Traveling to Rome: QoS specifications for automated storage system management

John Wilkes

Storage Systems Program, HP Laboratories,
1501 Page Mill Road, Palo Alto, CA 94304, USA
Tel: +1.650.857.3568 Fax: +1.650.857.5548
<wilkes@hpl.hp.com>

Abstract. The design and operation of very large-scale storage systems is an area ripe for application of automated design and management techniques – and at the heart of such techniques is the need to represent storage system QoS in many guises: the goals (service level requirements) for the storage system, predictions for the design that results, enforcement constraints for the runtime system to guarantee, and observations made of the system as it runs. Rome is the information model that the Storage Systems Program at HP Laboratories has developed to address these needs. We use it as an "information bus" to tie together our storage system design, configuration, and monitoring tools. In 5 years of development, Rome is now on its third iteration; this paper describes its information model, with emphasis on the QoS-related components, and presents some of the lessons we have learned over the years in using it.

1. Introduction

Designing, supporting, and managing storage systems is getting harder as they get larger and more complicated. And they are getting larger very quickly: compound annual growth rates of 150% in storage capacity are not unheard of. A data center of the immediate future could easily contain hundreds or thousands of logical volumes and file systems, hundreds of terabytes of disk drives, and handle tens to hundreds of gigabytes per second (GB/s) of storage traffic. Data availability is crucial: if the storage system goes down, so does the computer system that relies on it. Achieving all this requires a great deal of complexity: multiple disk array types, different data organizations, transactional support, automated fail-over schemes, and so on.

This complexity – compounded by the desire to avoid operator intervention because of errors and the dearth of skilled system administrators – means that the design and operation of large-scale storage systems is an area ripe for application of automated design and management techniques. At the heart of such techniques is the need to represent the QoS goals (service level requirements) of the storage system, the design that results, and observations made of the system as it runs.

Even though many of the same approaches used in the large literature on QoS for the networking domain (e.g., [1]) can be applied to storage systems, there are a number of differences that make the mapping non-trivial: (1) the low-level storage protocols (based on SCSI) are highly intolerant to packet loss, so dropping requests is

Published in the Proceedings of the *International Workshop on Quality of Service* (IWQoS), Karlsruhe, Germany, June 2001. © Springer-Verlag.

not a viable technique for handling overload or congestion; (2) there are very strong non-linearities in performance that result from the mechanical properties of disks drives and the use of large caches: it is easy to construct scenarios in which an inappropriate mix of I/O traffic to a disk drive can change the data transfer rate by a factor of 50; (3) there is no support for traffic shaping or QoS enforcement techniques in the storage system itself; (4) the cost of the storage system is often the dominant component of the overall system cost; and (5) dynamic quality adaptation at the application level is extremely rare. These factors mean that the primary approaches to providing QoS guarantees are provisioning and resource (re)-balancing, and that the only possible guarantees are probabilistic in nature. It also means that the portions of the QoS specifications for storage systems that describe I/O behavior have to be considerably richer than for many other domains, and the language used to describe them needs to be correspondingly more expressive than is usually the case for network traffic.

Our approach to the storage design problem has been to develop an architecture for a quality-of-service-based "attribute-managed" storage system [3]. This uses QoS goals to specify what is wanted to the storage system, which is then responsible for deciding how best to provide it – that is, the technically hard part of what we have built is an automated provisioning and load-balancing design tool that uses QoS goals as targets. We embed this design tool in a system that can automatically apply its decisions to a running system, monitor the result, and make better designs – all completely automatically.

In this scheme, the desired storage system goals are specified in terms appropriate to the client of the storage system (e.g., I/O requests per second, number of concurrent streams, access patterns, availability, etc.) and the storage system takes care of the details of deciding how many storage devices of what type to configure, how they should be connected, and how to lay out the data and balance the load across them.

We view this fundamentally as an optimization problem: something that computers are rather good at. Our contributions have been in defining the problem in a way that makes it tractable, applying our detailed knowledge of storage system components, developing techniques to understand and quantify the QoS specifications that such systems have to meet, and putting together design tools that can solve this problem, together with a complete suite of ancillary components to make it operational.

One of the most important – and certainly most central – of our contributions has been the Rome object model that is the subject of this paper. The Rome object (or information) model acts as an "information bus" to tie together our automated storage system design, configuration, and monitoring tools.

Rome is used to describe *everything* that we consider important about storage systems and their elements: the workloads presented to a storage system; the QoS goals of the system; the kinds of storage and network devices that storage systems can contain, and how they can be configured; how a specific storage system is configured; how the workloads are spread across those storage devices; how the storage components are connected together (e.g., through a FibreChannel or Ethernet-based SAN); end-user goals for the system and its behavior; both existing systems and potential "what if" designs that might better meet current or future needs; and information about the current state of a running system, and how it is behaving.

The tools we have developed either take in Rome description files, or emit them, or both. This makes it possible to compose these tools in many different ways, to achieve a wide range of different effects. It also makes it easier to write the tools:

each one need only concern itself with its portion of the problem, and can rely on other tools in the SSP suite to fill in the bits it doesn't deal with. In 5 years of development, Rome is now on its third iteration; this paper describes its information model, with emphasis on the QoS-related components, and presents some of the lessons we have learned over the years in using it.

2. The flight from Troy: automated storage system designs

Our first activity was to survey the literature and develop a formulation of the design problem as a constraint-based knapsack (bin packing) problem [12]: storage devices were represented as knapsacks, with multiple constraint dimensions (capacity, throughput, bandwidth, utilization, etc.); storage loads as the things to pack into them. The design problem is to select the appropriate set of storage devices, and a packing of storage loads onto them, to minimize some objective function (typically system cost).

Our second action was to develop a storage system designer (called *Forum*) that would take in storage system QoS requirements (expressed as I/O workload demands) and a library of storage device descriptions (recording their performance capabilities), and explore the search space of designs or assignments: bindings of workload elements to storage devices. This paper focuses on how we represent the QoS requirements in this system and its successors.

It happened that we had earlier developed a technique to control a storage-system simulator using the Tcl language [6, 9, 17], so it was natural for us to apply this to Forum. The basic idea was very simple: make everything an object, and add attributes to those objects to describe additional properties. Tcl's nested lists made this easy to record in the obvious way (see Fig. 1).

In this architecture, a `store` is a container for data, such as a logical volume; a `stream` represents a dynamic access pattern to it – the important part of a QoS specification. More than one stream can target a single store. The store has two

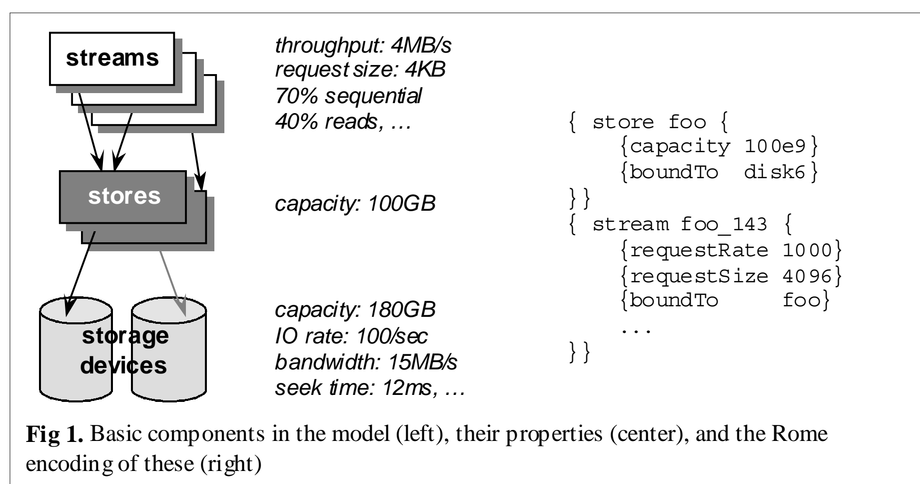

Fig 1. Basic components in the model (left), their properties (center), and the Rome encoding of these (right)

attributes: its size (`capacity`), and the fact that it is `boundTo` – in this case meaning *realized by* – a particular disk. Our Tcl-to-C++ interface made it easy to turn such Tcl statements into C++ objects. We chose to equate the statements with top-level objects, and the attributes with secondary objects hanging off the former, indexed by their names. An important behavior was that any attributes not recognized by a tool are merely passed through to the tool's output, but are otherwise ignored.

The basic object data structure with its attributes has served us very well. It made it easy to add new attributes – either to extend the standard set, or to support a tool-specific extension, or to try out a new idea. The "ignore things you don't understand rule" has made it easy to extend the set of attributes supported by some tools without affecting others: something that would have been very much harder if our primary interface had been an API instead of an object model.

We started with a very simple set of workload attributes: mean request rate (I/Os per second), mean request size (bytes), fraction of reads, and run length (the number of consecutive requests to sequential addresses) [12]. However, our prior work on understanding storage system behavior (e.g., [10, 11]) told us that these simple, fixed values would not be sufficient: storage system traffic is very bursty, and an understanding of this is vital to understanding its behavior (e.g., it is self-similar [7]). As a first step we augmented these simple mean values with variance, modeled on a normal distribution. Even though we suspected that this might not be all that good a fit to the real underlying process, it was compatible with our performance models, and therefore useful.

The QoS specification for a storage design could now be expressed in terms of a set of stores, each with zero or more streams directed to it, where the streams specified the required access patterns that were to be achieved. We found it helpful to distinguish between requirements and behaviors: *requirements* are the demands made on the system that its performance is to be measured against; *behaviors* are the actions taken by the load generator in asking for those requirements. For example, a requirement might be for a data rate of 1MB/s; a behavior might be to deliver this requirement as a stream of 1000 random-access 1KB reads per second – which would be a very different load on the storage system than a single 1MB request per second.

The obvious next question was: where should such data come from? Talking to customers and others trying to design storage systems, it became clear that few of them were comfortable providing estimates of the load currently imposed on their systems, let alone future loads. Despite this reluctance, it's important to point out that designers and customers are already forced to do just this by the existing manual methods of storage system design. These are still dominated by simple, rule of thumb "speeds and feeds" analyses: "this sounds a bit like the system we designed last week for _____ except that I think we should bump up the I/O rate by a bit (how about 20%?), and add a bit more random-access capacity in the disks for the indexing system." The main difference was in the degree of specificity required: we early on adopted the slogan that "the more you can tell us, the better a design job we can do". (For example, we once took advantage of anti-correlation data in the workload for a database benchmark to reduce the storage system cost by a factor of about 6.)

The two answers for where the data could come from were (1) measure an existing system, and, if necessary, extrapolate to a new one; (2) build up a library of prior workloads that we had met, and – just like our human designer counterparts – slowly accumulate wisdom about how workloads scaled, and capture these in tools that could

emit an appropriately-scaled workload approximation given a small number of knobs (much like the workload scaling parameters in the TPC database benchmarks [14]). The latter proved relatively easy for some simple workloads, but the full generality of mapping upper-level application QoS specifications down to low-level storage system behavior remains a difficult research topic, an experience shared with others in different fields.

The measurement path proved much more conducive to automation. The HP-UX operating system, which we used as an experimental platform, contains a measurement system that we could use for this very purpose: it is able to emit a trace record for every single physical I/O request, at a negligible cost (a couple of percent processor utilization). We wrote some tools to scan over such I/O traces, and generate the Tcl files describing the streams and stores.

One more component was needed: a representation of the performance characteristics of storage devices. For our first round, we restricted ourselves to disk drives, and were able to convert a subset of the data gathered for the simulation models used in the Pantheon simulator [17] into analytical models of disk drive's performance. The switch to analytical performance models inside the Forum design tool was required because the models can be called millions of times during the exploration of the design space, and simulations are simply too slow. To increase the flexibility of our models, we structured them as a set of composed components [13].

We completed the tool chain by developing an automatic configuration system called *Panopticon*: given a design of a storage system (i.e., a mapping of stores onto disk devices), it would construct the appropriate logical volumes to achieve this mapping, and we could then run a test on the resulting system.

2.1 Returning to the Imperial city: supporting disk arrays

Bolstered by our success with the early Forum results, we thought it would be a simple transition to extend our performance models and design language to encompass disk arrays. We were wrong. Disk arrays themselves need to be configured, and frequently have a great deal of internal complexity. In fact, choosing how best to configure a disk array to support a given workload is itself a challenging, NP-hard design problem.

Our first approach was called Minerva; it used "divide and conquer" to break the dependencies. A Minerva run begins by estimating the amount and kind of disk arrays that would be needed to meet the workload's requirements, using some very simple performance and capacity models, and pre-configures this set as the device descriptions input to Forum, which then attempted to pack the workload on to these disk arrays. If the workload did not completely fit, the estimating process was repeated with the left-over workload; if it did, then the Forum solver was re-run with the objective of evening out the load imbalance across the configured hardware, rather than minimizing the system cost.

By this time, the range of workload parameters we had begun to record was becoming rather too large for an *ad hoc* tool, so we developed a flexible data-analysis framework, *Rubicon*, that used a structure rather like that of a packet filter to perform analysis of an I/O trace and generate workload information.

Separating the internal and external representations also proved vital: we learned the hard way to distinguish between the information model and the manifestation of it

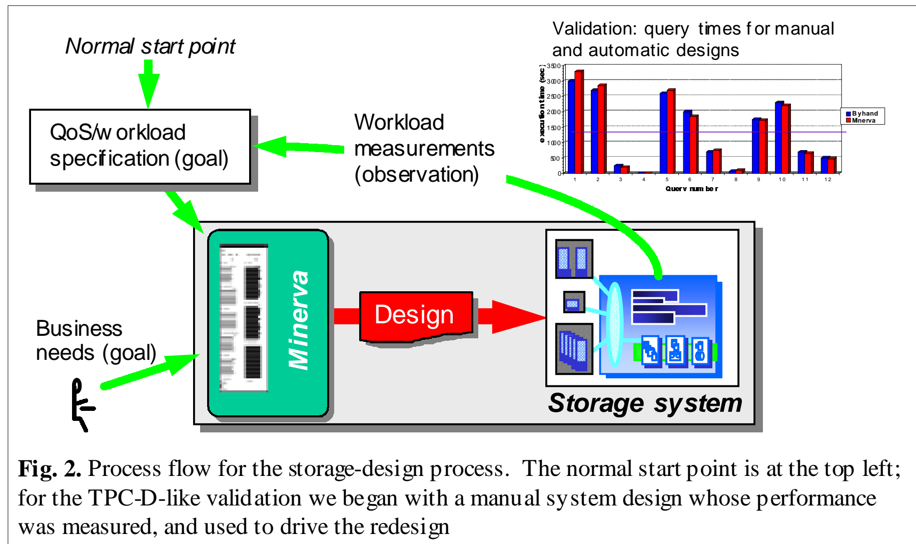

Fig. 2. Process flow for the storage-design process. The normal start point is at the top left; for the TPC-D-like validation we began with a manual system design whose performance was measured, and used to drive the redesign

in a tool as C++ objects: it is simply not possible to define a single representation of the important attributes of an object for all tools. (For example, performance data is completely ignored by the Panopticon configuration tool, so it would be counter-productive to insist that it use the same internal C++ data structures as Minerva.)

Using Minerva and its support tools, we were able to perform the following test (see Figure 2): we consulted a local database guru on how to configure the TPC-D benchmark [14] for our system; configured the system that way and ran the benchmark, measuring the I/O rates it achieved; then used these values as QoS goals input to the Minerva design tool, and asked it to match the performance QoS requirements at minimum cost. We used Panopticon to construct the resulting storage system design automatically (it had been extended to perform array configuration too); reloaded the database, and re-ran the benchmark. The performance result gave us query execution times within 2-3% of the original, but the Minerva-based design only needed 16 disk drives in a RAID-5 disk array configuration, whereas the manual design had required 30.

2.2 The arrival of the Visigoths

As time went on, the internal structure of the Minerva design tool started to slow us down, and Eric Anderson developed a new storage system designer (*Ergastulum*), which explored the disk array design space at the same time as it assigned work to the array. This had a number of advantages, not the least of which was that it eliminated the rather cumbersome two-phase design process that Minerva had used.

Ergastulum happened to be much faster than Forum, and showed that parsing the old Tcl structures was becoming a significant overhead, so we switched to a hand-coded parser, and simplified the language design a little. Ergastulum also needed a way to express the range of disk array designs that it could explore, and, rather than hard-code these into its logic, we chose slightly to extend the flexibility of the

language used to describe the storage devices in the device library. We called the result *Rome*.

3. Rome 2: The Renaissance – rebuilding St Peters

After some time, the first version of Rome became a slightly messy hybrid, with some backward-compatible components, and some forward-looking elements. It was sufficient to prompt us to undertake an overhaul of the result. The result is the present version, known as Rome 2.

Rome 2 was required to be good at describing real-life storage systems – their workloads, configurations, and the components they contain (devices, hosts, software, networks, etc.); extensible to encompass new storage devices, workload attributes, and even new target domains, such as internet data center configurations; rich enough to represent many levels of complexity; and capable of being represented in ways that are easy to parse, generate, and use by computer tools, and in (possibly different) ways that are understandable by humans.

Rome 2 achieves these goals by separating its underlying *object model*; from the linear encodings of it, which are known as *Latin* and *Greek*. Latin is the "native" language of Rome, and is used to specify it in the descriptions that follow; it uses a Tcl-like syntax derived from our earlier experiences. (Figure 3 shows a simple QoS specification written in Latin.) Greek is an XML-based linear encoding derived from Latin.

3.1 The Rome object model

The Rome object model is built on an object-type inheritance hierarchy; it provides the underlying structure used by the Rome object model to describe things of interest to the Rome tools. *Objects* in Rome represent things like disk arrays, or part of a storage-system workload such as a stream. Each object is introduced by a single *declaration*, has a unique *name*, and has an associated set of *attributes*, which provide additional information about the object. Attributes are modeled as objects in their own right; some of them represent internal *components*, such as the I/O controllers on a disk array. Much of the Rome object model specification is concerned with describing the attributes that each object type can have associated with it.

The Rome object model is defined in two layers. The lower level is known as the Rome *shallow semantics*. This occupies a middle ground between the syntax of the representation language (e.g., Latin), and the deep semantics, which is an ever-extending set of object types, components, and attributes, with their associated meanings. Because the meanings of the shallow-semantics objects are reasonably well standardized across the tools, we can build software libraries to handle the common operations on these object types. Examples of such objects include approximate values (of which more below) and common properties such as cost.

The Rome *deep semantics* defines the meanings of the remaining object types and their attributes. Examples include workload measurements and performance requirements (i.e., QoS specifications); data mappings (such as RAID); storage devices; interconnect fabrics (including SANs); and host hardware and software. The

```
{ store store1 {                                # a 100GB store
  { capacity 100e9 }                            # mapped to a logical unit
  { boundTo array4.lu_3 }                       #  on an array (not shown)
}}

{ stream stream1 {                              # a stream
  { boundTo store1 }                            #  bound to that store
  { source  host_A1}                            # originating at this host
  { interArrivalTimeOpen {                      # inverse of request rate
       { datamodelNormal best {                 # normal fit
           { mean 0.83e-3 }{ stddev 0.6e-3 }    # mean = 1200/sec
           { chiSquare 0.7 }                    # goodness of fit metric
       }}
       { datamodelExponential poor {            # exponential fit
           { mean 0.83e-3 } { chiSquare 0.2 }   # 1200/sec, less-good fit
       }}
  }}
  { requestSize {                               # a simple behavior
       { datamodelUniform {                     # uniform size in 4-12KiB,
            { mean 8192 }                       #  on 1024-byte boundaries
            { lbound 4096 } { ubound 12288 } { granularity 1024 }
       }}
  { responseTime {datamodelExponential {mean 50e-3}}}  # a goal
  { stream read {                               # just the read requests
      { filteredBy { opType read }}
      { interArrivalTimeOpen 1e-3 }             # 1000/sec
      { requestSize 9216 }                      # larger requests on avg.
  }}
  { stream write {
      { filteredBy { opType write }}
      { interArrivalTimeOpen 5e-3 }             # 200/sec
      { requestSize 4096 }
  }}
  { stream degraded {                           # something not right
      { filteredBy {{ outageDuration 3600 }     # 1 hour at a time
                    { outageFraction 0.002 }}}  # 17 hours/year
      { interArrivalTimeOpen 1.67e-3 } # 600/sec
      { stream write {
          { filteredBy { opType write }}
          { interArrivalTimeOpen 0.1 } # 10/sec
      }}
  }}
  { stream broken {                             # completely stopped
      { filteredBy {{ outageDuration 300 }      # 5 min at a time
                    { outageFraction 0.00001 }}} # 5 min/year
      { interArrivalTimeOpen inf }              # nothing: 0/sec
  }}
}}
```

Fig. 3. A (much simplified) sample workload specification example. One store is accessed by one stream. In normal mode, it gets 1200 requests/sec; in "degraded" mode, it can limp along for an hour at a time at half that rate; and it can be "broken" (non-accessible) no more than 5 minutes a year ("five nines availability"). For simplicity, most of the datamodels shown are simple numeric values; in practice, distributions would normally be used.

interpretation of the deep-semantics varies considerably from one tool to another, so it is much harder to provide a shared code library for them that does much more than define the basic object type.

Rome represents the idea that many things are alike by giving the objects that represent those things a common *object type* (or `objectType`). Object types are first-

class entities in Rome – that is, they are objects in their own right. An objectType declaration introduces (defines) a new Rome object type, after which objects and sub-objects (attributes) with that object type can be *declared*. Such declarations can occur at "the topmost level" (i.e., free-floating in the global namespace), or nested within another object. That is, an objectType object is loosely equivalent to a programming language class definition. Rome object types form a single-inheritance isA hierarchy.

An objectType attribute in an objectType declaration defines a sub-object type that only applies in the context of objects of the enclosing objectType. Such components (and their types) can be arbitrarily nested.

```
objectType <qualname> {
  [ { isA        <qualname_objectType> } ]
    { objectType <name> { <obj-list_component>} }*
  [ { occurrenceCount <number> | <numeric-range>}} ]
    <obj-list_type-paramaters>
}
```

This description comes directly from the Rome BNF-like specification; <angle-brackets> enclose non-terminals in the grammar, [square brackets] denote optional components, and an asterisk (*) indicates elements that can be repeated zero or more times. It means that an object type declaration takes as first argument a qualified name such as type.sub_type (i.e., dot-separated nesting is allowed), and a list of attributes. One of these attributes may be an isA attribute, which takes as argument the qualified name of another objectType, and defines the inheritance hierarchy. One or more type-specific attributes exist, defined by their own objectType declarations. An occurrenceCount attribute may be present to bound the number of instances of this object type that should be instantiated for every instance of the enclosing object. And, finally, there can be a list of objectType-specific attribute values that act as defaults for objects of the newly-declared objectType (shown as <obj-list$_{type\text{-}paramaters}$>).

The set of object types recognized by a Rome tool varies from tool to tool; it is based purely on the name of the object's objectType. Some tools have various object types built in (for example, a storage-system design tool will probably know about storage devices that it can design for); and some will not (e.g., a pretty-printer). Unrecognized object types are quite acceptable: in such circumstances a tool should either ignore the unrecognized object, or pass it on to its output. However, a tool may demand that certain objects or attributes exist, and the occurrenceCount attribute in an objectType declaration can specify the allowed number of instances of an object.

3.2 Attribute inheritance

Attribute inheritance is the process by which attributes are searched for in an object. If an attribute is present in the object, then it is used. If not, the attribute is looked for in the object's objectType declaration to see if a value is provided for it: that is, the search follows the isA type hierarchy, all the way up to the root, if necessary. Values provided closer to the point where the search originates take precedence. Each missing attribute is searched for independently.

This allows the `objectType` statement to store the attributes that all objects of that type have in common. Since adding an attribute or a component to an object does not change the type of the object itself, the number of base Rome object types is quite a bit smaller than in some other systems. Thus the Rome objectType `diskDrive_Quantum425S` is an instance of the `diskDrive` type, and includes values for the attribute parameters that don't vary across instances of Quantum 425S disk drives, such as capacity and performance parameters.

3.3 Approx values and datamodels

Rome treats data values that represent continuous, real-world values in a special way. It recognizes that such values are only approximate estimates of the underlying real-world process, and represents this by explicitly referring to such values as *approx values*. You can think of an *approx value* as the result of statistical sampling or characterization efforts on the real underlying process or value. Multiple independent measures or estimates can be provided; these are called *datamodels*. Datamodels can be simple or complex, ranging from a simple mean value up to complete distribution histograms of observed values. The currently supported set includes: Normal, Gamma, Exponential, Uniform, Constant, and Histogram.

There are three important properties of this idea. The first is that we use approx values almost everywhere that a value might be expected: this means that *everything* naturally becomes a statistical specification. The second is that approx values are used to express a range of allowed values, e.g., for a goal or a prediction. Most other QoS specification approaches that we are familiar with define a fixed, desired value, and then discuss what happens when variations from it occur. Our approach is to start by assuming the presence of variation, and then try to provision to support it. The third is that each datamodel has an associated random number generator that can produce values drawn from an equivalent distribution. This allows us, for example, to measure a trace, and then construct a similar one for replaying by simply generating I/O requests with similar distributions of inter-arrival time, request size, and so on.

Each datamodel type has its own particular set of parameters – as well as a set that can be applied to all datamodels. For example, a normal distribution datamodel fitted to the observed data would have mean and standard deviation parameters; it might also have data on the observed largest and smallest values observed, a count of the number of observations, how the data was gathered or filtered, and data about the underlying process that would otherwise be lost, such as the intrinsic minimum and maximum values.

Datamodels can represent truncated distributions: ones with hard upper and lower bounds, and ones with an intrinsic granularity, such as a block size.

A datamodel instance can have a name, so that there can be more than one of them: for example, two different datamodels of the same process, each of which can be provided with different goodness of fit measure to estimate how well it captures the underlying process. Note that a tool may choose to use a less-well fitting datamodel if it is unable to process the better model – for example, a tool using a queuing model may be restricted to exponential distributions, even though a Gamma model may be a better fit.

3.4 Storage workloads

A *workload* for a storage system is made up from one or more related *workload elements* (streams and the stores they target, and other workloads) that are applied to a system all together, or not at all. A workload can be used to represent an application, or part of an application, or a group of applications. Workloads may physically contain workload elements, or merely group ones that are defined elsewhere together – or they can do both. Loops are not permitted; nesting is.

```
workload <qualname> {
        { store   <name> { <obj-list_store> } }*
        { stream  <name> { <obj-list_stream> } }*
        { contains <qualname_workload-element-list> }*
        { workload <obj-list _workload> }*
}
```

We envisage that workloads will also capture relative importance of applications, and their security attributes, even though this doesn't yet occur.

3.5 Stores

A Rome store represents a container for file systems or database tables. A store can potentially handle many streams, and has only one intrinsic attribute: the capacity it provides to its clients, measured in bytes.

A store also demands backing space for its contents, and this is handled either by binding the store to a storage device (strictly, a logical unit on that device for block stores), or by mapping the store onto one or more lower-level stores through a `layout` attribute, which supports mappings such as mirroring, logical volume managers, and RAID data protection. These mappings may occur many times between the high-level logical volume seen by a database or file system, and the low-level disk mechanisms used to store their data.

3.6 Streams

A stream specifies the dynamic aspects of a workload imposed on a storage system. Each stream targets just one store. The stream attributes represent a combination of stream *requirements* and client *behaviors*. Requirements are goals that the storage system must meet (e.g., the request rate and request size); behaviors characterize the workload under which those requirements are to be provided (e.g., the request arrival process).

A stream can be looked at several different ways, and the specifications reflect this. The simplest is simply to record the desired, predicted, or observed access pattern. Others include filtering the accesses by (for example) operation type, or on the permitted degraded modes of operation. We have found every one of the attributes described here to be necessary; doubtless, as we progress with our workload modeling, we will add to this list.

<table>
<tr><td>streamType</td><td>block | NFS | CIFS | localUNIX | localWindowsNT</td><td>default is block</td></tr>
<tr><td colspan="3">Identifies the type of operations that the stream supports</td></tr>
<tr><td>boundTo</td><td><qualname$_{store}$></td><td></td></tr>
<tr><td colspan="3">Names the (one) store to which this stream is bound. A store can have multiple streams.</td></tr>
<tr><td>source</td><td><qualname$_{source}$></td><td></td></tr>
<tr><td colspan="3">The name of the host system or device that generates the load represented by this stream.</td></tr>
<tr><td>filteredBy</td><td><obj-list$_{filterTypes}$></td><td></td></tr>
<tr><td colspan="3">How (if at all) this substream was filtered down from the enclosing stream. The value is a list of parameters used for the filter, such as operation type, outage information, or phasing data.</td></tr>
<tr><td>interArrivalTimeOpen | interArrivalTimeClosed</td><td><approx$_{seconds}$></td><td>default is 0</td></tr>
<tr><td colspan="3">The time between requests issued to the storage system for this stream. If the arrival process is open, then this represents the rate at which requests will be generated regardless of the service time; if the arrival process is closed, it represents the “think time” between the completion of one request and the start of the next.</td></tr>
<tr><td>numOutstanding</td><td><approx$_{number}$></td><td></td></tr>
<tr><td colspan="3">The number of requests outstanding at a time for this stream. In a goal, this attribute dominates a closed interarrival process specification, and may act as a limiter on the effective arrival rate.</td></tr>
<tr><td>requestSize</td><td><approx$_{bytes}$></td><td></td></tr>
<tr><td colspan="3">The number of bytes read or written as a single request by this stream.</td></tr>
<tr><td>runCount</td><td><approx$_{I/O\text{-}count}$></td><td>default is 1</td></tr>
<tr><td colspan="3">A simple measure of spatial locality: the number of consecutive I/O requests that will be logically-consecutive addresses in the target store. There is no requirement that all the requests in a run have the same requestSize, nor need they all be reads or writes – this is solely a measure of the starting address of a set of consecutive requests.</td></tr>
<tr><td>jumpDistance</td><td><approx$_{bytes}$></td><td>default is random uniform across store</td></tr>
<tr><td colspan="3">A simple measure of spatial locality: the distance between the end of one request run and the beginning of the next request run in the I/O stream.</td></tr>
<tr><td>responseTime</td><td><approx$_{seconds}$></td><td></td></tr>
<tr><td colspan="3">The time that a single I/O request takes to complete, including any queuing delays. A distribution can be used to specify the range of allowed values for a goal; if a single <numeric-value> is provided, it means that both the desired and maximum-allowed response time have the same value.</td></tr>
<tr><td>onTogether</td><td><qualname$_{stream\text{-}phase}$-approx$_{overlap}$-list></td><td>default is independent</td></tr>
<tr><td colspan="3">The fractions of total time that this rill is in the current phase at the same time as the other listed streams are in theirs. In practice, the list of other phases is likely to be sparse: the most important combinations are probably the "on together" and the "this on, other off". If no value is specified, the value to assume is that for independence: the product of the fractions of time that each rill is in the given phase.</td></tr>
<tr><td>locationSkew</td><td><approx$_{bytes}$></td><td>default is no skew (uniform distribution)</td></tr>
<tr><td colspan="3">A distribution to describe the access-location skew, in terms of byte offsets within the store for the beginning of independent runs of requests. (That is, if the run length is exactly 2, the locationSkew attribute is used to specify the start address of exactly half the I/Os.) The attribute value is usually expected to be a histogram, or other non-point distribution. The distribution represents the relative rate at which an I/O request in the stream commences at the given portion of the target store's address space; a point value causes all runs to begin at that precise address.</td></tr>
</table>

We used to use the request rate to specify the I/O request-arrival process, but Rome 2 changed this to one based solely on inter-arrival times, to avoid recurring difficulties associated with knowing what the appropriate averaging interval should be for the arrival rate. Now we support the following:

- *Open* processes ignore the service time of requests they issue, and continue to generate requests at the same rate regardless of what the storage system response is. This means that there is no a priori upper bound on the number of outstanding I/O requests in flight at a time. Here, the `interArrivalTime` attribute dominates, and the `numOutstanding` attribute merely represents a measured value (e.g., it may not represent what is achieved in a new assignment).
- *Closed* processes have a fixed upper bound on the number of outstanding I/O requests in flight at a time. If they have a non-zero `interArrivalTime`, the number of outstanding requests may drop below this maximum. An "as fast as possible" arrival process can be specified by `{interArrivalTimeClosed 0}` together with some upper bound on the `numOutstanding`.

Not shown are new measures we are developing for use with the large data caches that are found in disk arrays. The basic idea is to include a measure of the LRU stack depth, or a richer (but much more expensive to measure), re-reference distance histogram, but we are still calibrating these measures against real disk arrays.

3.7 Substreams

A *substream* represents a portion of, or view onto, an enclosing stream specification. (We sometimes call them *rills*, from the Scottish word for a small stream.) We speak of the substreams as being "filtered from" the enclosing stream. This filtering can occur in a number of different ways:

- by target shard (a shard is a portion of a store, such as one of the back-end disks that the store layout maps to);
- by operation type (`opType`), such as `read` or `write`;
- by `phase`, which captures the idea that the stream accesses can be characterized by one pattern for a while, and then by another, and so on. This is expressed by use of a Markov-like phase transition model, with individual phases having their own properties, including a phase duration and a list of transition probabilities to other phases. Phases can be nested, and apply to multiple different time scales. They grew out of a simple on:off model, and are applied to handle the day-to-day change in activity levels as well as shorter-term burstiness effects.
- by a performability specification (see below).

There is no requirement for the set of filtered substreams to "cover" all the possible substreams that could be extracted from the enclosing stream. For example, a block stream might have just one substream, filtering for write operations in addition to data about the stream's overall requirements or behavior.

3.8 Performability

The top-level stream attributes describe the desired behavior in the absence of failures. We refer to it as the *baseline performability specification*. Failure to meet

the baseline performance goal is termed an *outage*. Some streams can tolerate such outages – especially if the outages can be bounded in duration or frequency or both [15]. For example, an application may be able to tolerate a short downtime period once a month; or may be able to operate with about half its usual workload for a while until a broken disk can be repaired. To represent this, the duration and frequency of these outage periods can be described, together with the tolerable levels of performance during the outages. Each such *performability specification* is written as a set of attributes that are override the baseline performance for the specified outage periods. The use of approx values in these attributes naturally supports probabilistic specifications.

Attribute	Value
outageDuration	<approx$_{\text{seconds}}$>
The longest tolerable outage duration.	
outageFrequency	<approx$_{\text{number per year}}$>
outageFraction	<approx$_{\text{fraction 0-1}}$>
The first specifies the allowed number of separate outages that is permitted (measured, etc) per year. The second specifies the fraction of the total time that can be outages, averaged over 1 year.	

3.9 Goals, observations, and designs

Although their specifications may look nearly identical when written down – indeed, our early tools took in workload observations and used them as QoS goals with no editing – we have learned that it is helpful to explicitly label QoS specifications with their purpose.

- *Goals* are desired target state(s) of the system. A goal is a form of requirements specification, or service level objective, with an associated utility function: the better the goal is met, the higher the utility function value (our use of approx attribute values seldom gives us binary goals). Goals are used as inputs to design tools, and as part of the input to evaluation tools that assess whether a design meets a set of goals, compares observations against the goals, or compares two or more designs against a set of goals. They may include cost bounds, or other constraints on a design step, and (potentially) durations for when they will apply.
- *Predictions* (and their associated designs) are the anticipated outcomes of offering the target workload to a design for a storage system. They represent estimates of future observations, if such a design were to be implemented, and allow comparative evaluations of designs. A *design* is a proposed realization of a way to achieve a goal. It captures the notion of "what if …"
- *Observations*, are descriptions of a system's behavior during some time interval. (An observation may or may not fit the original goal.) There can be multiple observations – the system might have been observed at different times, or with different sampling techniques.

These are obviously closely related. For example, our current storage design testbed takes measurements (observations) of a running system; feeds these as QoS specifications (goals) into a design step that attempts to optimize the resource usage in a running system while minimizing the amount of data movement required to do so. The result is a new design, whose likely performance we can predict.

What the Rome QoS specification does not include is the system objective function: essentially, what tradeoffs to make in the design process when faced with

too few resources, or more than necessary. Determining – let alone specifying – the objective functions that system designers use is currently somewhat of a black art. The nearest that Rome gets to this is the notion of *utility functions*, which express the benefit to be received from achieving a particular value for a specification parameter. It also turns out to be necessary to introduce priorities, or ranks: for example, in order to describe the notion of "business critical" applications in the face of disasters such as site outages.

And we have discovered that the objective functions often vary during the design process: although people may start by asking "what is a minimum cost design?" they then often switch to "how well-balanced can I make the result?" or "how fast can I make it go, if we fix the budget?" This remains an area of active research for us.

4. Related work

There is a great deal of work taking place on the use of QoS in designing systems (see, for example, the survey [1]). Most of the external academic work appears to be focused on network behavior; ours targets storage systems. Indeed, we deal with issues of network design only after the data placement decisions have been made. Part of this is because of the need to simplify the problem, but – perhaps more importantly – storage area networks (SANs) typically cost only a few percent of the total storage system cost, so simple over-provisioning works quite well.

As suggested in the introduction, even though storage workloads have many similarities to their network counterparts (burstiness, etc.), storage system workloads exhibit much greater disparity in their effective loads on the underlying system from behaviors like spatial locality, and the manner in which workloads interleave. We are not aware of other work addressing this issue in the same way as our approach.

We believe that the mapping between QoS goals and the design of the resulting system is itself a significant differentiator from most other work in this area. Although there have been a few examples of prior work in the storage system space, they have tended to assume very simple QoS models. Most work that we are aware of in the network space simply punts on the mapping issues – for example, the recent switch of emphasis to DiffServ in the IETF community merely pushes the design problem out to the people provisioning and using the network infrastructure. This probably makes sense for an environment where dynamic adaptation and congestion control with dropped packets is a very successful approach, but it doesn't seem to work for storage systems.

There is some work taking place in languages for specifying goals. For example, Bearden et al [2] discuss how goals might be represented in a CIM-like information model. Frølund and Koistinen [5] describe a rich language for expressing service level agreements (e.g., it has features to tackle the probabilistic comparisons that Rome's approx values support fairly simply). Both efforts focus a great deal of their expressiveness on describing things that it never occurred to us to write down. For example, both of these languages make a point of explicitly stating that if a response time goal is 100ms, then only response times that are less than 100ms are acceptable. Although this may make sense when very general contracts are being described, it seems to be less of a good idea in the rather narrower domain of storage systems – or networks, for that matter. Instead, we take such "goodness" comparisons as self-

evident – or, if you prefer, intrinsic properties of the design tools that use them. (Try inverting the sense of a comparison to see why this seems reasonable!)

The Quo system [8] supports different operating regimes (perhaps similar to Rome's "outages"), but appears to describe them in terms of visible implementation decisions that applications can pick between, or ask to be notified about. This is at odds with our slogan: "tell us what you want to accomplish, not how to do it".

Some people find similarities between our goal-directed system design and policy-based system management. We beg to differ: most work on policy-based systems has been on policy *rules*, not policy *goals*. (A policy rule is a statement of the form *if <condition> then <action>*.) Languages such as Ponder [4] make such policy rule-based systems easier to describe, but they don't help a great deal in the mapping from higher-level goals down to selection of which mechanisms to exercise – such as which policy rules to enable.

5. Summary and conclusions

Rome is an information model for capturing the important parts of the design and management problem for storage systems. Part of that model is a representation of QoS goals, predictions, and observations – together with the infrastructure to allow these to be turned into designs for storage systems to meet stated goals. QoS specifications for storage systems appear to need to be rather richer than their counterparts in the networking space, probably because of the much greater potential for non-linear performance interactions on mechanical storage devices, and caches. By combining the QoS specification system with the other portions of the storage system design problem, the Rome tool set can manipulate a common information model, which increases the ease with which a large set of functionality can be put together and developed incrementally.

The Rome 2 object model is quite simple – yet surprisingly powerful. Making an `objectType` a first class object enables a powerful, convenient attribute inheritance model. The freedom to add and override attributes has proven crucial to allow our tools to evolve gracefully, and has helped us avoid domino effects that often result from the traditional approach of hard-wiring an object's programmatic interface on a change. The result is a great deal of expressive power with relatively little overhead.

Rome is a living design: for example, it is actively evolving to encompass our improving understanding of the most important QoS attributes required to capture the nuances of new behavior patterns. (For example, we have only recently added file-level specifications to Rome.) It is also being actively extended in storage device modeling area: a topic for which space limitations prevented a discussion in this document. We believe that its inherent flexibility will allow these changes to be accommodated with relatively little difficulty.

Finally, we hope to make Rome publicly available for feedback and collaboration.

References

[1] Cristina Aurrecoechea, Andrew T. Campbell, and Linda Hauw. A survey of QoS architectures. *Multimedia Systems* **6**:138-151 (1998).

[2] M. Bearden, S. Garg and W. Lee. Integrating goal specification in policy based management. In *Proc. Policies for distributed networks and systems (Policy 2001)*, Bristol (Jan. 2001). Springer-Verlag *Lecture Notes in Computer Science* **1995**, pp 153-170.

[3] E. Borowsky, R. Golding, A. Merchant, L. Schreier, E.Shriver, M.Spasojevic, and J. Wilkes. Using attribute-managed storage to achieve QoS. Presented at *5th Intl. Workshop on Quality of Service*, Columbia University, New York (June 1997). Available from http://www.hpl.hp.com/SSP/papers/

[4] N. Damianou, N. Dulay, E. Lupu and M. Sloman. The Ponder specification language. In *Proc. Policies for distributed networks and systems (Policy 2001)*, Bristol (Jan. 2001). Springer-Verlag *Lecture Notes in Computer Science* **1995**, pp 18-38.

[5] Svend Frølund and Jari Koistinen. Quality of service specifications in distributed object system design. In *Proc. 4th USENIX Conf. on object-oriented technologies and systems. (COOTS)*, April 1998.

[6] R. Golding, C. Staelin, T. Sullivan, J. Wilkes. "Tcl cures 98.3% of all known simulation configuration problems" claims astonished researcher! Presented at the *Tcl Workshop*, New Orleans, May 1994. Available from http://www.hpl.hp.com/SSP/papers/

[7] M. E. Gómez and V. Santonja. Self-similarity in I/O workload: analysis and modeling. In *Workshop on Workload Characterization* (held in conjunction with the 31st annual ACM/IEEE International Symposium on Microarchitecture). Dallas, 1998.

[8] Joseph Loyall, Richard E. Schantz, John A. Zinky and David E. Bakken. Specifying and measuring quality of service in distributed object systems. In *Proc. of ISORC'98*, Kyoto, Japan (April 1998).

[9] John K. Ousterhout. *Tcl and the Tk toolkit*. Addison-Wesley, Professional Computing series (April 1994).

[10] Chris Ruemmler and John Wilkes. UNIX disk access patterns. *Proceedings of the Winter'93 USENIX Conference,* pages 405-420 (January 1993).

[11] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *IEEE Computer* **27**(3):17-28, March 1994.

[12] Elizabeth Shriver. A formalization of the attribute mapping problem. HP Laboratories technical report HPL–SSP–95–10 rev. D, (July 1996), available from http://www.hpl.hp.com/SSP/papers/

[13] Elizabeth Shriver, Arif Merchant and John Wilkes. An analytic behavior model for disk drives with readahead caches and request reordering. *Proceedings of SIGMETRICS'98* (Madison, WI, June 1998).

[14] Transaction Processing Performance Council. *TPC benchmarks: standard specifications.* Available from http://www.tpc.org

[15] John Wilkes and Raymie Stata. Specifying data availability in multi-device file systems. Position paper for the *4th ACM-SIGOPS European Workshop* (Bologna, Italy, 3-5 September 1990), published as *Operating Systems Review* **25**(1):56-59, January 1991.

[16] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems* **14**(1):108-136, February 1996.

[17] John Wilkes. The Pantheon storage-system simulator. HP Laboratories technical report HPL–SSP–95–14 (rev. 1, May 1996), available from http://www.hpl.hp.com/SSP/papers/