Application Availability:
An Approach to Measurement

David M. Fishman

SunUP High Availability Program

Office of the CTO

Sun Microsystems Inc.

ABSTRACT

Application Availability is widely sought after as a requirement for applications delivered over networks. While users know what they want—continuous application access with predictable performance—it's often difficult to establish concrete measures that show whether the service providers charged with delivering the application over a network can meet user requirements. Sifting out relevant, actionable data about availability is often as complicated as making decisions and acting based on that information. In order to structure discussion around definitions and appropriateness of measures, this paper sets forth:

- A definition of application availability
- An approach to decomposing applications for measurement
- A classification of service level indicators
- A presentation of measurement as a mode for service level contracting and feedback
- Generalized requirements for developing synthetic transactions

1. INTRODUCTION

Ironically, as the Internet demolishes many of the boundaries between IT and business, the measures that account for their operations are diverging. Traditionally, IT infrastructures, particularly platform resources, have accounted for how much work is getting done by systems using the same metrics as they use for control of resource management—CPU utilization, queries processed per hour, I/O operations, network packets, and the like. However, as systems become more distributed and networked, and as end-users in 24 time zones access systems round the clock—end users want to drive the measures of system availability since it affects their work immediately and directly.

End-users regard the contribution of IT infrastructure in terms of the value that it delivers, not operational metrics. The renaissance of Service Level Agreements (SLAs), once—like many aspects of centralized internet computing—the province of mainframe host-based environments, is driving IT management and end-users alike to seek a common currency to define their

shared objectives, without creating undue operational dependence between their domains.

That common currency is availability measurement. Application Availability Measurement (AAMe) from the end user perspective does not replace resource management, capacity planning, change management, performance analysis, or any of the other many practices that are the métier of the disciplined mission-critical shop. But in establishing and maintaining the value of an application to its users, none of these other disciplines can represent the system as a whole to the users as they see it.

This paper is written for IT and line-of-business executives looking for a way to identify meaningful indicators of application availability. It presents a generalized model for creating measures of application availability in user terms, and validating the application's user value. Specifically, it covers:

- An overview of the classic model for quantifying availability
- An approach for choosing what to measure
- An examination of measurement in the context of other feedback techniques
- A classification of different kinds of measures
- Some considerations in developing measures

There are a number of things I've assumed out of this paper. It does not directly address design of highly available, redundant application or system architectures, for a number of reasons. Any treatment of availability architectures short of book length would not do the subject justice. The measurement approach presented here is intended to be fairly architecture-independent. It's up to the availability architecture to determine how it tolerates, detects and recovers from failures at the many different component levels that make up the stack. I assume only that the application has the necessary mechanisms for doing so. Though the perspective is heavily skewed towards validation, there's plenty that needs to be inferred about the underlying design.

I use the terms "service", "application" and "system" fairly interchangeably; while some might argue for more semantic precision, the three notions are so fluid among themselves that one can argue for each to mean the same as one of the others. But to the extent precision can be applied, I use "system" to represent an end-to-end application environment. A "service" is defined as an application delivered over a network; it is the substrate of measurement for availability. Service-level indicators, or service level indicator metrics, are the result measures from tests that validate the service.

2. MEASURING AVAILABILITY

How Available is Available?

In its classic form, availability is represented as a fraction of total time that a service needs to be up. From a theoretical perspective, it can be quantified as the relationship of failure recovery time (also known as MTTR, mean time to recovery) to the interval between interruptions (MTBF or MTBI, mean time between failures or interruptions). A service

©2000 SUN MICROSYSTEMS INC., All Rights Reserved

that fails once every twenty minutes and takes one minute to recover can be described as having availability of 95%.

For an entire year of uptime—365 days times 24 hours times 60 minutes equaling roughly 525,600 minutes—uptime can be represented as "nines", as in the chart below.

One handy way to think of nines in a 365x24 year is in orders of magnitude: Five nines represents five minutes of downtime; four nines represents about 50 minutes; three nines, 500 minutes, etc. Every tenth of a percentage point per year is roughly 500 minutes of downtime. Of course, for services that don't need to operate 24 hours a day seven days a week, such as factory-floor applications in a single location, the outage minute numbers will vary based on the local operational window.

AVAILABILITY MEASURE	DOWNTIME PER YEAR	DOWNTIME PER WEEK
98%	7.3 days	202.15 minutes
99%	87.6 hours	101.08 minutes
99.5%	43.8 hours	50.54 minutes
99.8%	1,052 minutes	20.22 minutes
99.9%	526 minutes	10.11 minutes
99.95%	4.38 hours	5.05 minutes
99.99%	53 minutes	1.01 minutes
99.999%	5 minutes	6.00 seconds

Figure 1. Table of fractional outages

It should be readily apparent that getting past 1 minute of downtime per week can be quite an expensive proposition. Redundant systems that double the hardware required—in extreme cases, down to specialized fault-tolerant processes that compare instructions at every clock—and complex software that can handle the redundancy are just the beginning. The skills to deal with the complexity and the system's inability to handle change easily drive up the cost. Moreover, experience shows that people and process issues in such environments cause far more downtime than the systems themselves can prevent. Some IT operations executives are fond of saying that the best way to improve availability is to lock the datacenter door.

Be that as it may, any foray into high-availability goal-setting should begin with a careful analysis of how much downtime users can *really* tolerate, and what is the impact of any outage. The "nines" are a tempting target for setting goals; the most common impulse for any casual consumer of these "nines" is to go for a lot of them. Before you succumb to the temptation, bear in mind one thing: you can't decide how much availability you need without first asking "availability of *what?"* The concepts presented here should better prepare you to answer that question; once you've answered it, you can make more constructive use of your downtime target. As your availability goals mature, you'll find it more productive to choose user downtime targets rather than snappy formulations of uptime.

Availability Defined: User Relevance and Measurement Utility

What is the value of application availability? Let's set a definition of availability as *continuous application access with predictable performance.* In daily life, this is fairly intuitive: call your travel agency, and you don't care whether the servers are up or down, whether the network is saturated or not, or whether the client

©2000 SUN MICROSYSTEMS INC., All Rights Reserved

application can validate your credit card data. To you, the only value of the system is in whether the agent can book your ticket or not, or how long it takes. The value of the service—and the service level metric that indicates whether that value is realized—is measured at the end user's nose.

Naturally, to the user, the only measure of availability that matters is at the user—whether the user lives and breathes, or whether the user is some automated consumer of a service. In the online user world, that user's nose is a valuable spot: it represents the point where the application's value is highest—and usually becomes the most useful place to measure application availability. By implication, AAMe increases in value as it more closely approximates user experience. Service level objectives for AAMe must be tightly coupled to the value of the work done with the application.

Heisenberg: Measurement and its Discontents

Can application availability be measured? It's as much a philosophical question as a practical one. Most end-to-end applications are highly complex, dynamic and not deterministic in their behavior; with respect to bits speeding from point to point on the internet, this variation is a feature, not a bug. This also makes it difficult to pinpoint exactly how instrumentation at any given point will provide perfect information about the system's availability. Getting useful (actionable) information is a matter of scoping the end points around which the system may be measured.

As Heisenberg once showed, measurement distorts the measured event or element, making AAMe inherently an imperfect indicator. So at a minimum, in order to realize the value of AAMe, it is necessary that make certain that the measured application has enough capacity—i.e., processing cycles—to sustain measurement. And measures must be selected in such a fashion that their impact on the system is tolerable.

Be that as it may, for any dynamic system, no momentary snapshot constitutes a perfect measure of the application's availability. When the cost of measuring is easier to demonstrate than the benefit, it sets the bar high for any benefits that might accrue. But introducing slightly imperfect measures into a highly imperfect system does not perforce disqualify the act of measurement.

3. WHAT TO MEASURE

Where should an application be measured? Viewed as a service, an application delivered over a network can be understood in logistical terms, at its endpoints. To better understand the endpoints of a service, consider package delivery—the kind of package you can wrap in brown paper and hold in your hand. In the early days of transportation, the term "FOB" (literally, Freight on Board) was coined to describe the accountability for goods at any given point in the journey between seller and buyer. For manufactured goods, "FOB Factory" meant that a purchaser took ownership of the finished product from within the factory, through its transport, i.e., that transport was not the responsibility of the shipper, but of the receiver.

Any end to end service is often composed of subsidiary services. Let's take another simplified example as an analogy: delivery of

©2000 SUN MICROSYSTEMS INC., All Rights Reserved

fresh fish from ocean to dinner plate. The ultimate measure of fresh fish delivery is whether it tastes good when you eat it. But in real life, the logistical problems of fish delivery—specifically, measuring when the fish you'll eat for dinner got from point A to point B on its journey from ocean to plate—is the subject of contract relationships between independent service providers, stated in measurable service level terms. Before you eat, you value knowing when your fish left the ocean for the net, when the fish left the net for the ship's hold, if it was frozen, who cooked it, etc. Delivery from any one point in the chain to another may have multiple owners. At each point that control changes between service providers, be they fishermen, shippers, grocers, or chefs, or waiters, there's an implicit measurement point.

The analogy to service delivery, especially over the public Internet, lies in limitations on control and accountability for certain portions of the service-delivery chain. Can a service provider contract for end-user relevant AAMe metrics over an uncontrolled transport such as a public network? The answer may be no. But in every case, there's a scope boundary, up to which the service provider can take ownership, be measured and held accountable. And any given end to end service can be decomposed into subsidiary services.

This principle, of decomposition into component services, can be applied to most applications, regardless of whether they are internet-enabled. In the diagram below, a stylized "end-to-end" architecture (or "stack") for a web-based application can be decomposed into a set of measurement points for service level indicators. Any user or client of the application performing a transaction depends on all the layers below in order to complete a transaction. In this case, a user or client (i.e., a human or a browser) establishes a connection with a web-server over a network. The webserver connects with the application server, which processes business logic. The business logic in the application server connects to the DBMS for data retrieval as appropriate. And, of course, the DBMS runs on the operating system; it is only as available as the operating environment on which it executes. "Service" availability can be measured or tracked as only a subset of the complete end-to-end stack. With correct design allocating sufficient independence between layers, it's possible to speak of the availability of a series or set of services, each of which is a subset of what the end user requires to be up and running from end to end.

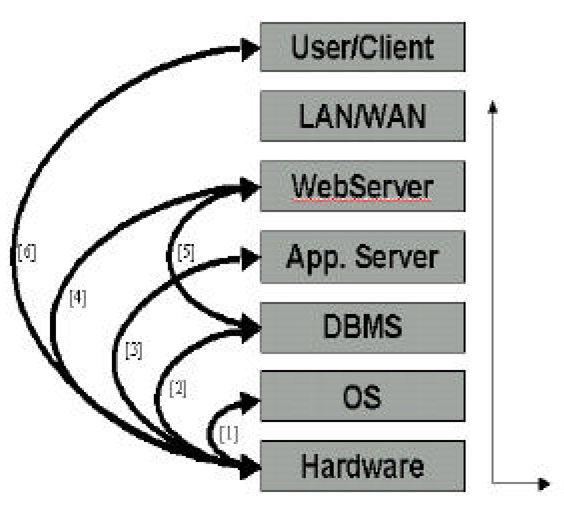

Figure 2. Service Decomposition

(1) Operating System service on hardware, presuming Hardware availability. Most platform vendors who refer to "99.9% uptime" refer only to this.

©2000 SUN MICROSYSTEMS INC., All Rights Reserved

(2) End-to-end Database Service, presuming OS and Hardware availability

(3) Application Service availability, including DBMS, OS and Hardware availability.

(4) Session availability, including all layers below.

(5) Application Server divorced from the database. In this scenario, it's conceivable that the business logic and connectivity to a data store could be measured (and managed) independently of the database component. Note that a combination of (2) and (5) are essentially the same as service (4), in the eyes of the user/client.

(6) A complete, end-to end measure, including the client and the network. While the notion of a "service" used here implies the network, I've included it in this diagram to show that you can establish the measure of availability for the stack as a whole with or without the network.. For internet-based applications, the notion of separating the network is important, because rarely, if ever, can service providers definitively establish and sustain service-levels across the public network. Moreover, when a user connects across the internet, it's important to understand how much of the user experience is colored by the vagaries of the internet, and how much is under the direct control of operational staff.

Decomposition into services takes the first step towards defining what availability is measured, and to what end. As we'll see below, indicating end user availability over time does not require that every service component be measured and tracked separately.

Comparison of Feedback Techniques

Measurement, monitoring, and management are three distinct feedback techniques, ways of taking data gathered about a system and applying it in changes made to the system. The industry and the market often confuse these three concepts. In fact, the differences between the three can provide a useful tool for choosing what data are useful in tracking availability for a service, or, in our case, a networked application.

Feedback Technique	Reporting Frequency	Intervention Frequency	Certainty /Scope
Measurement	periodic (cumulative)	periodic	low /loose
Monitoring	real-time	periodic	medium /limited
Management	real-time	real-time	high /well scoped

Figure 3. Feedback Mechanisms

In simplest terms, a feedback loop is composed of *reporting* and *intervention*, undertaken at certain intervals. Reporting frequency characterizes event sampling—how often, and how immediately are events known? Intervention frequency characterizes action taken based on this event data—how often do you draw conclusions from the data, and how immediately can you intervene to make changes to the system based on your conclusions?

- Measurement is periodic sampling with periodic intervention
- Monitoring is real-time sampling with periodic intervention

©2000 SUN MICROSYSTEMS INC., All Rights Reserved

- Management is real-time sampling with real-time intervention

One question to consider here is the value of information. Consider, by analogy, the relationship between bookkeeping, accounting, and audit of an organization's financial operations. Certainly, many of the same tools and techniques apply at all three levels. One key difference is in the audience: rarely does the CEO community require detailed, day-to-day information on smaller aspects; they seek a high-level indicator, such as profitability or cash flow, to give her the audit information she needs about the health of the business, and draw broad strategic conclusions. Yet the operations staff needs to understand intimately how individual data impact profitability in order to make that higher-level measure meaningful. Now, let's see how this notion applies in the services context.

4. CLASSIFYING SERVICE LEVEL METRICS

The best AAMe indicators track real work by real users as closely as possible. Most dot-coms and datacenters have service-level objectives of one sort or another that characterize system behavior; some formally quantify these objectives, either for internal management and alerting or as part of formal SLAs. Such objectives take the form of uptime of a database, correlated output of system management tools, delivered bandwidth and data streams, and a variety of levels in between. But which of these, if any, are useful in measuring application availability?

To this point, we've considered availability as an attribute of a service, as in "is it available?" In fact, availability is itself really a given service level, with quantifiable, measurable levels of attainment. A formal definition could take the form:

Availability:

a measure that checks the behavior of a system, using consistent tests repeated at set frequency over time, comparing accumulated test results with a goal.

Such a measure would be expressed using the formulation "test every sixty seconds, with a maximum of 50 test failures per week", to indicate 99.5% uptime. But taken individually, it can be difficult to translate these result measures into positive indicators of service availability.

Goal of Service Level Metric Characterization

Using a common framework can help service providers and service consumers better match how they work together over time on availability goals for their critical applications. Moreover, in a networked, distributed application model, certain subsystems (security, HTTP, application server, DBMS, etc.) can be characterized independently, as components of an end-to-end service level metric, in a way that supports good information about managing to service levels.

To better characterize what makes a useful service-level metric, we have formulated a simplifying hierarchy for ranking levels of application availability and service level metrics. The model, called SLIMTAX—for Service Level Indicator Metric Taxonomy—classifies metrics on how they test and what they test for. An important benefit is that as an organization's service-level tracking

©2000 SUN MICROSYSTEMS INC., All Rights Reserved

capability matures, SLIMTAX provides a roadmap for shifting towards application availability measurement up the hierarchy to levels that represent throughput and user work.

SLIMTAX Definitions

The SLIMTAX hierarchy incrementally adds features of an application service from the bare minimum of application existence, up through network delivery, service level thresholds, and complete user-centric status measurement.

- *A0: Key Process(es) Exists locally.*
  For example, a list of processes shows HTTP is running. Some applications have multiple processes, so just looking for those that show an application is up may not be enough.
- *A1: Local State.*
  Key process or processes can work to process inputs locally and produce correct output. For example, this is how a cluster tests the availability of the applications it is hosting. At this level, the test is local; in some instances, it is possible to derive availability from such a passive measure, such as the scanning of log files.
- *A2: Remote Session.*
  User can establish access (log in) over a network. Here, the notion of a synthetic transaction is introduced, though it need only run at intervals shorter than target failover times, in order to expose any failures. For example, an application that fails over in 15 minutes can show its availability in an A2 test that runs every 10 minutes, as the 15 minute failover will register as an outage.
- *A3: Transaction Response Time.*
  Key business operations are performing at a given rate. Here, a service level threshold or objective is used to measure whether a sufficient fraction of key transactions completes quickly enough; for example, 99.9% of the monitored transactions complete in 8 seconds or less.
- *A4: User Work.*
  Key population of users or clients are performing given units of user work over time. Such an indicator would account for 200 active users, sending and receiving an average of 30 emails each per hour, showing that an email system delivers 12,000 messages per hour with a given population. This measure can be based on session logs, or based on instrumented clients with a closed-loop that captures a user-centric picture of the end-to-end application.

SLIMTAX Features

It's important to understand that the SLIMTAX hierarchy doesn’t measure availability; it provides a "meta-metric", i.e., a framework for comparing metrics. In service level contracting, it can establish differences in requirements between end users and service providers, internal or external. Similarly, in architecting a systems and tools environment, SLIMTAX makes it possible to identify where feedback techniques for availability are applied, and to distinguish one feedback technique from another.

One kind of availability metric that shows operating system availability is a level A0 metric (at best) because it doesn't test if OS is doing something, it just checks if it's there. Similarly, many systems management tools check application availability just by looking

©2000 SUN MICROSYSTEMS INC., All Rights Reserved

for whether a certain process exists. Another example: in order to determine the failure of a subsystem, such as a node within a cluster, and take appropriate remedial action, an A0 or A1 measure may be adequate; test failures may be sufficiently well-defined to trigger automated recovery. But for other service level consumers, the fact that a cluster is "available" at the A0 or A1 level will be by definition understood not to encompass user-level metrics.

The SLIMTAX hierarchy also helps scope demarcation for bottom-up, top-down availability feedback techniques. In other words, measurement at the A0 level maps to interventions that are feasible at the local host level. Moreover, to the extent that the application stack includes redundancy built in to mask system failures from the users—as in a redundant server node running an application in standby mode—the metric can show the same mask underlying system failures that are not immediately visible to the user. With respect to user impact and user-experienced availability, the lower ranks are less informative, and higher ranks subsume lower ranks.

A corollary of the subsystem/supersystem demarcation is that availability metrics are inherently not comparable across levels. In other words, 98.2% availability at A1 cannot be compared with 99.2% availability at A2; the A1 level may not cover redundancies that mask failures at the next level up.

Moving up and down the hierarchy

In considering the range of different availability metrics described by the SLIMTAX hierarchy, it's worth noting the additional information that is added in moving from one level to another:

- *From A0 to A1*, a metric adds information showing that key processes in the application can accept inputs and do work.
- *From A1 to A2,* a metric adds information demonstrating the existence of a service, rising to meet the definition of a service as an application delivered over a network. Note that because the network is inherent from this level and above, there's no need to measure network availability alone, independent of application traffic making its way back and forth from the application. Performance is measured only with respect to failover time.
- *From A2 to A3,* a metric introduces a service level performance threshold, showing the performance of an application in terms of work executed over time (e.g., transactions per hour, seconds per transaction, etc.). Moreover, while the previous levels of the hierarchy are binary (was the system up or down?), an A3 metric allow for the possibility of performance degradation, as in a fraction of the system service level threshold. For example, a system targeting a performance level of 300 transactions per minute completed may operate below this target and still get useful work done; i.e., slow does not mean unavailable. This implies additional service level thresholds, such as a system that can do 85% its target rate. Contrast this with A2 synthetic transactions can show whether a system has experienced an outage.
- *From A3 to A4,* a metric adds user populations as the necessary complement to its response time requirements. For most systems, there's a material difference

©2000 SUN MICROSYSTEMS INC., All Rights Reserved

between 1 user driving 1000 transactions each minute, and 100 users driving 10 transactions each minute. By translating system work directly into user impact, the A4 metric provides the most complete indication of the impact of availability on the consumers of a service. Conversely, when a system's utilization drops with dips in a user population, the impact of downtime is adjusted appropriately.

In a perfectly instrumented system, the A4 metric would be enabled in closed loop fashion, so that administrators and end users would know exactly what throughput the system was achieving at any given time, in terms of user work. For example, administrators could log browser error messages on user workstations or PCs. For systems that are not perfectly instrumented, A2 and A3 level synthetic transactions provide the most representative picture of how much work the system is doing. However, for many applications, particularly those with named users, it's possible to establish exactly how many users are on the system at any given moment and derive an A4 metric by observing key transactions and their response time.

At the A0-A2 level, most indicators do not provide positive indication of system availability, though they can show when work *is* being done. The lack of transactions does not mean that the system is down; it may mean that no users are performing transactions, or that the entire population of users is on a lunch break, etc. But it may be possible to correlate a set of passive data measures into a record of how much work is being done at any one time. Log data can show how many transactions were completed over a particular period, but this is an incomplete positive indicator of application availability (i.e., a lack of logged operations does not mean the system was down).

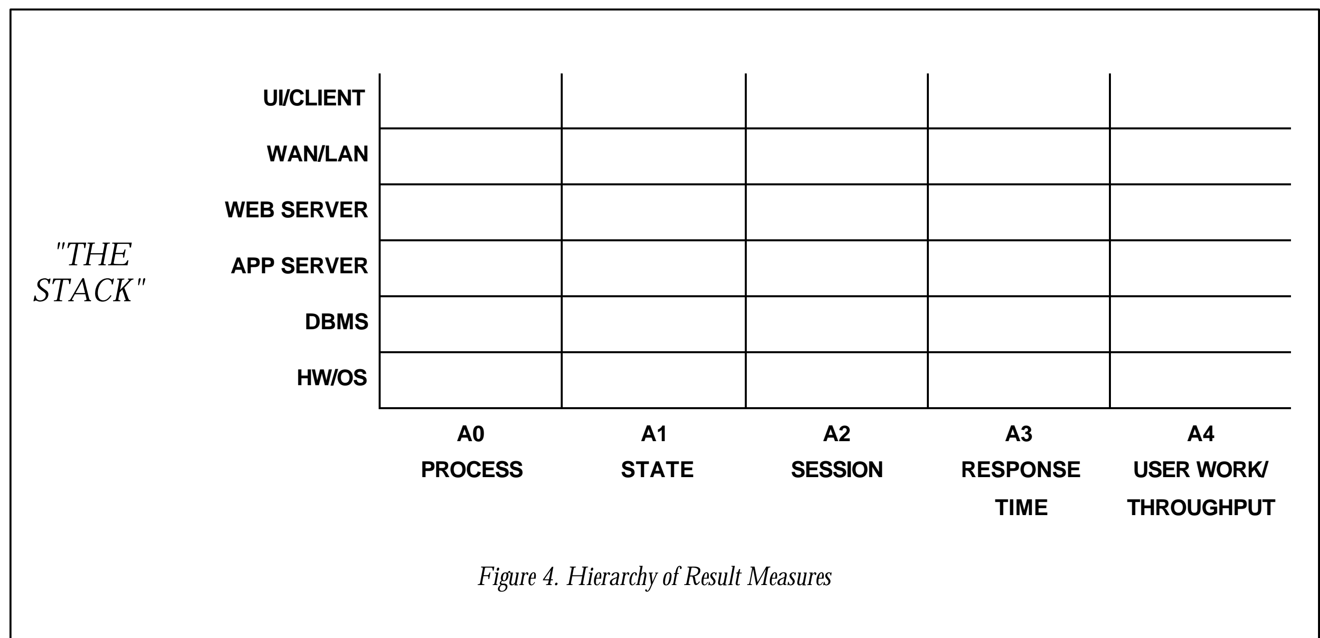

Figure 4. Hierarchy of Result Measures

©2000 SUN MICROSYSTEMS INC., All Rights Reserved

Applying SLIMTAX to application stack

Earlier, I described how an application may be decomposed into a stack of end-to-end services, or a "stack". It's a straightforward matter to conceive of tests that indicate service availability at a variety of points up and down the stack—either for the stack as a whole or for parts of the stack.

SLIMTAX can be represented as a simple matrix representing the service components of the stack along the vertical axis, and the hierarchy of result measures (A0-A4) along the horizontal axis, as shown above (Figure 4).

What the chart represents is a roadmap from server-centric, infrastructure-focused measures of uptime, at the lower left hand corner of the matrix, towards user-oriented measures, which are represented up and to the right. Most measures of application availability can be mapped into this hierarchy based on how they test for service availability (A0 through A4) and where in the stack they test for it.

There's often a temptation among operational groups just beginning to take on the challenge of user-oriented availability measures to "go for broke"—to focus only on measuring end-to-end service-level availability because "that's what users care about." It's important to recognize that it's often difficult to establish these kinds of measures at one stroke. Targeting a metric that represents only part of the stack or that only measures a certain degree of user activity offers solid incremental progress. It's also useful to move up the stack within a single result-measure class; for example, if you have a test and a metric that show A2-level database session availability over the network, a good next step would add A2 application server availability measurement over the network. It's not necessary to move directly from A2 to A3 in order to provide a more end-to-end measure.

Moving a test further up the application stack provides more information to the service consumer; ironically, it provides less actionable information to the service provider. For example, an end-to-end test might:

(1) begin at a browser;

(2) traverse the network;

(3) connect to a web server;

(4) submit data and transact business logic in the application server;

(5) retrieve data and insert a record in the database;

(6) perform read and write operations through an I/O device into a disk;

(7) return the correct result back through the stack to the user in under 15 seconds

Depending on the response time criterion for the test, it measures the service at either an A2 or and A3 level. If the test passes, the service is available; if it fails, the service is not. Designing this test correctly requires some analysis the application architecture to ensure that data retrieved from the database, for example, is not cached on the application server, to avoid masking a database failure.

SLIMTAX: Measurement vs. Management

The test transaction in the example above provides an interesting comparison between measurement data and management data. For measurement, this test case has two possible outcomes: pass or fail.

Contrast this with the management perspective. A diagnostic view of this test

©2000 SUN MICROSYSTEMS INC., All Rights Reserved

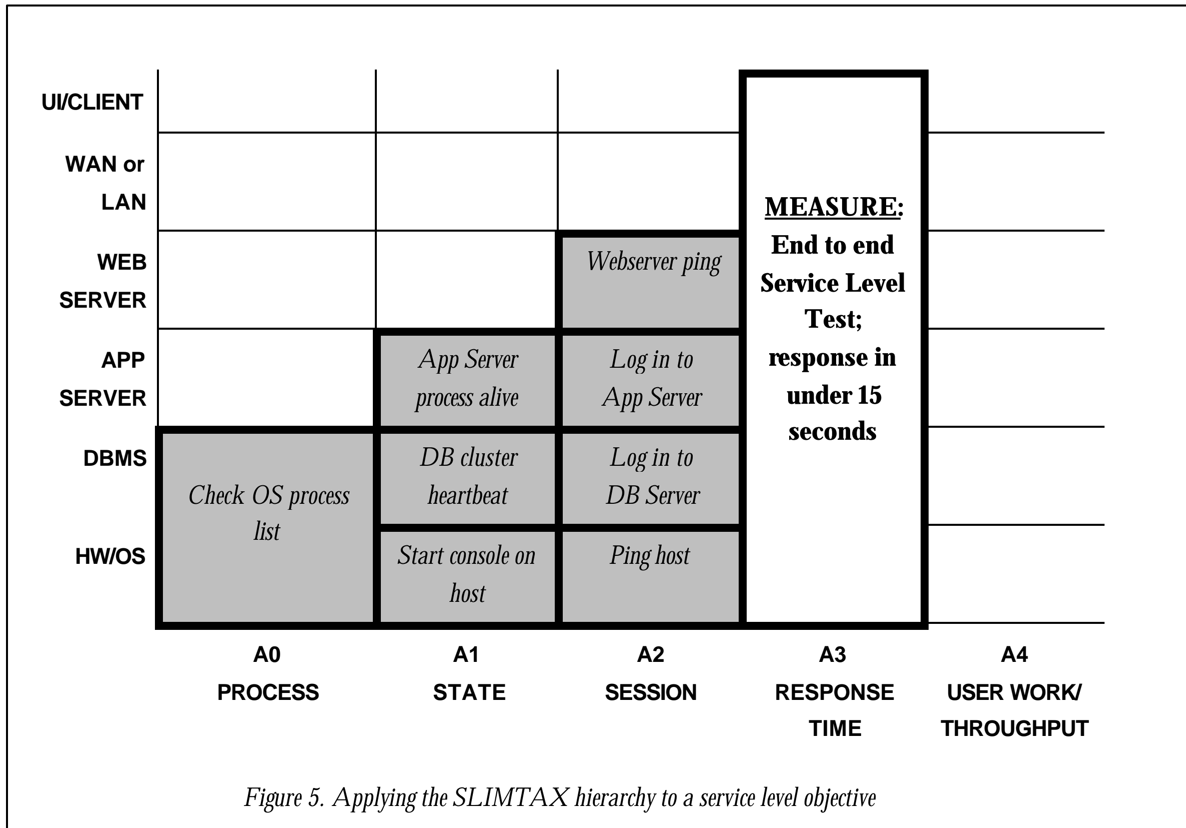

Figure 5. Applying the SLIMTAX hierarchy to a service level objective

would take into consideration results at each step of the way. For an administrator, having such data available can help deal with an outage, either with some policy-based automated recovery mechanism or as after-the fact diagnostic. Or, to make certain that she could deliver on the test result above, the administrator could complement the end-to-end test above with a set of A0-A2 tests, either in real time as an automated recovery mechanism, or as a set of diagnostics in dealing with an outage. It's up to the administrator and the application architect to make a number of choices:

- what tests short of the A3 end-to-end level are required in order to diagnose service outages and act upon service outages
- whether she needs to monitor them at all times, or only when there's a breach of the end-to-end service levels.
- What indicators, in combination, provide positive proof of the availability of the components of the service level.

This service level test, and its complementary diagnostic or management indicators, is mapped onto the SLIMTAX in Figure 5.

Note that the diagnostic examples set also shows the possibilities for applying the principle of service decomposition to create

©2000 SUN MICROSYSTEMS INC., All Rights Reserved

service-level targets within the application stack that capture its constituent parts. The database cluster heartbeat is a good example of management information that can be inferred from the system architecture, as a cluster is a local service level unto itself. While it may be part of a management mechanism with automated policies, it's not difficult to extract information about whether those policies are working and report it as part of the availability measurement effort.

5. TEST FREQUENCY, TIMING AND PERFORMANCE IN SLIMTAX

Sampling frequency

Now that we've seen how to test once for service level availability at different points in the stack, let's add a final dimension to the indicator: frequency of sampling. How do you know how frequently to run a test against your stack? It's only at this point that it is appropriate to consider your target downtime requirements, which I first mentioned at the outset of this paper.

Failure, interruption, recovery time, outage, timeout — all these terms represent the interval during which the application is not available. Since one goal of AAMe is to represent how long an application is up, a good test would be able to identify when the application stack is *not* available. For instance, given an architecture designed to recover from an outage in twenty minutes, there's an implied uptime goal for the service to keep interruptions to twenty minutes. Consequently the service level measure should test more than once every ten minutes to help make certain it never misses an outage, such as once every 2 minutes. This is a good place to apply an A2 metric; so long as the test can establish remote session into the monitored system every two minutes, then the application is up. More than 10 successive failures of the test is a good indicator that the application is missing its recovery-time target.

Outage duration is not the only consideration in deciding how often to test if the system is up. Take a service that's down four times in one hour, 3 minutes at a time, but once every 15 minutes. If a user tries to get onto the system, but logs in at exactly the same 15-minute intervals as the outage, she may well perceive that the system has been down for an hour. For users, frequency of outages matter as much as (or more than) their duration.

Note, again, the difference between management and measurement. From the system management perspective, even one failed A2 test should trigger an intervention, manual or automated. Measurement assumes that those responsible for management will try to do the right things and checks whether they have succeeded or not.

When performance is an availability issue

Traditionally, systems have been sized and configured based on workloads that account for performance, such as a particular throughput whenever the system is up. In other words, sizing for 3000 transactions per hour assumes that for 10,000 hours operation, the system will complete 30 million transactions. Naturally, there's a difference between 10,000 hours of systems operations, and 10,000 hours of the business being open. Each whole percentage point of downtime -- i.e., the difference between 98.5% and 99.5% availability -- amounts to 300,000 transactions.

©2000 SUN MICROSYSTEMS INC., All Rights Reserved

When this system comes up short 30 transactions per hour, it can miss its operational uptime targets.

Viewed this way, distinctions between measurement of performance and availability blur somewhat. However, erasing this distinction clashes with some fairly well established measures of system behavior. The challenge is in identifying given measures of availability and performance that can provide an indicator of overall system work.

SLIMTAX accounts for this in the distinction between A2 and A3 metrics, as I described above. Implicit in this distinction is a time-out; if an A2 session test cannot be established within the time that the application was to recover from a failure, it's a safe bet that there is a failure. For transactional environments, performance targets for transaction completion time are typically denominated in seconds; recovery from failure is typically slower, denominated in minutes. (For batch jobs and other long running transactions, completion time may be significantly longer. In these cases, other indicators such as work rates, rows processed, tables backed up, and the like, may be more appropriate.)

How does one account for slow performance as an availability problem? Since it's difficult to tell from a single transaction that missed its target speed whether there's a general performance problem, the system needs a performance profile that targets a performance threshold. At the A3 level, such a threshold could be characterized as "a minimum of 9000 transactions per hour, with an average rate of 15 seconds per transaction." One way to determine whether the system hits its performance target is to sample completed transaction logs retroactively, and inspect to see whether they completed on time. Given the sensitivity of most administrators to performance issues, they typically attend to such metrics much more closely. Moreover, it provides excellent evidence of their success or failure.

One important aspect of performance as availability is measurement of degraded operations. This can be applied in several ways. It's healthy for administrators to report to their users that the system is experiencing a "traffic jam" even if the cause has yet to be determined. In parallel, it may be possible to specify a target limitation for degraded operations; as in "98% of all synthetic transactions must complete within 15 seconds; no more than 5% must complete in more than 15 seconds, but less than 20 seconds."

But inspecting transactions locally doesn't show whether end users were driving transactions successfully. A more complete measure of application availability would drive a "synthetic transaction"—a set of user-level operations, scripted and driven by an automated tool that captures the result of any transaction, including elapsed time. Designed correctly—and this can often be done simply—such a synthetic transaction can show when a system is performing at the required level without actually creating a great deal of overhead on the system. A few well-placed synthetic transaction users around the network will reveal a great deal about the end-to-end performance of the system as a whole. Using synthetic transactions, outages can be declared when the system is performing at less than its target performance rate for a specified period of time. This service level objective

©2000 SUN MICROSYSTEMS INC., All Rights Reserved

would take the form, "98% of all synthetic transactions must complete within 15 seconds." Most website and internet monitoring today takes this form, using synthetic transactions to drive traffic at a website to emulate end user experience.

The most complete measure of performance as availability accounts for users on the system as well as the rate of their work as it proceeds through the system, designated by SLIMTAX as an A4 metric. Some infrastructures lend themselves to complete instrumentation, so administrators know in real time exactly how many concurrent users are doing how much work on the system. It's possible to infer this by inspection, either live or retroactively, to look at active, concurrent users (those who submit input at least once every 15 minutes, for example), and compare that to the total number of transactions, or correlate it with the rate of synthetic transactions.

6. METRICS AND TOOLS

Distinguishing Architecture and Operations from AAMe and Service Levels

Too often, service-providers—classically IT operations departments, help desks, and as they emerge, independent Application Service Providers and other "xSPs"—burden service consumers with management and monitoring parameters internal to the service. So long as the service provider meets set objectives, the consumer of the service need know nothing about how the provider manages or monitors the delivery of service. By implication, any corrective action undertaken to address missed service level targets is primarily the responsibility of the service provider.

Because service providers and IT operational staff are accustomed to viewing their infrastructure through the lens of management, such management tools have emerged as the preferred technique for measuring system and application availability. This tendency is more the product of tactical operational considerations—i.e., given a problem, find an action to be taken—than it is in tracking whether a system meets its desired availability goals. Knowing the behavior of one element of a system—such as hardware MTBF, or disk utilization, or network traffic—generally won't represent the behavior of an application and operating system software, nor will it expose dependencies between those layers.

The flaw at the heart of this "management fallacy" is not that such management tools are not useful. Rather, it is that they do not adequately represent whether user work is being undertaken and completed—in other words, does the end to end system provide continuous application access with predictable performance? For most networked application environments, the most useful technique to apply to begin to answer the question is measurement.

While it may seem obvious, service providers need to realize that exposure of management and monitoring information internal to a service is unnecessary so long as service consumers are not directly involved at the operational level. Any consumer of a service is more concerned with the attainment of service level targets than the underlying implementation. The challenge is to understand which metrics provide the necessary information to the service consumer, and to select measures that over

©2000 SUN MICROSYSTEMS INC., All Rights Reserved

time can provide information that keeps both sides focused on service level attainment.

Of course, the administrator is not alone in deciding how to manage the components of the service; it's up to the systems architect to account for service level and manageability architecture from the inception. In an ideal world, the two communicate; in reality, it's often up to the administrator to make inferences about the architecture in choosing what management information is most useful.

One implicit theme in the discussion of measurement so far is the virtue of simplicity. The primary question we've dealt with is not how to measure, but what to measure. In this final section, we'll explore different approaches one might take in taking specific measurements of a service.

Defining A Service-Level Indicator

As I mentioned earlier, measuring a service level requires the following:

- a test performed at regular intervals
- a tool to perform the test
- a goal for the results of the individual test (e.g., up/down, speed threshold)
- a goal for test results over time (e.g., 99.8% completion)
- a tracking mechanism for collecting and comparing results
- A presentation mechanism for showing results over time

The practice of testing for service levels has many elements in common with the broader discipline of software and system testing. (In fact, several test tool vendors are entering the market for service level monitoring, since the technology for service level measurement and monitoring is very similar to that used for test automation.) This is particularly true of test design, since it is important to apply many of the same architectural analysis skills. For example, when a test transaction selects a record from a database, which tables does it select from? Are these tables the ones most likely to show that the database is having a problem? However, testing for service levels need not be as complicated as regression testing, stress testing, or system envelope testing.

For simple applications with web interfaces, an automated test can use either a perl script or a servlet that logs its results to a flat file, spreadsheet, or database. Creating an A2 metric for database availability, for example, could be implemented by embedding some JDBC™-enabled calls into a perl script, and setting a flag, based on correctness of returned output. Of course, it still requires programming, maintenance, and attention to results over time to make sure that the test itself isn't broken. It's also possible to create synthetic transactions with formal automated testing tools. Such tools typically have more than capture/replay capabilities, adding language facilities to create logic in automated test scripts that can handle exception conditions, deal with conditional inputs and outputs, and log data in a repository.

To facilitate data collection and analysis over time, it's generally also useful to store outputs in a data store (or spreadsheet, at a minimum). This data store should enable you to easily retrieve and present trend information over time.

One practice that many operations personnel find useful is to post a "dashboard" that shows the state of key services on the

©2000 SUN MICROSYSTEMS INC., All Rights Reserved

organization's intranet. This allows any single user who needs a service to "check the traffic" when using an application. Several of the automated service-level monitoring tools in the market have this facility built-in. A good complementary practice for a web-based dashboard is to provide links to service-level definitions, as well as prose descriptions of service level targets, so that a user who sees a service-level trouble indicator can also check what the service level covers.

Defining a Synthetic Transaction

I've referred a number of times in the course of this paper to "synthetic transactions" as a means for checking the availability of an application across a network. By definition, a synthetic transaction constitutes an automated, self-contained set of user-operations that can be executed against service much as would be undertaken by any consumer of the service, such as a real user operating an application. For example, on the internet, a synthetic transaction includes a script that goes a public website for trading stocks, and tests looking up a stock price or portfolio valuation, and reporting whether the series of operations completed successfully. In other words, any fixed set of user operations that can be automated reliably can be defined as a synthetic transactions.

Requirements for a synthetic transaction, at a minimum, should address the following:

1. *Service Scope*. What are the boundaries of service level tested by the synthetic transaction? For example, does it include or exclude the local area network? Are there redundancies that it masks?

2. *Geographic Scope.* From which point(s) on the network should the synthetic transaction execute from? Executing a single synthetic transaction within the four walls of data center at the same time as the same transaction runs across the corporate network (or the internet, for that matter) can help establish the relative availability of the network compared to the service. Running the same transaction independently from multiple locations in the corporate network provides significant diagnostic information through triangulation, and comparison of results over time. This also represents a significant opportunity to add operational diagnostic capability; or, alternatively, to define a service level objective that excludes the outer levels of the network.

3. *Functional coverage*. This is the core of any test: what subsystems and functionality are exercised by the operations performed during the test? What subsidiary components of the service stack does the synthetic transaction exercise? For example, if a sample transaction includes data input and retrieval, does that data input cause the application server to do a select from the database, demonstrating its availability, or does it just retrieved cached information from a file server?

   Functional coverage analysis requires close collaboration between administrators and the system architects, but with a strong bias to end-user orientation. One good test of the functional coverage is to ask an average user how he or she knows the system is down, and see if he or she can readily perform the exact same transaction manually. It's also important to resist the temptation to create synthetic transactions that provide a wealth of

©2000 SUN MICROSYSTEMS INC., All Rights Reserved

diagnostic information; complexities introduced in the pursuit of optimization and troubleshooting data can make tests less robust and generate false negative results. Focus should be on a test that closely reflects what users do in spite of the possibilities for optimization

4. *Randomized think-time.* Users don't typically fire off inputs into an application as fast as they see outputs; typically, they pause between entries, either to think, or for interruptions or cups of coffee. Within reason, the synthetic transaction should be able to vary how much time passes between key inputs, to better simulate how users interact with the system.

5. *User operation demarcation (start/end).* Like any good test, a synthetic transaction needs to begin at a known system state, and end at a known state. Again, the focus of the test is on completion, not diagnosis.

6. *Response timeout.* When there is an outage, what is the service's recovery target? This applies at two levels. First, for each step of the test, how long should it wait to see the system respond to a single input? Second, for the synthetic transaction as a whole, the allotted completion time should also be a function of its SLIMTAX classification. A2 metrics can sustain longer response times, since it needs only to establish whether a service is there, not how quickly it responds.

   By contrast, A3 and A4 metrics should time out quickly, subject to the performance targets of the system. Again, brevity in a synthetic transaction is a virtue; many short transactions serve the purpose of measurement better than a few long-running ones. Again, the higher the metric rises in the SLIMTAX hierarchy, the more frequently tests need to execute.

7. *Operational windows.* When does the service under test have to be available? A simplified way to account for operational windows is to manually review availability data accumulated by the synthetic transaction, and consider only those outages that took place within operational constraints. Alternatively, the synthetic transaction can be programmed to consult a table of service parameters and ignore outages at certain times

8. *Service level goal: % success.* Appealing as it may seem to set goals in terms of percentage of 100% 24x7 uptime, both users and administrators will find it more constructive to work against outage budgets, denominated in minutes. Outage budgets can also be allocated to root causes following analysis of service-level attainment; network, application, and operating platforms can each be allocated a certain fraction of the outage budget, and managed to meet those targets independently.

©2000 SUN MICROSYSTEMS INC., All Rights Reserved

Acknowledgements:
Much of the explication in this paper is my own thinking, but underlying it are some strong ideas from many of the people I've been privileged to work with as I developed this exploration of the subject. I am indebted to Richard McDougall for his insights on the relationship of resource management to availability measurement, and to Michael Treese for both his ideas on the hierarchy of measures and his editorial skills. Thanks also to John Bongiovanni, Amir Raz, Farhad Shafa, Rob Sibley, and Jim Wright for their support and contributions.

About the Author:
David M. Fishman works in the SunUP™ High Availability Program under the Office of the CTO at Sun Microsystems, where he is responsible for application availability measurement strategies. Prior to that, he managed Sun's strategic technology relationship with Oracle, driving technology alignment on HA, Enterprise JavaBeans™ (EJB), scalability and performance. Before joining Sun in 1996, David held a variety technical and business development positions at Mercury Interactive Corporation, a software test automation tools company. There, he led product management efforts for automated GUI testing tools and load testing for packaged ERP implementations. As Mercury's business development manager, he helped drive its 1993 IPO. From 1988-1991, David worked at a VME board manufacturer in the defense electronics industry. He holds an MBA from the School of Management at Yale University. *Email: david.m.fishman@sun.com.*

Sun, Sun Microsystems, the Sun logo, Enterprise JavaBeans, JDBC are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

©2000 SUN MICROSYSTEMS INC., All Rights Reserved