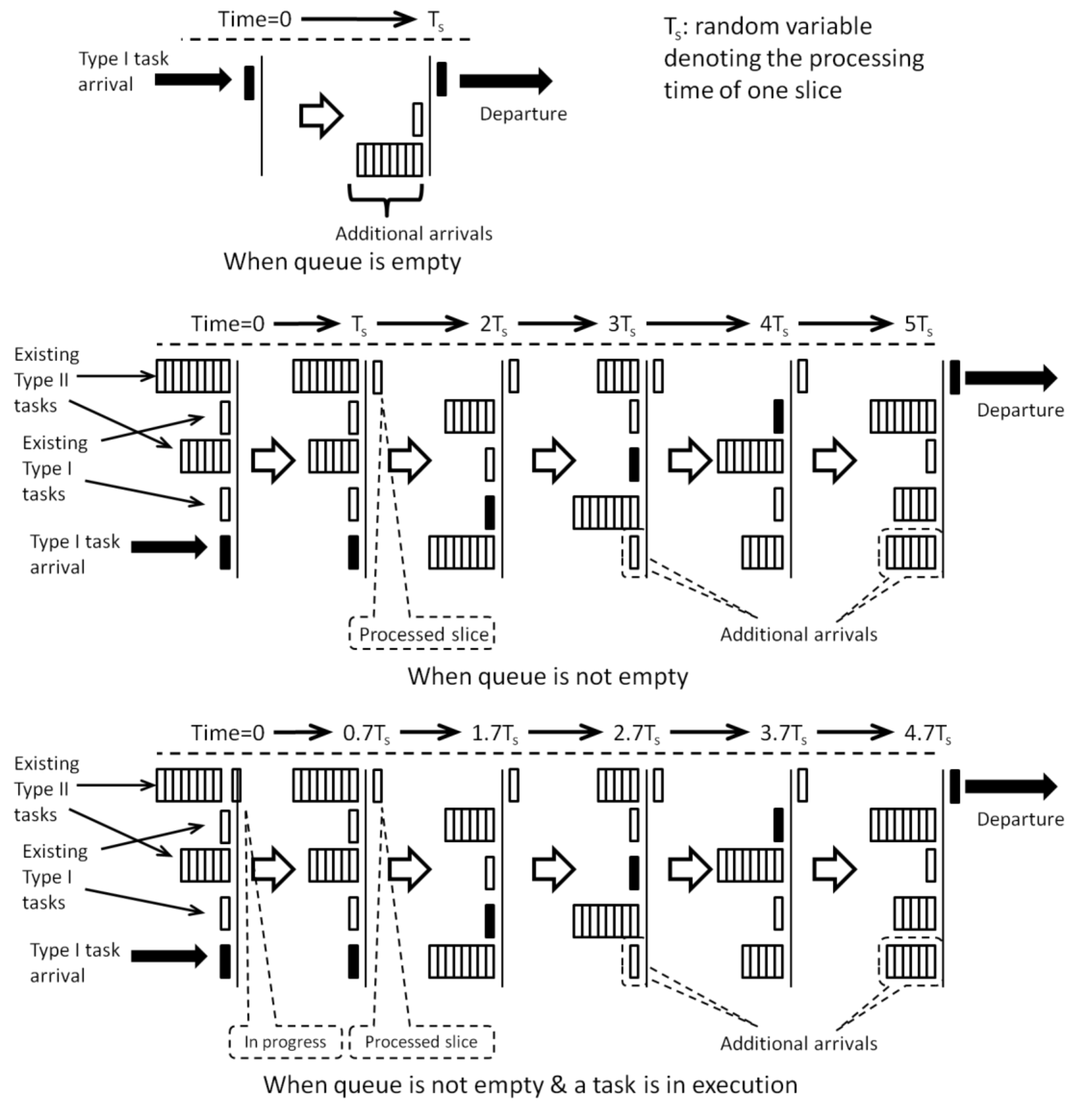


Figure 7.8. Illustration about Response Time Approximation

3) Denote the processing time of one slice as the random variable T_s and the number of other tasks that the arriving Type I task finds at the queue as the random variable N_A, then the response time of this arriving task, which is another random variable, can be expressed as:

$$RT = \left[\left(N_A + frac(N_A)\right) \cdot T_s\right] \cdot P(N_A) \tag{7.40}$$

The term "$P(N_A)$" is the probability of finding N_A tasks at the device upon arrival and is obtained from step 1 based on PS queue formulas, and equation 7.40 can in this respect be considered as an application of the law of total probability. The term "*frac*(N_A)" is a function of N_A and assumes either: a) unity if $N_A = 0$; or b) a uniform random value in the range (0, 1). The second case arises due to the explanation in 2).b. above. The distribution of *RT* can then be evaluated either through sampling or through direction convolution of the component distributions (i.e. T_s). The latter can be difficult in general, but may be straightforward in some cases, e.g. when T_s follows a normal distribution in which case the part within the bracket results in another normal distribution.

It is worthwhile to clarify the purpose of the approximation method at this point. The goal is not to compute the response time distribution of a PS queue, but rather to compute the distribution of a fair-queueing device as used in practice. Hence it is possible to introduce the notion of the "slices". The PS queue is simply employed to facilitate this procedure (by providing the device task number distribution as in step 1 described above).

Section 7.4 will look at the accuracy of the above approximation and present some numerical results regarding its accuracy. The accuracy of the method will also be further (indirectly) examined in the major quantitative results in Section 7.5.

7.3.4 Master availability model

The master availability model provides estimates on the availability of the master functionality. First I will describe the assumptions in the modeling.

1) The master and its backups (there are two backups) fail independently and the times-to-failure follow exponential distributions.
2) If the master fails and there is a backup available, the backup will switch in and start acting as the master. There is no interruption to metadata access functionality in this case. On the other hand, if the master fails while the backups have also failed, then metadata access becomes unavailable.
3) Recovery of master/backups are independent and times-to-recovery follows exponential distributions.
4) The network between the master and users, as well as that among the masters, does not suffer failure.

The first assumption is essentially the same as earlier assumptions. Regarding the rest:

- The second one is based on the fact that, as in GFS, the backups are already synchronized with the master and so it should be a very simple and quick matter to elevate one of the backup to the role of active master.
- For the third assumption, while it is true that in reality the recovery times are not necessarily exponentially distributed, it has been described in several works (e.g. [25]) that when the recovery is much faster than additional failure occurrence (which I believe is reasonable) using exponential failure times yields numerical results that are very close to those from more precise, but also more complex, models. Given that the benefit of using the latter does not seem to justify its complexity, in this thesis I will use a simple continuous time Markov chain (CTMC) for master availability modeling.

- The last assumption may be a relatively strong one. One of the reasons to make this assumption is to simplify the model and concentrate better on the structure of the GFS system. It may also be argued that, in real datacenter and/or computing clusters (where GFS is likely to be deployed) the network connections, especially those providing access to a component as critical as the master, are typically multihomed and therefore connection is unlikely to be totally lost.

Given the above assumptions, the CTMC model for master availability is presented in Figure 7.9.

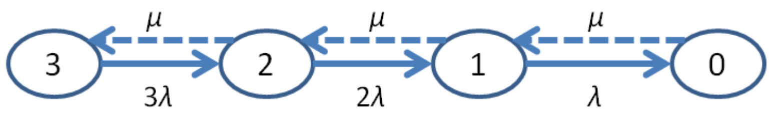


Figure 7.9. Master Availability CTMC Model

In the model, the labels of the states indicate how many nodes (master + backups) are still available. The "λ" and "μ" terms are the failure and recovery rates of individual master nodes respectively. The dashed arcs corresponding to transitions induced by completion of recoveries, and the solid ones are transitions caused by node failures. Solution of the model yields steady-state availability (i.e. the probability of not being in state 0) of the master subsystem.

7.3.5 Master Performance Model

The master performance model captures the metadata request processing that occurs on the active master and the backups. The relevant assumptions are:

1) There are fairness-preserving mechanisms in place on the master and backups, so that tasks from different sources are treated equally in the amount of resource they are allowed to consume.
2) Tasks that send metadata requests to the master can be classified into two types: small tasks that correspond to latency-sensitive user programs and are small in size (Type I) and large tasks that correspond to batch work that are much larger (Type II).
3) The two task types arrive according to their individual Poisson streams.
4) The network would not pose a bandwidth bottleneck; it simply adds certain random amount of delay.

The first three assumptions are essentially the same as those made in the data access performance model and share the same rationales as those discussed there. The fourth assumption is due to the fact that the tasks in this model contain transfer of metadata that are (compared to the actual data or the network capacity) very small, so I assume it is very unlikely for the network to become a bottleneck in terms of bandwidth. Also, the main purpose of this model is to produce an approximated distribution for the response time of Type I tasks, therefore throughput is not of primary concern.

Based on the assumptions, the main idea for the master performance model is to use queueing networks to obtain the response time distribution for Type I tasks. However, while the queueing network formalism is suitable for modeling queueing delays, generally speaking only mean quantities can be produced efficiently. This is deficient for two reasons: 1) because the goal is to obtain the distribution; 2) because GFS master configuration is that of a synchronized update, to evaluate the overall response time it is necessary to compute the maximum value of the parallel metadata operations on the master and the backups. This in turn requires the processing time distribution of each branch of the concurrent operations. To solve this issue, I would again use approximation schemes discussed earlier in the data block performance model (Section 7.5.2). With this in mind, the queueing network model is presented in Figure 7.10. The model uses queueing networks with one PS queue for each processing device (Network IO, CPU and Disk IO) on a (active or backup) master node. The master subsystem network, which is treated as an infinite-server (IS) queue, is not shown.

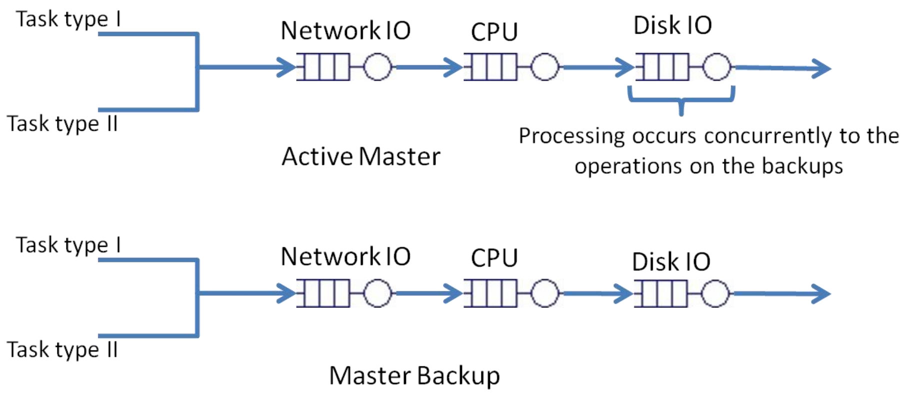


Figure 7.10. Master Performance Model

Note that during processing of a task the active master needs to log the operation to its disk, but this occurs in parallel to the operations at the backups. Thus the whole processing of a request at the master subsystem can be divided into two steps:

1) The request arrives at the active master through its Network IO and receives processing at its CPU. After the CPU processing the first step is done.
2) The request is logged into the active master disk (through Disk IO). Concurrently, copies of the request are also sent via the master subsystem network to the two backups. Upon arrival, a request copy would go through processing at the Network IO, CPU and Disk IO devices of the corresponding backup node before a reply is sent

back to the active master. The second step is complete only when the logging at the active master as done AND the two replies are received by the active master.

Because the ultimate quantity of interest is the (approximate) overall response time distribution, the distribution at each device on each node is needed. These are provided by the solution to the model in Figure 7.10 based on the approximation method. Then the response time distribution of each node and each step can be obtained from the approximate response time distributions at the devices, either through convolution of the device distributions (e.g., if the normal distributions are used this be easy) or via sampling methods. Denote the variables for the response times at the devices as $\{T_{NIO\_i}, T_{CPU\_i}, T_{Disk\_i}\}$, $i \in$ {Active Master: 0, Backup: 1, Backup: 2} and the transmission delay of the master subsystem network as T_{Net}, the overall response time variable of the master subsystem can be obtained as:

$$
\begin{aligned}
&RT_{Step1} = \left(T_{NIO\_0} + T_{CPU\_0}\right) \\
&RT_{Backup1} = T_{NIO_1} + T_{CPU_1} + T_{Disk_1} \\
&RT_{Backup2} = T_{NIO\_2} + T_{CPU\_2} + T_{Disk\_2} \\
&RT_{Step2} = \max\{T_{Disk_0}, \left(T_{Net} + RT_{Backup1}\right), \left(T_{Net} + RT_{Backup2}\right)\} \\
&RT_{master} = RT_{Step1} + RT_{Step2}
\end{aligned}
\tag{7.41}
$$

where the max{} term is over the three operation logging paths: the active master disk and the two backups. The distribution of RT_{master} may be evaluated either directly or through sampling.

7.3.6 Data access availability model

The data access availability model combines the master availability from the master availability model and the data block reliability from the data block reliability model to compute the overall access availability of a data block.

To translate data block reliability into the data block availability, it is necessary to introduce a recovery time quantity for the data block once it is lost from the replicated storage. While the previous models of storage backup can be used, here I will simply introduce an arbitrary distribution with mean ET_{rec} for the recovery time of a data block. Notice that to compute the steady-state availability only information about the mean is needed. With the recovery time given, the availability of the data block is given as:

$$A_{block} = ET_{fail}/(ET_{fail} + ET_{rec}) \tag{7.42}$$

where the term ET_{fail} is the mean time to block loss obtained from the block reliability model.

With the steady-state availability of a data block given, its access availability is simply the product of master subsystem availability and the availability of the block, i.e.,

$$A_{access} = A_{block} \times A_{master} \tag{7.43}$$

7.3.7 Data access performance model

Similar to the data access availability model, the data access performance model also combines the access response time distributions from the master performance model and block access performance model to compute the overall response time. Again this can be done either through direct convolution of distributions or sampling. The mean response time of Type II tasks can be directly obtained from the data block access performance model.

7.4 Accuracy of the distribution approximation method

In the previous section I described an approximation method to obtain the response time distribution from a product-form queueing network. In this section I provide some quantitative results regarding the accuracy of this method.

The PS scheduling specifies that the processing capacity a task receives from the station equals the total station capacity divided by the number of tasks at that station. This specification cannot be implemented efficiently in practice, and so what is typically done is a version named "fair-queueing". In such an implementation, the requests of each task (determined by some criteria such as source-destination information) are assigned to a separate internal queue. The processing device would assign some units of processing time (a "slice") to each non-empty internal queue in a round-robin fashion. The primary goal of such a scheme is to achieve "fairness" of processing resource allocation among the tasks, and there are variations of fair-queueing that emphasize different notions of "fairness".

In this thesis I will only look at the basic version of fair-queueing, and the goal is to check how close the response time distribution computed from the proposed method can approximate that of a fair-queueing scheduling device. The investigation is carried out by comparing discrete-event simulation results (performed using a Java program) of two fair-queueing devices in tandem and the numerical results from a corresponding analytical PS queueing station model and the approximation method. The scenario setup is a processing device that is subject to two Poisson task arrival streams, each with different parameters. The parameters are listed in Table 7.2. The task sizes follow normal distributions with CoV 0.01, and the size of the time slice is kept as the same as the size of the first stream tasks. The reason to choose this two-stage tandem setup is that this model piece is representative of the forms used in the major models.

Table 7.2. Approximation Method Parameter Setup

Stream ID	Arrival Rate (min^{-1})	Mean Size	Utilization
1	λ_1	0.05	$\rho_1=0.05\lambda_1$
2	λ_2	50	$\rho_2=50\lambda_2$

Figure 7.11 displays the approximation error in percentage between the empirical response time distribution from the simulation and the distribution generated by the approximation method. The simulations are executed until 2,000,000 samples are generated for the response time of stream 1. There is a burn-in period of 1,000 samples, and subsequent samples are generated at a sub-sampling period of 1,000 samples. As the results show, the accuracy of the approximation depends on the total utilization of the device as well as the relative magnitude of utilization between the target stream (stream 1 in this case) and the other, larger-volume traffic. While the accuracy is not satisfactory when the total utilization is small and when the small-size stream has a larger utilization (as in the first three pictures), it improves as the utilization increases and/or the portion of the small-size stream in utilization decreases (as in the later three pictures). Thus the approximation would be suitable for situations where the small tasks face nontrivial resource contention from larger tasks. As [56] points out, in the case of GFS (and perhaps in similar systems) an overwhelming majority of data transfer comes from the large flows. Hence I believe the approximation, while not perfect, can be quite useful in the context this thesis investigates.

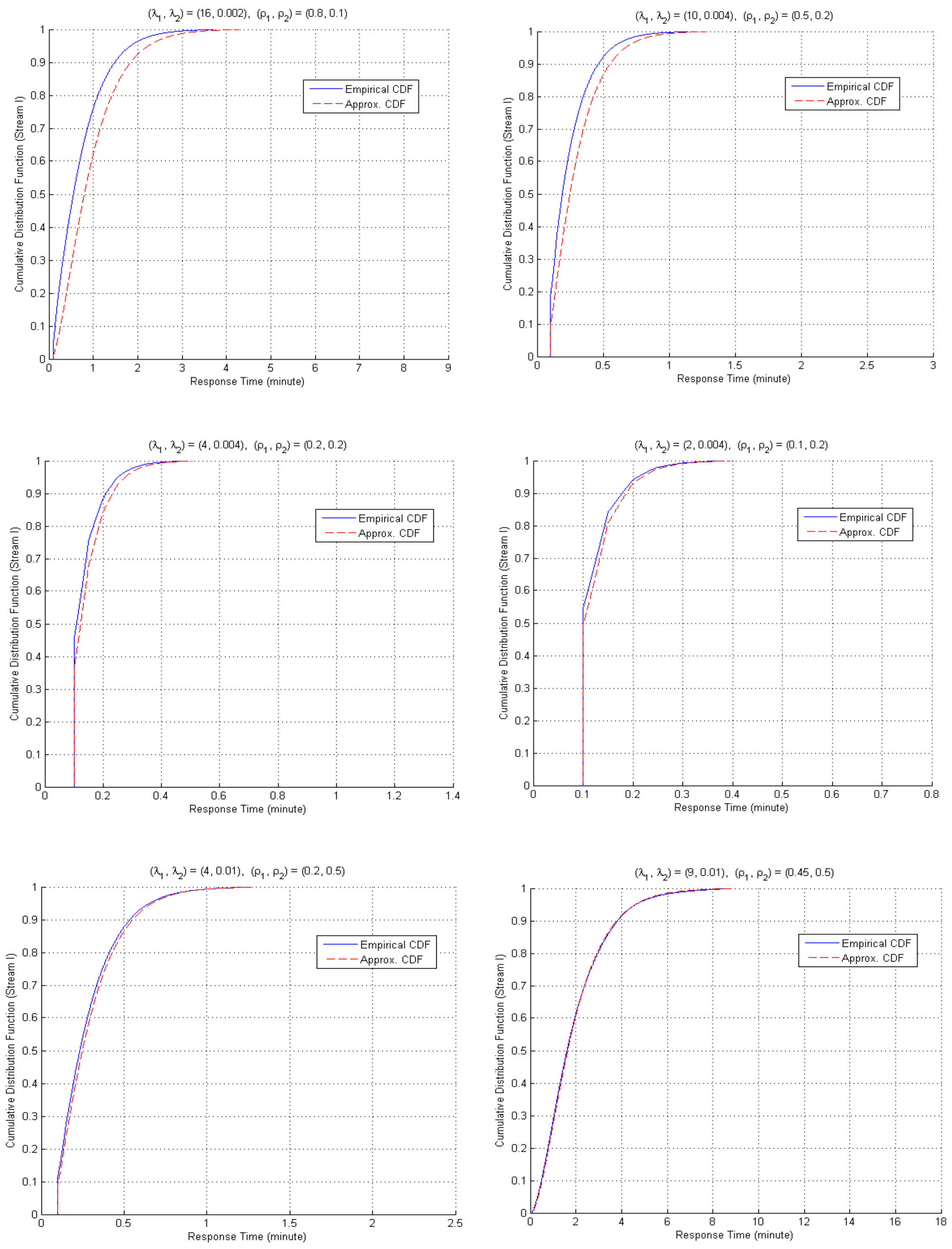


Figure 7.11. Accuracy Results

Before proceeding to the next section with the main numerical results, I would like to point out that the assumption of small tasks that can be completed in one slice assignment (as is the case for Type I) is essential to this approximation. In fact, for the other two types of tasks the response time distribution computed using the same approximation method would differ much from their simulation counterparts. Also, as the results suggest the approximation error increases as the slice size increases. This is because as the slice size increases the PS approximation of the fair-queueing becomes less effective. Therefore this approach is only used for the response time distribution of Type I tasks.

7.5 Main Numerical Results

In this section I will present the main numerical results concerning the replicated storage model. The overall purpose is to validate the developed models with simulation across a range of parameter settings. First I describe the base values for the parameters.

7.5.1 Base Parameters

Table 7.3 provides the base parameter values used in this section. In the table, "CoV" stands for coefficient of variation, and "Type I" and "Type II" refer to the two task types respectively. The "CPU Processing Rate" stands for how many tasks the CPU can process in one minute, and is typically a large value since CPU tends to be less loaded in a storage system. The time-to-failure data is obtained from [132] and the host processing power parameters are gleaned from [124]. The performance parameters for the master are for individual master nodes, and they are obtained by modifying their host-side counterparts based on the assumption that the master nodes are likely to have better capacity compared to storage nodes due to their importance. The data size parameters are determined partially from [124] and partially on the nature of the data, i.e. Type II tasks are larger than Type I tasks.

Table 7.3. Base Parameter Values

Parameter	Value	Parameter	Value
Type I task arrival rate	5.0e3 (sec^{-1})	Type II task arrival rate	0.3 (sec^{-1})
Type I metadata average size	5 (KB)	Type II metadata average size	250 (KB)
Type I metadata size CoV	0.1	Type II metadata size CoV	0.2
Type I data average size	50 (KB)	Type II data average size	1.0 (GB)
Type I data size CoV	0.1	Type II data size CoV	0.2
Master network IO bandwidth	10 (Gbps)	Host network IO bandwidth	1 (Gbps)
Master CPU Processing rate	5.0e5 (sec^{-1})	Host CPU Processing rate	1.0e5 (sec^{-1})
Master disk IO bandwidth	1.0 (Gbps)	Host disk IO bandwidth	500 (Mbps)
Master network average delay	0.5 (ms)	Host network average delay	1.0 (ms)
Master network delay CoV	0.1	Host network delay CoV	0.3
Master mean time to failure	8 (months)	Host mean time to failure	4.3 (months)
Master mean time to recovery	1 (hour)	Rack mean time to failure	10.2 (years)
Prob. of Type I data in memory	0.75	Prob. of Type II data in memory	0.5
Replication data average size	64 (MB)	Replication data size CoV	0.3

In the context of the models, the Markov chain-related ones can directly use the failure and recovery parameters. On the other hand, the queueing network-based performance models require synthesis of mean processing times (or alternatively processing rates) of individual tasks based on the data sizes and device bandwidth values. In the case of CPUs the rates are directly available. In the case of network and disk IOs the mean time values are obtained by dividing the respective task sizes with the device bandwidth, and the processing rates are the reciprocals of the mean values. The resulting processing rates for the queueing network models (i.e. the performance models) are given in Table 7.4. Note that the network delays (the last two entries) are assumed to be unaffected by the size of individual packets.

Table 7.4. Queueing Network Model Processing Rates

Queueing Station	Type I (ms^{-1})	Type II (ms^{-1})	Replication (ms^{-1})
Master network IO	250	5	N/A
Master CPU	500	10	N/A
Master disk IO	25	0.5	N/A
Host network IO	2.5	1.25e-4	2.604e-3
Host CPU	100	2	15.625
Host disk IO	1.25	6.25e-5	4.883e-4
Master network	2.0	2.0	N/A
Host network	1.0	1.0	1.0

Finally, in terms of data distribution across the hosts, I assume the system consists of N_{Host} = 20 hosts and each host has $N_{ReplicaPerHost}$ = 300 data replicas on it. This puts the total number of replicas in the system at $N_{Replica} = N_{ReplicaPerHost}*N_{host}$ = 6000. Given a replication factor of three, this means a total of $N_{Block} = N_{Replica}/3$ = 2000 data blocks are stored in the system. I further assume half of the data blocks are targets of Type I tasks and the rest of them for Type II tasks, and that an incoming task is equally likely to read or write to the target block. This means the number of replicas (blocks) for Type I and II accesses are both $N_{ReplicaPerType}$= 3000 ($N_{BlockPerType}$ = 1000). In the case of read, I assume the incoming task goes equally likely to any item of the corresponding type; in the case of write it will contribute to the arrival rate at all three replica hosts. These assumptions provide a case of reasonable complexity for the study, but are in no way essential for the model to work. Under these assumptions, the arrival rates of the two types of tasks at the hosts can be computed based on the following procedure (those for the master are already given in Table 7.3):

- For a read task, because it only needs to contact one replica, its probability of arriving at a given host would be $P_{readArrival}$ = 1/20. The reason is that a task may visit any replica (of the same type, thus there are 3,000) with the same probability, and since a host has $N_{ReplicaPerHostI}/2 = 150$ replicas (or blocks since no two replicas of the same block are stored on the same host) of that type, the probability of this host being involved would be 150/3000 = 1/20.

- For a write task, because it needs to contact all three replicas, the probability that it would contribute to the workload at a particular host would be $P_{writeArrival} = 3/20$, since $(N_{ReplicaPerHostI}/2)/N_{BlockPerType} = 150/2000 = 3/20$ is the probability that the task accesses a data block that has a replica on the host.

Given the above reasoning, the arrival rates of the two types of tasks at a given host would be:

$$\begin{aligned}\text{Rate}_{\text{Task\_Arrival\_TypeI}} &= (P_{readArrival} + P_{writeArrival})*\lambda_{\text{I}}/2 = 0.1\lambda_{\text{I}} = 0.5\ \text{ms}^{-1}\\ \text{Rate}_{\text{Task\_Arrival\_TypeII}} &= (P_{readArrival} + P_{writeArrival})*\lambda_{\text{II}}/2 = 0.1\lambda_{\text{II}} = 3.0\text{e-}5\ \text{ms}^{-1}\end{aligned} \tag{7.44}$$

for each type, where λ_{I} and λ_{II} are the arrival rates for the two types as in the first row of Table 7.3. The division by 2 is due to the assumption that read and write requests are equally likely to happen. Also, the arrival rate of replication execution, based on (7.39), would be:

$$\text{Rate}_{\text{Replication\_Arrival}} = N_{ReplicaPerHost}*(\lambda_h+0.75\lambda_r) \approx 2.7616\text{e-}8\ \text{ms}^{-1} \tag{7.45}$$

7.5.2 Data Block Access Performance Model

The data block access performance model is validated first because, as the import graph in Figure 7.3 indicates its analysis does not depend on other models. As described earlier in Section 7.3, the model provides:

- The approximated response time distribution of Type I tasks. This is obtained based using the approximation method. The model inputs needed for this purpose are the utilization values at the three queueing stations in Figure 7.6.

- The mean response time of Type II tasks. This can be used to estimate the throughput value of Type II tasks.
- The mean completion time of a replication execution. This quantity can be directly obtained from the model as the mean response time of the replication process.

Note that the first metric is obtained at the granularity of a single host. The overall response time distribution of Type I tasks will be composed by the overall access performance model to be described later.

Figure 7.12 compares the Type II mean response time and mean replication completion time results from the model and the simulation while varying arrival rates. All simulations are executed until at least 1,000 samples of task completion are generated for all arrival streams (task Type I, Type II and replication), and the confidence interval is at a confidence level of 95%. There is a burn-in period of 1,000 samples, and subsequent samples are sub-sampled at a period of 1,000 samples. The results are produced from a range of Type II task arrival rates.

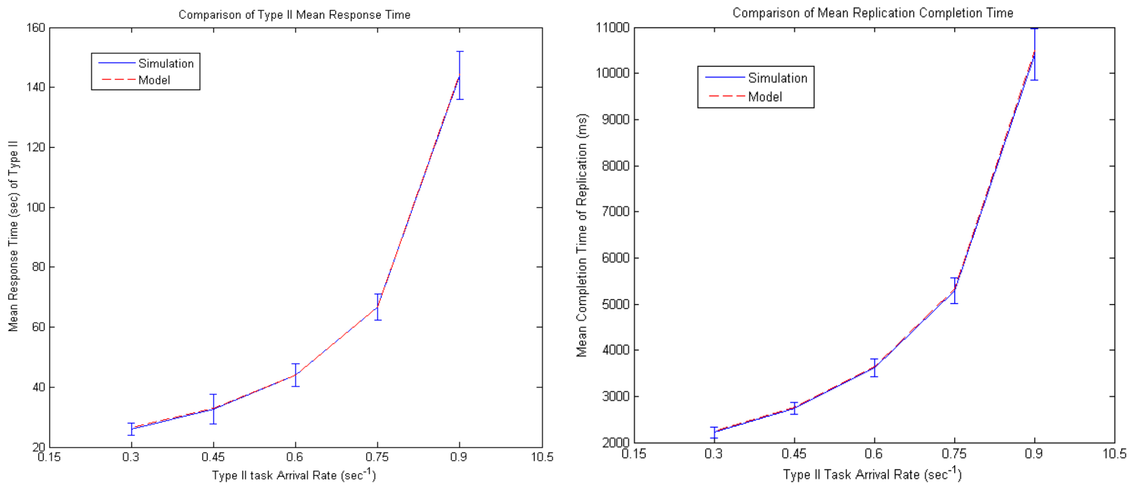
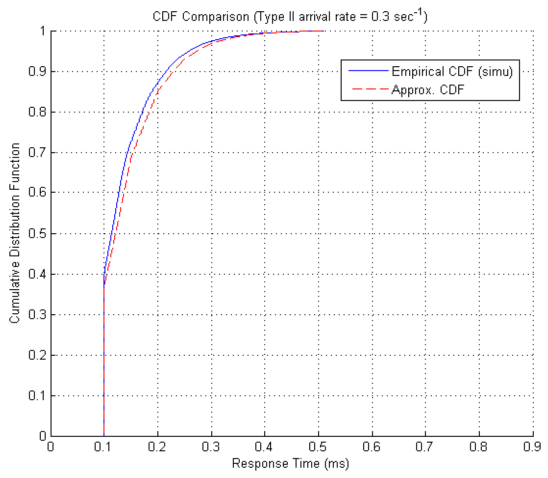
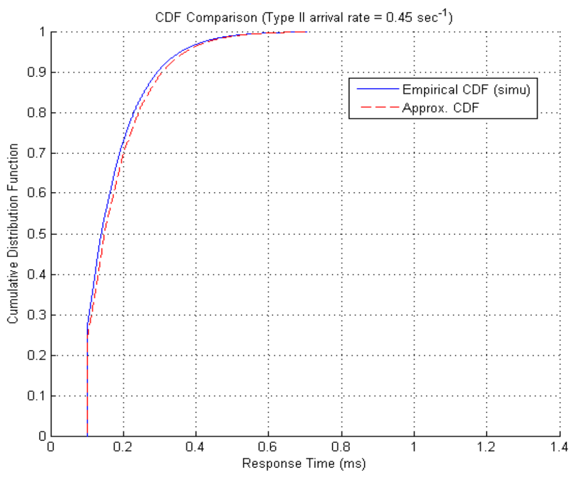


Figure 7.12. Host Performance Model – Type II Mean Response Time and Mean Replication Time

As Figure 7.12 shows, these metrics have good match between the model and simulation. Note the mean response time result shows that, as more tasks arrive during the processing for a given task, the latter faces greater resource contention, completes slower, and thus experiences longer execution.

Figure 7.13 compares the approximated response time distribution of Type I tasks from the model against the distribution from the simulation. As the results indicate, when the workload from Type II tasks grows the accuracy of the approximation method also improves. This is similar to the results from the earlier approximation method validation, and indicates the method can be effective in approximating the response time distribution in such an environment.





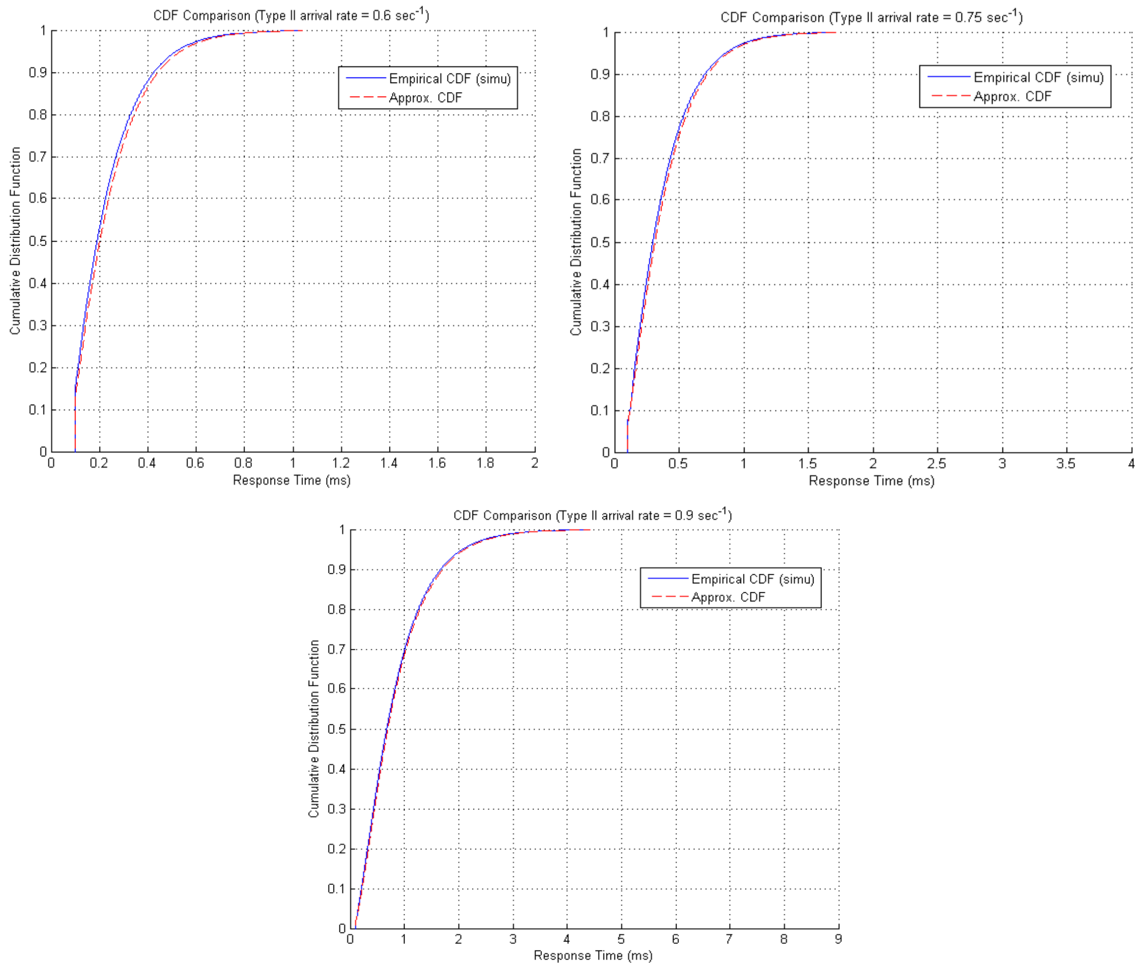


Figure 7.13. Host Performance Model – Type I Response Time Distribution

7.5.3 Data Block Reliability Model

For this model I present comparison of expected time to the loss of a block. As Figure 7.14 suggests, the model is very accurate. Such accuracy is to be expected since the reliability model is exact, subject to the assumptions made.

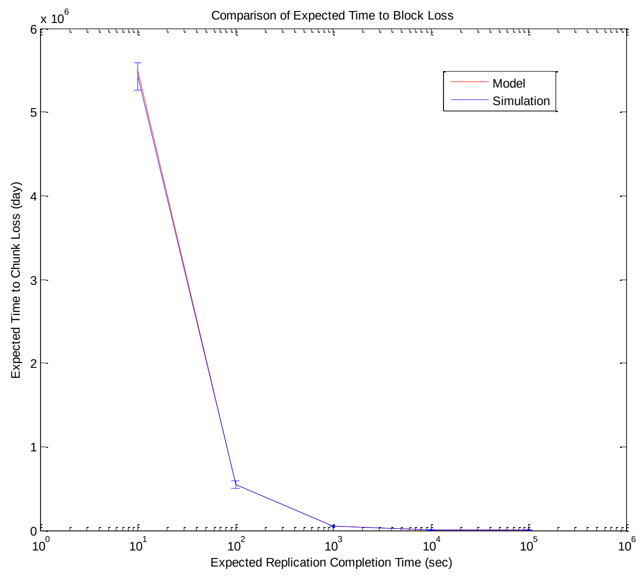


Figure 7.14. Data Block Reliability Model Results

7.5.4 Master Availability Model

The validation of the master availability model mainly concerns the accuracy of using a CTMC model, where some non-exponential distributions in the actual system dynamics are substituted with exponential ones, under different parameter settings and recovery time distributions. The results are presented Table 7.5, where the recovery time probability distributions have identical mean (1) and coefficient of variation values (0.2). The first row shows the results from the model, while the second and third rows contain the simulation results using the corresponding recovery time distributions. The master node mean time to failure is eight months (as in Table 7.3) which translates to a failure rate of 1.736e-04 per hour.

Table 7.5. Master Subsystem Unavailability Comparison

Rec. time dist.\ MTTR multiplier	0.1	1	10	50	100
Exponential	3.138E-11	3.140E-09	3.123E-08	3.823E-06	2.979E-05
Normal	4.635E-11	4.652E-09	4.851E-08	5.210E-06	4.260E-05
Lognormal	4.011E-11	4.080E-09	4.217E-08	5.022E-06	4.082E-05

While the above results suggest that there is difference between the results from the simulation and those from the CTMC model, in the practical range of parameters (e.g. the first three columns) the absolute differences are very small, and therefore I consider the usage of the CTMC model still acceptable. Furthermore, the trends in unavailability change under the different distribution settings are consistent. This means the model is still useful for investigating system unavailability changes under different parameters. Viewing such observations together with the reduced modeling complexity of the CTMC, I consider the benefit of using the CTMC model to outweigh its drawbacks.

7.5.5 Master Performance Model

The main focus of validation for the master performance model is to check how the response time distribution of Type I task behaves when parameters change. More specifically, while the accuracy of the response time distribution approximation for a single node has been investigated earlier in Section 7.4, it is still unknown how the distribution at the master subsystem would be given the synchronization at master processing (i.e. the logging at both the active master disk and the two master backups). Figure 7.15 looks into this issue by varying the arrival rates of the two task types (indicated in the title of each picture). From the figures we can see that when the workload from Type I is much heavier than that from Type II (as is the case in all four pictures where the utilization of Type I at the devices are generally ten times larger than that of Type II), the approximated response time distribution displays a deviation pattern from the simulation result similar to that discussed in Section 7.4. Even though the approximation in this case is not very accurate, it could still be useful for analysis into task response time quantiles for two reasons: 1) the absolute difference in response time for the same quantile is still reasonably small; and 2) the approximation tends to overestimate the response time, and thus provides a safer bound (as compared to, e.g., when it underestimates) when the distribution is used for planning purposes.

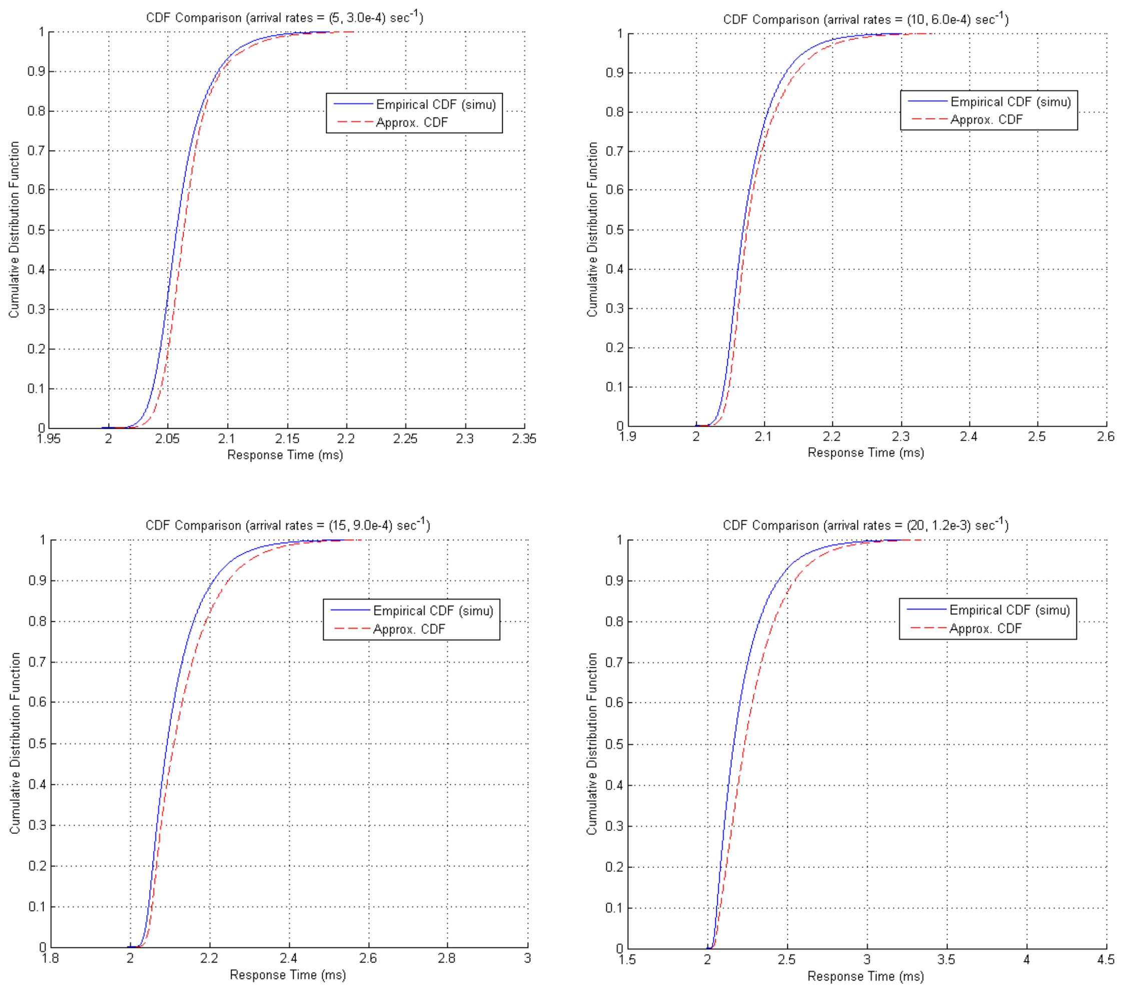


Figure 7.15. Master Performance Model Results set I

In Section 7.4 the results show that the approximation becomes better when the workload from other tasks (i.e. Type II in this context) increases hence improving the accuracy of the PS assumption. However in the case of the master, where the maximum of multiple variables needs to be calculated due to the presence of synchronization, additional approximation error may occur. The following Figure 7.16 looks into this issue. If we compare these results with those in Figure 7.15, we can see that increasing the workload from other tasks (Type II) does still improve the accuracy of the approximation. The left figure shows the case when the arrival rate of the Type II task is doubled. While there is still a bit of difference it is not major and is obviously improved compared to the first picture in Figure 7.15 which serves as the base case. If we increase the rate of Type II arrival further, for example as in the right figure above where the rate is increased one hundred times. The difference in this case becomes negligible. It is worthwhile to point out though that in the system parameter setting used in this chapter the arrival rate corresponding to the right figure would be beyond the system capacity. Nevertheless even if we stick to the value used in the left figure the method still proves to be accurate enough.

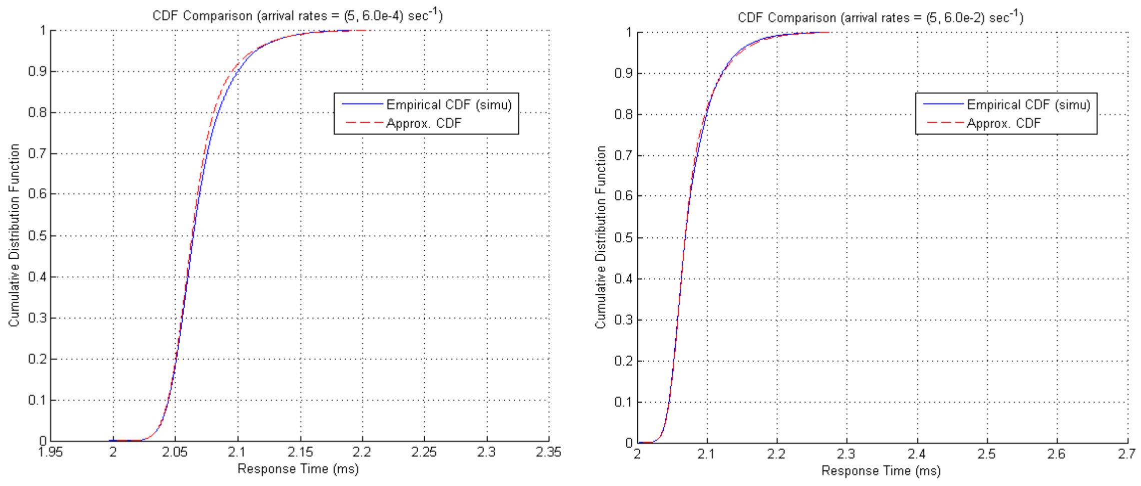


Figure 7.16. Master Performance Model Results set II

7.5.6 Access Availability Model

As described in Section 7.3, the access availability model provides the steady-state probability that a data block is accessible, which is given by the product of master subsystem availability and that of the block itself. Due to the data block model being confirmed previously as very accurate, here I present results concerning how the overall availability changes subject to different master node mean-time-to-recovery and recovery time distributions. Assuming that the mean data block recovery time is 10 hours, Table 7.6 gives the results (in unavailability, since the availability values are very close to one).

Table 7.6. Access Unavailability Comparison

Rec. time dist.\ MTTR multiplier	0.1	1	10	50	100
Exponential	2.392e-10	3.348e-09	3.144e-08	3.823e-06	2.979e-05
Normal	2.541e-10	4.860e-09	4.872e-08	5.210e-06	4.260e-05
Lognormal	2.479e-10	4.288e-09	4.238e-08	5.022e-06	4.082e-05

As the table shows, the access unavailability displays similar trends of change to those in the master availability section. The absolute differences are not significant in practical range of parameter values, and the usefulness of the model remains valid as discussed previously.

7.5.7 Access Performance Model

The access performance model provides the overall access performance measures. Figure 7.17 presents the comparison of results from the combined model and overall simulation. As the results indicate, as the workload from Type II task increases the approximation has increased accuracy. Even though the previous master validation section indicates that in the range of parameter used here there is some error in that model, globally such difference is absorbed due to the fact that the actual data access dominates the master part in terms of response time. Thus for the scenarios considered in this chapter, the approximation method provides good estimates on the response time distribution when the system is loaded by larger tasks, and remains useful when the system becomes lighter loaded.

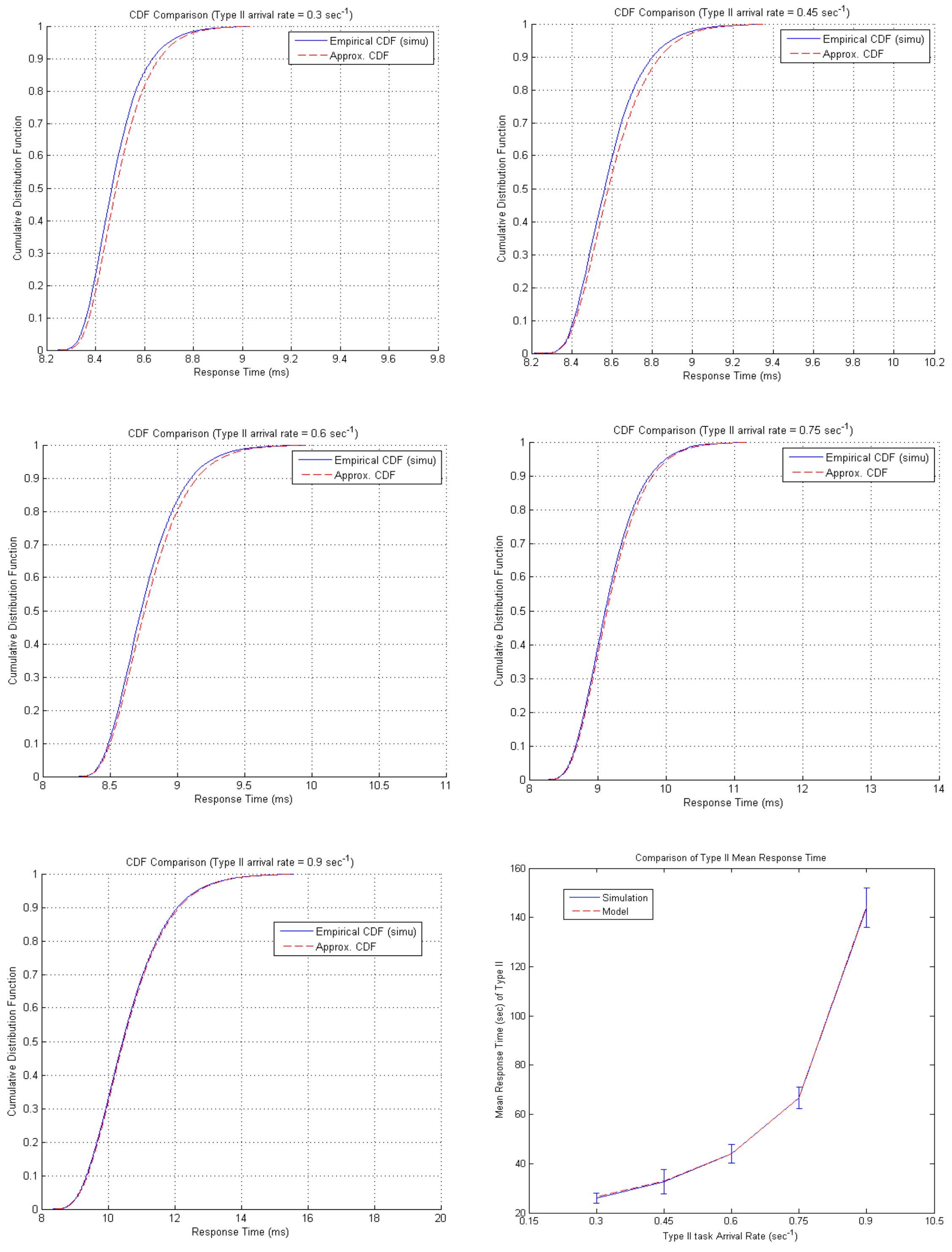


Figure 7.17. Overall Access Performance Model Results

7.6 Application Scenarios and Possible Extensions

In this section I provide discussion on how the current model can be applied in practice, how to apply it to systems that have characteristics different from those of GFS, and how it may be further extended.

7.6.1 Usage Scenarios

The models developed in previous sections provide a range of utilities that are useful for many purposes in replicated storage design and operation, including:

- Predicting data block reliability. This utility is useful in helping determine suitable replication levels for blocks, as well as designing suitable backup strategies. Given that the recovery process from data backup can be much slower than replication, it is an open question as to how backups should be made. The reliability parts of the model would assist in answering such a question.
- Predicting response time distribution for latency-sensitive tasks. The SLAs available today typically is unable to provide detailed specification on application performance in hosting environment due to the difficulty in predicting performance dynamics in such an environment. The response time distribution model developed in this chapter could help with such an issue by providing the basis for analyzing quantile information of application response times.
- Predicting level of resource contention. The queueing network formalisms employed is very suitable for capturing resource contention in the system being modeled. The addition of the response time approximation allows much more insight into the effect of

a particular system capacity configuration, hence providing more comprehensive information on which more effective capacity planning can be based.

7.6.2 Application to Different Systems

This subsection discusses how to apply the modeling approach to replicated storage systems that are different from GFS.

- For a system that issues parallel requests (instead of using a pipeline) to all data replicas during a write. This can essentially be modeled in the same way as the master part. If throughput is of main concern, the queueing formalism by itself would suffice. If response time distribution is desired, it would be necessary to check if the approximation method is still applicable, e.g. determine if the tasks of interest is small and the nodes can be treated as PS queues. If that is the case, as may be expected for tasks that put premium on response time distributional info in the first place, then the same technique applies.
- For a system that utilizes weaker forms of consistency, e.g. eventual consistency. In such a case, because synchronization is relaxed it would actually be easier to use the currently developed techniques to study the performance and availability aspects. On the other hand, an additional measure of interest in such a system would be the time to recover consistency, which is treated as a property regarding the global knowledge about the data. Because consistency reasons about the status of a data, a state space model such as a Markov chain can be developed to study this issue. Part of the parameterization of such a model could come from the models developed in this

chapter, which can provide quantities such as the time to propagate data updates opportunistically across replicas subject to possible resource contention in the system.

7.6.3 Possible Model Extensions

In this subsection I discuss some directions to extend the current models.

- Introduce a network model to better capture the possible correlation in availability and performance among multiple data blocks. The current model covers some correlation, specifically in the resource contention in the queueing models (for performance) and the possibility of rack failures (for availability/reliability). Additional network components can be added in similar fashions, where each device is presented both as a queue (in a performance model) and a possible source of failure (in the availability/reliability part). The introduction of such a model may also allow analysis of block replica distribution strategies to be carried out.
- Apply the model to a larger scale, e.g. at the level of geographically separated datacenters each hosting the replica of a large set of data. One prerequisite to achieve this goal is to verify whether, at the datacenter level, it can be assumed that the access flows to the different data replicas are approximately handled in a PS fashion. If the answer is positive, then the same methods in this chapter can be applied, essentially by replacing the model components with their larger-scale counterparts.
- Introduce more task classes. Currently the task type classification is binary and may be too coarse-grained. Additional task types can be introduced easily if a detailed distributional analysis of its response time is not needed. However, if the distribution is needed, it remains an open question as to whether (or how far) the approximation used

here can be applied in that case. Generally speaking I would expect the method to remain largely valid so long as the task type in question is not the primary contributor to the utilization of system resources.

7.7 Related Work

The utility of replicated storage is widely recognized and it is an area that has garnered significant amount of attention. Aside from the prominent instances from industry (such as [121][124][122][128]), there is also an extensive body of innovative works from the academia on replicated storages, some examples being [137][138][139][140][141][142]. The majority of these works focus on implementing replicated storage functionalities in novel ways and have contributed to a growing pool of available technologies that can be employed to build new replicated storage systems. On the other hand, the work presented in this chapter focus on quantitatively predicting the behavior of replicated storage systems built from some combination of such technologies. In this regard the models developed (and the modeling methods) are complimentary to the implementation works and may be used to assist system design and implementation efforts.

There are also quite a number of research works focusing on evaluating particular aspects of replicated storage systems, and they are arguably closer to the work in this chapter. For example, [143] uses a simple Markov model to study the persistence of a replicated data item in the system, which can be regarded as a simpler and less detailed version of the models in Section 7.3.2. Somewhat related, [144] studies the tradeoff of performance and availability in a replicated storage using a combination of a simple Markov model and trace analysis. Due to its coverage of both the performance and availability perspectives it may be considered as a simpler alternative to the methods presented in Section 7.3.3 and 7.3.6. Similar to the analysis of particular system design choices in this chapter, [145] compares erasure coding and replication, two major ways of achieving data reliability that may be employed jointly in practice, in a quantitative manner. In contrast, the work of [146] focuses on the cost perspective of replicated storage systems which provides an interest angle that is complementary to the system measures studied in this chapter. [147] goes further and discusses the connection between the cost and availability aspects of replicated storages.

As discussed earlier in this chapter, consistency is one of the most important and complex issues in a replicated storage systems. Research works abound on this topic, some examples include [148][149][150], which examine the issue from various perspectives, typically with a mixture of emphasis on evaluation/reasoning and design/implementation of actual techniques or systems. While this chapter does not cover the consistency issue directly, it provides useful methods to analyze the performance and availability/reliability implications of particular consistency techniques. Due to the highly dependent nature of replicated storage performance and its consistency goal/implementation, the work in this chapter acts as useful compliments to such research works.

7.8 Conclusion

In this chapter I described some detailed models for capturing replicated storage system performance and availability. The models are constructed from a combination of queueing networks and Markovian models. To obtain task response time distribution for latency-sensitive tasks, which is difficult for queueing network models, I devised an approximation method that has been shown to be accurate in many practical situations, as well as still useful in others where the accuracy is decreased. The models developed can be used to provide a range of utilities, including task response time prediction and data reliability analysis, that would be very useful for system design, planning as well as runtime control The models can also be extended in several directions with moderate effort to capture wider real scenarios.

8. Summary

In this chapter I summarize the content of this dissertation. The focus of this dissertation is on predictive analysis of modern storage systems with stochastic modeling techniques. To this end, a variety of models are utilized to capture different aspects of such systems.

Chapter 3 presented models for comparing different RAID configurations. The models investigated the use of the MRGP formalisms, instead of the traditional CTMC models that are simpler but potentially less accurate, in this context. Numerical results indicated that the CTMC formalism may have sufficient accuracy when used for RAID availability/reliability analysis, but not for more performance-oriented aspects. Further results based on the more accurate MRGP model suggested the relative merits of the RAID6 and RAID10 configurations would most likely depend on the mix of workload they are subject to, as they tend to show different performance trends when subject to different data access patterns.

Chapter 4 presented the stochastic models for analyzing cloud storage resource provisioning. The models were motivated by earlier work that was not sufficiently accurate, and employs novel observation about the problem structure to create a more efficient and accurate modeling approach for the system. The work in this chapter particularly focused on model scalability and accuracy, and to improve on these two fronts three different types of model solution methods are examined: numerical solution, simulation and MCMC sampling. The relative merits of the methods in terms of efficiency are discussed, and detailed proof for the validity of the MCMC approach in this context was presented. The work of this chapter thus provides a pool of general analysis methods that can be employed to similar system analysis scenarios.

Data backup is a commonly employed method to ensure data reliability. Chapter 5 presented stochastic models for comparing data backup configurations and policies in terms of the level of data protection they provides and their impact on system availability. The model covers significant amount of details related to data backup execution and variants of backup policies.

Partially continuing the focus of the previous chapter, Chapter 6 presented a framework for data backup planning based on the MDP optimization formalism. The modeling approach covers important high-level requirements on the level of data protection needed and also accounts for the potentially limited system resource available for backup execution. To deal with the classic issue of large state space in MDP application, a novel approximation method was devised to decompose the original problem into smaller independent ones with an exponential decrease in model size/complexity. Extensive numerical comparison was performed to compare the quality of planning generated by the MDP-based framework against commonly used heuristics and showcase the superiority of the former. The decomposition is also shown to lead to much better model scalability compared to the direct MDP approach.

Finally, Chapter 7 presented detailed models for replicated storage systems, using the Google File System as a case study. The models cover different system aspects in much details using various formalisms as appropriate. To capture the response time distribution of latency-sensitive tasks, an important system performance metric that is difficult to predict and thus often ignored, a novel approximation scheme is proposed and shown via comparison to simulation to have good accuracy subject to moderate assumptions that are reasonable for the replicated storage systems in general.

One of the primary directions for future work is to extend the model to cover more aspects of storage systems. The contention of resource on and availability of the communication paths and networks may be considered to make the model more comprehensive. Inclusion of such details introduces greater level of interdependencies among the system elements currently modeled, so alternative formalisms that better account for the additional correlation effect may need to be employed.

Another future direction that can greatly expand the work presented in this dissertation is the integration of real system measurement/operation data. Assuming availability of the data, two main extension approaches exist. It is possible to use the data to validate the probabilistic distributions used in the modeling, or develop more accurate ones. Alternatively, the data can be used as the basis of a learning framework which updates the parameters of the models dynamically. In the context of the work presented in this thesis, the former approach can be applied to the state-space models (such as CTMC, DTMC, SMC or, less often, MRGP) through the phase-expansion approach to approximate real distributions (as determined by the data) with Coxian distributions. As for the second approach, the foremost candidate would be the planning framework in Chapter 6, since extensions of MDP (e.g., Q-learning) is already widely used in Machine Learning contexts. On the other hand, it remains an open question as to how to incorporate such learning faculties into the other model components, in particular the queueing network models due to their complex solution structure.

Hurdles may still need to be overcome regarding the availability of storage system data. Due to the fact that storage systems tend to be a part of the computing infrastructure of the owner organizations, which in many cases are reluctant to reveal details of such infrastructural components, there is a scarcity of storage system operation data available to the researchers. It seems the most likely way by which this issue may be addressed would be through storage systems designed for supporting scientific research in the first place, which in turn is likely to depend much on the development of opensource tools and system architectures. With the popularity of opensource projects such as openstack [53] and openflow [150], such a prospect may not be very far, but much work remains to be done.

Appendix A

Selected Sets of Program Code (Java, Matlab, SHARPE, SPNP)

Chapter 4 - The RPDE model (SHARPE)

```
format 12

bind
  lambda_s   1
  beta_s   1
  lambda   20
  delta   1200
end

markov RPDE
0_on 0_off lambda_s
0_on 1_on lambda
1_on 0_on delta
1_on 1_off lambda_s
1_on 2_on lambda
2_on 1_on delta
2_on 2_off lambda_s
2_on 3_on lambda
3_on 2_on delta
3_on 3_off lambda_s
3_on 4_on lambda
4_on 3_on delta
4_on 4_off lambda_s
4_on 5_on lambda
5_on 4_on delta
5_on 5_off lambda_s
5_on 6_on lambda
0_off 0_on beta_s
1_off 1_on beta_s
1_off 0_off delta
3_off 3_on beta_s
3_off 2_off delta
2_off 2_on beta_s
2_off 1_off delta
4_off 4_on beta_s
4_off 3_off delta
5_off 5_on beta_s
```

```
5_off 4_off delta
6_on 5_on delta
6_on 6_off lambda_s
6_off 6_on beta_s
6_off 5_off delta
end
end


echo output start:

var drop_prob prob(RPDE,6_on)+prob(RPDE,6_off)
echo drop probability:
expr drop_prob

var nonempty_prob 1-prob(RPDE,0_on)-prob(RPDE,0_off)
echo nonempty probability:
expr nonempty_prob

var exp_number (prob(RPDE,1_on)+prob(RPDE,1_off))+2*(prob

(RPDE,2_on)+prob(RPDE,2_off))+3*(prob(RPDE,3_on)+prob

(RPDE,3_off))+4*(prob(RPDE,4_on)+prob(RPDE,4_off))+5*(prob

(RPDE,5_on)+prob(RPDE,5_off))+6*(prob(RPDE,6_on)+prob

(RPDE,6_off))
echo expected number of tasks in RPDE:
expr exp_number

end
```

Chapter 4 - The pool model - simulation (MATLAB)

```
clear all;
clc;

n_h = 2;
m = 2;
Lh = 2;
lambda = 10;
beta_h = 12;
mu = 1;

N_batch = 10000;
```

```
N_event = 500000;
N_run = 1;
drop = zeros(N_run,1);

tokens = zeros(m+1,Lh+1);
tokens(1,1) = n_h;
% The array for the epoch of next event. (1) for arrival, (2) for
provision, (3) for service
next_epoch = [inf,inf,inf];
arrival_sample = exprnd(1/lambda,N_batch);
next_epoch(1) = arrival_sample(1);
arrival_count = 1;

% Record the current total rate of provision and service, so as to
% possibly save computation
total_beta = 0;
total_mu = 0;

tic;
for run_ind = 1:N_run
  RandStream.setDefaultStream(RandStream.create('mrg32k3a','NumStreams',
    N_run,'StreamIndices',run_ind));
  drop_occ = 0;
  count = 0;
  total_time = 0;
  % The main simulation process
  while(count<=N_event)
    [time,index] = min(next_epoch);
    % next_epoch = next_epoch - time;
    total_time = total_time + time;
    count = count + 1;

    % Transfer tokens appropriately
    % If an arrival occurs
    if index == 1
      num_full = sum(tokens(:,Lh+1));
      % If all tokens are at the full column, drop the task
      if num_full == n_h
        drop_occ = drop_occ + 1;
      else
        % Otherwise, randomly assign to a PM with capacity
        chosen = randi(n_h-num_full,1);
        for line = 1:m+1
          if chosen <= sum(tokens(line,1:Lh))
            target = 0;
            while(chosen>0)
              target = target + 1;
              chosen = chosen - tokens(line,target);
            end
            tokens(line,target) = tokens(line,target)-1;
            tokens(line,target+1) = tokens(line,target+1)+1;
            % Update the next provision time if necessary
            if line < m+1 && target == 1
              total_beta = total_beta + beta_h;
            end
            break;
          else
            chosen = chosen - sum(tokens(line,1:Lh));
          end
        end
      end
      % next_epoch(1) = exprnd(1/lambda);
      next_epoch(2) = exprnd(1/total_beta);
      next_epoch(3) = exprnd(1/total_mu);
      % Sample the next arrival time
      arrival_count = arrival_count + 1;
      next_epoch(1) = arrival_sample(arrival_count);
      if arrival_count == N_event
        arrival_sample = exprnd(1/lambda,N_batch);
        arrival_count = 0;
      end

    % If a provision occurs
    elseif index == 2
    % Randomly choose a PM as the one on which a provision is done
      num_nprov = sum(tokens(:,1))+sum(tokens(m+1,2:Lh+1));
      if num_nprov < n_h
        chosen = randi(n_h-num_nprov,1);
        for line = 1:m
          if chosen <= sum(tokens(line,2:Lh+1))
            target = 1;
            while(chosen>0)
              target = target + 1;
              chosen = chosen - tokens(line,target);
            end
            tokens(line,target) = tokens(line,target)-1;
            tokens(line+1,target-1) = tokens(line+1,target-1)+1;
            % Update provision and service rate if necessary
            if line == m || target == 2
              total_beta = total_beta - beta_h;
            end
            total_mu = total_mu + mu;
            break;
          else
            chosen = chosen - sum(tokens(line,2:Lh+1));
          end
        end
      end
      next_epoch(2) = exprnd(1/total_beta);
      next_epoch(3) = exprnd(1/total_mu);
      arrival_count = arrival_count + 1;
      next_epoch(1) = arrival_sample(arrival_count);
      if arrival_count == N_event
        arrival_sample = exprnd(1/lambda,N_batch);
        arrival_count = 0;
```

```
      end

    % If an execution completes
    else
    % Randomly choose a PM as the one on which an execution is done
      num_empty = sum(tokens(1,:));
      toChoose = zeros(1,m);
      if num_empty < n_h
        for line = 2:m+1
          toChoose(line-1) = sum(tokens(line,:));
        end
        chosen = randi(toChoose*(1:m)',1);
        for line = 2:m+1
          if chosen <= sum(tokens(line,:))*(line-1)
            target = 0;
            while(chosen>0)
              target = target + 1;
              chosen = chosen - tokens(line,target)*(line-1);
            end
            tokens(line,target) = tokens(line,target)-1;
            tokens(line-1,target) = tokens(line-1,target)+1;
            % Update provision and service as necessary
            if line == m+1 && target > 1
              total_beta = total_beta + beta_h;
            end
            total_mu = total_mu - mu;
            break;
          else
            chosen = chosen - sum(tokens(line,:))*(line-1);
          end
        end
      end
      next_epoch(2) = exprnd(1/total_beta);
      next_epoch(3) = exprnd(1/total_mu);
      arrival_count = arrival_count + 1;
      next_epoch(1) = arrival_sample(arrival_count);
      if arrival_count == N_event
        arrival_sample = exprnd(1/lambda,N_batch);
        arrival_count = 0;
      end
    end
  end

  drop_rate = drop_occ/total_time;
  drop(run_ind) = drop_rate;
end
running_time = toc;
```

Chapter 5 - The partial priority approximation (see page 143 of the main text)

The outside wrapper (Python)

```
import os
import sys
import subprocess
import math
import string
import fileinput

basis = []
for line in fileinput.input("PP_basis.txt"):
   basis.append(line.strip())

util_h = 0
util_l = 0
tput = 0
flag = 1
thres = 0.00000000001

mu = [0]*256
arrival = 1.8988
for i in range(0,len(mu)):
   while flag:
      flag = 0
      input_file = open("Test.txt","w")
      for line in basis:
         if "rfCPUu" in line:
            temp = line.split()
            if temp[0] == "rfCPUu":
               rate_l = eval(temp[1])*(1-util_h)
               input_file.write(temp[0]+" "+str(rate_l)+"\n")
               continue
         elif "N_user" in line:
            temp = line.split()
            if temp[0] == "bind" and temp[1] == "N_user":
               input_file.write(temp[0]+" "+temp[1]+" "+str(i+1)+"\n")
               continue
         input_file.write(line+"\n")
      input_file.close()

      os.system("sharpe Test.txt > Result.txt")
      results = []
      for line in fileinput.input("Result.txt"):
         if "UTIL1" in line:
            util_temp = eval(line.split()[1])
            if abs(util_temp - util_h) > thres:
               flag = 1
            util_h = util_temp
         if "THRU1" in line:
            tput = eval(line.split()[1])
   mu[i] = tput

product = 1
denominator = 1
for i in range(0,len(mu)):
   #print mu[i]
   product = product*mu[i]
   denominator += arrival**(i+1)/product
Prej = arrival**256/product/denominator
print Prej
```

The basis file ('PP_basis.txt', SHARPE)

```
format 12

mpfqn Inner(N_user)

chain Apache

CPU fDiskIO p_disk
CPU fNetIO 1-p_disk
fDiskIO CPU 1
end

chain backup
 fDiskIO Sender 1
 Sender fNetIO 1
 fNetIO Network 1
 Network bNetIO 0.5
 Network fDiskIO 0.5
 bNetIO Receiver 1
 Receiver bDiskIO 1
 bDiskIO Generator 1
 Generator Network 1
end
end

* Rates
```

```
fNetIO fcfs rfNetIO
end
fDiskIO fcfs rfDiskIO
end

Sender ps rfCPUb
end
CPU ps rfCPUu
end

Network is rNetwork
end

bNetIO fcfs rbNetIO
end
Receiver fcfs rReceiver
end
bDiskIO fcfs rbDiskIO
end
Generator fcfs rGenerator
end

end

* Bind number of jobs in two chains
Apache N_user
backup N_backup
end



* Bind branching probabilities
bind p_disk 1/3
bind ratio 3

* Bind rates
bind
  rfNetIO 46.875
  rfDiskIO 52.734375
  rfCPUu 850
  rfCPUb 500
  rNetwork 50
  rbNetIO rfNetIO*ratio
  rReceiver 1000
  rbDiskIO rfDiskIO*ratio
  rGenerator 1500
end

* Bind number of backup batches
*bind  N_backup 8
*bind N_user 10
bind  N_backup 5
bind N_user 5

* User request arrival rate
bind arrival 9.0722*2

bind UTIL1 mutil(Inner,Sender;N_user)
expr UTIL1
bind THRU1 mtput(Inner,CPU;N_user)*0.5
expr THRU1
end
```

Chapter 6 - The MDP Framework (Java)

The Basic Framework

MDP.java

```
import java.io.BufferedWriter;
import java.io.File;
import java.io.IOException;
import java.nio.charset.Charset;
import java.nio.file.Files;
import java.util.*;

public class MDP
{
    public LinkedHashMap<String, MDPstate> states = new LinkedHashMap<String, MDPstate>();
    double opt_reward = 0;
    double totalProb = 0;

    public void addState(MDPstate state)
    {
            if(states.containsKey(state.id))
            {
                        System.out.println("Duplicate state ID. Abort.");
                        return;
            }
            states.put(state.id, state);
    }

// Class for data source availability model
class State
{
            String id;
            String[] nextID;
            // Type of transition to this state, i.e. exponential or general (only uniform for now).
            String[] tranType;
            // Transition probabilities.
            double[] tranProb;
            // Values associated with the transition, rate for exponential, mean for uniform.
            double[] tempTran;
            // Steady-state probability in the availability model.
            double probability = 0;
            // Steady-state probability of the embedded DTMC.
            double embed_Prob = 0;
            // Temporary variable used in MDP solution and state probability computatition.
            double tempVal = 0;
            // Mean sojourn time of this state
            double sojourn = 0;

            public State(String pid, int num_next)
            {
                        id = pid;
                        nextID = new String[num_next];
                        tranType = new String[num_next];
                        tranProb = new double[num_next];
                        tempTran = new double[num_next];
            }
}
```

MDPstate.java

```
import java.util.ArrayDeque;

// MDP state class
public class MDPstate
{
        public String id;
        public ArrayDeque<MDPaction> actions = new ArrayDeque<MDPaction>();
        public MDPaction optimAction = null;
        MDPaction tempAct = null;
        int num_action = 0;
        public double value = 0;  // State reward value from MDP solution
        double probability = 0; // steady-state probability under current policy
        double tempVal = 0; // Temporary variable used in solution
        // -1/0/1 for state being impossible/not completely checkeded/possible.
        int possible = 0;
        int cur_action = 0; // Used during MDP construction

        public MDPstate(String pid)
        {
                id = pid;
        }

        public void addAction(MDPaction act)
        {
                actions.add(act);
                num_action ++;
        }
}
```

MDPaction.java

```
import java.util.ArrayDeque;

// MDP action class
public class MDPaction
{
  public String id;
  public ArrayDeque<MDP_nextState> next_states=new ArrayDeque<MDP_nextState>();
  double value;
  int num_states;

  public MDPaction(String pID, String sid)
  {
      id = pID;
      next_states.add(new MDP_nextState(sid));
      num_states = 1;
      value = 0;
  }

  public void addNextState(String sid)
  {
      next_states.add(new MDP_nextState(sid));
      num_states ++;
  }
}
```

MDP_nextState.java

```
// MDP next-state
public class MDP_nextState
```

```
{
	public String id;
	MDPstate ref;
	double probability = 0;
	public double reward = 0;

	public MDP_nextState(String sid)
	{
		id = sid;
	}
}
```

DataSource.java (this version is for the DCDV paper. Modification should be easy for the PRDC)

```
import java.util.LinkedHashMap;

public class DataSource
{
	public String name;
	public int maxSkip;
	public int maxPB;
	// Data quantities for partial/full backup, and partial/full recovery
	public double partial;
	public double full;
	public double rec_partial;
	public double rec_full;
	// Ratio of overwrite
	public double OWfactor;
	// Expected chunk loss within a decision interval.
	public double expLoss;

	// Rate of re-replication with two/one replicas remaining;
	// rate values for doing partial/full recovery
	double repRate2;
	double repRate1;
	double rateRecPartial;
	double rateRecFull;

	// Expected number of partial replicas to process
	double expPartial;

	// Auxiliary variables
	double totalProb = 0;
	double sojourn_exp = 0;
	// Data for availability computation
	LinkedHashMap<String, State> avail_states = new LinkedHashMap<String,
State>();

	public DataSource(String pName, int pmSkip, int pmPB, double[]
back_params,
				double[] avail_params)
	{
		name = pName;
		maxSkip = pmSkip;
		maxPB = pmPB;
		full = back_params[0];
		partial = back_params[1];
		rec_full = back_params[2];
		rec_partial = back_params[3];
		OWfactor = back_params[4];

		// Assume exponential time-to-failure
		expLoss = 1-Math.exp(-avail_params[0]*24);
		repRate2 = avail_params[0];
		repRate1 = avail_params[1];
		rateRecFull = avail_params[2];
		rateRecPartial = avail_params[3];

		expPartial = 0;
	}

	// Compute the availability model, an SMP, transition probabilities of
the data sources
	// This is for now done manually
	public void compute_avail()
	{
		System.out.println("Computing availability");
		double uni_factor = 0.9;
		State curState = avail_states.get("1");
		double lam1 = curState.tempTran[1];
		double lam2 = curState.tempTran[2];
		double T = curState.tempTran[0];
		curState.tranProb[0] = Math.exp(-(lam1+lam2)*T)*uni_factor;
		curState.tranProb[1] = lam1/(lam1+lam2)*(1-Math.exp(-
(lam1+lam2)*T))*uni_factor;
		curState.tranProb[2] = lam2/(lam1+lam2)*(1-Math.exp(-
(lam1+lam2)*T))*uni_factor;

		curState = avail_states.get("2");
		lam1 = curState.tempTran[1];
		lam2 = curState.tempTran[2];
		double lam3 = curState.tempTran[3];
		T = curState.tempTran[0];
		curState.tranProb[0] = Math.exp(-(lam1+lam2+lam3)*T)*uni_factor;
		curState.tranProb[1] = lam1/(lam1+lam2+lam3)*(1-Math.exp(-
(lam1+lam2+lam3)*T))*uni_factor;
		curState.tranProb[2] = lam2/(lam1+lam2+lam3)*(1-Math.exp(-
(lam1+lam2+lam3)*T))*uni_factor;
		curState.tranProb[3] = lam3/(lam1+lam2+lam3)*(1-Math.exp(-
(lam1+lam2+lam3)*T))*uni_factor;

		curState = avail_states.get("3");
```

```
			lam1 = curState.tempTran[1];
			lam2 = curState.tempTran[2];
			T = curState.tempTran[0];
			curState.tranProb[0] = Math.exp(-(lam1+lam2)*T)*uni_factor;
			curState.tranProb[1] = lam1/(lam1+lam2)*(1-Math.exp(-
(lam1+lam2)*T))*uni_factor;
			curState.tranProb[2] = lam2/(lam1+lam2)*(1-Math.exp(-
(lam1+lam2)*T))*uni_factor;

			curState = avail_states.get("B1");
			curState.tranProb[0] = 1*uni_factor;
			curState = avail_states.get("B2");
			curState.tranProb[0] = 1*uni_factor;
			curState = avail_states.get("B3");
			curState.tranProb[0] = 1*uni_factor;
			curState = avail_states.get("D1");
			curState.tranProb[0] = 1*uni_factor;
			curState = avail_states.get("D2");
			curState.tranProb[0] = 1*uni_factor;
		}

		// Compute the availbility model steady-state probabilities based on EMC
		// probabilities and transition parameters. For now this is manual.
		public void compute_avail_prob()
		{
			sojourn_exp = 0;
			double tempRate;
			State tempState;
			tempState = avail_states.get("B1");
			sojourn_exp += tempState.embed_Prob * (tempState.sojourn =
tempState.tempTran[0]);
			tempState = avail_states.get("B2");
			sojourn_exp += tempState.embed_Prob * (tempState.sojourn =
tempState.tempTran[0]);
			tempState = avail_states.get("B3");
			sojourn_exp += tempState.embed_Prob * (tempState.sojourn =
tempState.tempTran[0]);
			tempState = avail_states.get("D1");
			sojourn_exp += tempState.embed_Prob * (tempState.sojourn =
tempState.tempTran[0]);
			tempState = avail_states.get("D2");
			sojourn_exp += tempState.embed_Prob * (tempState.sojourn =
tempState.tempTran[0]);
			tempState = avail_states.get("1");
			tempRate = tempState.tempTran[1]+tempState.tempTran[2];
			sojourn_exp += tempState.embed_Prob * (tempState.sojourn =
1.0/tempRate*(1-Math.exp(-tempRate*tempState.tempTran[0])));
			tempState = avail_states.get("2");
			tempRate =
tempState.tempTran[1]+tempState.tempTran[2]+tempState.tempTran[3];
```

```
              sojourn_exp += tempState.embed_Prob * (tempState.sojourn =
1.0/tempRate*(1-Math.exp(-tempRate*tempState.tempTran[0])));
              tempState = avail_states.get("3");
              tempRate = tempState.tempTran[1]+tempState.tempTran[2];
              sojourn_exp += tempState.embed_Prob * (tempState.sojourn =
1.0/tempRate*(1-Math.exp(-tempRate*tempState.tempTran[0])));

              tempState = avail_states.get("1");
              tempState.probability =
tempState.embed_Prob*tempState.sojourn/sojourn_exp;
              tempState = avail_states.get("2");
              tempState.probability =
tempState.embed_Prob*tempState.sojourn/sojourn_exp;
              tempState = avail_states.get("3");
              tempState.probability =
tempState.embed_Prob*tempState.sojourn/sojourn_exp;
              tempState = avail_states.get("B1");
              tempState.probability =
tempState.embed_Prob*tempState.sojourn/sojourn_exp;
              tempState = avail_states.get("B2");
              tempState.probability =
tempState.embed_Prob*tempState.sojourn/sojourn_exp;
              tempState = avail_states.get("B3");
              tempState.probability =
tempState.embed_Prob*tempState.sojourn/sojourn_exp;
              tempState = avail_states.get("D1");
              tempState.probability =
tempState.embed_Prob*tempState.sojourn/sojourn_exp;
              tempState = avail_states.get("D2");
              tempState.probability =
tempState.embed_Prob*tempState.sojourn/sojourn_exp;
       }
}
```

SystemSpec.java (again DCDV version)

```
import java.util.ArrayDeque;
import java.util.Arrays;
import java.util.Iterator;

// System specifications in term of the requirements from the data sources and the
// available system resources
public class SystemSpec
{
       public ArrayDeque<DataSource> sources = new ArrayDeque<DataSource>();
       int num_source = 0;
       int maxResource;
       double Time = 24;
       // The three parameters for impact function
}

	double[] impactParams = {1,0,0};

	public void setResource(int res)
	{
		maxResource = res;
	}

	public void addSource(DataSource src)
	{
		sources.add(src);
		num_source ++;
	}

	public void computeResult()
	{
		double reward = 0;
		Iterator<DataSource> itr1 = sources.iterator();
		DataSource tempSrc;
		State tempState1;
		while(itr1.hasNext())
		{
			tempSrc = itr1.next();

			// The downtime part of the penalty
			tempState1 = tempSrc.avail_states.get("D1");
			reward += tempState1.probability;
			tempState1 = tempSrc.avail_states.get("D2");
			reward += tempState1.probability;
			tempState1 = tempSrc.avail_states.get("B1");
			reward += tempState1.probability;
			tempState1 = tempSrc.avail_states.get("B2");
			reward += tempState1.probability;
			tempState1 = tempSrc.avail_states.get("B3");
			reward += tempState1.probability;
		}
		System.out.println("Expected penalty for downtime:
"+reward/num_source*24*365);
	}

	public void setImpactParam(double[] sImpactParams)
	{
		impactParams = Arrays.copyOf(sImpactParams, sImpactParams.length);
	}
}
```

Generator.java

```
import java.util.*;

public class Generator
{
    public static MDP genMDP(SystemSpec spec)
    {
      int num_state = 1;
      String stateID;
      String next_stateID;
      MDP mdp1 = new MDP();
      MDPaction tempAct;

      int num_src = spec.num_source;
      if(num_src<=0)
      {
             System.out.println("Non-positive number of data sources. Abort");
             return(null);
      }
      int num_action = (int) Math.pow(3, spec.num_source);

      // Find out maximum number of states and get info on state requirements
      int [] reqs = new int[num_src*2];
      int n = 0;
      for(DataSource temp : spec.sources)
      {
             num_state *= (temp.maxPB+1)*temp.maxSkip;
             reqs[n*2] = temp.maxSkip;
             reqs[n*2+1] = temp.maxPB;
             n ++;
      }

      // Construct the state space
      MDPstate tempState1, tempState2;
      Stack<MDPstate> stack1 = new Stack<MDPstate>();
      for(int i=0; i<num_state; i++)
      {
             stateID = genStateID(i, num_src, reqs);
             if(!mdp1.states.containsKey(stateID))
             {
                    tempState1 = new MDPstate(stateID);
                    mdp1.addState(tempState1);
             }
             else
                    tempState1 = mdp1.states.get(stateID);

      // If the state was checked before and found impossible, ignore it
             if(tempState1.possible == -1)
                    continue;
```

```
// If the state has been checked before and found to be possible, ignore
// it since it must have been thoroughly checked by this point.
        if(tempState1.possible == 1)
                continue;
// If the state has not been checked before, push it onto the stack.
        stack1.push(tempState1);
        while(!stack1.isEmpty())
        {
                tempState1 = stack1.peek();
// If there is no action left for the stack head, mark it as impossible
// if it is not already marked as possible. Then pop it. If the state is
// possible, mark its parent on the stack as the same.
                if(tempState1.cur_action > num_action-1)
                {
                        if(tempState1.possible <1)
                                tempState1.possible = -1;
                        stack1.pop();
                        if(!stack1.isEmpty())
                                if(tempState1.possible == 1)
                                        stack1.peek().possible = 1;
                                else
                                        stack1.peek().actions.removeLast();
                                continue;
                }
// Generate the ID of the next-states corresponding to this action.
// The first entry in the returned ArrayDeque instance is the state that
// the system will reach IF there is no failure. The rest are states the
// system will reach if some data sources fails.
                next_stateID = genNextID(tempState1.id,
                   tempState1.cur_action, num_src, spec.maxResource, reqs);
// Check whether the next_stateID is null.
// If it is, then either the first next-state is impossible or the
// action violates the resource constraint. In this case the action will
// not be valid as it depends on the intended next-state.
// Thus, proceed to check the next action of the current state.
                if(next_stateID == null)
                {
                        tempState1.cur_action ++;
                        continue;
                }
// Otherwise, check the intended next-state. If it's new to the MDP,
// then at this moment we do not know if the action is valid. In this
// case, add the action along with the intended next-state to the MDP
// and push it onto the stack.
// If the intended next-state is already in the MDP, then check it's
// possibility status. If it is impossible, proceed directly to the next
// action. If it's not known but on the stack (0) or possible (1), then
// the action is guaranteed to be valid (and the current state possible),
// so the action can be added to the current state.

// The exact reachability of the unintended next-states can be checked
```

```
        // later via the loop outside the stack.

        // If the intended next-state is new to the MDP
                    if(!mdp1.states.containsKey(next_stateID))
                    {
                        tempState2 = new MDPstate(next_stateID);
                        mdp1.states.put(next_stateID, tempState2);
                        stack1.push(tempState2);
                        // Add the action
                        tempAct = new
                            MDPaction(genActionID(tempState1.cur_action,
                        num_src), next_stateID);
        // The uniformization step: add a superficial self-loop to every action,
        // so as to avoid potential periodicity of the constructed MDP.
                        tempAct.addNextState(tempState1.id);
                        tempState1.actions.add(tempAct);
                        tempState1.cur_action ++;
                        continue;
                    }
        // Otherwise, check if the next-state is possible. If -1, the next-state
        // is impossible and so is the action. Check the next action.
        // If 0 or 1, the next-state and the action are both possible. Add the
        // action to the current state. Check the next action.
                    else
                    {
                        tempState2 = mdp1.states.get(next_stateID);
                        if(tempState2.possible == -1)
                        {
                            tempState1.cur_action ++;
                            continue;
                        }
                        else
                        {
                            tempState1.possible = 1;
                            tempAct = new
                                MDPaction(genActionID(tempState1.cur_ac
                            tion, num_src), next_stateID);
        // The uniformization step: add a superficial self-loop to every action,
        // so as to avoid potential periodicity of the constructed MDP.
                            tempAct.addNextState(tempState1.id);
                            tempState1.actions.add(tempAct);
                            tempState1.cur_action ++;
                            continue;
                        }
                    }
                }
        }
        return(mdp1);
    }

    // Remove impossible states and assign values to probabilities and rewards.
```

```
public static void updateMDP(MDP mdp1, SystemSpec spec1)
{
        MDPstate curState;
        DataSource curS;
        Iterator<String> itr1 = mdp1.states.keySet().iterator();
        Iterator<DataSource> itr4;
        int digit1, digit2, actDigit;
        String curStateID, nextStateID, actID;
        double reward, recData, backData;
        double[] impactParams = spec1.impactParams;
        // The uniformization factor
        double uni_factor = 0.9;
        while(itr1.hasNext())
        {
                curState = mdp1.states.get(itr1.next());
                // Remove states that are impossible.
                if(curState.possible == -1)
                {
                        itr1.remove();
                        continue;
                }
                // Check each action of the state.
                curStateID = curState.id;
                for(MDPaction curAct: curState.actions)
                {
                        actID = curAct.id;
                        for(MDP_nextState curNext : curAct.next_states)
                        {
                           reward = 0;
                           nextStateID = curNext.id;
                           curNext.ref = mdp1.states.get(nextStateID);
                           // The transition introduced by uniformization
                           if(nextStateID == curAct.next_states.peekLast().id)
                           {
                              curNext.probability = 1-uni_factor;
                              curNext.reward = 0;
                              continue;
                           }
                           itr4 = spec1.sources.iterator();
                           for(int i=0; i<spec1.num_source; i++)
                           {
                             actDigit  = Integer.parseInt(actID.substring(i,  i+1));
                // For the case one full backup is always performed each day
                             digit1 = Integer.parseInt(curStateID.substring(i*2,
                                     i*2+1));
                              digit2  = Integer.parseInt(curStateID.substring(i*2+1,
                                     i*2+2));
                               curS = itr4.next();
                               recData = curS.expLoss;
                               backData = digit1;
                               switch(actDigit)
```

```
                                {
                                case 0:
                                {
                                        recData *= curS.rec_full + 
digit2*curS.rec_partial;
                                        reward += recData;
                                        break;
                                }
                                case 1:
                                {
                                        recData *= curS.rec_full + 
(digit2+1)*curS.rec_partial;
                                        reward += recData + curS.full*(1-Math.exp(-
backData*curS.partial));
                                        break;
                                }
                                default:
                                {
                                        recData *= curS.rec_full;
                                        reward += recData + curS.full;
                                        break;
                                }
                        }
                }
                curNext.probability = uni_factor;
                curNext.reward = impactParams[0]*reward*uni_factor;
                }
        }
    }
}

static String genStateID(int input, int num_src, int [] reqs)
{
        String ID = "";
        // 1, 2 refers to the number of skipped backups, partial backups.
        int digit1, digit2;
        for(int i=1; i<=num_src; i++)
        {
                digit2 = input%(reqs[(num_src-i)*2+1]+1);
                input /= (reqs[(num_src-i)*2+1]+1);
                digit1 = input%(reqs[(num_src-i)*2]);
                input /= (reqs[(num_src-i)*2]);
                ID = Integer.toString(digit1+1) + Integer.toString(digit2) + ID;
        }
        return ID;
}

static String genActionID(int input, int num_src)
{
        String id = "";
        // 0 for no backup; 1 for partial backup; 2 for full backup
        for(int i=0; i<num_src; i++)
        {
                id = Integer.toString(input%3) + id;
                input /= 3;
        }
        return(id);
}

static String genNextID(String curID, int actionNum, int num_src, int num_res,
int[] reqs)
{
        String id1;
        int temp, digit1, digit2, t1;
        int num_op = 0;
        // Generate the representation of the intended next-state
        id1 = "";
        for(int i=0; i< num_src; i++)
        {
                temp = actionNum % 3;
                t1 = num_src-i;
                // digit1 for number of skipped backup, digit2 for partial backup
                digit1 = Integer.parseInt(curID.substring(t1*2-2, t1*2-1));
                digit2 = Integer.parseInt(curID.substring(t1*2-1, t1*2));
                // No backup for a data source
                if(temp == 0)
                {
                        // If maxSkip has been reached
                        if(digit1 >= reqs[t1*2-2])
                        {       return(null);}
                       id1 = Integer.toString(digit1+1) + Integer.toString(digit2)
+ id1;
                }
                // Partial backup for a data source
                else if(temp == 1)
                {
                        // If maxPB has been reached
                        if(digit2 >= reqs[t1*2-1])
                        {       return(null);}
                        id1 = "1" + Integer.toString(digit2+1) + id1;
                        num_op ++;
                }
                else if(temp == 2)
                {
                        id1 = "10" + id1;
                        num_op ++;
                }
                actionNum /= 3;
        }
        // Check for resource violation.
        if(num_op > num_res)
        {       return(null);}
        // For the case where one full backup is always performed each day
        return(id1);
    }
}
```

Solver.java

```
import java.util.*;

public class Solver
{
// Solve the given MDP via value iteration. The returned value indicates
// whether there is a change in the optimal policy.
  public static int valueIteration(MDP mdp1, double discount, double threshold)
  {
        double difference, old_value, new_value;
        int count;

        // Initialization
        for(MDPstate curState : mdp1.states.values())
        {       curState.value = 0; }
        count = 0;

        // Record the current optimal policy
        MDPaction[] old_policy = new MDPaction[mdp1.states.size()];
        int ind = 0;
        for(MDPstate curState : mdp1.states.values()) {
                old_policy[ind] = curState.optimAction;
                ind ++;
        }

        // Main part
        while(true) {
                difference = 0;
                // Compute new state values
                for(MDPstate curState : mdp1.states.values()) {
                        old_value = curState.value;
                        new_value = findOptimal(curState, discount);
                        curState.optimAction = curState.tempAct;

                        difference = difference<Math.abs(new_value-old_value) ?
                                        Math.abs(new_value-old_value) : difference;
                }
                // Update the state values
                for(MDPstate curState : mdp1.states.values()) {
                        curState.value = curState.tempVal;
                        curState.tempVal = 0;
                }
                count ++;
                if(difference <= threshold) {
                        break;
```

```
			}
			if(count >= 1000) {
				System.out.println("Maximum number of iteration reached.");
				 break;
			}
		}

		// Check if there is policy change and return appropriately.
		ind = 0;
		int diff_count = 0;
		for(MDPstate curState : mdp1.states.values()) {
			if(old_policy[ind] != curState.optimAction)
				diff_count ++;
			ind ++;
		}
		return(diff_count);
	}

	static void policyIteration(MDP mdp1, double discount, double threshold)
	{
		int count1, count2;
		double difference, old_val, new_val;
		boolean different;

		// Initialization
		for(MDPstate curState : mdp1.states.values()) {
			curState.value = 0;
			curState.tempVal = 0;
			curState.optimAction = curState.actions.peekFirst();
		}
		count1 = 0;

		// Main part
		while(true) {
			// Compute new state values
			count2 = 0;
			while(true) {
				difference = 0;
				for(MDPstate curState : mdp1.states.values()) {
					evalAction(curState.optimAction, discount);
					curState.tempVal = curState.optimAction.value;
				}
				// Update the state values
				for(MDPstate curState : mdp1.states.values()) {
					old_val = curState.value;
					new_val = curState.tempVal;
				       difference = difference < Math.abs(new_val-old_val) ?
							Math.abs(new_val-old_val):difference;
					curState.value = curState.tempVal;
					curState.tempVal = 0;
				}
```

```
                count2 ++;
                if(difference <= threshold)
                {       break; }
                if(count2 >= 10000)
                {       break; }
            }

            System.out.println(count2);
            for(MDPstate curState : mdp1.states.values()) {
                System.out.print(curState.id + " ");
                System.out.println(curState.value);
            }

            // Compute the new policy
            different = false;
            for(MDPstate curState : mdp1.states.values()) {
                findOptimal(curState, discount);
                if(!curState.optimAction.id.equals(curState.tempAct.id))
                    different = true;
                curState.optimAction = curState.tempAct;
            }

            count1 ++;
            if(!different) {
                System.out.println("Optimal policy found.");
                break;
            }
            if(count1 >= 1) {
               System.out.println("Maximum number of iteration reached.");
                break;
            }
        }
        System.out.println(count1);
}

static double findOptimal(MDPstate state1, double discount)
{
        MDPaction opt_action = null;

        for(MDPaction curAct : state1.actions) {
            evalAction(curAct, discount);
            if(opt_action == null)
                opt_action = curAct;
            else if(opt_action.value > curAct.value)
                opt_action = curAct;
        }
        state1.tempAct = opt_action;
        state1.tempVal = opt_action.value;
        return(state1.tempVal);
}
```

```
// Return the expected value of an action (i.e. computing the Q function)
static void evalAction(MDPaction action1, double discount)
{
	double exp_reward = 0;
	for(MDP_nextState curNext : action1.next_states) {
		exp_reward += (curNext.reward + 
curNext.ref.value*discount)*curNext.probability;
	}
	action1.value = exp_reward;
}

// Solve the MDP for its steady-state probabilities under the current policy
public static void power_MDP(MDP mdp1, double threshold)
{
	double difference, old_value, new_value;
	int count;

	// Initialization
	for(MDPstate curState : mdp1.states.values()) {
		curState.tempVal = 0;
		curState.probability = 1.0/mdp1.states.size();
	}
	count = 0;

	// Main part
	while(true) {
		mdp1.totalProb = 0;
		for(MDPstate curState : mdp1.states.values()) {
			for(MDP_nextState curNext : 
curState.optimAction.next_states) {
				curNext.ref.tempVal += 
curState.probability*curNext.probability;
				mdp1.totalProb += 
curState.probability*curNext.probability;
			}
		}

		difference = 0;
		for(MDPstate curState : mdp1.states.values()) {
			old_value = curState.probability;
			new_value = curState.tempVal/mdp1.totalProb;
			difference = difference < Math.abs(new_value-old_value) ?
					Math.abs(new_value-old_value) : difference;
			curState.probability = new_value;
			curState.tempVal = 0;
		}

		count ++;
		if(difference <= threshold) {
			break;
		}
```

```
            if(count == 10000) {
                  System.out.println
                                    ("Power MDP: Maximum number of runs for power
method reached.");
                  break;
            }
      }
}

// Solve the availablity models for steady-state probabilities under the
current policy
static void power_MC(DataSource src1, double threshold)
{
      Iterator<String> itr1;
      double uni_factor = 0.9;
      State curState;
      double difference, old_value, new_value;
      int count;

      // Initialization
      itr1 = src1.avail_states.keySet().iterator();
      while(itr1.hasNext()) {
            curState = src1.avail_states.get(itr1.next());
            curState.tempVal = 0;
            curState.embed_Prob = 0;
      }

      itr1 = src1.avail_states.keySet().iterator();
      src1.avail_states.get(itr1.next()).embed_Prob = 1;
      count = 0;

      // Main part
      while(true) {
            itr1 = src1.avail_states.keySet().iterator();
            src1.totalProb = 0;
            while(itr1.hasNext()) {
                  curState = src1.avail_states.get(itr1.next());
                  for(int i=0; i<curState.nextID.length; i++) {
                        src1.avail_states.get(curState.nextID[i]).tempVal +=
                              curState.embed_Prob*curState.tranProb[i];
                        src1.totalProb +=
                              curState.embed_Prob*curState.tranProb[i];
                  }
                  curState.tempVal += 1-uni_factor;
                  src1.totalProb += 1-uni_factor;
            }

            difference = 0;
            itr1 = src1.avail_states.keySet().iterator();
            while(itr1.hasNext()) {
                  curState = src1.avail_states.get(itr1.next());
```

```
                        old_value = curState.embed_Prob;
                        new_value = curState.tempVal/src1.totalProb;
                        difference = difference < Math.abs(new_value-old_value) ?
                                        Math.abs(new_value-old_value) : difference;
                        curState.embed_Prob = new_value;
                        curState.tempVal = 0;
                }

                count ++;
                if(difference <= threshold) {
                        System.out.println
                        ("Power MC: Steady-state probabilities found by
convergence.");
                        System.out.println("Number of iterations:
"+Integer.toString(count));
                        break;
                }
                if(count == 10000) {
                        System.out.println
                        ("Power MC: Maximum number of runs for power method
reached.");
                break;
        }
    }
}

// Update the parameters of the data sources based on the MDP state
distribution under the current policy
static void updateParams(MDP mdp1, SystemSpec spec1)
{
        String curID;
        int numPartial;
        for(DataSource curSrc : spec1.sources)
                curSrc.expPartial = 0;

// Update the mean time to recover of the availability models with
probabilities of the MDP model
        for(MDPstate curState : mdp1.states.values()) {
                curID = curState.id;
                int i=0;
                for(DataSource curSrc : spec1.sources) {
                        numPartial = Integer.valueOf(curID.substring(i*2+1,
i*2+2));
                        curSrc.expPartial += curState.probability * numPartial;
                        i++;
                }
        }

        // Update the parameter values for the data sources.
        for(DataSource curSrc : spec1.sources) {
                System.out.println("ExpLoss: "+curSrc.expLoss);
```

```
            System.out.println
                    ("ExpRecT:
"+(curSrc.rateRecFull+curSrc.expPartial*curSrc.rateRecPartial));
                    System.out.println("New ExpLoss: "+curSrc.expLoss);
            }
        }
}
```

The Decomposition Framework

SimuDataSource.java

```
public class SimuDataSource
{
        // "ID" is also used to implement some heuristic policies
        public int ID;
        public int RPO, RTO;
        public int skipped, curPB;
        public double backFB, backPB, recFB, recPB;
        public double failRate, OWfactor;
        public double prevFailTime;
        // The action adopted by the most recent decision point
        public String curAction;
        // Value function entries from approximate MDP analysis
        public double[] approxValues;

        public SimuDataSource() {}

        public SimuDataSource(int pID, int pRPO, int pRTO, double[] back_params,
                double[] avail_params)
        {
                ID = pID;
                RPO = pRPO;         RTO = pRTO;
                backFB = back_params[0];  backPB = back_params[1];
                recFB = back_params[2];   recPB = back_params[3];
                OWfactor = back_params[4];
                failRate = avail_params[0];
                skipped = 1;
                curPB = 0;
                prevFailTime = 0;
                approxValues = new double[RPO*(RTO+1)];
        }

        public double recTime()
        {
                return(recFB+curPB*recPB);
        }

        public double backupTime(int action)
        {
```

```
		switch(action) {
			case 0:
				return 0;
			case 1:
				return PBTime();
			default:
				return FBTime();
		}
	}

	private double FBTime()
	{
		double time = backFB; // + backPB*skipped;
		return(time);
	}

	private double PBTime()
	{
//		double time = skipped*backPB*OWfactor;
		double time = backFB*(1-Math.exp(-skipped*backPB));
		return(time);
	}
}
```

DecompMDP.java (many of the comments are for different scenarios; uncomment as needed)

```
import java.io.BufferedWriter;
import java.io.IOException;

import analytical.*;
import simulative.SimuDataSource;
import mosek.fusion.*;

public class DecompMDP
{
	static int factor1 = 1;
	static int factor2 = 1;
	// Mean data quantity for full, for partial, mean quantity for recovery from full, partial
	// Then the expected number of chunk losses which will be updated during iteration
	static double[][] back_params = {
			{10*factor1, 1.5*factor2, 10.0/500*factor1, 1.5/500*factor2, 1},
			{9*factor1, 1.2*factor2, 9.0/500*factor1, 1.2/500*factor2, 1},
			{11*factor1, 1.4*factor2, 11.0/500*factor1, 1.4/500*factor2, 1},
			{14*factor1, 1.3*factor2, 14.0/500*factor1, 1.3/500*factor2, 1},
			{12*factor1, 1.1*factor2, 12.0/500*factor1, 1.1/500*factor2, 1},
			{15*factor1, 1.0*factor2, 13.0/500*factor1, 1.2/500*factor2, 1}};
```

```
//	static double[][] back_params = {{10*factor1, 1.5*factor2, 10.0/500*factor1,
1.5/500*factor2, 1},
//			{9*factor1, 1.2*factor2, 9.0/500*factor1, 1.2/500*factor2, 2},
//			{11*factor1, 1.4*factor2, 11.0/500*factor1, 1.4/500*factor2, 3},
//			{14*factor1, 1.3*factor2, 14.0/500*factor1, 1.3/500*factor2, 2},
//			{12*factor1, 1.1*factor2, 12.0/500*factor1, 1.1/500*factor2, 1}
	// rate for re-replication with 2 and 1 replicas, two rate parameters for recovery time
	// (full and partial copy)
	static double[][] avail_params = {{60.0/30, 60.0/3, 1.0/1, 1.0/0.1},
			{60.0/23, 60.0/2.2, 1.0/0.9, 1.0/0.12},
			{60.0/25, 60.0/2.3, 1.0/0.9, 1.0/0.1},
			{60.0/20, 60.0/1.5, 1.0/1.1, 1.0/0.11},
			{60.0/27, 60.0/2.0, 1.0/1.0, 1.0/0.13}};
	// RPO/RTO requirements
	static int[][] req = {{5,3}, {4,3}, {5,5}, {3,1}, {3,5}, {4,4}};
	// Resource constraint
	static int resource = 2; //5;

	public static void main(String[] args)
	{
//		MainMDP(back_params, req, resource);
	}

	// Use penalty term to decompose original MDP into one per source (no multiplier)
	public static void MainMDP(SimuDataSource[] sources, int resource, double discount,
			double penalty, BufferedWriter writer1) throws IOException
	{
		// Penalty for violating RTO/RPO
		double penalty1 = penalty;
		int num_src = sources.length;
		SimuDataSource curS;
		for(int n=0; n<num_src; n++) {
			curS = sources[n];
			Model M = new Model("MainMDP");
			try
			{
				// Variables
				Variable Vs =
M.variable("S"+n,curS.RPO*(curS.RTO+1),Domain.greaterThan(0));

				// Construct constraints
				double value1, value2, value3;
				for(int m=1; m<=curS.RPO; m++)
					for(int k=0; k<=curS.RTO; k++) {
```

```
                                        String name = n+"s-"+m+"_"+k;
                                        Variable tempV1 = Vs.index((m-
1)*(curS.RTO+1)+k);

                                        Variable tempV2 = Vs.index(0);
                                        Expression fullCon, partialCon, noneCon;
                                        value1 = curS.backFB +
                                                        (1-Math.exp(-
curS.failRate*24))*curS.recFB;

                                        value2 = curS.backFB*(1-Math.exp(-
m*curS.backPB)) +
                                                        (1-Math.exp(-
curS.failRate*24))*(curS.recFB+(k+1)*curS.recPB);
                                        // value2 = m*curS.backPB*curS.OWfactor +
                                        //              (1-Math.exp(-
curS.failRate*24))*(curS.recFB+(k+1)*curS.recPB);
                                        value3 = (1-Math.exp(-
curS.failRate*24))*(curS.recFB+k*curS.recPB);
                                        // Introduce additional penalties for states
based on how far they are
                                        // from the RPO of the given source
//                                      if(m==curS.RPO) {
//                                      value1 += penalty1*(m-1)/curS.RPO;
//                                      value2 += penalty1*(m-1)/curS.RPO;
//                                      value3 += penalty1*(m-1)/curS.RPO;
                                        value1 += penalty1*Math.pow(1.0*(m-
1)/curS.RPO, 3);
                                        value2 += penalty1*Math.pow(1.0*(m-
1)/curS.RPO, 2);
                                        value3 += penalty1*Math.pow(1.0*(m-
1)/curS.RPO, 1);
//                                      }

                                        fullCon = Expr.sub(tempV1,
Expr.mul(discount,tempV2));
                                        M.constraint(name+"full", fullCon,
Domain.lessThan(value1));
                                        if(k<curS.RTO) {
                                                tempV1 = Vs.index((m-
1)*(curS.RTO+1)+k);

                                                tempV2 = Vs.index(k+1);
                                                partialCon = Expr.sub(tempV1,
Expr.mul(discount,tempV2));
                                                M.constraint(name+"partial",
partialCon, Domain.lessThan(value2));
```

```
                                                }
//                                              else {
//                                                      tempV1 = Vs[n].index((m-
1)*(curS.RTO+1)+k);
//                                                      tempV2 = Vs[n].index(k);
//                                                      partialCon = Expr.sub(tempV1,
Expr.mul(discount,tempV2));
//                                                      M.constraint(name+"partial",
Expr.add(partialCon, Vs[0]),
//
        Domain.lessThan(value2+penalty));
//                                              }
                                                if(m<curS.RPO) {
                                                        tempV1 = Vs.index((m-
1)*(curS.RTO+1)+k);
                                                        tempV2 = Vs.index(m*(curS.RTO+1)+k);
                                                        noneCon = Expr.sub(tempV1,
Expr.mul(discount,tempV2));
                                                        M.constraint(name+"none", noneCon,
Domain.lessThan(value3));
                                                }
//                                              else {
//                                                      tempV1 = Vs[n].index((m-
1)*(curS.RTO+1)+k);
//                                                      tempV2 = Vs[n].index((m-
1)*(curS.RTO+1)+k);
//                                                      noneCon = Expr.sub(tempV1,
Expr.mul(discount,tempV2));
//                                                      M.constraint(name+"none", noneCon,
Domain.lessThan(value3+penalty));
//                                              }
                                        }

                                // Define the objective
                                Expression objExpr = Expr.constTerm(new double[] {0});
                                double factor;
                                factor = 1.0;///(curS.RPO*(curS.RTO+1))/1;
                                for(int m=0; m<Vs.size(); m++)
                                        objExpr = Expr.add(objExpr,
Expr.mul(factor,Vs.index(m)));
                                M.objective("obj", ObjectiveSense.Maximize, objExpr);

                                // Solve the problem
//                              M.acceptedSolutionStatus(AccSolutionStatus.Anything);
```

```
                                        M.solve();

                                        // Retrieve approximate value function entries
                                        int index;
                                        double[] tempV;
                                        tempV = Vs.level();
                                        for(int m=1; m<=curS.RPO; m++)
                                                for(int k=0; k<=curS.RTO; k++) {
                                                        index = (m-1)*(curS.RTO+1)+k;
                                                        curS.approxValues[index] = tempV[index];
                                                }

                                        // Check which constraints are tight so as to recover policy
                                        curS = sources[n];
                                        double dual1, dual2, dual3, total;
                                        for(int m=1; m<=curS.RPO; m++)
                                                for(int k=0; k<=curS.RTO; k++)
                                                {
                                                        String name = n+"s-"+m+"_"+k;
                                                        dual1 = 0; dual2 = 0; dual3 = 0;
//                                                      System.out.print(n+"-"+m+"-"+k+": ");
                                                        writer1.write(n+"-"+m+"-"+k+": ");
//                                                      System.out.println(Vs.level((m-
1)*(curS.RTO+1)+k)+" ");
//                                                      System.out.println(" full:
"+M.getConstraint(name+"full").level(0));
//                                                      System.out.println("--------------->
"+M.getConstraint(name+"full").dual()[0]);
                                                        dual1 = M.getConstraint(name+"full").dual()[0];
                                                        if(k<curS.RTO)
                                                                dual2 =
M.getConstraint(name+"partial").dual()[0];
                                                        if(m<curS.RPO)
                                                                dual3 =
M.getConstraint(name+"none").dual()[0];
                                                        total = dual1+dual2+dual3;
//                                                      if(m<curS.RPO && Math.abs(dual3/total)>1e-
10)
//                                                              System.out.print("none:
"+dual3/total+"; ");
//                                                      if(k<curS.RTO && Math.abs(dual2/total)>1e-10)
//                                                              System.out.print("partial:
"+dual2/total+"; ");
//                                                      if(Math.abs(dual1/total)>1e-10)
//					System.out.print("full: "+dual1/total);
//					System.out.println();
						if(m<curS.RPO && Math.abs(dual3/total)>1e-
10)
							writer1.write("none: "+dual3/total+";
");
						if(k<curS.RTO && Math.abs(dual2/total)>1e-10)
							writer1.write("partial: "+dual2/total+";
");
						if(Math.abs(dual1/total)>1e-10)
							writer1.write("full: "+dual1/total);
						writer1.write("\n");
//						System.out.println("none: "+dual3/total+";
partial: "+dual2/total+" full: "+dual1/total);
//						if(k<curS.RTO)
//						{
//							System.out.println(" partial:
"+M.getConstraint(name+"partial").level(0));
//							System.out.println("--------------->
"+M.getConstraint(name+"partial").dual()[0]);
//						}
//						if(m<curS.RPO)
//						{
//							System.out.println(" none:
"+M.getConstraint(name+"none").level(0));
//							System.out.println("--------------->
"+M.getConstraint(name+"none").dual()[0]);
//						}
//						System.out.println();
					}
				} catch (SolutionError e) {
					e.printStackTrace();
				}
				finally
				{ M.dispose(); }
			}
//			System.out.println("+++++++++++++++++++++++++++++++++");
		}

		// Another decomposed version, using the original MDP solver
		public static void MainMDP2(SimuDataSource[] sources, int resource, double discount,
double penalty)
		{
			SystemSpec spec1;
		int num_src = sources.length;
		double[] back_param = new double[4];
		double[] avail_param = new double[4];
		SimuDataSource curS;
		MDP mdp1;
		int curFB, curPB;
		double[] impactParams = {1.0, 1.0, 5.0};
		for(int n=0; n<num_src; n++) {
			curS = sources[n];
			spec1 = new SystemSpec();
			spec1.setResource(1);
			spec1.setImpactParam(impactParams);
			back_param[0] = curS.backFB;
			back_param[1] = curS.backPB;
			back_param[2] = curS.recFB;
			back_param[3] = curS.recPB;
			avail_param[0] = curS.failRate;
			avail_param[1] = 0; avail_param[2] = 0; avail_param[3] = 0;
			spec1.addSource(new
DataSource("src"+n,curS.RPO,curS.RTO,back_param,avail_param));

			mdp1 = Generator.genMDP(spec1);
			Generator.updateMDP(mdp1, spec1);
			if(mdp1.states.size() == 0) {
				System.out.println("The MDP is infeasible.");
				return;
			}

			Solver.valueIteration(mdp1, discount, 0.01);
			Solver.power_MDP(mdp1, 0.001);

			for(MDPstate curState : mdp1.states.values()) {
				curFB = (int)curState.id.charAt(0)-48;
				curPB = (int)curState.id.charAt(1)-48;
				curS.approxValues[(curFB-1)*(curS.RTO+1)+curPB] =
curState.value;
			}

			for(int m=1; m<=curS.RPO; m++)
				for(int k=0; k<=curS.RTO; k++) {
					System.out.println(n+"-"+m+"-"+k+": "+
							curS.approxValues[(m-
1)*(curS.RTO+1)+k]);
```

```
                                                curS.curAction = 
mdp1.states.get(m+""+k).optimAction.id;
                                                if(curS.curAction.equals("0"))
                                                        System.out.println("none");
                                                else if(curS.curAction.equals("1"))
                                                        System.out.println("partial");
                                                else {
                                                        System.out.println("full"+" "+curS.curAction);
                                                }

                                        }
                                }
                        }
}
```

Chapter 7

The Chunk Reliability Model (MATLAB)

```
clear all;
clc;

repTime = [10, 100, 1000, 10000, 100000];
failR = 1.0./[4.3*30*24*3600, 10.2*365*24*3600];
aux1 = [3,2];
aux2 = [2,1];

Pend = zeros(length(repTime), 2);
ET_final = zeros(length(repTime), 2);
for n = 1:length(repTime)
    T = repTime(n);
    % Some common terms
    term1 = aux1*failR';
    term2 = aux2*failR';
    term3 = exp(-T*term1);
    term4 = exp(-T*term2);
    term5 = exp(-2*T*sum(failR));

    % Transition matrix
    Tmat = zeros(5,5);
    Tmat(4,4) = 1;
    Tmat(5,5) = 1;
    Tmat(1,5) = 0.5*(1-term5);
    Tmat(1,1) = 0.5*(1-term5);
    Tmat(1,2) = term5;
    Tmat(3,5) = -failR(1)/term1*(1-term3) + 0.5*(1-term5);
    Tmat(3,1) = -failR(2)/term1*(1-term3) + 0.5*(1-term5);
    Tmat(3,3) = sum(failR)/term1*(1-term3);
    Tmat(3,2) = (1-exp(-failR(1)*T))*term5;
    Tmat(3,4) = term3;
    Tmat(2,5)  = sum(failR)/term2*(1-term4)  - sum(failR)/term1*(1-term3);
    Tmat(2,1) = failR(1)/term2*(1-term4) + failR(2)/term1*(1-term3);
    Tmat(2,2) = failR(1)/term1*(1-term3);
    Tmat(2,3) = (1-exp(-T*sum(failR)))*term4;
    Tmat(2,4) = term3;

    initP = [failR(2), 2*failR(1), sum(failR)]/term1;

    % Expected time to transition
    ET = zeros(3,5);
    ET(1,5) = 0.5*(-T*term5+(1-term5)/2/sum(failR));
    ET(1,1) = ET(1,5);
    ET(1,2) = T*term5;
    ET(3,5) = 0.5*(-T*term5+(1-term5)/2/sum(failR)) - ......
                failR(1)/term1*(-T*term3+(1-term3)/term1);
    ET(3,1) = 0.5*(-T*term5+(1-term5)/2/sum(failR)) - ......
                failR(2)/term1*(-T*term3+(1-term3)/term1);
    ET(3,3) = sum(failR)/term1*(-T*term3+(1-term3)/term1);
    ET(3,2) = T*Tmat(3,2);
    ET(3,4) = T*Tmat(3,4);
    ET(2,5) = sum(failR)/term2*(-T*term4+(1-term4)/term2) - ......
                sum(failR)/term1*(-T*term3+(1-term3)/term1);
    ET(2,1) = failR(1)/term2*(-T*term4+(1-term4)/term2) + ......
                failR(2)/term1*(-T*term3+(1-term3)/term1);
    ET(2,2) = failR(1)/term1*(-T*term3+(1-term3)/term1);
    ET(2,3) = T*Tmat(2,3);
    ET(2,4) = T*Tmat(2,4);

    % Compute absorption probabilities
    Q = Tmat(1:3,1:3);
    C = Tmat(1:3,4:5);
    Pend(n,:) = initP/(eye(3)-Q)*C;

    % Compute mean time
    Tmod4 = Tmat(1:4,1:4);
    Tmod4 = Tmod4./repmat(sum(Tmod4,2),1,4);
    Tmod5 = Tmat([1,2,3,5],[1,2,3,5]);
    Tmod5 = Tmod5./repmat(sum(Tmod5,2),1,4);

    Q = Tmod4(1:3,1:3);
    EV4 = initP/(eye(3)-Q);
    ET4 = sum(ET(:,1:4),2)./sum(Tmat(1:3,1:4),2);
    ET_final(n,1) = EV4*ET4;

    Q = Tmod5(1:3,1:3);
    EV5 = initP/(eye(3)-Q);
    ET5 = sum(ET(:,[1,2,3,5]),2)./sum(Tmat(1:3,[1,2,3,5]),2);
    ET_final(n,2) = EV5*ET5;
end
```

The distribution time approximation (here use host performance model; MATLAB)

```
clear all;
clc;

sizes = [0.05, 1000, 64];
variants = [1, 1.5, 2, 2.5, 3];
mu_net = [2.5, 1.25e-4, 2.604e-3];
mu_CPU = [100, 2, 15.625];
mu_disk = [1.25, 6.25e-5, 4.883e-4];
prob = [1,1,0.25; ......
        1,1,0.5; ......
        1,1,0.5];

% Depth: variant;
% Row: stream
% Column: station (net, CPU, disk)
util = zeros(length(variants),3,3);
EN = zeros(length(variants),3,3);
ERT = zeros(length(variants),3,3);
ERT_total = zeros(length(variants),3);
tput = zeros(length(variants),3);
util_device = zeros(length(variants),3);
for n = 1:length(variants)
    lambda = [0.5, 3e-5*variants(n), 2.7618e-8];

    lambda_net = lambda.*prob(:,1)';
    lambda_CPU = lambda.*prob(:,2)';
    lambda_disk = lambda.*prob(:,3)';

    util(n,:,1) = lambda_net./mu_net;
    util(n,:,2) = lambda_CPU./mu_CPU;
    util(n,:,3) = lambda_disk./mu_disk;

    util_T1 = reshape(util(n,1,:),1,3);
    util_T2 = reshape(util(n,2,:),1,3);
    util_R = reshape(util(n,3,:),1,3);
    util_device(n,:) =
sum(reshape(util(n,:,:),size(util,2),size(util,3)),1);

    EN(n,2,:) = util_T2./(1-util_device(n,:));
    EN(n,3,:) = util_R./(1-util_device(n,:));
    ERT(n,2,:) = EN(n,2,:)/lambda(2);
    ERT(n,3,:) = EN(n,3,:)/lambda(3);
    ERT_total(n,2) = sum(ERT(n,2,:));
    ERT_total(n,3) = sum(ERT(n,3,:));

    tput(n,2) = sizes(2)/ERT_total(n,2);
end
```

References

[1] R. Ghosh, F. Longo, V. K. Naik, and K. S. Trivedi, “Modeling and Performance Analysis of Large Scale IaaS Clouds”, Elsevier Future Generation Computing Systems (FGCS), 2012.

[2] A. Reibman, and K. S. Trivedi, “Numerical Transient Analysis of Markov Models”, Computers & Operations Research 15, no. 1 (1988): 19-36.

[3] M. Malhotra, J. K. Muppala, and K. S. Trivedi. “Stiffness-Tolerant Methods for Transient Analysis of Stiff Markov Chains”, Microelectronics Reliability 34, no. 11 (1994): 1825-1841.

[4] V. Kulkarni. Modeling and Analysis of Stochastic Systems. Chapman-Hall, 1995.

[5] D. Logothetis, K. S. Trivedi, and A. Puliafito. “Markov Regenerative Models”, In Proceedings of International Computer Performance and Dependability Symposium, 1995, pp. 134-142. IEEE, 1995.

[6] K. S. Trivedi, Probability &Statistics with Reliability, Queuing and Computer Science Applications. John Wiley & Sons, 2002.

[7] G. Ciardo, A. Blakemore, P. F. Chimento, J. K. Muppala, and K. S. Trivedi. “Automated Generation and Analysis of Markov Reward Models Using Stochastic Reward Nets”, In IMA Volumes in Mathematics and its Applications: Linear Algebra, Markov Chains and Queueing Models, C. Meyer, R. J. Plemmons (editors), volume 48, pages 145-191. Springer, 1993.

[8] M. A. Marsan, G. Conte, and G. Balbo. “A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of the Multiprocessor Systems”, ACM Transactions on Computer Systems, 2(2):93-122, 1984.

[9] F. Baskett, K. M. Chandy, R. R. Muntz, and F. G. Palacios. “Open, Closed, and Mixed Networks of Queues with Different Classes of Customers”, Journal of the ACM (JACM) 22, no. 2 (1975): 248-260.

[10] G. Bolch, S. Greiner, H. de Meer, and K. S. Trivedi. “Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications”, John Wiley & Sons, 2nd ed., 2006.

[11] K. M. Chandy, J. H. Howard Jr, and D. F. Towsley. “Product Form and Local Balance in Queueing Networks”, *Journal of the ACM (JACM)* 24, no. 2 (1977): 250-263.

[12] D. F. Towsley. “Queueing Models with State-Dependent Routing”, Journal of the ACM, 27(2):323-337, April 1980.

[13] J. Buzen. “Computational Algorithms for Closed Queueing Networks with Exponential Servers”, Communications of the ACM, 16(9):527-531. September 1973.

[14] M. Reiser, and S. Lavenberg. “Mean-Value Analysis of Closed Multichain Queuing Networks”, Journal of the ACM, 27(2):313-322, April 1980.

[15] K. S. Trivedi, and R. Sahner. “SHARPE at the Age of Twenty Two”, In ACM SIGMETRICS Performance Evaluation Review, Volume 36, Issue 4 (March 2009).

[16] M. L. Puterman. Markov Decision Processes: Discrete Stochastic Dynamic Programming. Vol. 414. Wiley.com, 2009.

[17] P. W. Glynn, and D. L. Iglehart. “Importance Sampling for Stochastic Simulations”, Management Science 35, no. 11 (1989): 1367-1392.

[18] J. Morio, R. Pastel, and F. Le Gland. “An Overview of Importance Splitting for Rare Event Simulation”, European Journal of Physics 31, no. 5 (2010): 1295.

[19] B. Schroeder, and G. A. Gibson. “Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you?” In Proc. of 5th USENIX Conf. on File and Storage Technologies (FAST07), 2007.

[20] J. G. Elerath, and M. Pecht, “Enhanced Reliability Modeling of RAID Storage Systems”, in Proc. of 37th IEEE/IFIP Int'l Conf. on Dependable Computing and Networks (DSN07), pp.175-184, 2007.

[21] J. G. Elerath, “A Simple Equation for Estimating Reliability of an N+1 Redundant Array of Independent Disks (Raid)”, In Proc. of IEEE/IFIP Int'l Conf. on Dependable Computing and Networks (DSN09), pp.484-493, 2009.

[22] E. Rozier, W. Belluomini, V. Deenadhayalan, J. Hafner, KK Rao, and P. Zhou, “Evaluating the Impact of Undetected Disk Errors in RAID Systems”, In Proc. of IEEE/IFIP Int'l Conf. on Dependable Computing and Networks (DSN09), pp.484-493, 2009.

[23] K. M. Greenman, J. S. Plank, and J. J. Wylie, “Meantime to Meaningless: MTTDL, Markov Models, and Storage System Reliability”, In Proc. of USENIX Workshop on Hot Topics in Storage and File Systems, pp.1-5, 2010.

[24] J. G. Elerath, and J. Schindler, “Beyond MTTDL: A Closed-Form RAID 6 Reliability Equation”, ACM Trans. Storage, Vol. 10, No. 2, 2014.

[25] V. Venkastesan, and I. Iliadis, “A General Reliability Model for Data Storage Systems”, In Proc. of 9th Int’l. Conf. on Quantitative Evaluation of Systems (QEST2012), pp. 209-219, 2012.

[26] I. Iliadis, R. Haas, X. Hu, and E. Eleftheriou, “Disk Scrubbing Versus Intra-disk Redundancy for RAID Storage Systems”, ACM Trans. Storage, 7, 2, 2011.

[27] I. Iliadis and V. Venkastesan, "Rebuttal to 'Beyond MTTDL: A Closed-form RAID 6 Reliability Equation'", ACM Trans. Storage, Vol. 11, No. 2, 2015.

[28] F. Machida, J. Xiang, K. Tadano, Y. Maeno, and T. Horikawa, "Performability Analysis of RAID10 versus RAID6", In Supplemental Proc. of 43rd Int'l Conf. on Dependable Computing and Networks (DSN13), 2013.

[29] K. V. Vishwanath, and N. Nagappan, "Characterizing Cloud Computing Hardware Reliability", In Proc. of 1st ACM Symposium on Cloud computing (SoCC), pp. 193-204, 2010.

[30] V. Venkastesan and I. Iliadis, "Effect of Codeword Placement on the Reliability of Erasure Coded Data Storage Systems", In Proc. of 10th Int'l. Conf. on Quantitative Evaluation of Systems (QEST2013), pp. 241-257, 2012.

[31] fio, http://freecode.com/projects/fio

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

[33] M. D. Beaudry, "Performance-Related Reliability Measures for Computing Systems", IEEE Trans. on Computers, vol. C-27, pp. 540-547, June 1978.

[34] R. Huslende, "A Combined Evaluation of Performance and Reliability for Degraded Systems", ACM/SIGMETRICS, pp.157-164, 1981.

[35] R. M. Smith, K. S. Trivedi, and A. Ramesh, "Performability Analysis: Measures, An Algorithm and a Case Study", IEEE Trans. on Computers, Vol. C-37, No. 4, pp. 406-417, Apr, 1988.

[36] K. S. Trivedi, E. Andrade, and F. Machida, "Combining Performance and Availability Analysis in Practice", Advances in Computers, Vol. 84, pp. 1-38, 2012.

[37] D. Logothetis, and K. S. Trivedi, "The Effect of Detection and Restoration Times for Error Recovery in Communication Networks", Network and Systems Management, Vol. 5 No. 2, pp. 173-195, 1997.

[38] H. Sun, T. Tyan, S. Johnson, R. Elling, N. Talagala, and R. B. Wood, "Performability Analysis of Storage Systems in Practice: Methodology and Tools", In Proc. of the 3rd Int'l Conf. on Service Availability, pp. 62-75, 2006.

[39] A. Thomasian, and M. Blaum, "Mirrored Disk Organization Reliability Analysis", IEEE Trans. on Computers, Vol. 55 No.12, pp. 1640-1644, 2006.

[40] A. Thomasian, and J. Xu, "Reliability and Performance of Mirrored Disk Organizations", Computer Journal, Vol. 51, No.6, pp. 615-629, 2008.

[41] A. Thomasian, and Y. Tang, "Performance, Reliability, and Performability of a Hybrid RAID Array and a Comparison with Traditional RAID1 Arrays", Cluster Computing, Vol. 15, No. 3, pp. 239-253, 2012.

[42] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler. "An Analysis of Latent Sector Errors in Disk Drives", In Proc. of ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, pp. 289-300, 2007.

[43] D. C. Sawyer, "Dependability Analysis of Parallel Systems using a Simulation-based Approach", NASA-CR-195762, 1994.

[44] T. J. E. Schwarz, Q. Xin, E. L. Miller, D. D. E. Long, A. Hosopdor, and S. Ng, "Disk Scrubbing in Large Archival Storage Systems". In Proc. of the 12th IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), pp. 409-418, 2004.

[45] A. Dholakia, E. Eleftheriou, X.-Y. Hu, I. Iliadis, J. Menon, and K. K. Rao, "A New Intra-disk Redundancy Scheme for High-Reliability RAID Storage Systems in the Presence of Unrecoverable Errors", ACM Trans. on Storage, Vol. 4, No. 1, pp. 1-42, 2008.

[46] N. Mi, A. Riska, E. Smirni, and E. Riedel, "Enhancing Data Availability in Disk Drives through Background Activities", In Proc. of 38th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 492-501, 2008.

[47] B. Schroeder, S. Damouras, and P. Gill, "Understanding Latent Sector Errors and How to Protect against Them", In Proc. of 8th USENIX Conference on File and Storage Technologies (FAST), pp. 71-84, 2010.

[48] X. Wu, J. Li, and H. Kameda, "Reliability Modeling of Declustered-parity RAID Considering Uncorrectable Bit Errors", IEICE Trans. on Fundamentals of Electronics, Communications and Computer Sciences, Vol. 80, No. 8, pp. 1508-1515, 1997.

[49] E. W. D. Rozier, W. Belluomini, V. Deenadhayalan, J. Hafner, K. K. Rao, P. Zhou, "Evaluating the Impact of Undetected Disk Errors in RAID Systems", In Proc. of 39th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 83-92, 2009.

[50] Amazon S3, http://aws.amazon.com/s3/

[51] Microsoft Azure Store, http://azure.microsoft.com/en-us/services/storage/

[52] EMC Atmos, http://www.emc.com/storage/atmos/atmos.htm

[53] Openstack Swift, http://docs.openstack.org/developer/swift/

[54] Amazon S3 SLA, http://aws.amazon.com/s3/sla/

[55] L. Tomek, and K. S. Trivedi, “Fixed-Point Iteration in Availability Modeling”, in: M. Dal Cin (Ed.), Informatik-Fachberichte, Vol. 91: Fehlertolerierende Rechensysteme, Springer-Verlag, Berlin, 1991.

[56] A. K. Mishra, J. L. Hellerstein, W. Cirne, and C. R. Das. “Towards Characterizing Cloud Backend Workloads: Insights from Google Compute Clusters”, *ACM SIGMETRICS Performance Evaluation Review* 37, no. 4 (2010): 34-41.

[57] D. R. Cox, “A Use of Complex Probabilities in the Theory of Stochastic Processes”, In Mathematical Proceedings of the Cambridge Philosophical Society, vol. 51, no. 02, pp. 313-319. Cambridge University Press, 1955.

[58] W. Fischer, and K. Meier-Hellstern. "The Markov-modulated Poisson Process (MMPP) Cookbook." Performance Evaluation 18, no. 2 (1993): 149-171.

[59] I. L. Mitrany, and B. Avi-Itzhak, “A Many-Server Queue with Service Interruptions”, Operations Research, 16.3 (1968), 628-638.

[60] M. F. Neuts. “Matrix-Geometric Solutions in Stochastic Models: An Algorithmic Approach”, Courier Dover Publications, 1981.

[61] R. Chakka, “Spectral Expansion Solution for Some Finite Capacity Queues”, Annals of Operations Research, 79(1998), 27-44.

[62] N. Akar, N. C. Oğuz, and K. Sohraby, “A Novel Computational Method for Solving Finite QBD Processes”, Stochastic Models 16.2 (2000), 273-311.

[63] A. Bobbio, and K. S. Trivedi, “An Aggregation Technique for the Transient Analysis of Stiff Markov Chains”, In IEEE Transactions on Computers, 100.9 (1986), 803-814.

[64] C. Hirel, B. Tuffin, and K. S. Trivedi, “SPNP: Stochastic Petri Nets. Version 6.0”, In Computer performance evaluation: Modeling tools and techniques; 11th International Conference; TOOLS 2000, Schaumburg, Il., USA, B. Haverkort, H. Bohnenkamp, C. Smith(eds.), Lecture Notes in Computer Science 1786, Springer Verlag, 2000.

[65] K. Mishra, and K. S. Trivedi, “An Unobtrusive Method for Uncertainty Propagation in Stochastic Dependability Models”, International Journal of Reliability and Quality Performance (IJRQP), Vol. 3, No. 1, (2011) 49 –65.

[66] S. Geman, and D. Geman, “Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images”, Pattern Analysis and Machine Intelligence, IEEE Transactions on 6 (1984): 721-741.

[67] P. D. Hoff, A first course in Bayesian statistical methods. Springer Science & Business Media, 2009.

[68] C. M. Bishop, Pattern Recognition and Machine Learning. Springer, 2006.

[69] E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good, “The Cost of Doing Science on the Cloud: the Montage Example”, In Proceedings of the 2008 ACM/IEEE conference on Supercomputing (p. 50). IEEE Press. Nov. 2008.

[70] A. Iosup, S. Ostermann, M. N. Yigitbasi, R. Prodan, T. Fahringer, and D. H. Epema, “Performance Analysis of Cloud Computing Services for Many-Requests Scientific Computing”, In IEEE Transactions on Parallel and Distributed Systems, 22(6), pp. 931-945. 2011.

[71] H. Liu, and S. Wee, “Web Server Farm in the Cloud: Performance Evaluation and Dynamic Architecture”, In Cloud Computing, Springer Berlin Heidelberg, 2009, pp. 369-380.

[72] I. Al-Azzoni, and D. Kondo, “Cost-Aware Performance Modeling of Multi-tier Web Applications in the Cloud”, in: Networked Digital Technologies, Springer Berlin Heidelberg, 2012, pp. 186-196.

[73] A. Bhadani, and S. Chaudhary, “Performance Evaluation of Web Servers using Central Load Balancing Policy over Virtual Machines on Cloud”, In Proceedings of the Third Annual ACM Bangalore Conference (p. 16). ACM. Jan. 2010.

[74] Y. Ueda, and T. Nakatani, “Performance Variations of Two Open-Source Cloud Platforms”, In 2010 IEEE International Symposium on Workload Characterization (IISWC), pp. 1-10. IEEE. Dec. 2010.

[75] G. Juve, E. Deelman, K. Vahi, G. Mehta, B. Berriman, B. P. Berman, and P. Maechling, “Scientific Workflow Applications on Amazon EC2”, In E-Science Workshops, 2009 5th IEEE International Conference on (pp. 59-66). IEEE. Dec. 2009.

[76] W. Voorsluys, J. Broberg, S. Venugopal, and R. Buyya, “Cost of Virtual Machine Live Migration in Clouds: A Performance Evaluation”, In Cloud Computing, Springer Berlin Heidelberg, 2009, pp. 254-265.

[77] H. Liu, C. Z. Xu, H. Jin, J. Gong, and X. Liao, “Performance and Energy Modeling for Live Migration of Virtual Machines”, In Proceedings of the 20th international symposium on High performance distributed computing, pp. 171-182. ACM. Jun. 2011.

[78] S. Toyoshima, S. Yamaguchi, M. Oguchi, “Storage Access Optimization with Virtual Machine Migration and Basic Performance Analysis of Amazon EC2”, In Advanced Information Networking and Applications Workshops (WAINA), 2010 IEEE 24th International Conference on (pp. 905-910). IEEE. Apr. 2010.

[79] J. Dejun, G. Pierre, and C. H. Chi, “EC2 Performance Analysis for Resource Provisioning of Service-Oriented Applications”, In Service-oriented Computing. ICSOC/ServiceWave 2009 Workshops (pp. 197-207). Springer Berlin Heidelberg. Jan. 2010.

[80] P. Desnoyers, T. Wood, P. Shenoy, R. Singh, S. Patil, and H. Vin, "Modellus, Automated Modeling of Complex Internet Data Center Applications", In ACM Transactions on the Web (TWEB), 2012.

[81] H. Khazaei, J. Misic, and V. B. Misic, "Modeling of Cloud Computing Centers using M/G/M Queues", In Distributed Computing Systems Workshops (ICDCSW), 2011 31st International Conference on (pp. 87-92). IEEE. Jun. 2011.

[82] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya, "CloudSim: A Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms", In Software: Practice and Experience, 41(1), 23-50. 2011.

[83] G. Sakellari, and G. Loukas, "A Survey of Mathematical Models, Simulation Approaches and Testbeds used for Research in Cloud Computing", In Simulation Modeling Practice and Theory. 2013.

[84] A. Chervenak, V. Vellanki, and Z. Kurmas, "Protecting File Systems: A Survey of Backup Techniques", *Proc. Joint NASA and IEEE Mass Storage Conference*, 1998.

[85] F. Machida, E. Andrade, D. S. Kim, and K. Trivedi, "Candy: Component-based Availability Modeling Framework for Cloud Service Management using SysML", Proc. IEEE Symposium on Reliable Distributed Systems (SRDS), pp. 209 –218, 2011.

[86] http://httpd.apache.org/

[87] http://rsync.samba.org/

[88] X. Yin, J. Alonso, F. Machida, E. Andrade, and K.S. Trivedi, "Availability Modeling and Analysis for Data Backup and Restore operations", SRDS 2012.

[89] Mural, I., A. Bondavalli, X. Zang, and K. S. Trivedi, "Dependability Modeling and Evaluation of Phased Mission Systems: A DSPN Approach", Dependable Computing for Critical Applications 7, pp. 319-337, IEEE, 1999.

[90] K. Sevcik. "Priority Scheduling Disciplines in Queuing Network Models of Computer Systems". Proc. IFIP 7th World Computer Congress, pp. 565- 570, 1977.

[91] K. M. Chandy, U. Herzog, and L. Woo. "Parametric Analysis of Queuing Networks," IBM J. Res. Dev, vol. 19, no. 1, pp. 36-42, Jan. 1975.

[92] W. Xie, H. Sun, Y. Cao, and K. S. Trivedi, "Modeling of User Perceived Webserver Availability", Proc. IEEE International Conference on Communications, vol. 3, pp. 1796–1800, 2003.

[93] A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller, "Measurement and Analysis of Large-Scale Network File System Workloads," Proc. of the USENIX Annual Technical Conference, pp. 213–226, 2008.

[94] EMC Backup Advisor, http://www.emc.com/products/detail /software/backupadvisor.htm. 2013.

[95] NEC HYDRAstor, http://www.necam.com/HYDRAstor/. 2013.

[96] CA ARCserve, http://www.arcserve.com/ solutions/backup-and-archiving.aspx. 2012.

[97] Symantec Backup Exec, http://www.symantec.com/backup-exec. 2012

[98] L. Cherkasova, A. Zhang, and X. Li, "DP+IP = Design of Efficient Backup Scheduling", Proc. Int. Conf. on Network and Service management (CNSM), pp. 118-125, 2010.

[99] K. Keeton, C. Santos, D. Beyer, J. Chase, and J. Wilkes, "Designing for Disasters", Proc. 3rd Conf. on File and Storage Technologies (FAST04), pp. 59-72, 2004.

[100] K. Keeton, and A. Merchant, "A Framework for Evaluating Storage System Dependability", Proc. of Intl. Conf. on Dependable Systems and Networks (DSN04), pp. 877-886, 2004

[101] S. Gaonkar, K. Keeton, A. Merchant, and W. H. Sanders, "Designing Dependable Storage Solutions for Shared Application Environments", Proc. of Intl. Conf. on Dependable Systems and Networks (DSN06), pp. 371-382, 2006.

[102] E. Rozier, W. Sanders, P. Zhou, N. Mandagere, S. Uttamchandani, and M. Yakushev, "Modeling the Fault Tolerance Consequences of Deduplication", Proc. IEEE Symp. on Reliable Dist. Systems, 2011.

[103] D. Geer, "Reducing the Storage Burden via Data De-duplication", Computer, vol. 41, pp. 15-17, Issue. 12, Dec. 2008.

[104] K. Renuga, S. Tan, Y. Zhu, T.C. Low, and Y. Wang, "Balanced and Efficient Data Placement and Replication Strategy for Distributed Backup Storage Systems", in Proc. of Int. Conf. on Computational Science and Engineering (CSE '09), pp.87-94, 2009.

[105] H. Wang, K. Zhou, and L. Yuan, "Fault-Tolerant Online Backup Service: Formal Modeling and Reasoning", in Proc. IEEE Int. Conf. on Networking, Architecture, and Storage (NAS), pp.452-460, 2009.

[106] S. Gnanasundaram, and A. Shrivastava. Information storage and management. Wiley Publishing, Inc., 2009.

[107] D. M. Smith. "The Cost of Lost Data", Journal of Contemporary Business Practice 6, no. 3 (2003).

[108] N. Meuleau, N. Hauskrecht, K. E. Kim, L. Peshkin, L. P. Kaelbling, T. L. Dean, and C. Boutilier. "Solving very Large Weakly Coupled Markov Decision Processes", In AAAI/IAAI, pp. 165-172. 1998.

[109] C. Guestrin, D. Koller, R. Parr, and S. Venkataraman. "Efficient Solution Algorithms for Factored MDPs", Journal of Artificial Intelligence Research. pp 399-468. 2003.

[110] D. Adelman, and A. J. Mersereau, "Relaxations of Weakly Coupled Stochastic Dynamic Programs", Operations Research, vol. 56, no. 3 (2008), pp. 712-727.

[111] G. Sun, Y. Dong, D. Chen, and J. Wei. "Data Backup and Recovery Based on Data De-duplication", In Artificial Intelligence and Computational Intelligence (AICI), 2010 Int'l Conf. on, vol. 2, pp. 379-382. IEEE, 2010.

[112] J. Kaiser, D. Meister, A. Brinkmann, and S. Effert, "Design of An Exact Data De-duplication Cluster", In Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on, pp. 1-12. IEEE, 2012.

[113] D. N. Tran, F. Chiang, and J. Li. "Friendstore: cooperative Online Backup using Trusted Nodes", In Proc. of the 1st Workshop on Social Network Systems, pp. 37-42. ACM, 2008.

[114] L. Toka, D. Matteo, and P. Michiardi. "Online Data Backup: A Peer-assisted Approach", In Proc. of 10th Int'l Conf. on Peer-to-Peer Computing (P2P), pp. 1-10, 2010.

[115] L. M. Kaufman, "Data Security in the World of Cloud Computing", Security & Privacy, IEEE 7, no. 4,pp. 61-64, 2009.

[116] L. Courtes, O. Hamouda, M. Kaaniche, M-O. Killijian, and D. Powell, "Dependability Evaluation of Cooperative Backup Strategies for Mobile Devices", In Proc. of 13th Pacific Rim International Symposium on Dependable Computing (PRDC2007), pp. 139-146, 2007.

[117] R. Xia, X. Yin, J. Alonso, F. Machida, and K. S. Trivedi, "Performance and Availability Modeling of IT Systems with Data Backup and Restore", IEEE Tran. on Dependable and Secure Computing, in press.

[118] S. Nakamura, K. Nakayama, and T. Nakagawa, "Optimal Backup Interval of Database by Incremental Backup Method", In Proc. of Int'l Conf. on Industrial Engineering and Engineering Management (IEEM 2009), pp. 218-222, 2009.

[119] K. Renuga, S. S. Tan, Y. Zhu, T. Low, and Y. Wang, "Balanced and Efficient Data Placement and Replication Strategy for Distributed Backup Storage Systems", In Proc. of Int'l Conf. on Computational Science and Engineering (CSE'09), vol. 1, pp. 87-94, 2009.

[120] S. Gaonkar, K. Keeton, A. Merchant, and W. H. Sanders, "Designing Dependable Storage Solutions for Shared Application Environments", IEEE Transactions on Dependable and Secure Computing, vol. 7, no. 4,pp.366-380, 2010.

[121] F. B. Schmuck, and R. L. Haskin, “GPFS: A Shared-Disk File System for Large Computing Clusters”, In FAST, vol. 2, p. 19. 2002.

[122] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, “The Hadoop Distributed File System”, In Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on, pp. 1-10. IEEE, 2010.

[123] F. Wang, S. Oral, G. Shipman, O. Drokin, T. Wang, and I. Huang, “Understanding Lustre Filesystem Internals”, Oak Ridge National Laboratory, National Center for Computational Sciences, Tech. Rep (2009).

[124] S. Ghemawat, H. Gobioff, and S. Leung, “The Google File System”, In ACM SIGOPS operating systems review, vol. 37, no. 5, pp. 29-43. ACM, 2003.

[125] S. Gilbert, and N. Lynch, “Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services”, ACM SIGACT News 33, no. 2 (2002): 51-59.

[126] A. S. Tanenbaum, and M. V. Steen. Distributed Systems. Prentice-Hall, 2007.

[127] W. Vogels, “Eventually Consistent”, Communications of the ACM 52, no. 1 (2009): 40-44.

[128] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, “Dynamo: Amazon's Highly Available Key-value Store”, In ACM SIGOPS Operating Systems Review, vol. 41, no. 6, pp. 205-220. ACM, 2007.

[129] J. Dean, and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters”, Communications of the ACM 51, no. 1 (2008): 107-113.

[130] K. McKusick, and S. Quinlan. "GFS: Evolution on Fast-forward”, Communications of the ACM 53, no. 3 (2010): 42-49.

[131] G. Ciardo, and K. S. Trivedi. "A Decomposition Approach for Stochastic Reward Net Models”, Performance Evaluation 18, no. 1 (1993): 37-59.

[132] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V. Truong, L. Barroso, C. Grimes, and S. Quinlan, “Availability in Globally Distributed Storage Systems”, In OSDI, pp. 61-74. 2010.

[133] Y. Liu, and K. S. Trivedi, “Survivability Quantification: The Analytical Modeling Approach”, *International Journal of Performability Engineering* 2, no. 1 (2006): 29.

[134] D. P. Heyman, and M. J. Sobel (2003). "5.8 Superposition of Renewal Processes". Stochastic Models in Operations Research: Stochastic Processes and Operating Characteristics.

[135] P. J. Denning, and J. P. Buzen, "The Operational Analysis of Queueing Network Models", *ACM Computing Surveys (CSUR)* 10, no. 3 (1978): 225-261.

[136] R. W. Wolff, "Poisson Arrivals See Time Averages", Operations Research 30.2 (1982): 223-231.

[137] D. K. Gifford, "Weighted Voting for Replicated Data", In Proceedings of the seventh ACM symposium on Operating systems principles, pp. 150-162. ACM, 1979.

[138] K. Petersen, M. Spreitzer, D. Terry, and M. Theimer, "Bayou: Replicated Database Services for World-wide Applications", In Proceedings of the 7th workshop on ACM SIGOPS European workshop: Systems support for worldwide applications, pp. 275-280. ACM, 1996.

[139] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi et al., "Oceanstore: An Architecture for Global-scale Persistent Storage", ACM Sigplan Notices 35, no. 11 (2000): 190-201.

[140] A. Gharaibeh, S. Al-Kiswany, and M. Ripeanu, "Thriftstore: Finessing Reliability Trade-offs in Replicated Storage Systems", Parallel and Distributed Systems, IEEE Transactions on 22, no. 6 (2011): 910-923.

[141] J. Baker, C. Bond, J. C. Corbett, J. J. Furman, A. Khorlin, J. Larson, J. Leon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing Scalable, Highly Available Storage for Interactive Services", In CIDR, vol. 11, pp. 223-234. 2011.

[142] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha, "Spanstore: Cost-effective Geo-replicated Storage Spanning Multiple Cloud Services", In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, pp. 292-308. ACM, 2013.

[143] S. Ramabhadran, and J. Pasquale, "Analysis of Long-Running Replicated Systems", In INFOCOM, vol. 2006, pp. 1-9. 2006.

[144] J. Zhang, and P. Honeyman, "Performance and Availability Tradeoffs in Replicated File Systems", In Cluster Computing and the Grid, 2008. CCGRID'08. 8th IEEE International Symposium on, pp. 771-776. IEEE, 2008.

[145] H. Weatherspoon, and J. D. Kubiatowicz, "Erasure Coding vs. Replication: A Quantitative Comparison", In *Peer-to-Peer Systems*, pp. 328-337. Springer Berlin Heidelberg, 2002.

[146] H. Stockinger, K. Stockinger, E. Schikuta, and I. Willers, "Towards A Cost Model for Distributed and Replicated Data Stores", In Parallel and Distributed Processing, 2001. Proceedings. Ninth Euromicro Workshop on, pp. 461-467. IEEE, 2001.

[147] H. Yu, and A. Vahdat, "The Costs and Limits of Availability for Replicated Services", ACM Transactions on Computer Systems (TOCS) 24, no. 1 (2006): 70-113.

[148] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch, “Session Guarantees for Weakly Consistent Replicated Data”, In *Parallel and Distributed Information Systems, 1994., Proceedings of the Third International Conference on*, pp. 140-149. IEEE, 1994.

[149] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, “Stronger Semantics for Low-Latency Geo-Replicated Storage”, In *NSDI*, pp. 313-328. 2013.

[150] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li, “Paxos Replicated State Machines as the Basis of a High-Performance Data Store”, In *NSDI*. 2011.

[151] https://www.opennetworking.org/Openflow

Biography

Ruofan Xia was born in Dazhou, Sichuan Province, China on November 24th, 1986. He enrolled in the Electronic Engineering program in Tsinghua University, Beijing, China in August 2005, and received a Bachelor of Science degree in July, 2009. He then pursued a Master of Science degree at the Department of Electrical and Computer Engineering at Duke University, NC, USA, receiving the degree in May, 2011. Ruofan Xia received a department fellowship from the Duke Computer Science Department from 2011 to 2013, two IBM PhD fellowships for the year 2014 and 2015 respectively, and a summer research fellowship from the Allen and Joyce Temple Graduate Fellowship Fund for the summer of 2015.

Academic Publications:

[1] Optimizing Replicated Storage Backup Operation using Markov Decision Process. Ruofan Xia, Fumio Machida, and Kishor Trivedi. Accepted by the 21st IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2015).

[2] A Markov Decision Process Approach for Optimal Data Backup Scheduling. Ruofan Xia, Fumio Machida, and Kishor Trivedi. In Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on (pp. 660-665). IEEE.

[3] Performance and Availability Modeling of IT Systems with Data Backup and Restore. Ruofan Xia, Xiaoyan Yin, Javier Alonso, Fumio Machida and Kishor Trivedi. IEEE Transaction on Dependable and Secure Computing (TDSC), issue 04, volume 11.

[4] Stochastic Model Driven Capacity Planning for an Infrastructure-as-a-Service Cloud. Rahul Ghosh, Francesco Longo, Ruofan Xia, Vijay Naik and Kishor Trivedi. IEEE Transaction on Service Computing (TSC), 2013, 1-1.

[5] Performability Modeling for RAID Storage Systems by Markov Regenerative Process. Fumio Machida, Ruofan Xia and Kishor Trivedi. Submitted to IEEE Transaction on Dependable and Secure Computing (TDSC), under review.

[6] Semi-Markov Models of composite Web services for their performance, reliability and bottlenecks. Zheng Zheng, Kishor S. Trivedi, Kun Qiu, and Ruofan Xia. Accepted to IEEE Transaction on Service Computing.

[7] Quantification of System Survivability. Kishor Trivedi and Ruofan Xia. Telecommunication System Journal (Springer), issue 67.