Automating data dependability

Kimberly Keeton and John Wilkes
Storage Systems Department, Hewlett-Packard Laboratories, Palo Alto, CA, USA
{kkeeton,wilkes}@hpl.hp.com

Abstract

If you can't make your data dependable, then you can't make your computing dependable, either. The good news is that the list of data protection techniques is long, and growing. The bad news is that the choices they offer are getting more complicated: how many copies of data to keep? whether to use full or partial redundancy? how often to make snapshots? how to schedule full and incremental backups? what combination of techniques to use? The stakes are getting higher: web access means that services must have 24x7 availability, and users are willing to switch if services are unavailable. Finally, human administrators can (and often do) make mistakes. These factors compel us to simplify and automate data dependability decisions as much as possible.

We are developing a system that will automatically select which data protection techniques to use, and how to apply them, to meet user-specified dependability (i.e., reliability and availability) goals. This paper describes our approach and outlines our initial descriptions for user requirements, failure characteristics and data protection techniques.

1 Motivation

A dependable system is one that just works: it does what you want, when you want it, to meet your needs. More and more, those needs are associated with access to information – without that information, highly dependable endpoints are useless. Getting it wrong is expensive: e-commerce sites, such as Amazon.com and ebay.com, may lose up to $200,000 per hour of downtime, and financial services may lose as much as $2.5M to $6.5M per hour of downtime [11].

The information had better be reliable, too – "losing" data is even worse than failing to provide access to it when it is needed. Many businesses, including financial institutions, pharmaceutical companies, and trading companies, must retain data for multi-year periods to meet legal requirements. As a result, losing data may have far-reaching ramifications.

This paper describes our approach to the data dependability problem. We concentrate on enterprise-scale storage systems, because their information needs affect so many of today's computing services. Today's enterprise storage systems range in size from many terabytes to a few petabytes of storage, with high rates of growth. Storage users demand predictable performance, very high availability, and extreme levels of data reliability. The sheer size of the systems means that administrators want more cost-effective and high-performance solutions than the traditional solution of "copy everything to tape." Administrators also want techniques to protect against important problems, such as user and software errors. For example, in today's SAN-connected systems, mirroring defends against device failure, but still propagates mistakes or software errors.

Over the last decade, the set of data protection techniques has increased significantly. In addition to traditional tape-based backup, online techniques using high-density disks and incremental snapshots are becoming attractive, as the price of disk storage capacity plummets [10]. Geographic distribution, replication and partial redundancy through RAID levels or erasure codes have also been enabled by cheaper wide area network performance.

Each technique provides some portion of the protection that is needed; combined, they can cover a much broader range. The key question, then, is how to determine the appropriate combination of different techniques to provide the data protection desired by the user. For example, one popular combination includes local RAID-5, remote mirroring, snapshot and backup to tape. RAID-5 provides protection against disk failures, remote mirroring guards against site failures, snapshots address user errors, and tape backup protects against software errors and provides archival.

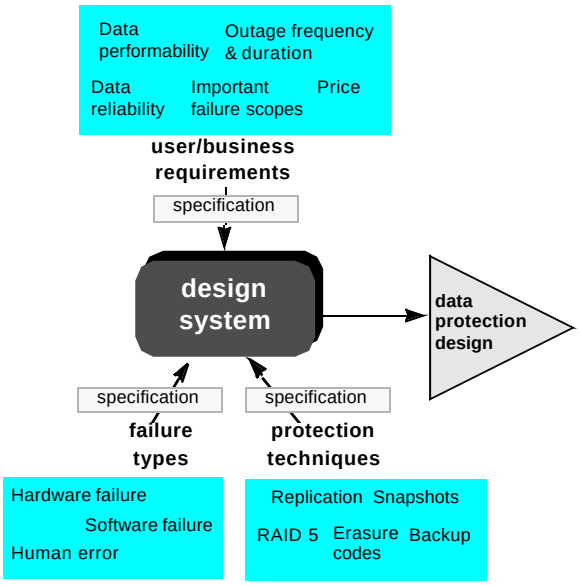

Figure 1: our approach to automating data dependability.

Selectivity in picking techniques matters – assigning different levels of protection to different kinds of information can save money or free up resources for providing more protection to important data. The mixture of techniques needs to change over time as user and business requirements change, as the environment changes and as new techniques become available. Unfortunately, the mix of techniques is too rich for people to reason about well, with the result that they either grossly over-provision, or accidentally omit coverage for events that later destroy their data. The former is merely expensive – storage systems typically comprise about half of the capital cost of current computer systems, with even higher operational costs. The latter can be catastrophic.

2 Our approach

We believe there is a better way: instead of asking humans to perform large-scale, complicated optimization problems, get the storage design system to help:

- Instead of asking people to describe how to implement data protection, have them specify their needs to the storage system, and let the system determine which implementation best meets them.
- Instead of asking people to reason about infrequent failure events, have them specify how available and reliable data must be, and let the system figure out which failures must be handled to meet these goals.
- Instead of expecting people to choose between available techniques and reason about their side-effects and interactions, let the system automatically choose the appropriate techniques, and incorporate new techniques as they become available.
- Instead of expecting people to monitor the system, and adapt its implementation to unexpected changes such as increased failure rates for a component, get the storage system to do it.

Figure 1 illustrates this automatic goal-directed design process. We have successfully applied this methodology to performance-related storage goals [1, 2, 3]. We believe the time is right to extend it to dependability. In particular, we concentrate on the *reliability* (whether the system discards, loses or corrupts data) and *performability* (whether the data can be accessed at a particular performance level at a given time) aspects of dependability [5].

We focus here on reliability and performability for data, rather than for applications. Even though users ultimately care about their end-to-end service needs, data dependability is a prerequisite for application dependability. We choose to start here because the narrower storage interface may provide more straightforward solutions, and these solutions will benefit a wide range of applications. We begin by examining the block-level storage system, and plan ultimately to extend the approach to higher-level data from file systems and databases.

2.1 Automating data dependability

Our approach to automating this problem has five main components (see Figure 1):

1. A description of the user's requirements (a *requirements specification*) for data performability and data reliability.
2. A description of the *failures* to be considered, including their scope and likelihood of occurrence.
3. A description of the *data protection techniques* available, including what failures they can tolerate, and how they recover from these failures.
4. A *design system*: a set of tools, including ones that select which techniques to use to satisfy the requirements.
5. The output is a *data protection design* specifying which data protection techniques should be deployed against which pieces of data and parameter settings (e.g., frequencies, retention times, etc.)

We note that the user is involved in only the first component, specifying the data protection requirements. Users may optionally designate that certain failures are important (e.g., electric utility unreliability may be problematic at a site), or conversely, unimportant (e.g., hurricanes in California are infrequent). They may also optionally express preferences for certain data protection techniques over others (e.g., from manufacturer X instead of Y). The underlying storage design system is responsible for enumerating and describing failures and the behaviors of the data protection techniques.

To automate the process of data dependability, our specifications for user requirements, failures and data protection techniques are made in a quantitative, declarative fashion [16]. Although the storage design system can work with only partial specifications for any of the components, more detailed information will lead to higher quality data protection designs.

2.2 The storage design system

The storage design system is a suite of tools that chooses (or, initially, assists with choosing) data protection techniques that meet the specified goals. We view the problem as an optimization one, by analogy to our earlier work on storage system design for performance [1, 3]. We envision a solution including the following tools:

- *user interrogation tool*: a graphical user interface (GUI) that asks users what they want, and then maps their intuitively specified goals into a quantitative specification. This tool removes the need for users to supply reams of numbers.
- *design checker*: this tool applies well-known modeling techniques [9] to predict whether a data protection design will satisfy a set of user goals.
- *design comparator*: using the design checker, this tool determines which of two designs more effectively meets the user's goals.

– *design tool*: this tool uses optimization techniques to automatically find the best data protection design to satisfy the user's goals, employing the design comparator to compare alternative designs [1, 3].

The design tool produces a data protection design, which records the design decisions, including what techniques should be applied to each data object and how the technique configuration parameters should be set. Given such a design, several additional management tools are desirable:

– *configuration tool*: this tool implements the design created by the design tool, including executing storage system and host configuration commands and potentially migrating data to the appropriate devices.
– *solution monitoring tool*: once a data protection design has been deployed, this tool monitors the solution to determine whether the predicted behavior is realized and whether environmental failures occur as expected.
– *adaptive management:* the design tool, configuration tool, and solution monitoring tool can be used in concert to provide an iterative approach to adapt to changes in requirements or the environment [2].

3 Specifications for automating data dependability

In this section, we provide a brief overview of our approach's declarative specifications for user requirements, failures and data protection techniques.

As described in [16], we believe it is necessary to distinguish between the following when specifying users' data dependability requirements and data protection techniques:

- the *content* we are trying to protect (called *data*)
- the *access patterns* to the content (called *streams*)
- the *containers* in which data resides (called *stores*)

Distinguishing between data and stores allows us to separate the requirements of the content from the properties of the container. Data descriptions include attributes such as capacity, data loss rate, retention and expiration times, and recovery dependencies; they are described more in Section 3.1.1. Store properties, which are provided by the underlying data protection techniques, are described in more detail in Section 3.3.

Describing data separately from streams allows us to separate reliability from performability. Our primary technique for specifying data accessibility is the use of stream performability requirements, which indicate how often certain performance levels should be achieved [17]. These stream requirements are described in more detail in Section 3.1.3.

Figure 3 (in the appendix) is an example that folds together many of the points we describe here. It is structured using the Rome data model [16]. The example represents the format that would be seen by one of our design tools, rather than the form seen by people specifying this information, for which user-oriented GUIs are more appropriate. We encourage the reader to follow along as these concepts are introduced.

3.1 Specifying user requirements

In this section, we discuss the requirements for which the users must provide input, including data and stream attributes.

3.1.1 Data: content

Data is the information content that is stored. At the storage system level, a data item is relatively large – a logical volume, a complete file system, or a database table. Higher-level software, such as a file system or database, maps smaller data objects such as files or records into these larger-scale data objects. We assume the existence of a primary copy of a data item, possibly with completely or partially redundant secondary copies.

Reliability is the most important data property: how much data loss is tolerable? Although users are likely to answer "none," they must decide how much they are willing to pay to bound the amount or rate of data loss. For a large system, it makes sense to think of a "mean data loss rate" that is achievable for a given price.

Different types of data may have different reliability needs. For instance, it might be more cost-effective to allow up to 30% of the data for an Internet search application to be lost than to pay for its protection, as this will only impact the quality of answers it can provide, without impacting its ability to provide an answer [6]. Similarly, a database index can be rebuilt from the raw data, so it can be protected with less expensive techniques than the data it indexes, as queries can be posed against the main table(s) temporarily.

Thus, we believe that the appropriate data-related dependability properties are as follows:

- *capacity*: the size of the content, in bytes.
- *dataLossRate*: the allowed rate at which a particular size of data loss can occur, specified as a <bytes, time interval> tuple. When the size is one byte, the interval is the inverse of the traditional "mean time to data loss" metric. Some failure modes and store designs are such that only a portion of the data may be lost (e.g., at most a day's worth of updates). This measure allows different designs to exhibit different properties, even if their traditional "reliability" values are the same.
- *dependsOn*: data-level recovery or integrity dependencies. For instance, the indices for an order-entry database depend on the underlying tables, implying that the tables should be recovered first after a failure. Additionally, applications may share data, making the description of these dependencies a DAG rather than a tree.

3.1.2 Retrieval points

Keeping read-only copies of a data item's state is a classic technique in data protection. These copies have been called versions, generations, snapshots, checkpoints or backups; we prefer the term *retrieval points* to separate the intent from the technique. A retrieval point permits "time-travel" in the storage system by capturing the state of a data item at some moment, with the expectation that this state can be accessed in the future. Retrieval points can be used to satisfy many needs, including protection against device failure, protection against user error or malicious actions [12, 14], protection against software errors or data corruption, legal requirements (e.g., audits), preserving a particular data state or a related set of data item states (e.g., archiving all the designs for a particular aircraft engine), and wanting to look at prior versions "in case they still have value" (e.g., old RCS versions or the file system structure used last year).

Retrieval point properties depend on the purpose of the retrieval point. Some retrieval points are single-shot archives (e.g., the design for an airplane engine). Such an archive may have required retention times, and accessibility and reliability properties. Some retrieval points are better thought of as a series of related point-in-time snapshots, generated by some automatic technique. A series may be better described by the number of retrieval points it should contain and a bound on the intervals between them, which implicitly specifies how many recent updates the user is willing to lose.

The useful lifetime of a retrieval point depends on the time to discover the need to recover data. For example, a person may take a few seconds to realize that he or she made a mistake, or a file system consistency checker may only be run once an hour. This time-to-detect acts as a natural lower bound on how long an associated retrieval point should be kept. Often, time-to-detect is quite short: research indicates that humans typically catch about 70% of their own errors, often within a few minutes of making them [11].

We describe retrieval point properties as follows:

- *capacity* and *dataLossRate*: defined as for a data item. It is useful to distinguish between the dataLossRate for the retrieval point and that of the parent data item, because people may value these versions differently.
- *retention* and *expiration times*: how long the content must be retained, and when it must be expunged. For example, retain the data forever and never expunge it, or keep it for seven years and then discard it immediately. The expiration date must be no earlier than the retention date. Associating these times with a read-only retrieval point avoids the difficulties of deciding what to do for data items that are being updated, where the starting point of the retention/expiration period is often unclear: is it the creation time, the update time, or the last access time?
- *count*: the number of retrieval points of this type to retain. This attribute can implicitly specify how long retrieval points should be kept. Note that the user cannot specify both a count and a retention time.
- *interval*: the interval between retrieval points. This attribute is a measure of the amount of data loss that can be tolerated on a failure or mistake. It can be specified as a time (e.g., 30 minutes), an amount of data (4MB), or a count of updates (1 million writes).

A data item can have a series of retrieval points, each with associated resiliency and lifetime properties. For example, specifying the three retrieval point series <interval=60 seconds, count=60, dataLossRate=1 B / 5 minutes>, <4 hours, 2, 1 B / 12 hours>, <3 months, 1 B / 50 years> might cause the system to take a low-resiliency snapshot every minute, a more resilient one every four hours, and a nearly indestructible one once every quarter.

3.1.3 Streams: access patterns

Streams describe the access patterns to an associated data item. Performance requirements may be as simple as request rates and request sizes, or rich enough to include other quantities, such as spatial and temporal locality, phasing behavior, correlations between accesses to different parts of the storage system, and response time goals. (A more detailed description can be found in [16].)

We specify data accessibility using stream-based performability requirements, which indicate how often certain performance levels should be achieved. In particular, we describe both the baseline performance requirements under normal operation and the performance requirements needed during *outages*, which are periods of degraded operation. The allowed frequency and duration of outages can also be specified.

This approach is in contrast to the traditional, rather simplistic, "number of nines" availability metric, which implicitly supports only "all" or "none" as its two performance levels. In our scheme, the traditional "no availability" case is an outage with zero access performance. In addition, several intermediate outage levels can be described, each with its own performance specification. For example, if a system can tolerate operating at 50% of normal performance for a while, then this time can be used to bound the recovery time for a mirrored disk failure.

We describe outages using the following attributes:

- *outage duration*: the time duration of the longest tolerable outage. Note that this quantity implicitly includes the time to detect, diagnose, repair and recover from the failure, as described in Section 3.2. As a result, the overall recovery time must be strictly less than or equal to the outage duration.
- *outage frequency*: the maximum number of separate outages that are permitted (or measured) during a user-specified period (e.g., a year).
- *outage fraction*: the fraction of the total time (averaged over the user-defined period) that can be outages.

A common approach to handling the case where data has to be retrieved from a secondary copy is the use of a

"time to first byte" specification, to cover the recovery time. We believe that this case is better handled by specifying the allowed delay as an outage, which can also be used to put bounds on the number and frequency of such recoveries.

A final category of stream characteristics governs the amount of recently-written data that may be lost on a failure (e.g., from a file system write buffer in volatile memory). This amount is related to the update rate and the fraction of the data that is changing, so we believe these notions to be most appropriately categorized as stream (rather than data) characteristics:

- *recent-write data loss*: the amount of the most-recently-written data that can be lost on a failure. This parameter may be specified in terms of time (e.g., no more than the last 30 seconds) or in terms of volume (e.g., no more than 1MB).
- *recent-write loss frequency*: an occurrence frequency can be associated with this event to bound how often it may happen.

Note that individual retrieval points probably have different stream characteristics from each other and from the parent data item. Applications may only update the data item (e.g., the most recent copy), while they may read either the most recent copy or one of the retrieval points. For example, a retrieval point implemented as a file system snapshot may be read by the backup system to copy the data to archival media. Alternately, a retrieval point may be read to restore data upon software or user error. A retrieval point may also be directly accessed by an application. For example, a data mining algorithm may be run on a retrieval point for data from the OLTP database. The characteristics of such streams will have a large impact on what techniques can be used to provide the retrieval point (e.g., rapid recovery from user error and tape backup are not a good match).

3.1.4 Common attributes

Several attributes cut across the data and stream attribute categories, including:

- *application*: the business functionality that corresponds to a particular data item or stream (e.g., order-entry OLTP database or email server). The application is used to group related data items and streams.
- *dependsOn*: application-level recovery dependencies or ordering requirements. For instance, after a site failure, the OS data should be recovered before any of the application data.
- *utility*: Trade-offs can be specified using utility functions [8]: utility is positive when the system is delivering value (e.g., performing at or above its goal), and negative (e.g., a penalty) when the system under-achieves the goals. Such a utility function would allow us to prioritize applications differently. For example, getting 100 requests/sec for the order entry data may be more important than getting any requests to email data, but that once email achieves 50 requests/sec, then increasing the order-entry capability would be valuable.

  The reader can think of a utility function being provided for any of the above metric values. It is important that there is a "common currency" for utility; ultimately, we believe that this should be the same as the currency used to pay for the system. We are currently considering how best to specify generalized utility functions.

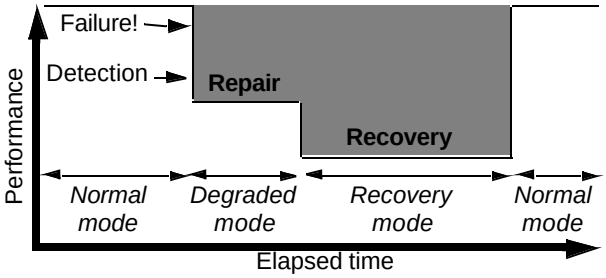

Figure 2: failure behavior modes and performance.

3.2 Specifying storage system failures

Failures can cause a system to lose data or to lose accessibility to data. They may occur at many points in the system, including the hardware, the software, and the humans interacting with the system. In fact, the literature suggests that humans are now responsible for the largest fraction of failures in many systems [11].

Our model of failure behavior [17] is illustrated in Figure 2. A *failure* occurs; some (hopefully short) time later this is *detected* and *diagnosed*, and the system enters *degraded mode* operation. If the technique(s) mask the failure, operation will continue at a (potentially reduced) non-zero performance level. After the failure is repaired, *recovery* is then initiated, and the system returns to normal operation (which may or may not be identical to the initial state, due to load balancing issues, etc.)

We characterize failures by the likelihood of their occurrence and the severity of their effects:

- *failure frequency*: what is the likely rate of failures of this type (expressed as an annual failure rate per entity)?
- *failure scope*: what part of the system is affected by the failure? The scope includes how many objects are affected at a time (e.g., all disks of a particular type or all files in a directory). The design system itself will calculate the effects of cascading, dependent failures.
- *failure correlations*: how often are failures not independent? For example, multiple disks may fail at the same time if the problem is excessive temperature.
- *failure manifestation*: how does the failure manifest itself? Possibilities include fail-stop (our focus for now), fail-stutter [4] or Byzantine failures.
- *early warning*: how much warning time is provided

before the failure? For true failures, this amount is likely zero; however, planned maintenance operations may provide as much as weeks of warning.

- *failure duration*: how long is the failure expected to last? For planned maintenance, we may be able to estimate the duration of the outage.
- *failure fix unit*: what's the minimum field replaceable unit (FRU) to fix this failure? For example, if a disk fails, it is more cost-effective to replace the defective disk than to replace the entire array. Alternately, if a SCSI bus fails, an entire disk array enclosure may need to be replaced. The failure fix unit will ultimately determine how much data must be recovered.

Examples of hardware-oriented failure scopes include:

- *components* (e.g., sector, disk, controller, cache memory, link, bus, UPS, fan, cable, plug)
- *subsystems* (e.g., array, host, switch, LAN, air conditioning unit, building power transformer)
- *racks* (e.g., a set of subsystems)
- *rooms* (e.g., a set of racks)
- *buildings* (e.g., a set of rooms)
- *sites* (e.g., a set of buildings)
- *areas* (e.g., a set of sites, city, earthquake zone)

Software- and user-oriented failures have similar kinds of scopes: a file, all files owned by a user, a file system, a logical volume, an operating system partition, a cluster file system, etc. Similar scopes exist for a database: record, table, table-space, and the entire database.

An alternate approach to failure scopes and correlations is to specify dependence relationships between components, which can be used to construct hierarchical fault trees [13]. We are currently investigating which alternatives are most appropriate.

We assume that initial failure estimates are provided to the storage design system by the provider of the hardware or software components. This information may be augmented by monitoring the operation of a deployed system.

3.3 Specifying data protection techniques

Stores are containers in which data items reside. The family of containers used to store data is organized as a hierarchy: host logical volumes reside on one or more disk array logical units, which are comprised of one or more disks, etc. Stores may employ different data protection techniques, which use redundant data representations to ensure that data is not damaged if something fails.

Different techniques provide a wide range of performance, failure tolerance and cost properties. They differ in their:

- baseline performance under normal operation
- degraded- and recovery-mode (i.e., outage) performance
- ability to tolerate different failure properties
- repair and recovery costs (e.g., time, money)
- cost of implementation (e.g., space, time, money)

Selecting the best data protection techniques to satisfy user requirements requires understanding these properties for each candidate technique. We assume this information will be supplied by the provider of the data protection technique, estimated using models, or calculated by the storage design system, as described below.

To capture the performance categories, the design system needs one or more *performance models* of the technique's behavior – typically, one for each mode in which it can operate (normal, degraded, recovery). Several techniques have been described in the literature for estimating storage device performance under normal mode quickly and efficiently enough to be used in a storage design tool (e.g., [15]).

The design system needs to determine the probability of each performance mode for each technique. This requires knowledge of:

- *failure tolerance*: which failure scopes can the technique handle? We believe that this is best thought of as the set of performance modes entered for each associated failure scope.
- *data loss*: how much data is lost for a failure scope? Zero means that the technique masks or tolerates the failure.
- *repair time*: how long does repair take, before recovery can commence? This time includes physical repair, and can be short if the system includes hot spares for the failed component.
- *data to be reconstructed*: this value will be computed from the failure fix unit for a particular failure scope. For instance, if an entire disk enclosure must be replaced, all disk array logical units that use disks in that enclosure must reconstruct data.
- *recovery time*: this value will be calculated as a function of the desired performance levels and tolerable outages. For example, if recovery is achieved by copying data, then the speed of recovery can be varied to control the amount of disruption to the foreground load. Working backwards from the user's maximum allowed outage duration and frequency allows us to calculate the recovery traffic that will achieve recovery within the tolerable outage bounds. This traffic will have the least effect on the foreground load. Whether the resulting foreground performance is adequate is then a function of the workload requirements. If not, then the design system must choose some other technique to provide adequate protection and performance.

We are still in the early stages of mapping this space, and expect to expand this specification framework as we gain more experience with how the specifications, performance models and design tools interact.

4 Related work

Data dependability is a vast space. Much of the existing work in the systems community is on devising new data protection techniques, rather than on providing guidance on how to choose between them. CMU's PASIS project is trying to understand trade-offs in the design of survivable storage systems [18]. Their focus is primarily on threshold coding schemes and cryptographic techniques to increase data dependability (including availability, reliability and security) in the wide area.

Additional work in the systems literature deals with evaluating the performability of a system. Wilkes and Stata [17] propose a method for describing performability using variations in quality of service under normal and degraded modes of operation. Brown and Patterson [7] describe how to measure the availability of RAID systems using a similar performability framework.

In the performance space, we have successfully used the approach of declarative goal specification [16] to automate the mapping of performance and capacity requirements onto storage designs [1, 2, 3].

Although the system administration community chooses between different data protection techniques regularly, there is little published literature on the topic. The dependable systems community has developed a vocabulary for describing dependability and failure behavior [5] and techniques for modeling various aspects of dependability [9] [13]. We are in the process of understanding the considerable body of literature from this community.

5 Conclusions

Increasing data dependability is an important problem, whose value will continue to increase as service dependability becomes more important. The proliferation of techniques, the complexity of interactions and the richness of demands are making purely manual methods cumbersome. We believe that automating the selection of techniques is both necessary and promising, and that the correct goal is a broadly-scoped automated design tool for data protection. This paper has outlined our approach, and described the data model we use to describe user requirements, failure characteristics and data protection technique behaviors.

We see a number of areas of future work. First, how do we intuitively ask users what they want? Second, how do we build a suite of design tools that automates the selection of data protection techniques? Finally, we challenge developers of new techniques to describe their behavior under different operational modes.

6 Acknowledgements

The authors thank Eric Anderson, Christos Karamanolis, Mahesh Kallahalla, Ram Swaminathan, and Jay Wylie for their valuable insights and comments on earlier versions of this paper.

7 References

[1] G. Alvarez, et al. "Minerva: an automated resource provisioning tool for large-scale storage systems," *ACM Transactions on Computer Systems*, 19(4):483-518, November 2001.

[2] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. "Hippodrome: running circles around storage administration," *Proc. of the USENIX Conference on File and Storage Technologies (FAST)*, January 2002, pp. 175 - 188.

[3] E. Anderson, M. Kallahalla, S. Spence, R. Swaminathan, and Q. Wang. "Ergastulum: an approach to solving the workload and device configuration problem," HP Laboratories SSP technical memo HPL-SSP-2001-05, May 2002.

[4] R. Arpaci-Dusseau and A. Arpaci-Dusseau. "Fail-stutter fault tolerance," *Proc. of the Eighth Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, May 2001, pp. 33 - 38.

[5] A. Avizienis, J.-C. Laprie and B. Randell. "Fundamental concepts of dependability," *Proc. of the 3rd Information Survivability Workshop*, October 2000, pp. 7 - 12.

[6] E. Brewer. "Lessons from giant-scale services," *IEEE Internet Computing*, 5(4):46-55, July 2001.

[7] A. Brown and D. Patterson. "Towards availability benchmarks: a case study of software RAID systems," *Proc. of the 2000 USENIX Annual Technical Conference*, June 2000, pp. 263 - 276.

[8] G. Candea and A. Fox. "A utility-centered approach to dependable service design," *Proc. of the 10th ACM-SIGOPS European Workshop*, September 2002.

[9] B. Haverkort, R. Marie, G. Rubino and K. Trivedi, eds. *Performability modeling: techniques and tools*, John Wiley and Sons, Chichester, England, May 2001.

[10] K. Keeton and E. Anderson. "A backup appliance composed of high-capacity disk drives," *Proc. of HotOS-VIII*, May 2001, p. 171.

[11] D. Patterson. "A new focus for a new century: availability and maintainability >> performance," Keynote speech at *USENIX FAST*, January 2002. Available from http://www.usenix.org/publications/library/proceedings/fast02/.

[12] D. Santry, et al. "Deciding when to forget in the Elephant file system," *Proc. of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, December 1999, pp. 110 - 123.

[13] D. Sieworek and R. Swartz. *Reliable computer systems: design and evaluation*, A. K. Peters, Third Edition, 1998.

[14] J. Strunk. et al. "Self-securing storage: protecting data in compromised systems," *Proc. of Operating Systems Design and Implementation (OSDI)*, San Diego, CA, October 2000, pp. 165-180.

[15] M. Uysal, G. Alvarez and A. Merchant. "A modular, analytical throughput model for modern disk arrays," *Proc. of the 9th Intl. Symp. on Modeling, Analysis and Simulation on Computer and Telecommunications Systems (MASCOTS)*, August 2001, pp. 183 - 192.

[16] J. Wilkes. "Traveling to Rome: QoS specifications for automated storage system management," *Proc. of the Intl. Workshop on Quality of Service (IWQoS)*, June 2001, pp. 75 - 91.

[17] J. Wilkes and R. Stata. "Specifying data availabilty in multi-device file systems," *Proc. of the 4th ACM-SIGOPS European Workshop*, September 1990; published as *Operating Systems Review* **25**(1):56-59, January 1991.

[18] J. Wylie, et al. "Selecting the right data distribution scheme for a survivable storage system," Technical report CMU-CS-01-120, Carnegie Mellon University, May 2001.

8 Appendix: example

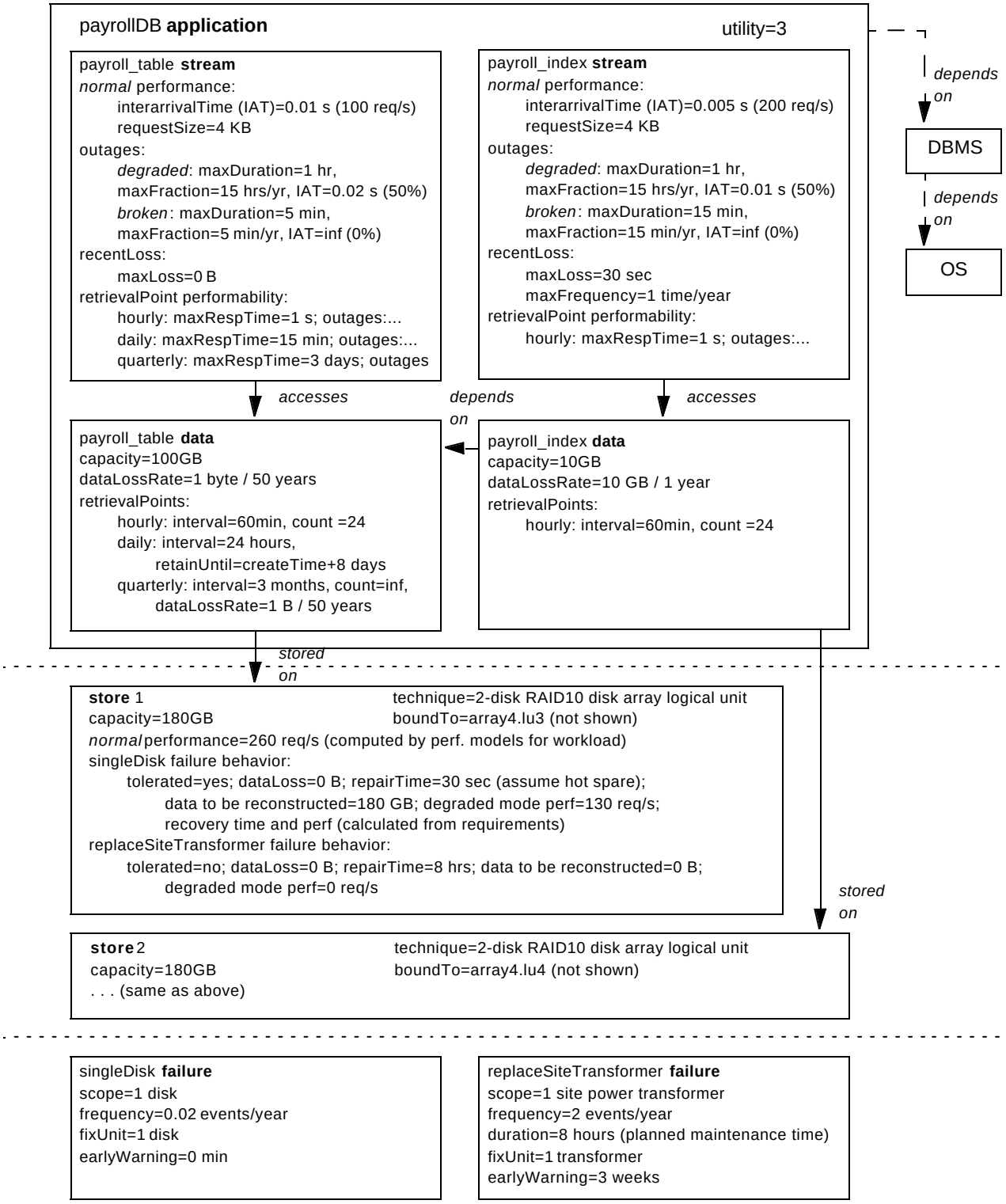

Figure 3: **a (simplified) sample data dependability specification.** A payroll database contains both a table and an index. In normal mode, the table gets 100 requests/sec; in "degraded" mode, it can operate at half that rate for an hour at a time; and it can be "broken" no more than 5 minutes a year ("five nines availability"). The index is accessed at twice the rate. Three retrieval points are defined for the table, with different time intervals and retention periods. The data is stored on RAID10 disk array logical units, whose performance under normal mode and various failure conditions is provided by performance models. We note that these stores tolerate the single disk failure, but not the site transformer replacement.