Request Success Rate of Multipathing I/O with a Paired Storage Controller

Gangadhar Enagandula and Varsha Apte
Department of Computer Science and Engineering
IIT Bombay, Mumbai - 400 076, India
Email: {gangadhar,varsha}@cse.iitb.ac.in

Bipul Raj*

***Abstract*—**

The success probability of I/O requests in presence of failures is increased by a combination of failover mechanisms built into the storage server, multiple access paths from I/O clients to the server, and timeout-retry mechanisms at the client itself. We define and evaluate a unified availability metric, *request failures per million* (RFPM), which quantifies request failure probability while taking into account client-side as well as server-side mechanisms. We calculate this metric using a two-level model of I/O service - a probability tree that captures the I/O driver behaviour, and a set of CTMC (Continuous Time Markov Chain) models that capture failover mechanisms at the server. The I/O driver model captures detailed timeout-retry mechanisms including retries at multiple ports ("multipathing"). The server model captures transient phenomena such as failure detection, takeover and emulation behaviour of a paired storage controller. The model shows that client retry mechanisms provide significant improvement in request success probability. The model is then used to study the sensitivity of RFPMs to parameters such as timeouts, reboot time and failure detection delay. The results show that the model can help in answering several what-if questions related to how system parameters impact request success rate.

Keywords*—*availability, storage controllers, Markov chains, analytical model

I. Introduction

Users today expect guaranteed 24x7 access to their data which is stored on networked storage systems. Storage systems use a plethora of techniques to provide high availability access to data, such as high availability disk arrays, and redundant storage controllers ("nodes"), with *failover* mechanisms built into them, so that the chance of a I/O request succeeding, improves [1]. An example mechanism is of *pairing* nodes - where each node monitors its partner node's health, and *takes over* the partner's function if it detects failure. When the partner recovers, control is *given back* to it.

The success probability of an I/O request can also be improved by recovery mechanisms at the client. Specifically, client I/O drivers can *detect* failures, and then try to *recover* from them. Failure detection is based either on receipt of error messages from the storage nodes, or on lack of any response, resulting in a *timeout*. Clients respond to these error signals by *retries*.

Retries can be made to the same port, or can exploit *multipathing* [2]. Multipathing implies the provision of multiple connection paths between hosts and controllers [2]. A multipathing I/O driver uses timeout and retry mechanisms to check various paths to a controller, before reporting an I/O failure [3].

Considering the myriad mechanisms used, it is important that rigourous analysis be carried out to determine the effect of these on increasing the success probability of an I/O request. This can be done using *analytical models* of the systems. In the storage domain, models of availability have been developed in great detail for the disk subsystem, specifically for RAID architectures [4]. However, not much work has been done to date in developing analytical models of storage *nodes* that would capture the redundancy and failover mechanisms described above. To our knowledge, there is also no work in the storage domain that captures the effect of the *client's* failure recovery mechanisms. A recent model by Trivedi et al [5] in the telecommunication domain captures client and server recovery mechanisms in detail. Our work is inspired by the approach proposed there, and takes it further by incorporating the specifics of the storage system that we studied.

We use the Netapp "FAServer" High Availability Pair ("HA pair") [6] as our storage controller system, and the SCSI protocol with Asymmetric Logical Unit Access ("ALUA") as our client I/O protocol [3], and apply our modeling approach to compute request success (failure) probability for this unified system.

The contributions of this work are as follows:
1. We propose that storage system availability be measured in terms of probability of I/O request success. In the presence of client recovery mechanisms, this probability is different from the instantaneous server availability, and needs to be studied and quantified.
2. We create a probability tree model that captures the client I/O protocol's behaviour, including multipathing and retry, in response to the state it finds the storage node in. Calculating the branching probabilities of this tree requires (a) determining steady-state probabilities of the states of the server system and (b) determining the probability of transitioning from one set of states to the other.
3. We develop a set of CTMC models to do this. A root CTMC model is developed that captures the HA pair's failure-recovery behavior, and can give us the required steady-state probabilities. Then we create a set of derived CTMCs, where we make some states "absorbing". We use the root CTMC to set initial state probabilities of the derived CTMCs, and carry out *transient analysis* to quantify probabilities such as

*This work was done while this author was with NetApp, Bangalore, India

"Probability a request finds the controller down, but it finishes rebooting before the client times out".
4. Numerical evaluation of our model offers various insights: first, it shows that it is worthwhile calculating request success probability while incorporating client recovery mechanisms. Equating this to instantaneous system availability results in significant underestimation of probability of request success. Second, our study of sensitivity to system parameters shows some non-obvious results - e.g. we see that under certain conditions, request success probability can be more sensitive to giveback processing than to rebooting time. We also show that under certain conditions, taking over too soon from a failed node may not increase request success probability at all (in fact, it can *decrease* it slightly).

In summary, our model gives the designers of such systems an effective tool using which they can set values of configurable parameters, or decide which processes should be sped up to get the most improvement in request success probability.

The rest of the paper is as follows: Section II describes the high availability storage controller architecture, and the I/O protocol's multipathing and retry behavior. Section III describes our model of the HA pair and of the SCSI ALUA protocol. Section IV presents some numerical results and insights based on the results. We conclude the paper in Section V.

II. Background

In this section we will first provide an overview of a high availability dual storage controller architecture (modeled on the Netapp FAServer "HA pair"), and then describe the relevant features of the SCSI protocol that impact I/O request failure rate. Finally, we review existing work in availability analysis of storage systems.

A. High Availability Storage Controller Pair

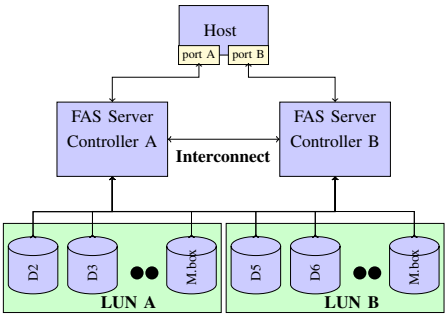


Fig. 1: HA Pair Architecture.

The Netapp High Availability Storage Controller Pair ("HA Pair") architecture [6] is depicted in Figure 1. The storage array is divided into two logical units, controlled by two "FAServers" in a paired configuration. In this configuration, each controller is connected to its partner through an interconnect, and is also connected to its partner's disk shelves. The logical units have a *logical unit number* (LUN) and are conventionally termed as "LUNs".

The HA Pair configuration is *asymmetric*; i.e. FAServer A is the primary owner of LUN A and FAServer B is the primary owner of LUN B. Thus, if the host sends a request on port A to FAServer A, but it is meant for LUN B, FAServer A forwards the request to FAServer B.

Each controller monitors its partner's health by listening to a heartbeat that is sent over the interconnect. In addition, each controller writes its heartbeat status into a special "mailbox disk store". Failure is detected if there is absence of the heartbeat both from the interconnect and the mailbox disk for a given amount of time. After this *detection delay*, a failover process is started which results in the surviving node assuming control of the LUN that was owned by the failed node. The time required for this is called *takeover time*. After takeover is completed, we say that the failed controller is being *emulated* by the surviving controller. When the failed node recovers, the surviving node must return ownership of the taken over LUN to its partner. This process involves several steps, and is called *auto-giveback*.

Failures are of two types - one where the controller recovers by an auto-reboot, and one that requires manual repair. A reboot may also fail, in which case we have to default to manual repair. If takeover fails, emulation will not be possible and the controller will be up only when it reboots or repairs. The auto-giveback may also fail, in which case we have to use a process of manual give-back.

The data on the LUN which the failed controller owns is *unavailable* during *failure detection time, takeover time* and *giveback time* [6].

B. SCSI Protocol with ALUA

The SCSI protocol with *Asymmetric Logical Unit Access* (ALUA) is a standard that allows the host to discover and use multiple paths to a logical unit, where each path may have different performance characteristics [2].

ALUA supports a storage server architecture such as the one described above - where a request to logical unit A sent on port A will have better performance than if it were sent on port B. ALUA specifies standard commands by which ports and their association with LUNs as active/optimized, active/non-optimized, standby, unavailable, etc can be discovered.

While the standard specifies this discovery, the precise behavior of how a request times out and is retried, is up to specific driver implementations and their configurations [3]. For the purpose of our analysis, we assume the following (representative) behavior: First, we assume that each port is in the "active/optimized" state for the LUN it is directly connected to (through one controller), and in "active/non-optimized" for the other LUN. We also assume that as a default, I/O requests for a particular LUN are only sent out on the port that is "active/optimized" for that LUN.

Suppose an I/O request arrives for LUN A; the driver will first send it out on port A, and start a timer. A successful response may be received in time, or the controller may respond with an error code which indicates that it is rebooting. In this case, the request will be retried multiple times at the port A up to a certain time (termed the *retry window* [5]). The server may also respond with an error code which indicates

that it is down and *not* rebooting. If there is a timeout, or a non-rebootable error response from server A, the driver retries the request at port B. At port B also, if there isn't immediate success, the driver may retry up to a second *retry window*. Note that the request may now succeed if A reboots, because controller B can forward the request to A. It will also succeed if controller B starts *emulating* controller A within this time.

When a request arrives, if one of the ports (say port A) is "unavailable, then there are two possibilities: the network path from port A itself has failed, or that failover has happened for controller A, and B is now emulating A (and thus the port that connects to A is of no use and is marked as "unavailable"). In this case requests to either LUNs will be routed through port B. If the request is for LUN A and A is up, then B will forward the request to A. If A is not up, B will still be able to serve the request if it is emulating A.

C. Related Work

Most of the storage availability analysis work deals with disk subsystem failures [7], which is not the focus of this paper. Availability analysis of storage systems considering various layers has been presented in a few papers [8], [9]. Ford et al [8] have presented a thorough analysis of an year's worth of failure data corresponding to thousands of storage nodes. Apart from statistically characterizing and determining the values of quantities such as mean time to various types of failures, the paper also presents a Markov model of stripe availability. The model captures batch failures of data "chunks" in stripes, and is sufficiently accurate in estimating system availability metrics. However, the model only considers recovery by reconstruction of data lost due to a "chunk" failure in a stripe. Their storage nodes do not seem to employ takeover and emulation mechanisms.

A fairly detailed model of the complete storage system, including all the components at various layers was proposed by Amiri and Wilkes [9] for helping design storage systems that meet certain availability requirements. They also generate a CTMC to model the failure and repair of the various components, and to represent the impact of some recovery policies. However, again, they do not consider systems with takeover and emulation mechanisms. They do take a "request view", in the sense that performance of the request is considered in declaring a system as available

However, both these papers do not model recovery mechanisms of any I/O protocol.

Our work draws significantly from work done in the context of SIP (Session Initiation Protocol) server systems availability and call dropping probability by Trivedi et al [5]. Their system availability model incorporates the notion of reboot vs manual repair failures, detection delays etc. Furthermore, their analysis introduced the idea of computing request success probability while taking into account its *retry* behavior. While we use these contributions as a starting point, our system differs from theirs in many ways. Specifically our servers have data ownership roles and processes of takeover, emulation and giveback, and our request protocol has multipathing. These are behaviors not modeled earlier, and lead to significant difference and complexity in the modeling methodology.

Up states		Down States	
UP	Working normally	RBT	Rebooting
DET	Detecting partner node failure	REP	Under manual repair
TKO	Taking over service of partner	ERB	Rebooting while in emulation mod
EML	Emulating partner	ERP	Undergoing repair while in emulation mode
GVB	Giving back LUN control to partner	TKB	Taking back control of LUN

TABLE I: Possible Controller States

III. Hierarchical Model for Computing I/O Request Failure Probability

An I/O request for data which is on a particular LUN succeeds if: a) the controller which owns that LUN (and a path to it) is *available* at the time of arrival of a request, or b) it is *unavailable* when the request arrives, but becomes available sometime before the driver completely times out and reports a failure. Here a controller is "available" if it is either itself operational, or being emulated by its partner. Note that we do not take into consideration the possibility that there is a failure (or any server state change) during the processing of a request, since request processing times are negligibly small.

As described in Section II, when a failure is detected, either by the client, or by the partner node, each one independently triggers recovery mechanisms. We develop a hierarchical model that captures the intricate interplay between the driver's and the HA pair's failure recovery mechanisms, as follows:

- A higher level *Probability Tree* for I/O Request Experience: this model uses a probability tree that represents the various states that an arriving request might find the HA pair in at various points in time after its arrival. The leaf nodes of this tree denote the eventual fate of this request (success, or failure). The probability tree is parameterized by its *branching probabilities*. These branching probabilities are either (a) the steady state probabilities of finding the HA pair in a certain state (e.g. "Controller A is Down"), or (b) the probability of moving from a particular state that an arriving request found the HA pair in (e.g. Controller A is Down), to a particular other state, within the retry window of the request (e.g. moving to "Controller A is being Emulated", within 30 seconds). Once the probability tree is created, the probability of success can be found trivially, if the branching probabilities can be quantified. This is done using a set of HA pair state models.
- Lower level *Continuous Time Markov Chain (CTMC)* models of the HA pair: A CTMC model can be developed easily for the HA pair, which captures all the failover, emulation, takeover and giveback events and corresponding state changes. This gives us the steady state probability of finding the HA pair in a certain state. For finding the probability of *moving* from one state to the other within a certain time, we define a series of "modified" CTMCs, derived from the original CTMC, on which we carry out *transient analysis*. The steady state probabilities of the main CTMC are used to set the initial state probabilities of these modified CTMCs.

Now we present the details of each sub-model.

A. CTMC Availability Model of the HA Pair

Our HA pair availability model focuses only on the failure probability of the controller itself. For the analysis presented in

this paper, we assume that the LUNs are always available to the controllers (justified by the fact that disk array MTTF is much higher than controller MTTF [8]) and that the interconnect never fails. We do consider the possibility that the path from the host to the controller may fail due to hardware failure, however that is not part of the HA pair availability model presented in this section.

We assume that the distribution of all times are exponential. With this assumption we can use a CTMC [10] to capture the system failure/repair behavior. The state of the CTMC is a tuple which represents the state of the controller A and the state of the controller B. The possible states that a controller can be in is shown in Table I. E.g., the state (TKO, RBT) represents the state that Controller B is rebooting and Controller A has started the takeover process. Similarly, state (GVB, TKB) represents the state where controller B is now taking back control from Controller A. These states will be considered as unavailable states for requests for LUN B. On the other hand, the state (EML, RBT) represents controller A emulating controller B, while B is rebooting. This is an available state for both LUNs.

Building a CTMC involves creating the state-transition graph, where the arcs are labeled with *rates* of various events. We consider the following events in generating this graph: rebootable failure, non-rebootable failure, reboot, repair from non-rebootable failure, failure detection, takeover, auto-giveback and manual giveback. We also use the following coverage factors (i.e. probability of success of): reboot, takeover and giveback.

The CTMC modeling the HA pair (not shown, due to space constraints) is a finite, irreducible CTMC, which can be evaluated for its steady-state probabilities [10] using standard methods and tools [11]. We refer to this CTMC as the CTMC M_0 and let S denote the set of states of this CTMC. Let π_s denote the steady-state probability of a particular state $s \in S$. For a subset $C \subset S$, we also denote by π_C the steady state probability of the CTMC being in any $s \in C$. This probability is given by $\sum_{s \in C} \pi_s$. The transient probabilities at time t, $P_s^0(t)$ and $P_C^0(t)$ are similarly defined. These state probabilities can also be calculated numerically by specifying this CTMC in CTMC solver tool [11].

B. I/O request success model

Figures 2 and 3 show a probability tree (split into four parts) that models the success/failure path of an I/O request. The nodes in the tree represent various possibilities such as the state that the request finds the HA pair in, or events that happen after a request arrives. The nodes are labeled with numbers for easy reference. Each node is labeled with an event, and the branches are labeled with the possible outcomes, and the probabilities of those outcomes.

Consider a new I/O request arriving at a host. The request can be for either logical unit A or B (Figure 2a, Node 1). Suppose it is for LUN A. The default is then to try to send it out on port A. This can only be done if the port A is active. Port A can be inactive in two ways: either the network hardware has failed, or network hardware has not failed, but port A has been marked "unavailable" because controller A had failed over and controller B is either emulating A or "giving back" to A. These possibilities are considered in Nodes 2 and 3.

If port A is active, the request is sent out on port A to controller A. If A is up (Node 6), the request succeeds. If not, then the host driver behaviour depends on the type of error it gets (Node 8).

If A sends a "rebooting" error response , the driver retries at port A upto its retry window (see Figure 3b, Node 24) . If A reboots within this window, then the request succeeds. If A does not reboot in time, the driver retries at port B (and starts a new timer). If port B is inactive (either hardware down, or marked unavailable), the request now fails (Nodes 25, 26).

If port B is active, and if controller B is up (Node 27), then it can serve the request if either A reboots, or if B starts emulating controller A within the driver's retry window. If the driver now exhausts its retry window, the request fails (Node 29).

If B itself was down (Node 28), then the request can be successful only if the following happens within the retry window: B had a reboot failure, and it itself reboots and A also reboots, or only B reboots and starts emulating A.

Now lets go back to Node 2. If port A hardware has failed, then the request is tried directly at port B (Node 4). In this case it is possible that port B hardware has failed, or that it is marked unavailable. In either case, the request fails. If port B is active, the request is sent out on the port. Then the request success depends on whether controller B is up or down.

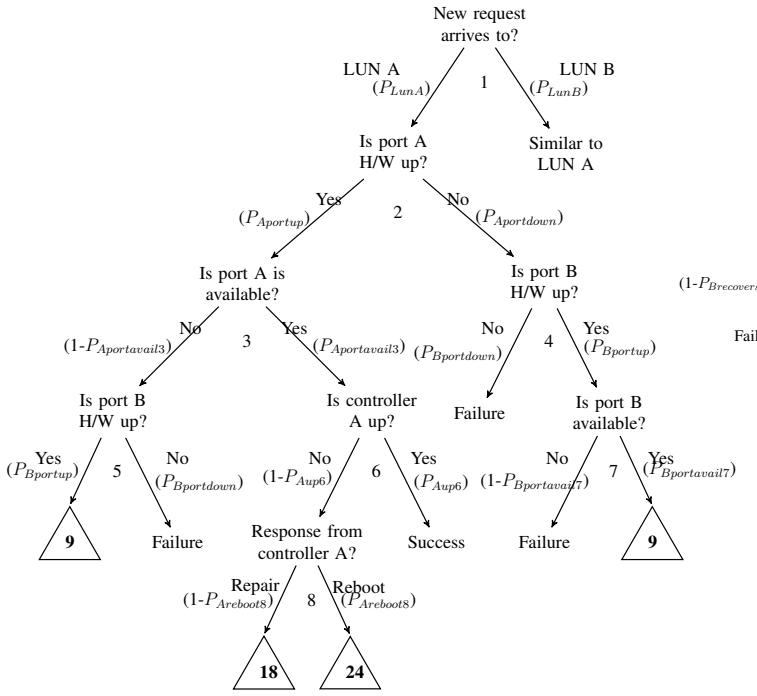
Figure 2b shows the tree continued from the Nodes 5 and 7, "Yes" branch. In either case, Node 9 is reached because the request was for LUN A, but port A was inactive but port B was active. Note that although the structure of the subtree under Node 9 is the same, the probabilities of the branches are different when the subtree is under Node 5 vs under Node 7.

Now if controller B is down and experiencing a non-rebootable failure, then the request will fail (Nodes 9, 10). If it responds with a rebootable failure code, the driver will retry the request. If B reboots within the retry window (Node 12), then the request will succeed if A is either up or being emulated by B. Note that this probability will also be different based on whether we reached here through Node 5 or Node 7.

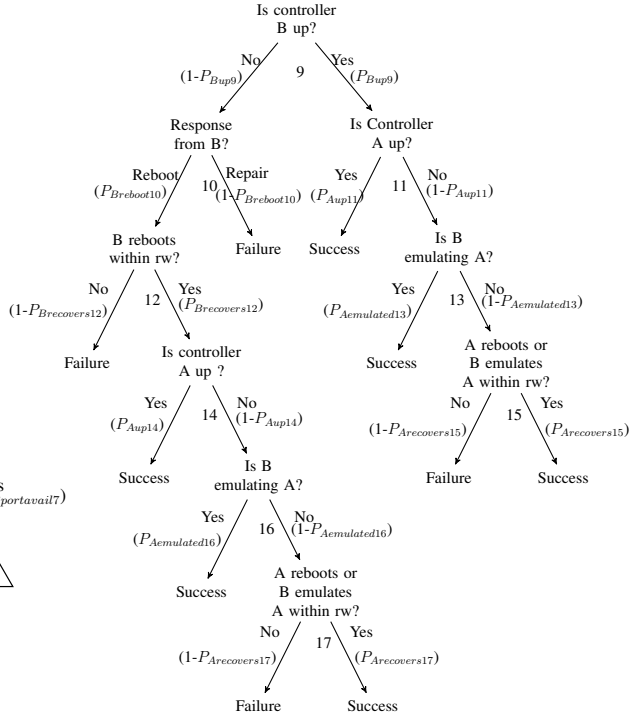
The rest of the tree, shown in subtrees in Figures 2 and 3 can be explained in a similar way. The nodes and branches are labeled in a self-explanatory way. Once the tree has been parameterized with branching probabilities, the probability of request success is simply the sum of the probabilities of reaching one of the leaf nodes labeled "success". The probability of reaching a particular leaf is the product of the branch probabilities in the path from the root to the leaf node. In the following subsection we describe how we obtain these branching probabilities.

C. Calculating the branching probabilities of the tree

Some of the branching probabilities of the probability tree are input parameters: the probability of a new request arriving for either LUN A or LUN B (P_{LunA}, P_{LunB}) and the probability that a host port is down because of a network

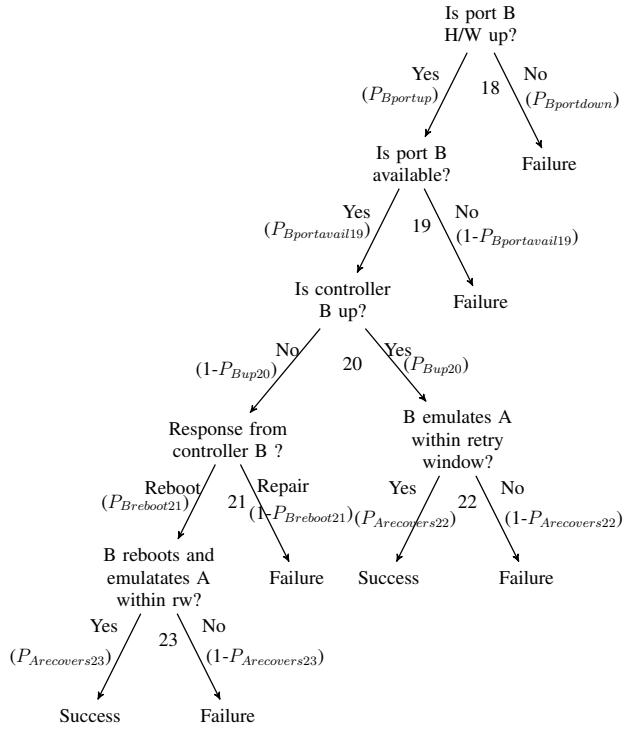


(a) Top level tree

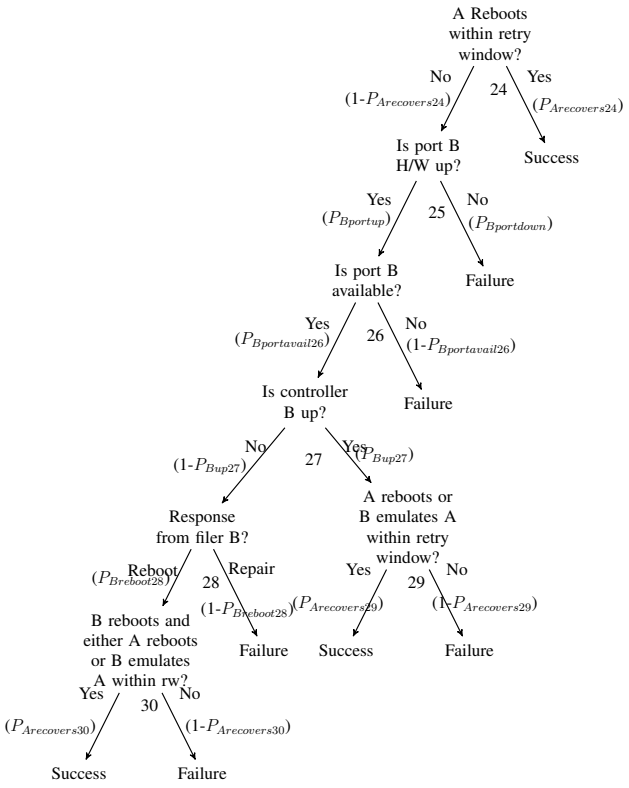


(b) Subtree rooted at Node 9, under "Yes" branch at Nodes 5 and 7

Fig. 2: Request modeling of SCSI protocol with ALUA



(a) Subtree rooted at Node 18, under Node 8,"repair" branch



(b) Subtree rooted at Node 24, under Node 8, "reboot" branch

Fig. 3: Request modeling of SCSI protocol with ALUA

Subset Name	Corresponding states of the CTMC
Aup	{(UP/DET/TKO/EML, RBT/REP), (GVB, TKB), (UP, UP) }
Bup	{(REP/RBT, UP/DET/TKO/EML), (TKB, GVB), (UP, UP) }
Adown	S - Aup
Bdown	S - Bup
Aemulated	{ (REP/RBT, EML), (REP/RBT, ERB), (REP/RBT, ERP) }
Bemulated	{ (EML, REP/RBT), (ERB, REP/RBT), (ERP, REP/RBT)}
Anotemulated	S - Aemulated
Bnotemulated	S - Bemulated
Aportunavail	{ (REP/RBT, EML), (REP/RBT, ERB), (REP/RBT, ERP), (TKB, GVB) }
Bportunavail	{ (EML, REP/RBT), (ERB, REP/RBT), (ERP, REP/RBT), (GVB, TKB) }
Aportavail	S - Aportunavail
Bportavail	S - Bportunavail
Areboot	{(RBT, UP/EML/ERB/ERP/TKO/DET/RBT), (ERB, RBT/REP/TKB) }
Breboot	{(UP/EML/ERB/ERP/TKO/DET/RBT, RBT), (RBT/REP/TKB, ERB) }
Arepair	{(REP, UP/EML/ERB/ERP/TKO/DET/RBT), (ERP, RBT/REP/TKB) }
Brepair	{(UP/EML/ERB/ERP/TKO/DET/RBT, REP), (RBT/REP/TKB, ERP) }

TABLE II: Controller States and corresponding CTMC states

Branch Probability	calculation from Availability model
$P_{Aportavail3}, P_{Bportavail7}$	$\pi_{Aportavail}, \pi_{Bportavail}$
$P_{Aup6}, P_{Arecoot8}$	$\frac{\pi_{Aportavail \cap Aup}}{\pi_{Aportavail}}, \frac{\pi_{Aportavail \cap Adown \cap Areboot}}{\pi_{Aportavail \cap Adown}}$
$P_{Bup9-5}, P_{Brecoot10-5}$	$\frac{\pi_{Aportunavail \cap Bup}}{\pi_{Aportunavail}}, \frac{\pi_{Aportunavail \cap Bdown \cap Breboot}}{\pi_{Aportunavail \cap Bdown}}$
$P_{Aup11-5}$	$\frac{\pi_{Aportunavail \cap Bup \cap Aup}}{\pi_{Aportunavail \cap Bup}}$
$P_{Aemulated13-5}$	$\frac{\pi_{Aportunavail \cap Bup \cap Adown \cap Aemulated}}{\pi_{Aportunavail \cap Bup \cap Adown}}$
P_{Bup9-7}	$\frac{\pi_{Bportavail \cap Bup}}{\pi_{Bportavail}}$
$P_{Brecoot10-7}$	$\frac{\pi_{Bportavail \cap Bdown \cap Breboot}}{\pi_{Bportavail \cap Bdown}}$
$P_{Aup11-7}$	$\frac{\pi_{Bportavail \cap Bup \cap Aup}}{\pi_{Bportavail \cap Bup}}$
$P_{Aemulated13-7}$	$\frac{\pi_{Bportavail \cap Bup \cap Adown \cap Aemulated}}{\pi_{Bportavail \cap Bup \cap Adown}}$
$P_{Bportavail19}$	$\frac{\pi_{Aportavail \cap Adown \cap Arepair \cap Bportavail}}{\pi_{Aportavail \cap Adown \cap Arepair}}$
P_{Bup20}	$\frac{\pi_{Aportavail \cap Adown \cap Arepair \cap Bportavail \cap Bup}}{\pi_{Aportavail \cap Adown \cap Arepair \cap Bportavail}}$
$P_{Brecoot21}$	$\frac{\pi_{Aportavail \cap Adown \cap Arepair \cap Bportavail \cap Bdown \cap Breboot}}{\pi_{Aportavail \cap Adown \cap Arepair \cap Bportavail \cap Bdown}}$

TABLE III: Branching probabilities from steady-state analysis

path failure from that port to the corresponding controller ($P_{Aportup}, P_{Bportup}$).

The rest of the branching probabilities are calculated from CTMC models of the HA pair. Some of these probabilities correspond to the request finding the HA pair in a certain set of states on its arrival. We assume that a *new* arrival finds the system in steady state, so we use the steady state probabilities of CTMC M_0 for this calculation. Some branching probabilities are probabilities of transitioning from one subset of states to another subset of states in a given amount of time. These are found by evaluating the transient state probabilities of derived CTMCs. The details of how we do this is as follows.

For convenience of calculation of these probabilities, we first define states that a controller can be in, and the subset of states of the CTMC that this corresponds to. These controller states, and their corresponding CTMC subsets are shown in Table II.

1) Probabilities based on steady state analysis: Consider the branching probability at a node to which the path from the root of the tree does not contain a node of the type "reboots/takes over/emulates within rw". Thus the path to this node contains branches based only on the steady state of the system. For such a node, the branching probability is the conditional probability that the request finds the system in the state corresponding to this branch, *given* that the request had found the system in the subset of states common to all the branches on the path to this branch. E.g. consider Node 9 when it is attached under Node 5. Here the probability that controller B is up is given by

$$\pi_{Bup|Aportunavail} = \frac{\pi_{Bup \cap Aportunavail}}{\pi_{Aportunavail}}$$

The intersection set $Bup \cap Aportunavail$ is easily defined based on Table II. Note that, in this calculation, we do not include the probabilities of port hardware being up or down, as the CTMC state probabilities are independent of these probabilities. The rest of the branching probabilities of this type are defined similarly and are given in Table III.

2) Probabilities based on transient analysis: Now consider the nodes of the type "reboots/takes over/emulates within rw" and their children nodes. E.g. consider Figure 3b, Node 24 ("A reboots within retry window?"). This node represents the event that the request is for LUN A, port A was active but controller A sent a "rebooting" response. In this case, the request will be retried until its "retry window" (rw) is reached. Then the probability that the request succeeds at port A is given by the probability that A reboots before this retry window [5]. This is equal to the probability of the CTMC starting in states $S_{24} = A_{portavail} \cap A_{down} \cap A_{reboot}$ and entering states: $F_{24} = A_{up}$ within time rw. Note that we ignore the possibility that the controller will have a second failure in this duration.

This probability can be found by modifying the original CTMC such that all outgoing arcs from the states F_{24} are removed. For this modified CTMC (M_{24}), let the transient probabilities of being in state s at time t be denoted by $P_s^{24}(t)$. The initial state probability for a state $s \in S_{24}$ is the conditional steady-state probability of being in a specific state $s \in S_{24}$, given that the CTMC is in one of the states of S_{24}:

$$P_s^{24}(0) = \begin{cases} \frac{\pi_s}{\sum_{s \in S_{24}} \pi_s}, & \forall s \in S_{24}, \\ 0, & otherwise. \end{cases}$$

and the required branching probability is given by

$$P_{Arecovers24} = \sum_{s \in F_{24}} P_s^{24}(rw) = P_{F_{24}}^{24}(rw)$$

Since $F_{24} = A_{up}$ is an absorbing state in CTMC M_{24}, the above probability corresponds to *reaching* this state anytime before time rw [5].

Note that branching probabilities of the children nodes of Node 24 (Nodes 26, 27, 28) will also now be transient probabilities. These states are seen by the request after its retry window is exhausted, which is a time rw after its arrival. Therefore, we have to calculate the probability of the request finding the system in the corresponding states, rw time after its arrival. These are the state probabilities of the CTMC M_{24} evaluated at time rw.

The exact probabilities required are shown in Table IV. For all the branching probabilities whose values correspond to transient probabilities, a modified CTMC is built (denoted by M_i for Node i), and transient analysis carried out. Transient probabilities for state s (or subset S) of CTMC M_i are denoted by $P_s^i(t)$ (or $P_S^i(t)$). For transient analysis, we must specify the initial state probabilities and for each node, we must

define the transient probability that we are interested in (which states, at what time). We have summarized this information in Table IV. The table displays: the label of the required branching probability, the CTMC on which transient analysis is carried out, the set of states in which the CTMC can start, and their initial state probabilities, the final states (these are those states whose outgoing arcs have been removed) and the other states for which we will calculate transient probabilities. The last column shows the expression for the branching probability. Note that the nodes of the subtree under Node 9 have different branching probabilities depending on whether they are attached under Node 5 or Node 7, because this changes the initial conditions. The notation in the table ("-5" and "-7") reflects this difference.

Some of these probabilities need further explanation. Consider branch probabilities for Nodes 14 and 16 (Figure 2b). For Node 14, we want to find P_{Aup14}, which is the probability that the request will find controller A up, given that controller B is up, at one of its retries. The retries may happen at random times within the retry window rw. We approximate this event by the probability that the request finds the system in this state at time $rw/2$ after request arrival. Thus, we find the probability that controller A is up at time $rw/2$ after arrival, given that controller B was also up at that time.

Since the request is assumed to arrive at steady state, we evaluate the original CTMC (M_0) for its transient probabilities, and determine the required state probabilities at time $rw/2$. The initial state probabilities are the same as for Node 12 (conditional steady state probabilities). Then, P_{Aup14} is given by $P^0_{Aup|Bup}(rw/2)$. $P_{Aemulated16}$ is calculated similarly (see details in Table IV).

$P_{Arecovers17}$ is calculated as follows: we define a CTMC M_{17} with arcs for states $F_{17} = Bup \cap (Aup \cup Aemulated)$ removed. We set the initial state probabilities also by evaluating the CTMC M_0 for transient probabilities at time $rw/2$ which gives us the probability that the request finds controller B up but A down and not emulated when it retries (at $rw/2$). Then $P_{Arecovers17}$ is given by the probability that A recovers within the time *remaining* in the retry window ($rw/2$), which is given by transient probabilities of CTMC M_{17} evaluated at $rw/2$.

The remaining transient analysis based branching probabilities are defined in Table IV.

IV. Results

We evaluate the model described in the above sections using standard CTMC solver tools, to study the impact of various system and protocol parameters and mechanisms on the request failure probability. We convert this probability into a metric called *request failures per million* - RFPMs [5] -, which we calculate as

$$\text{RFPM} = \text{Request failure probability} \times \quad 10^6.$$

Note that a validation against measurement of the results predicted by our models was not possible as data regarding failures was not available. However, we believe that this makes this modeling effort all the more important, as in absence of any data, it is the only quantification of the availability and of relative improvements of this system.

Furthermore, while values of some system parameters (reboot time, takeover time, giveback time) were available (values used in this paper are close to, but not exactly equal to the actual values for HA Pair), other system parameters such as mean time to failure, mean time to repair, the coverage factors, were not. For this purpose, we carry out a sensitivity analysis of our metric to these and other parameters. The default values used are shown in the caption of the Figure 4.

Firstly, this model should be able to quantify the improvement in availability by going from a single node to a paired configuration. Next, it should quantify the improvement provided by multipathing and timeout/retry mechanisms employed by SCSI with ALUA.

To make this comparison, we convert system unavailability also to a "request failures per million" metric. Thus RFPMs for the single node and HA Pair, without taking into account protocol behaviour were calculated as: $10^6\times$ Pr[system is unavailable]. Here, we consider the HA Pair available only if both the controllers are available (through normal operation, or through emulation).

Note that in the graphs, unless otherwise noted in the legend, the plot corresponds to the RFPMs of an HA pair with multi-pathing and retry mechanisms.

Figure 4a compares RFPMs vs reboot time for the single node without retries, the HA pair without retries, and then the HA pair with retries for two retry windows. For these set of values, the paired configuration brings down RFPMs by almost 70% of those for the single node. The client retry and multipathing mechanisms provided an additional 67% reduction in the RFPMs over those of the HA Pair with no retries. The main cost of this improvement is of performance. If the application is delay-tolerant, this might be acceptable.

In the rest of the graphs, we study client RFPMs vs various system parameters.

Figure 4b shows the "law of diminishing returns" in action for RFPMs vs mean time to rebootable failures. Whether this MTTF should be improved will depend upon which part of the graph the current MTTF is in.

Figure 4c shows low level of sensitivity of RFPMs towards reboot time. This is possibly because in the single node, the repair time dominates the unavailability figure. This is seen in Figure 4d, which shows higher sensitivity to repair time (especially the single node). In case of HA pair with or without client retries, other factors such as takeover and emulation reduce the impact of reboot times, since a partner may takeover and start emulating a node that is rebooting.

This is confirmed in Figures 4e and 4f, which show higher level of sensitivity towards takeover and (auto-) giveback. This is because the takeover time is now the actual time for the LUN to be available again, rather than the reboot time. In this scenario, the giveback processing time is also very important, since a LUN remains unavailable during giveback processing.

We notice another interesting trend in the RFPM vs takeover time which is also seen in RFPM vs detection

Branch Probability label	CTMC Label	Initial States	Initial State probabilities $P_s(0)$	Final States (F) or Other States (R)	Branch Probability
$P_{Brecovers12-5}$	M_{12}	$S_{12-5} = Aportunavail \cap Bdown \cap Breboot$	$\frac{\pi_s}{\pi_{S_{12-5}}}, \ \forall s \in S_{12-5}$	$F_{12} = Bup$	$P_{F_{12}}^{12-5}(rw)$
$P_{Aup14-5}$	M_0	S_{12-5}	$\frac{\pi_s}{\pi_{S_{12-5}}}, \ \forall s \in S_{12-5}$	$R'_{14} = A_{up} \cap B_{up}$, $R_{14} = A_{down} \cap B_{up}$	$\frac{P_{R'_{14}}^{0-5}(\frac{rw}{2})}{P_{F_{12}}^{0-5}(\frac{rw}{2})}$
$P_{Aemulated16-5}$	M_0	S_{12-5}	$\frac{\pi_s}{\pi_{S_{12-5}}}, \ \forall s \in S_{12-5}$	$R_{16} = R_{14} \cap Aemulated$	$\frac{P_{R_{16}}^{0-5}(\frac{rw}{2})}{P_{R_{14}}^{0-5}(\frac{rw}{2})}$
$P_{Arecovers17-5}$	M_{17}	$S_{17} = Bup \cap Adown \cap Anotemulated$	$P_{S_{17}}^{0-5}(\frac{rw}{2}), \ \forall s \in S_{17}$	$F_{17} = Bup \cap (Aemulated \cup Aup)$	$P_{F_{17}}^{17-5}(\frac{rw}{2})$
$P_{Brecovers12-7}$	M_{12}	$S_{12-7} = Bportavail \cap Bdown \cap Breboot$	$\frac{\pi_s}{\pi_{S_{12-7}}}, \ \forall s \in S_{12-7}$	F_{12}	$P_{F_{12}}^{12-7}(rw)$
$P_{Aup14-7}$	M_0	S_{12-7}	$\frac{\pi_s}{\pi_{S_{12-7}}}, \ \forall s \in S_{12-7}$	R'_{14}, R_{14}	$\frac{P_{R'_{14}}^{0-7}(\frac{rw}{2})}{P_{F_{12}}^{0-7}(\frac{rw}{2})}$
$P_{Aemulated16-7}$	M_0	S_{12-7}	$\frac{\pi_s}{\pi_{S_{12-7}}}, \ \forall s \in S_{12-7}$	R_{16}	$\frac{P_{R_{16}}^{0-7}(\frac{rw}{2})}{P_{R_{14}}^{0-7}(\frac{rw}{2})}$
$P_{Arecovers17-7}$	M_{17}	S_{17}	$P_{S_{17}}^{0-7}(\frac{rw}{2}), \ \forall s \in S_{17}$	F_{17}	$P_{F_{17}}^{17-7}(\frac{rw}{2})$
$P_{Arecovers15-5}$	M_{15}	$S_{15-5} = Aportunavail \cap Bup \cap Adown \cap Anotemulated$	$\frac{\pi_s}{\pi_{S_{15-5}}}, \ \forall s \in S_{15-5}$	$F_{15} = F_{17}$	$P_{F_{15}}^{15-5}(rw)$
$P_{Arecovers15-7}$	M_{15}	$S_{15-7} = Bportavail \cap Bup \cap Adown \cap Anotemulated$	$\frac{\pi_s}{\pi_{S_{15-7}}}, \ \forall s \in S_{15-7}$	F_{15}	$P_{F_{15}}^{15-7}(rw)$
$P_{Arecovers22}$	M_{22}	$S_{22} = Aportavail \cap Adown \cap Arepair \cap Bportavail \cap Bup$	$\frac{\pi_s}{\pi_{S_{22}}}, \ \forall s \in S_{22}$	$F_{22} = F_{17}$	$P_{F_{22}}^{22}(rw)$
$P_{Arecovers23}$	M_{23}	$S_{23} = Aportavail \cap Adown \cap Arepair \cap Bportavail \cap Bdown \cap Breboot$	$\frac{\pi_s}{\pi_{S_{23}}}, \ \forall s \in S_{23}$	$F_{23} = F_{17}$	$P_{F_{23}}^{23}(rw)$
$P_{Arecovers24}$	M_{24}	$S_{24} = Aportavail \cap Adown \cap Areboot$	$\frac{\pi_s}{\pi_{S_{24}}}, \ \forall s \in S_{24}$	$F_{24} = Aup$, $R_{24} = Adown$	$P_{F_{24}}^{24}(rw)$
$P_{Bportavail26}$	M_{24}	S_{24}	, ,	$R_{26} = Adown \cap Bportavail$	$\frac{P_{R_{26}}^{24}(rw)}{P_{R_{24}}^{24}(rw)}$
P_{Bup27}	M_{24}	S_{24}	, ,	$R_{27} = Bup \cap R_{26}$ $R'_{27} = Bdown \cap R_{26}$	$\frac{P_{R_{27}}^{24}(rw)}{P_{R_{26}}^{24}(rw)}$
$P_{Breboot28}$	M_{24}	S_{24}	, ,	$R_{28} = Breboot \cap R_{27}$	$\frac{P_{R_{28}}^{24}(rw)}{P_{R'_{27}}^{24}(rw)}$
$P_{Arecovers29}$	M_{29}	$S_{29} = Adown \cap Bportavail \cap Bup$	$\frac{P_s^{24}(rw)}{P_{S_{29}}^{24}(rw)}, \ \forall s \in S_{29}$	$F_{29} = F_{17}$	$P_{F_{29}}^{29}(rw)$
$P_{Arecovers30}$	M_{30}	$S_{30} = Adown \cap Bportavail \cap Bdown \cap Breboot$	$\frac{P_s^{24}(rw)}{P_{S_{30}}^{24}(rw)}, \ \forall s \in S_{30}$	$F_{30} = F_{17}$	$P_{F_{30}}^{30}(rw)$

TABLE IV: Branching probabilities from transient analysis

delay (Figure 4g). We see that for large retry windows, the RFPMs show an initial insensitivity to increasing takeover time or detection delay. This is because, if a takeover happens quickly (if detection delay and takeover processing is very fast), before a failed node reboots, but later the failed node reboots successfully, this is an "unnecessary" failover, which resulted in the system being unavailable again during giveback processing. Thus although unavailability due to rebooting is reduced, it increases due to giveback processing. Thus taking over "too soon" may not be useful, especially for requests with long retry windows.

Figure 4h shows a high level of sensitivity of RFPMs, and a linear decrease w.r.t. reboot coverage factor for requests with short retry windows. Having longer retry windows (as usual) reduces the sensitivity to this parameter also.

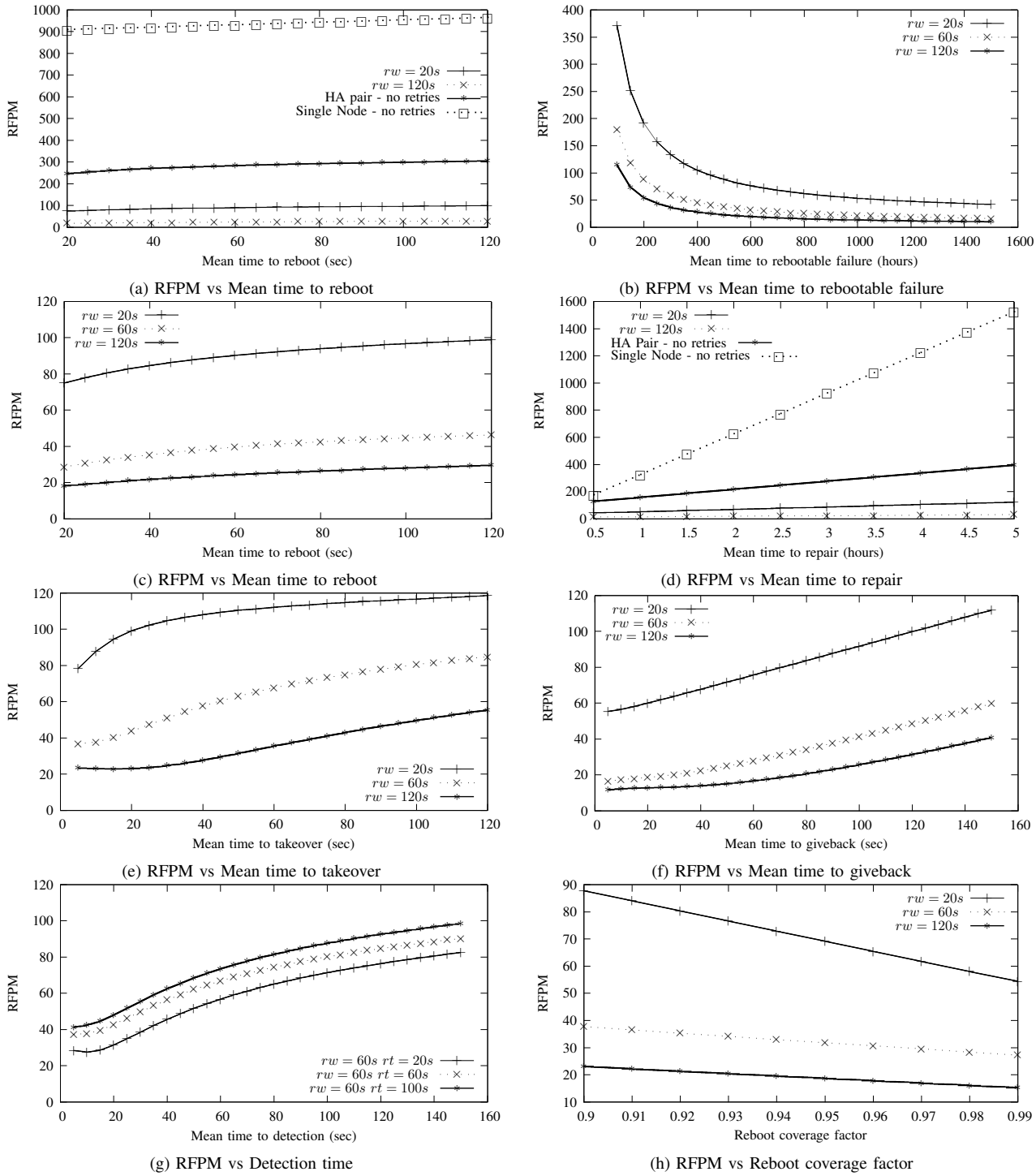
V. Summary and Conclusions

The availability analysis of systems is usually carried out by taking into account the failure and repair behaviour of the *server* components. However, client protocols also have "failure recovery" mechanisms built into them that must be analyzed. In this paper, we presented a model that took into account not only the details of the transient failover and recovery mechanisms of the server system, but also captured the request failure recovery mechanisms at the client in great detail. The analysis was able to provide insights into how various configuration parameters affect the request success probability. For example, it showed that in presence of takeover mechanisms, reboot time becomes less important. Additionally, it showed that if time available for retries (the retry window) is large, then takeover should not be triggered too soon - the system should wait to see if reboot is working.

There are several future directions for this work. The main direction is to scale this model so that it can work for larger (N-way redundant) systems. The model should also be compared against data to validate its prediction accuracy. Finally, since retry mechanims affect performance (response time of requests) a combined performance+availability study of the HA pair must be carried out.

Acknowledgment

This work was supported by a grant from the Netapp Advanced Technologies Group, Bangalore, India.



(a) RFPM vs Mean time to reboot

(b) RFPM vs Mean time to rebootable failure

(c) RFPM vs Mean time to reboot

(d) RFPM vs Mean time to repair

(e) RFPM vs Mean time to takeover

(f) RFPM vs Mean time to giveback

(g) RFPM vs Detection time

(h) RFPM vs Reboot coverage factor

Fig. 4: RFPMs sensitivity. Unless otherwise specified, values used are: Mean time to non-rebootable failure: 10000hrs, Mean time to rebootable failure: 500hrs, Mean time to failure detection: 15s, Mean time to takeover: 10s, Mean time to reboot (rt): 50s, Mean time to repair: 3hrs, Mean time to auto-giveback: 90s, Mean time to forced giveback : 100mins, Probability that auto-takeover succeeds : 0.9, Probability that auto-reboot succeeds: 0.9, Probability that auto-giveback succeeds: 0.9, Probability that request is for LUN A or B: 0.5, Probability that port hardware is up: 0.9999, Retry window (rw):20,60,120s

REFERENCES

[1] J. White, "Storage subsystem resiliency guide," Netapp, Tech. Rep. TR-3437, Sep. 2011.

[2] A. T and V. Rao, "Netapp host multipath strategy," NetApp, Tech. Rep. WP-7135, May 2011.

[3] *Linux Enterprise Server 11 SP2: Storage Administration Guide*, Novell, April 2012.

[4] M. Malhotra and K. S. Trivedi, "Reliability analysis of redundant arrays of inexpensive disks," *Journal of Parallel and Distributed Computing*, no. 17, pp. 146–151, 1993.

[5] K. S. Trivedi, D. Wang, and J. Hunt, "Computing the number of calls dropped due to failures," in *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, Nov. 2010, pp. 11–20.

[6] M. Liese, "High-availability pair controller configuration overview and best practices," NetApp, Tech. Rep. TR-3450, October 2011.

[7] 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 *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, 2007.

[8] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan, "Availability in globally distributed storage systems," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*. USENIX Association, 2010, pp. 1–7.

[9] K. Amiri and J. Wilkes, "Automatic design of storage systems to meet availability requirements," Technical Report HPL–SSP–96–17, Hewlett-Packard Laboratories, Tech. Rep., 1996.

[10] K. S. Trivedi, *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*. New York, 2001: John Wiley and Sons, 2001.

[11] M. Kwiatkowska, G. Norman, and D. Parker, "Prism 4.0: Verification of probabilistic real-time systems." in *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, ser. LNCS, vol. 6806. Springer, July 2011, pp. 585–591.