Software Engineering Institute

Characterizing Technical Software Performance Within System of Systems Acquisitions: A Step-Wise Methodology

Bryce L. Meyer
James T. Wessel

April 2010

TECHNICAL REPORT
CMU/SEI-2010-TR-007
ESC-TR-2010-007

Acquisition Support Program

Carnegie Mellon

Table of Contents

List of Figures

List of Tables

Acknowledgments

The authors thank Ceci Albert, Stephen Blanchette, and Lisa Marino of the Carnegie Mellon® Software Engineering Institute (SEI) for their guidance and review of early drafts of this report. Their comments greatly improved the final product.

® Carnegie Mellon is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

Abstract

The characterization of software performance (SWP) in complex, service-oriented architecture (SOA)-based system of systems (SoS) environments is an emergent study area. This report focuses on both qualitative and quantitative ways of determining the current state of SWP in terms of both test coverage (what has been tested) and confidence (degree of testing) for SOA-based SoS environments. Practical tools and methodologies are offered to aid technical and programmatic managers:

- a stepwise methodology toward SWP selection
- SWP and system architecture design considerations
- resource limiters of SWP
- SWP and test event design considerations
- organizational and process suggestions toward improved SWP management
- a matrix of measures including test fidelity and realism levels

These tools are not complete, but do offer a good starting point with the intent to encourage contributions to this growing body of knowledge.

This report is intended to benefit leaders within the varied acquisition communities, Program Executive Offices, and Program Management Offices. It provides detailed guidance for use by technical leadership as well.

1 Overview

The SEI, through its Acquisition Support Program, facilitated holistic improvement in a complex service-orientated architecture(SOA)-based system of systems (SoS). The improvement took both a technical and programmatic perspective, in the design and use of software performance (SWP) tools and methodologies, which enabled the vital association of hardware to software performance. Impacts include improved:

- programmatic and technical visibility into software performance earlier in the development life cycle, providing “actionable intelligence”
- understanding of the ability of software deployed on diverse networked systems to meet software quality characteristics
- traceability of desired program capabilities
- coordination and synchronization of organizational software performance asset investments
- identification of consequential program issues and risks

The characterization of SWP in complex, SOA-based SoS environments is an emergent study area. Based on experiences with a real world Department of Defense (DoD) SOA SoS acquisition environment, along with industry experience and academic research, this report documents the Software Engineering Institute’s ongoing investigation of these questions:

- Will the performance of fielded software enable envisioned capabilities, in an end-to-end user environment?
- What performance data should technical and program managers use as “actionable intelligence” to aid decision making?
- What organizational structures and resources need to support SWP management?

We include discussions of a recommended step methodology for choosing SWP measurements, SWP and software system architecture design, SWP resource limiters, SWP and test event design, and continuous SWP process improvement.

We also detail impacts for acquisition programs, including improved

- programmatic and technical visibility into software performance earlier in the development life cycle, providing “actionable intelligence”
- understanding of the ability of software deployed on diverse networked systems to meet software quality characteristics
- traceability of desired program capabilities
- coordination and synchronization of organizational software performance asset investments
- identification of consequential program issues and risks

We focus on both qualitative and quantitative ways of determining the current state of SWP in terms of both test coverage (what has been tested) and confidence (degree of testing) for complex

SOA-based SoS)environments. We offer practical tools and methodologies to aid technical and programmatic managers:

- a stepwise methodology toward SWP selection
- SWP and system architecture design considerations
- resource limiters of SWP
- SWP and test event design considerations
- organizational and process suggestions toward improved SWP management
- a matrix of measures including test fidelity and realism levels

These tools are not complete, but do offer a good starting point with the intent to encourage contributions to this growing body of knowledge.

A Scenario

We present our methodology in the context of a blade environment, although the general concepts should apply across hardware instantiations. The following scenario provides a context of SOA SoS usage for ensuing SWP discussions.

Networked SoS projects link multiple hardware platforms (e.g., a truck or plane) to perform some defined activity together, where each platform consists of a system of processing units linked by one or several communication networks (e.g., local area network, radio). These systems use a SOA in which software elements provide "offers" of functionality to a "discovery engine." Each processor, or blade, in the processing unit runs a common middleware that encapsulates the operating system for each processor.

A coarse and simplified analogy is a series of websites on the public Internet; each one offering information or function (i.e., a service); each providing an interface, with a standard format with Internet Protocol (IP); and discovered using a search engine's discovery feature. The search engine is hosted in a distributed fashion on every website server to facilitate system performance e.g., swifter discovery and system design attributes (e.g., fault tolerance). The processors of each website are varied in type, count, and power. The websites are developed and hosted by a diversity of developers from varied organizations.

A user enters the keywords into the search engine, located on the nearest server, which provides a graphical user interface (GUI) that allows the user to link to the websites. The more users and more websites, the harder the search engine has to work' the more used the connections to the websites become, the more effort the websites' servers must exert to provide the offered service.

In this scenario, you seek measures to determine SWP for each website, excluding the network[1] metrics themselves:

- How would you decide what measures would be required?
- What kinds of testing are needed to elucidate the performance of the service's software and underlying structure?

[1] For this report, sample metrics of 'network aspects are provided only as examples, such as router/FW/radio. Delving deeply into network metrics could encompass the better part of many books.

- How do you know the state of SWP testing, and what gaps remain?

Typically, diverse test events in varied stages exist, analogous to exploring a cavern of unknown size using intersecting flashlights (see Figure 1). Only where the flashlight beams intersect do we have enough illumination to know the true character of a portion of the cavern. Replace the flashlights with test result documentation, and the analogy holds for quantifying the software performance of a large, complex SOA SoS.

This report provides a method to better plan metrics use toward illuminating software performance for an objective SOA SoS. Included is a set of initial SWP metric categories, along with a method to assess the current state of testing.

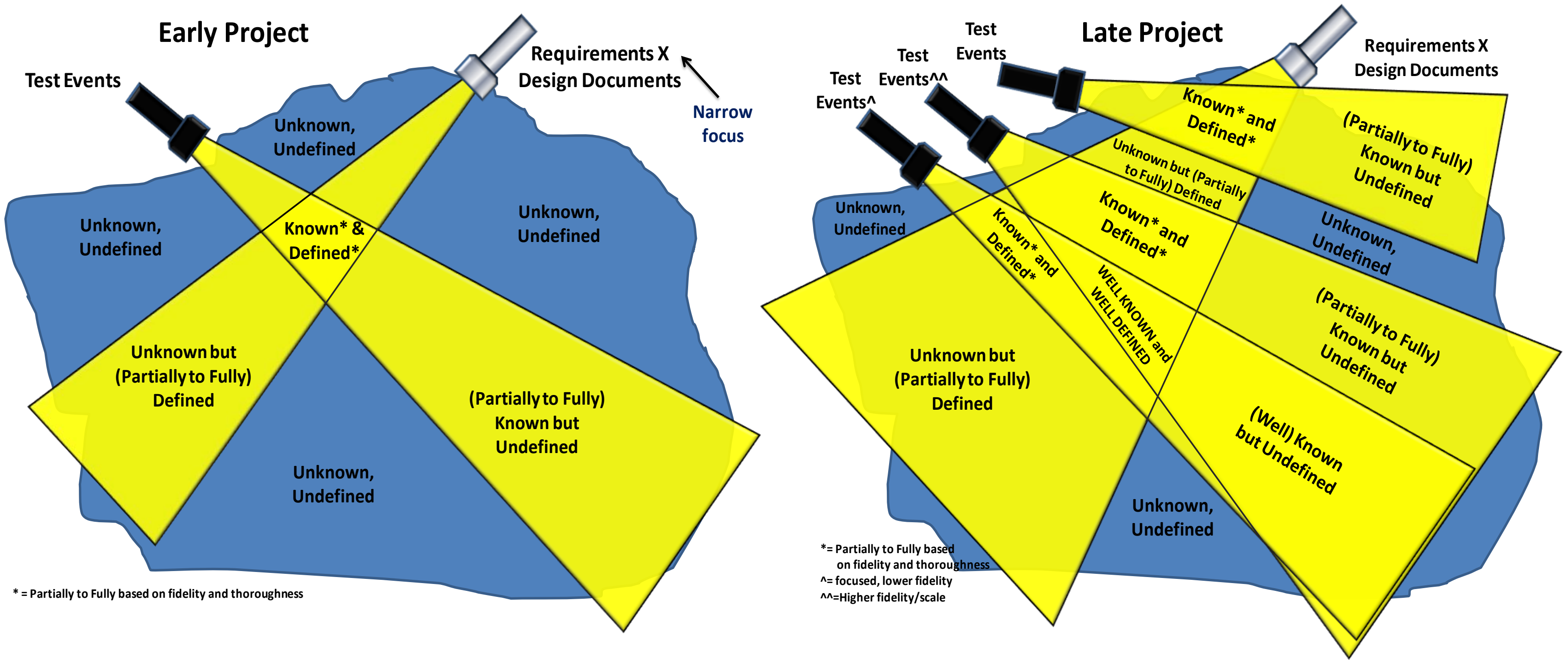

Early Project
Test Events
Requirements X Design Documents
Narrow focus
Unknown, Undefined
Unknown, Undefined
Unknown, Undefined
Known* & Defined*
Unknown but (Partially to Fully) Defined
(Partially to Fully) Known but Undefined
Unknown, Undefined
* = Partially to Fully based on fidelity and thoroughness
Late Project
Test Events
Test Events^^
Test Events^
Requirements X Design Documents
Known* and Defined*
(Partially to Fully) Known but Undefined
Unknown but (Partially to Fully) Defined
Known* and Defined*
Unknown, Undefined
Unknown, Undefined
Known* and Defined*
WELL KNOWN and WELL DEFINED
(Partially to Fully) Known but Undefined
Unknown but (Partially to Fully) Defined
(Well) Known but Undefined
Unknown, Undefined
*= Partially to Fully based on fidelity and thoroughness
^= focused, lower fidelity
^^=Higher fidelity/scale

Figure 1: Flashlights in a Dark Cavern, a Project Phase Analogy

2 Defining Software Performance in a SOA SoS

The SoS environment in this context included software implemented within a service context, with common middleware, deployed on diverse systems, and networked together by a wide area network (WAN). One objective is to better understand the software performance quality characteristics, especially timing and resource consumption, under full-scale usage conditions, using representative or complete software elements, on real or high fidelity representative hardware. To facilitate SWP management, a roadmap needs to be developed that depicts the SWP progress (meeting performance targets) from initial testing through final testing. The early collection of SWP metrics (e.g., paper analysis) is informative material. It is essential that the end-state test environment include the following:

- testing at full end-to-end scale
- use of complete software builds on production hardware (as possible)
- use of objective network and platform elements

The idea is to, as closely and early as possible, resemble the fielded operational environment. A fundamental goal is to select software performance metrics tied to software performance characteristics, which are linked to system (and ultimately system of systems') ability to perform as required.

2.1 Organizational Issues in Improving Performance for SOA SoS

Any method to assess and improve software performance must allow for coordination and input from diverse groups, each with its own concerns. Tests occur at many levels of fidelity, of varied scales, using models of many types, with different stakeholder collection goals. Ideally, testing should eventually include ways to improve performance testing for all interest groups. An overall objective is to maximize the effectiveness and efficiencies of test events holistically, across all interest groups. This needs to be a planned and managed effort. A formal definition of *Roles and Responsibilities* is useful to avoid confusion or management gaps. In this case, it was not clear to all who collected valued metrics that they were also responsible for publishing their data to the common repository to allow for a more complete program-level SWP analysis.

2.2 A Process for Determining SWP Using Select SOA SoS Metrics

This process is an initial leap in assessing the current SWP state. The included 10-step process is intended to facilitate the development of a common view. A common vocabulary for respective performance metrics for test events is offered, accompanied by a method for linking test events and metrics via a SWP metric matrix.

The ten steps are:

1. Develop a SOA SoS Performance View.
   Develop a SOA SoS layout performance view.
2. Review Key SOA SoS Resource Limiters.
   Review key resource limiters from the layout.

3. Develop Sample Scenarios and Determine Respective SWP Impacts.
   Develop a series of scenarios, then list the performance impacts in each step and section of the scenario.
4. Create an Initial List of SWP Metrics for Your SoS
   List the metrics that affect or quantify the impacts to software performance in each scenario, and combine all the impacts into a common list of metrics.
5. Add Required Software Performance Metrics from Other Sources
   Add required software performance metrics from other sources (e.g., sub-contractors).
6. Determine All Test Events and Rate Their Maturity
   Determine all test events (including integration events) that have occurred at every level and in each organization in the SOA SoS. Rate the fidelity of each event for each metric. Add to the columns of the metrics list to form the "metrics matrix."
7. Determine What Metrics and Events are Missing
   Circulate and vet the metrics matrix throughout all architecture and engineering test organizations in the SOA SoS, asking: What metrics or test events are missing? Update the matrices.
8. Plan Future Tests and Mine Data from Existing Data Sets
   Use the populated and vetted metrics matrix to plan future events: What gaps exist in infrastructure, test methods, or test plans?
9. Tie in Architecture to the Metrics
   Using traceability, tie-in architecture to improve software performance of the SOA SoS. What elements are tied to each metric?
10. Determine the Refresh Schedule
    Determine how often the last nine of these ten steps will be repeated.

Figure 2 shows the 10 steps in a process flow format. This 10-Step Method follows the general tenets of the Practical Software and Systems Measurement (PSM) Measurement Process Model of Plan, Measure, Evaluate, and Re-plan.

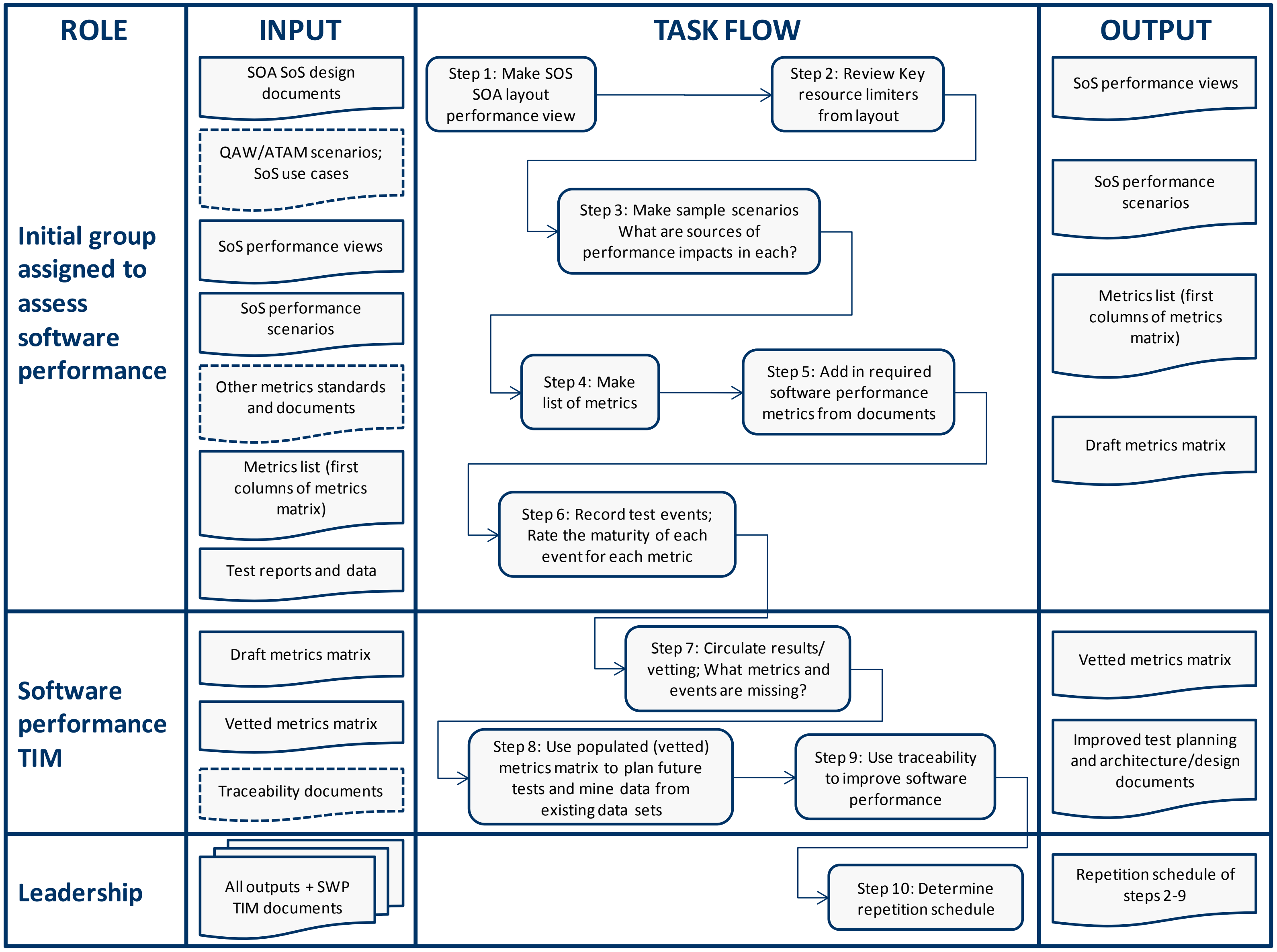

ROLE
INPUT
TASK FLOW
OUTPUT
Initial group assigned to assess software performance
SOA SoS design documents
QAW/ATAM scenarios; SoS use cases
SoS performance views
SoS performance scenarios
Other metrics standards and documents
Metrics list (first columns of metrics matrix)
Test reports and data
Step 1: Make SOS SOA layout performance view
Step 2: Review Key resource limiters from layout
Step 3: Make sample scenarios What are sources of performance impacts in each?
Step 4: Make list of metrics
Step 5: Add in required software performance metrics from documents
Step 6: Record test events; Rate the maturity of each event for each metric
SoS performance views
SoS performance scenarios
Metrics list (first columns of metrics matrix)
Draft metrics matrix
Software performance TIM
Draft metrics matrix
Vetted metrics matrix
Traceability documents
Step 7: Circulate results/ vetting; What metrics and events are missing?
Step 8: Use populated (vetted) metrics matrix to plan future tests and mine data from existing data sets
Step 9: Use traceability to improve software performance
Vetted metrics matrix
Improved test planning and architecture/design documents
Leadership
All outputs + SWP TIM documents
Step 10: Determine repetition schedule
Repetition schedule of steps 2-9

Figure 2: Software Performance Process Flow – 10 Steps

3 Detailed 10-Step Method for Software Performance in a SOA SoS Environment

3.1 Step 1: Develop a SOA SoS Performance View

The four sub-steps below outline the process for generating a SOA SoS performance view.

3.1.1 Notional System of Systems Hierarchy

There needs to be a common understanding of the hierarchy from the SWP perspective of the SOA SoS. This hierarchy can then be used for further decomposition. Start by obtaining a high-level view of the SoS (a common artifact to most systems) and determine at what levels of the hierarchy are services instantiated. Each system has one or more processing units. Each processing unit has one or more blades. Each blade can host one or more services, as in Figure 3. Note that your SoS may differ. You might want to, for example, replace system for a rack or LAN, and blade and processing unit for server, etc. This decomposition is notional to illustrate the techniques in this report.

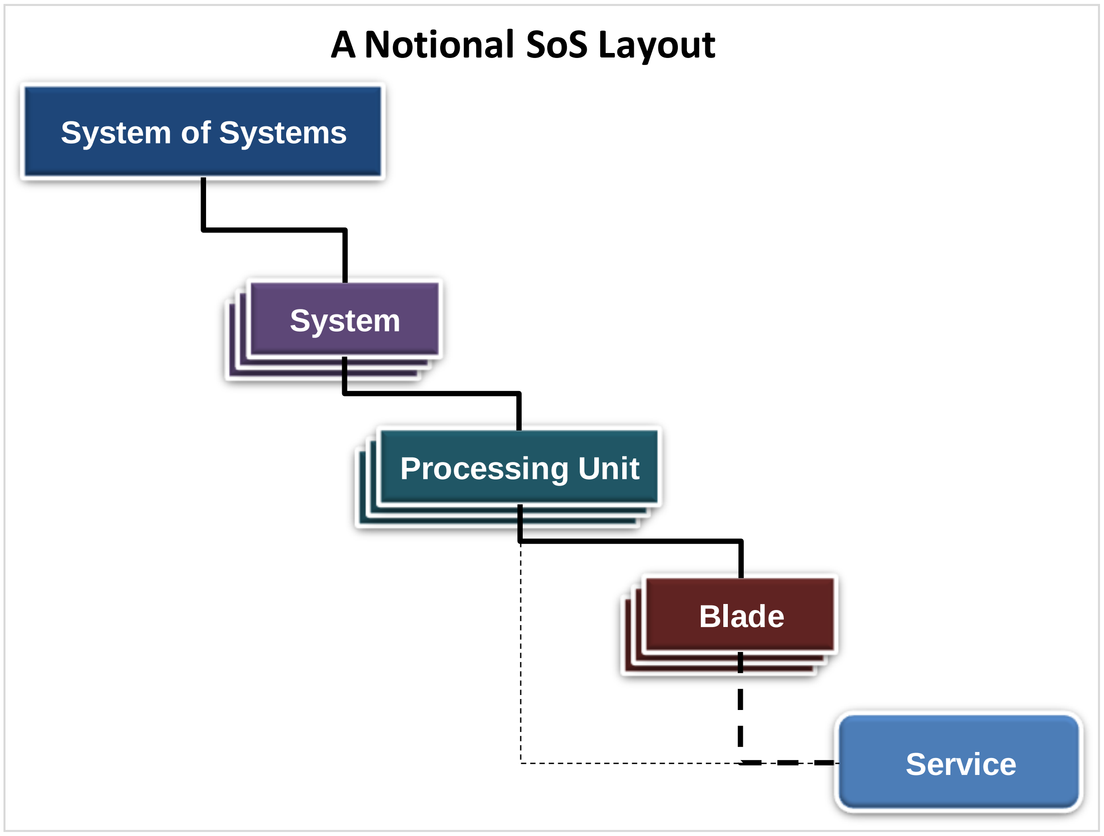


Figure 3: Notional SOA SoS Hierarchy Layout

3.1.2 Bottom-Up Decomposition: The Blade

A card or blade is a server (see Figure 4), with its own CPU or CPUs, caches, DRAM, a slot for a flash card with flash memory, possibly its own drives, and an interface (backplane interface) that allows it to link to other blades over a common backplane. (The backplane is a switch or bus structure on the processing unit, at the next level up.) If we look at delays to access various components, it is faster to reach data in process on the same CPU than it is to reach out to the L1 cache—which is faster than reaching to the L2 cache, which is faster than accessing either the on-blade flash card or going off-card to the processing unit level via the backplane interface.

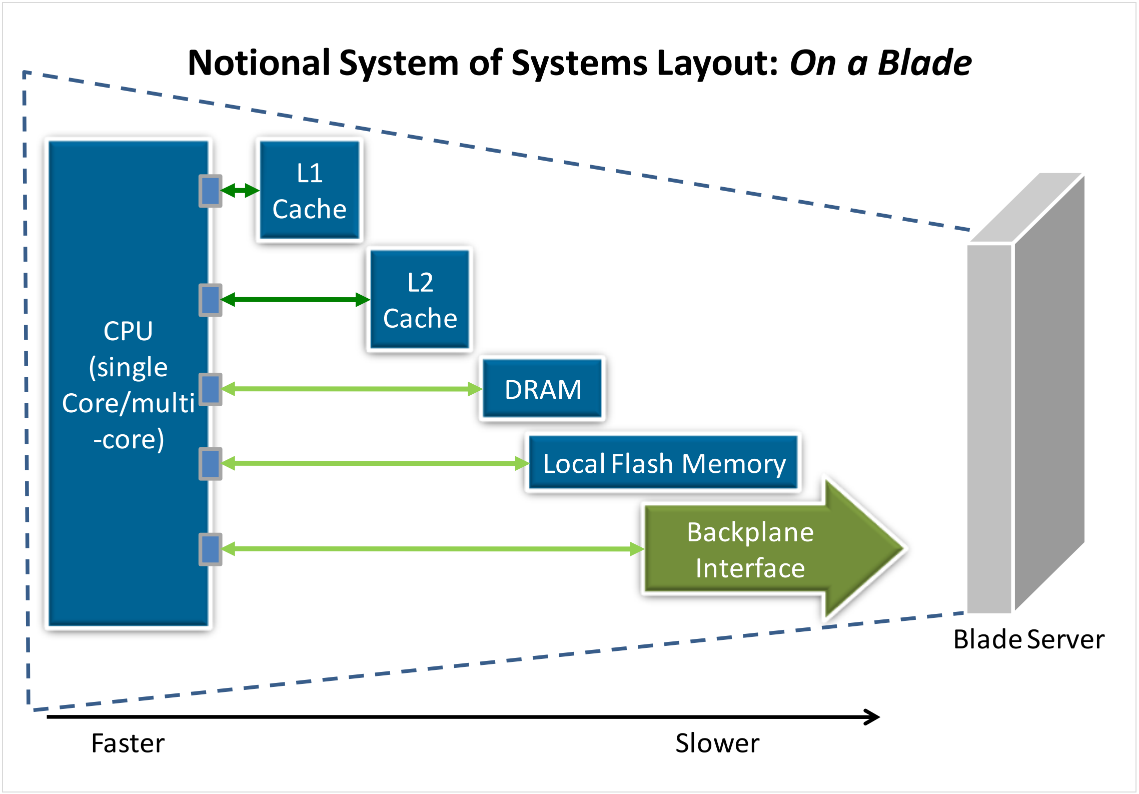


Figure 4: Single Server (Blade) Performance Measurement Points

3.1.3 Bottom-Up Decomposition: The Processing Unit

The next level up is the processing unit (see Figure 5). A processing unit is defined as the enclosure that holds one or more blades, each blade in the processing unit linked by a common backplane. This backplane is a structure similar to a switch backplane (e.g., Ethernet), or may be similar to a bus backplane. In the processing unit, the backplane links to a specialized blade that acts as a firewall and router used to link to the system's LAN (e.g., fiber optic gigabit Ethernet). We will refer to this LAN router/firewall as the processing unit's "LAN blade." The processing unit has a fiber channel interface to the system's RAID array. A specialized processing unit in the system will provide workstation access to the processing units in the platform. For delays (think SWP), accessing data on the same blade is faster than going between blades in the same processing unit, or going to the fiber channel RAID adapter (which introduces fiber channel and RAID delays), or going to the LAN router/firewall to use resources on the LAN provided by other processing units or WAN to other systems. Thus, the system architecture directly affects SWP.

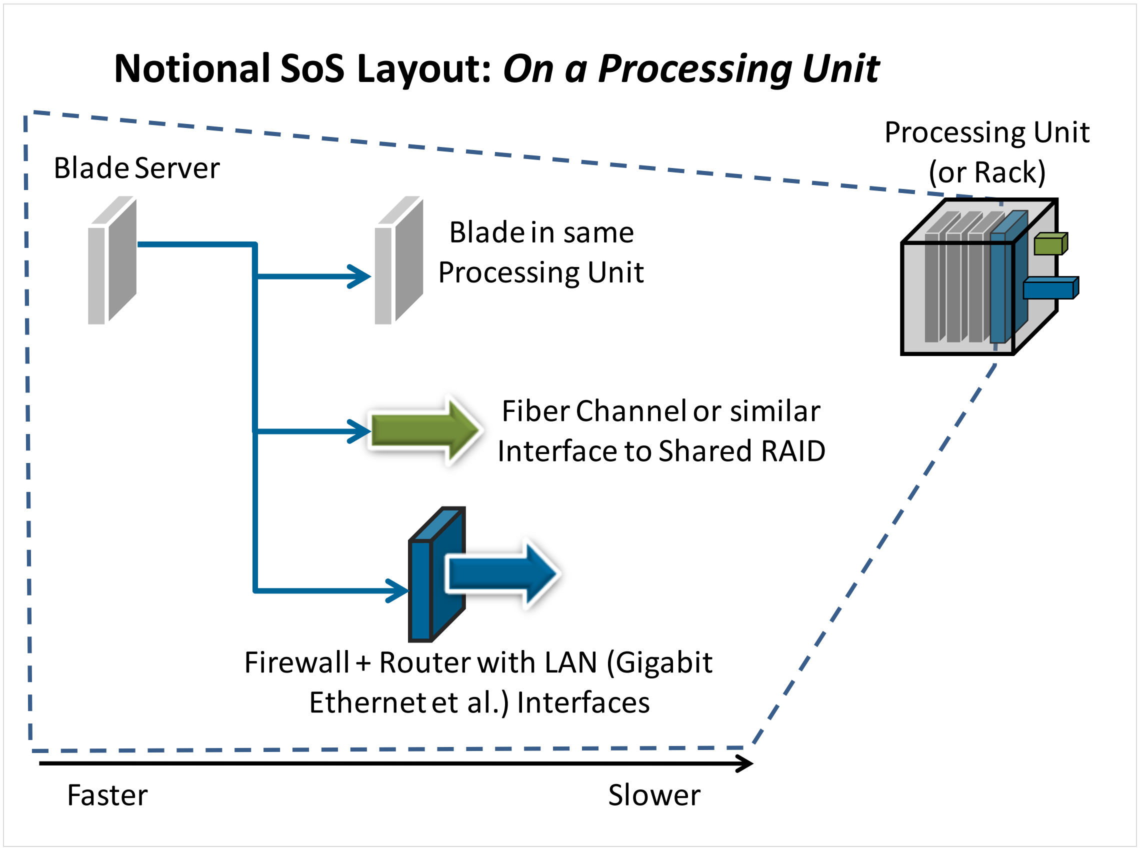

Notional SoS Layout: *On a Processing Unit*
Blade Server
Processing Unit (or Rack)
Blade in same Processing Unit
Fiber Channel or similar Interface to Shared RAID
Firewall + Router with LAN (Gigabit Ethernet et al.) Interfaces
Faster
Slower

Figure 5: Decomposition of the Processing Unit

3.1.4 Bottom-Up Decomposition: The System

The system is the primary element in our SoS. The example system consists of multiple processing units, and three units for radio interfaces to the SoS WAN, linked by a system LAN (e.g., fiber optic gigabit Ethernet), and a RAID linked to each processing unit via a fiber channel interface. The processing units on a system LAN can be of various compositions, types, groups (for security purposes, for example) and usages. In this example we list a short-range radio unit (which includes a router and firewall for the short-range WAN segments), a long-range radio (again including router and firewall for the WAN segment), and a satellite connection (with router and firewall capability). Systems have a router/firewall with connections to a physical fiber optic network WAN, although our example precludes a system-to-system tether. Access is significantly faster on the LAN because of low latency and higher bandwidth; whereas WAN connections experience more delays making access significantly slower (see Figure 6).

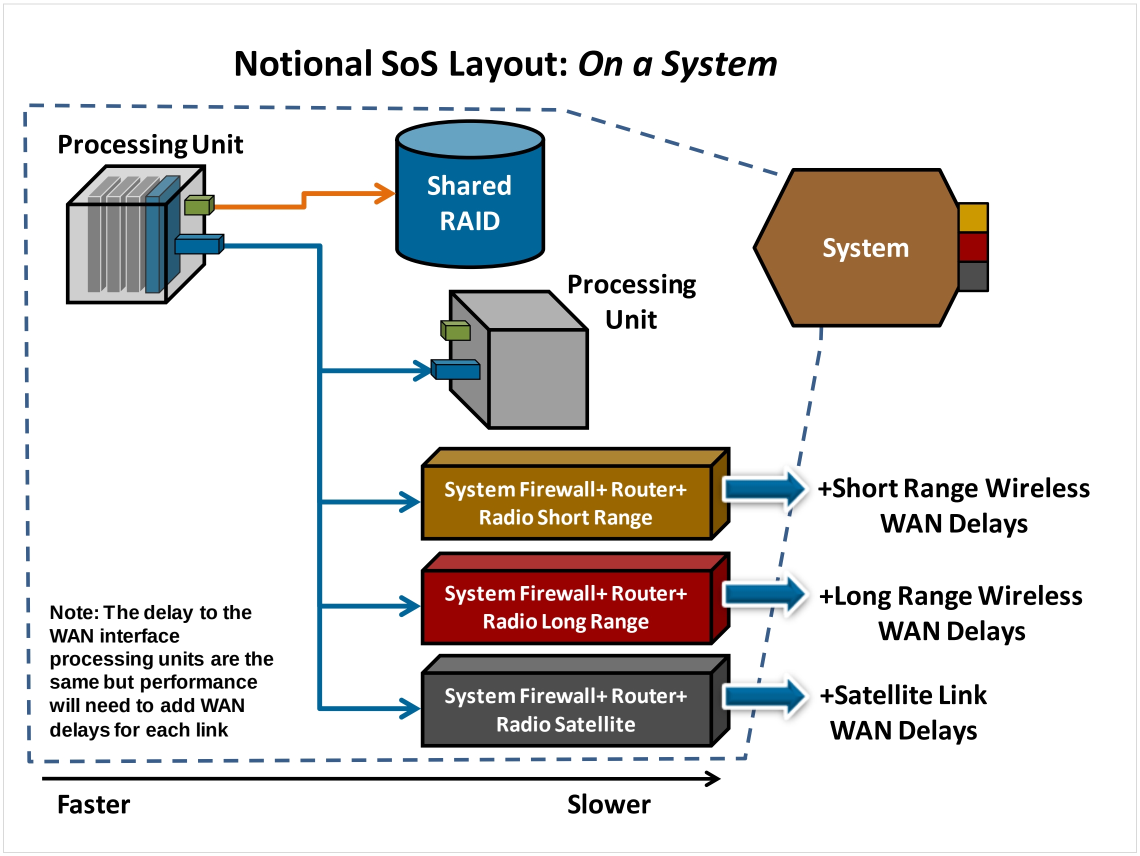

Notional SoS Layout: *On a System*
Processing Unit
Shared RAID
System
Processing Unit
System Firewall+ Router+ Radio Short Range
+Short Range Wireless WAN Delays
System Firewall+ Router+ Radio Long Range
+Long Range Wireless WAN Delays
Note: The delay to the WAN interface processing units are the same but performance will need to add WAN delays for each link
System Firewall+ Router+ Radio Satellite
+Satellite Link WAN Delays
Faster
Slower

Figure 6: Decomposition of the System

In the notional SoS we assume that the short-range connections are faster than the long-range connections over the long-range radio, which are faster than those to connected systems or resources over the satellite network. We also assume that an indirect connection (two short-range radio hops) is longer in delay than a single long-range radio hop, and that an indirect long-range or short-range indirect multi-hop connection to another system entails less delay than reaching over the satellite network. Indirect connections involve network delays over the connection medium, plus open system interconnection (OSI) delays for switching and routing for each hop. The satellite connection must consider the satellite's OSI switching delays in addition to the long physical distance to the satellite and back.

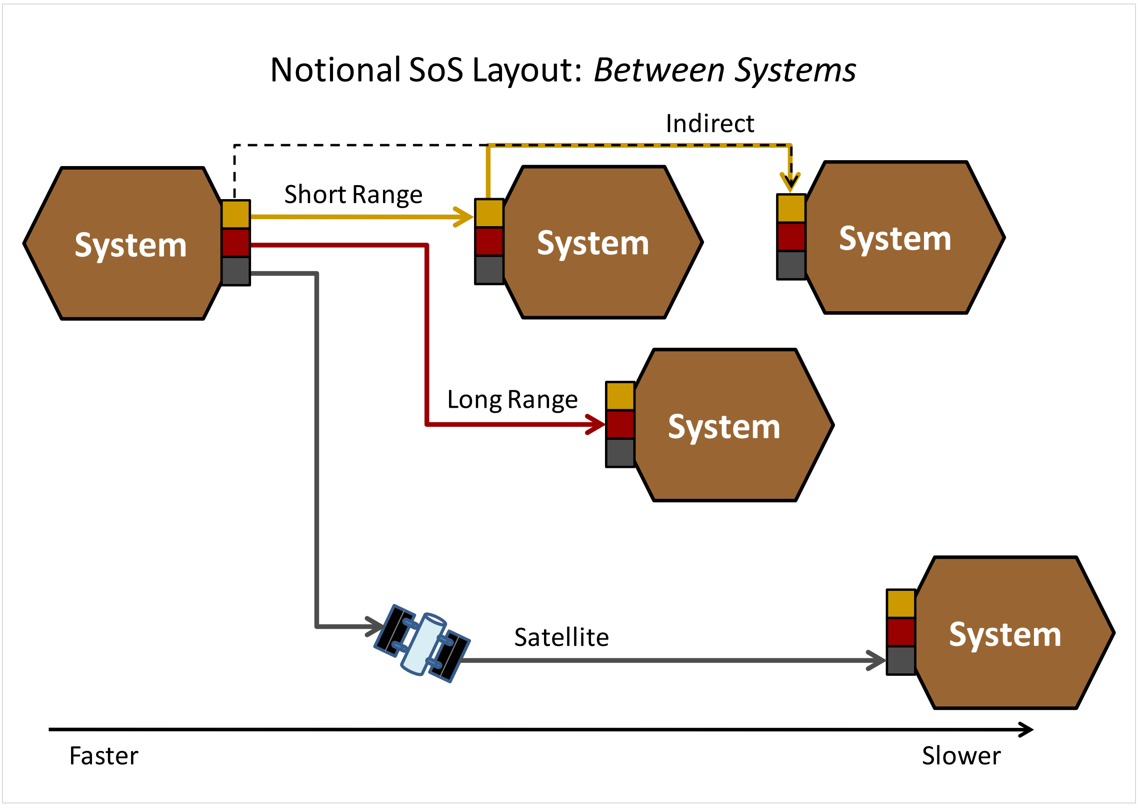


Figure 7: WAN Connections between Systems in the Notional SoS

3.1.5 What about the Services in Our SOA SoS?

Services, the “S” in SOA, are sets of discoverable items that perform a function or functions, defined by the service offer and interface to the service. The service exists as an instance on top of a common middleware, which in turn is loaded onto a blade (Figure 8). This middleware (which may have variants) encapsulates the OS of the host blade (or across blades, though not covered here) across the system of systems.

Each service instance on a blade is found by the larger SoS using a service offer that specifies what the service does, how it does it, and how to connect to the service using an interface. The middleware of other blades and systems then discovers the service offer, and adds the service to the list of available services.

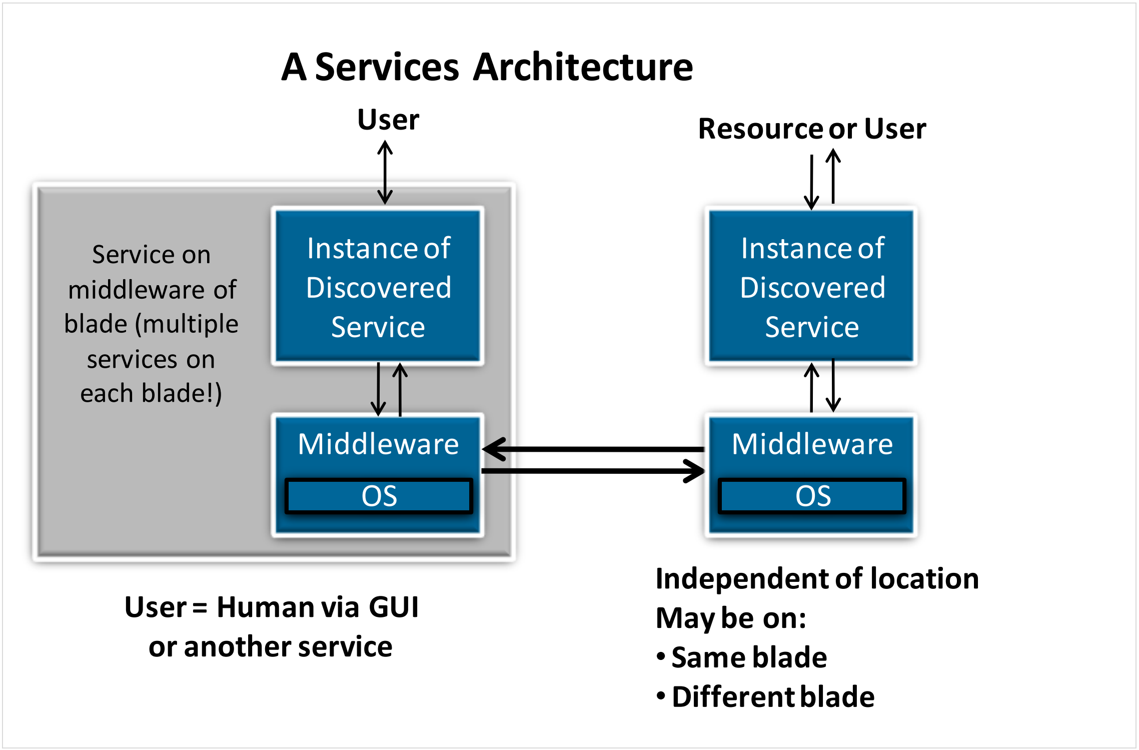

A Services Architecture
User
Resource or User
Service on middleware of blade (multiple services on each blade!)
Instance of Discovered Service
Instance of Discovered Service
Middleware
OS
Middleware
OS
User = Human via GUI or another service
Independent of location
May be on:
• Same blade
• Different blade

Figure 8: A Generic Service Architecture: Service, Middleware, Blade OS

Note that the service instance could be accessed from anywhere in the SOA. The middleware hides all but the interface from the service. As in our Internet example, a user of a search engine (service) is unaware of the physical location of the web page. The service in our notional SOA SoS has no awareness. As a result, in many real-world SOA SoS systems with WAN concerns, a quality of service schema using priority settings in the service interfaces is used in the middleware and WAN routers to control traffic and usage of services.

3.2 Step 2: Review Key SOA SoS Resource Limiters

Many SWP effects are related to (1) the time it takes to retrieve data from various sources, and (2) the use of key data resources. Additionally, performance may suffer from errors, faults, or sub-optimally maintained system resources. This step provides a basic look at some of these factors.

3.2.1 Performance Factors That Affect Software Performance

SWP can be boiled down into two top-level factors: delay to data and critical resource utilization. Looking at the initial shaping factors provided below for delay, timing can be orders-of-magnitude different on the representative performance wheel, as shown in Figure 9. One approach to optimize performance is to minimize slower interactions. The availability of slower assets tends to promote their usage; a good software design, however, manages how often slower or key resources are used.

Errors of various types, abandoned processes, threads, or service offers can consume resources. To better manage SWP you need to understand the general types of issues inherent in the architecture. We recommend that managers answer the following questions for each level of the SoS hierarchy for Step 1 (on page 8), and use the answers in Step 3 (on page 17):

- What measures are needed to quantify software performance for improvement?
- What metrics show use of critical assets at each level of the SoS, including inefficiencies?
- What errors delay or prevent availability?

The appendix (see page 44) provides a list of measures that were utilized within our experience.

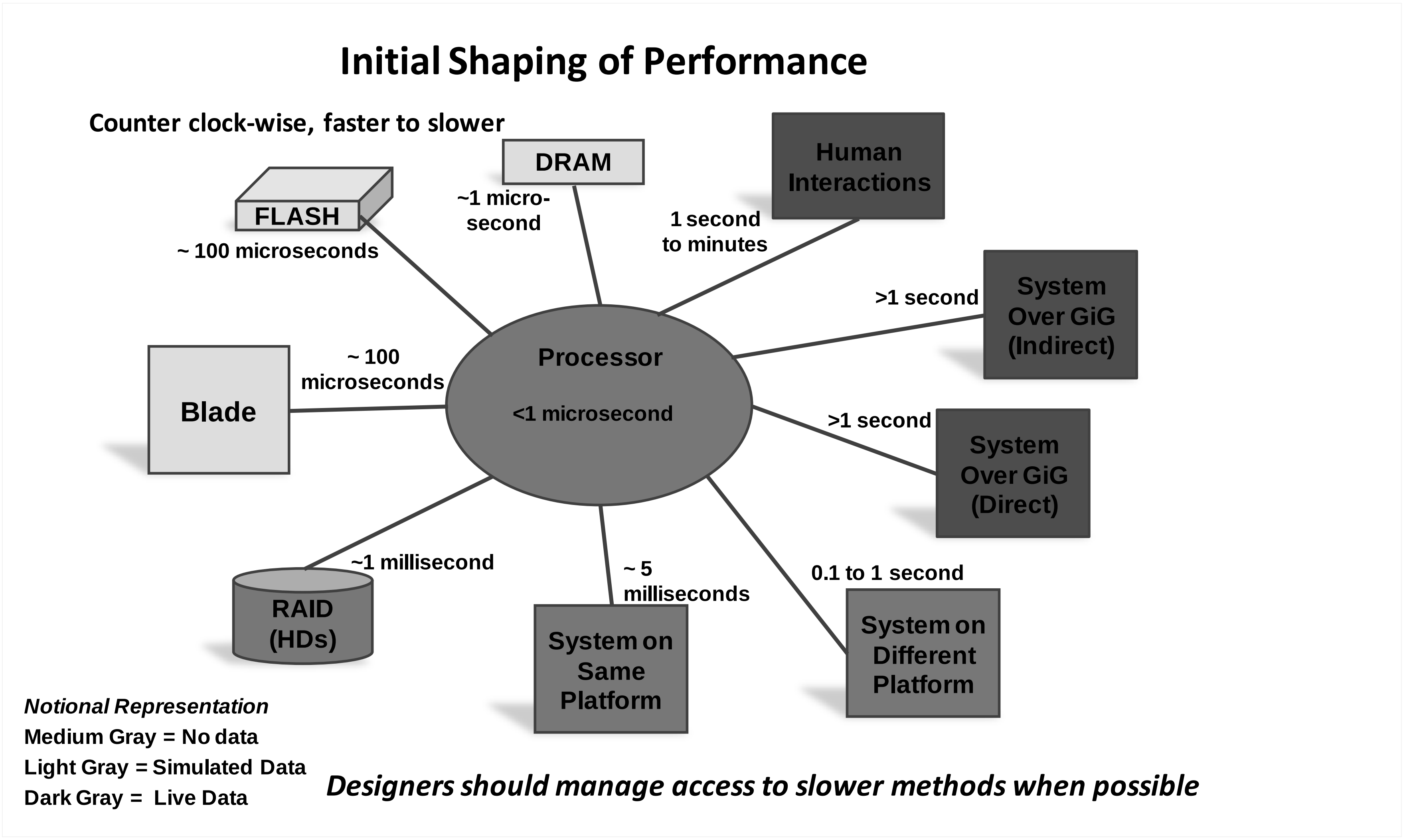


Figure 9: Initial Performance Shape for the SoS

3.2.2 The Effects of Scale

The work of each blade and system will increase based upon the total number of systems in the SoS. Each increase in scale increases resource needs per Service hosting blade, memory consumption at various levels, time to discover the increased numbers of service offers, WAN utilization, etc. Hierarchy decisions in software architecture can determine scalability, such as how to

- disseminate data to various systems in the SoS
- discover services and aggregate discovery
- allocate resource location and network addresses
- balance processing versus network usage

The appendix (page 45) has sample metrics that can be used to determine how scale will affect the SoS's software performance.

3.3 Step 3: Develop Sample Scenarios and Determine Respective SWP Impacts

In this step sample scenarios are used to determine performance impacts or factors.

Note: The outputs from an SEI Quality Attribute Workshop (QAW) or Architecture Tradeoff Analysis Method (ATAM) effort, targeted at software performance, could also aid in arriving at a list of scenarios for this step [Barbacci 2003].

3.3.1 Sample SOA SoS Scenarios and Factors

A group should be assembled to develop scenarios similar to the sample scenario that appears in Figure 10. In each step, determine which components at each level—and which interactions between components (both called "factors")—will affect software performance.

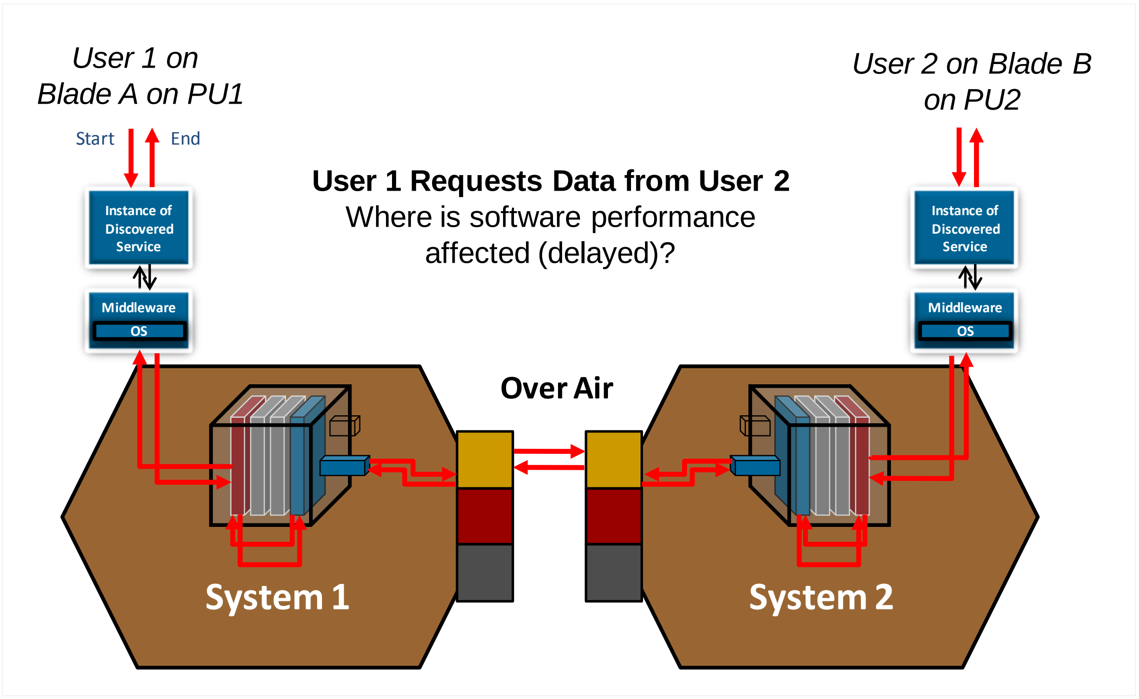

User 1 on Blade A on PU1
Start
End
User 2 on Blade B on PU2
User 1 Requests Data from User 2
Where is software performance affected (delayed)?
Instance of Discovered Service
Middleware
OS
Instance of Discovered Service
Middleware
OS
Over Air
System 1
System 2

Figure 10: Possible Scenario for Our SOA SoS

For example, examining just the delays involved in this scenario where User 1 uses a service to request data from User 2 and User 2's response, we have at least:

- User 1's human delay[2]
- GUI delay for User 1's interface
- On Blade A:
  - service call to middleware

[2] There are various causes of performance issues and you will want to find where in your architecture 'snags' can occur. To improve performance you might need to either perform software tasks while waiting for human response or limit the need to ask user for responses.

  - middleware to OS
  - delays between Blade 1 and Processing Unit 1’s LAN blade
  - processing on PU 1’s LAN blade
  - LAN latency to short-range Router/FW/Radio 1
  - delays on short-range Router /FW /Radio 1
  - latency over the air to System 2’s short-range router/FW/radio
  - delays on short-range Router/FW/Radio 2
  - LAN latency from short-range Router/FW/Radio 2 to PU2’s LAN blade
  - processing on PU2’s LAN blade
  - delays between PU2’s LAN blade and Blade B
- On Blade B:
  - OS to middleware
  - middleware to service call
  - GUI delays
- User 2’s human delays

The group can derive more factors then these. Each of these factors will be tied to the metrics that determine the successful accomplishment of the scenario, as defined in system of system-level requirements. List these factors, and the associated scenarios that generated them.

3.4 Step 4: Create an Initial List of SWP Metrics for Your SoS

Utilizing use case scenarios and a respective list of performance factors, we are now in a position to generate a common list of software performance metrics.

3.4.1 Getting Metrics from Factors

What are software performance metrics? They are measurable items that inform the ability of the SoS software as deployed on systems to meet software quality characteristics and SoS requirements, especially timing and resource consumption. A metric is attributable to one or more factors that affect the performance of scenarios. One metric may affect several factors in many scenarios.

Software performance metrics may be tied to system-level requirements for system performance. System-level test scenarios, if they exist and are defined in requirements, might also define SoS-level performance. Further, existing standardized sets of software performance metrics from other projects or papers, such as this one, may also exist. All these together can be used to define a rough list of software performance metrics.

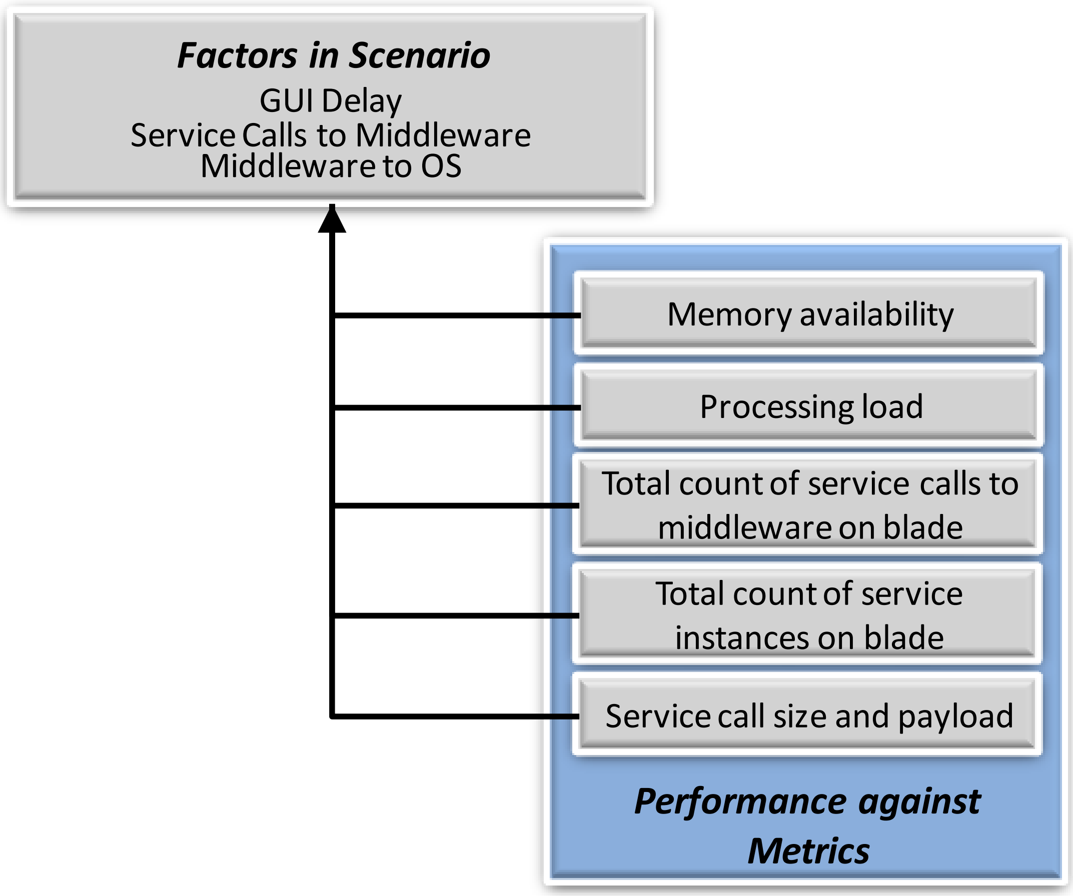

Factors in Scenario
GUI Delay
Service Calls to Middleware
Middleware to OS
Memory availability
Processing load
Total count of service calls to middleware on blade
Total count of service instances on blade
Service call size and payload
Performance against Metrics

Figure 11: Decompose Metrics from Each Factor from Each Scenario

The SoS and SOA decomposition above can be used to brainstorm performance metrics. *From the software perspective, which measurements at each layer could be used to predict or affect software performance or the scalability of the system? How are the factors in the scenario measured?* For example, you might consider *how many calls are made to the middleware on each blade for each test scenario and condition*—this relates to the performance of User 1's use of service to middleware of Blade A of CPU 1 in Figure 11.

You can also ask, *how many times does a service use each type of memory, or reach off-system to touch other systems' memory?* This question incorporates factors such as delays between Blade 1 and Processing Unit 1's LAN blade. It also indicates which service components can be re-hosted to swap slow off-system calls for faster LAN calls to the RAID, or other on-system processing units.

A question-and-answer round of this type with a knowledgeable and diverse team of architects or engineers can produce a list of performance metrics.

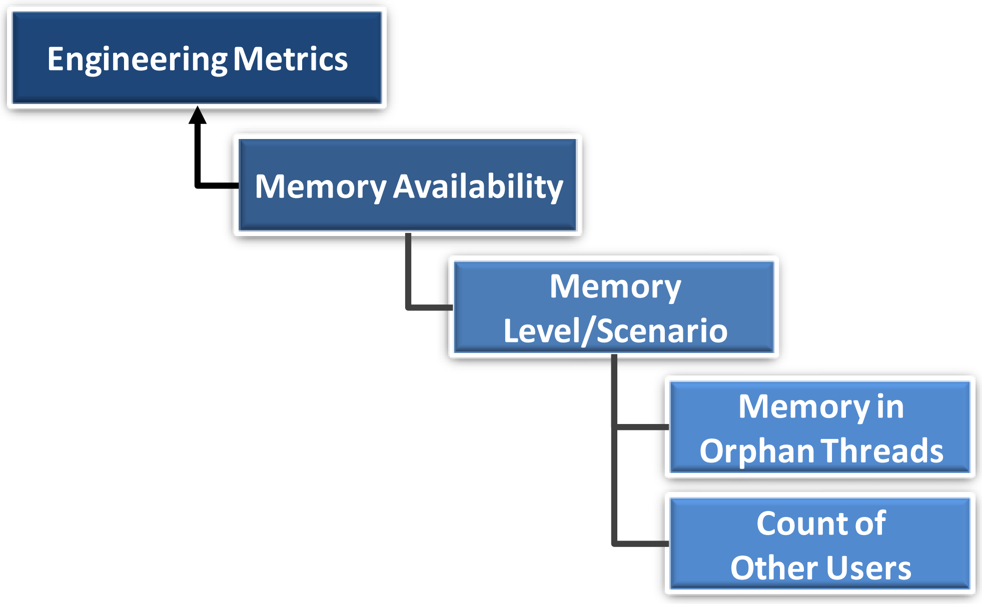


Figure 12: Decomposition of Performance-Related Metrics

Given a list derived by Steps 1-3 above, and the rounds of questions-and-answers above, you can derive more specific metrics by decomposing system-level metrics into software performance metrics. Which metrics should be broken up into easier to measure or more software related sub-metrics? For each metric found, record the reason the metric is needed and a rough idea of how it might be collected. Make a note of a few keywords that can be used for metadata tagging any data that contains the metric. Also consider noting a short name—preferably using a standard naming convention—that can be used in data fields. It is helpful to number each metric. Some metrics can be rolled up into metric families.

3.4.2 (Optional) Values of Metrics from Requirements: The Why

If requirements include system performance metrics (this is not always the case), it is useful to find the ensuing values of software performance metrics as part of the decomposition process above (see Figure 13). Performing this optional step will make Step 9 in the process much less time consuming. These values typically are maximum allowable values for timing and utilization and minimum allowable values for quality. Use a question-and-answer round as in Section 3.4.1 to allocate the minimum and maximum values of metrics if there is knowledge of expected behavior.

An example might be that in our sample scenario in Figure 10, a requirements document may state that User A must be able to complete the scenario in 20 seconds. If memory utilization is too high (greater than 90%), processor utilization on blade A is too high (above 90%), and the like for each factor, then the task will take too long to perform because the blade cannot update the GUI on the screen fast enough. This occurs similarly for quality type requirements, for example, "the system must remain functional 99.9999% of the time." It is recommended an analysis occur to determine which metrics in a factor are the key limiters. For example: if a blade is at greater than 100% utilization, or if the LAN is at 100% utilization, the system is unavailable to other commands. The worst case scenario will dictate the value for the SoS in most cases. If known, indicate these requirements and source documents in the *Why?* column of the Metrics List in Table 1.

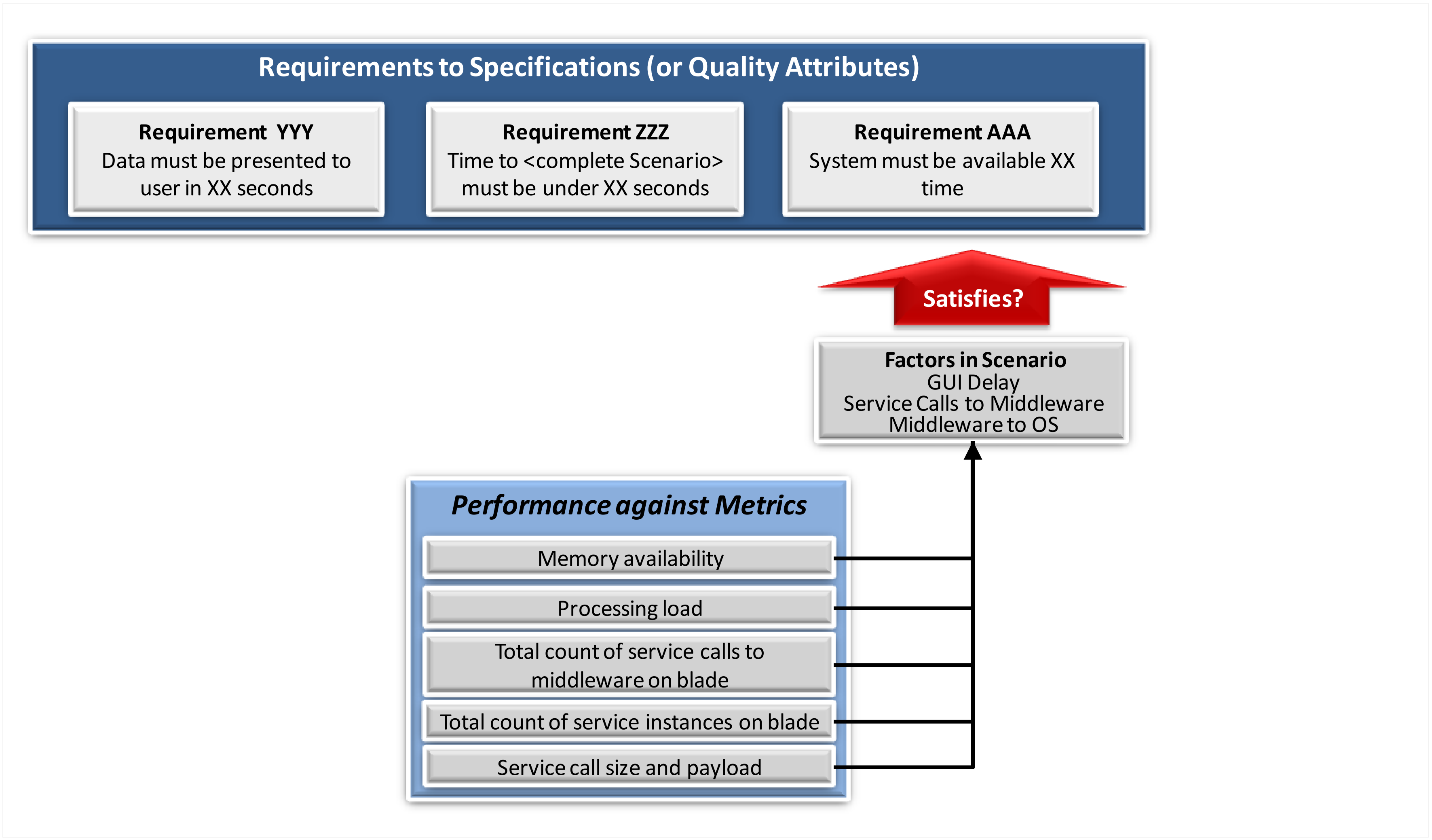

Requirements to Specifications (or Quality Attributes)
Requirement YYY
Data must be presented to user in XX seconds
Requirement ZZZ
Time to <complete Scenario> must be under XX seconds
Requirement AAA
System must be available XX time
Satisfies?
Factors in Scenario
GUI Delay
Service Calls to Middleware
Middleware to OS
Performance against Metrics
Memory availability
Processing load
Total count of service calls to middleware on blade
Total count of service instances on blade
Service call size and payload

Figure 13: Tying Factors and Their Ensuing Metrics to Requirements

3.4.3 Generating the Initial Metrics List

Once the list of metrics is combined and duplicates are removed, add a tag that describes the need that generated the metric (Need Type) and add the high-level category used above ("Engineering metrics" as in Figure 12). Each metric constitutes a row. If further decomposition is used, a dotted-number schema may be used to insert secondary metrics in the rows of the list.

Table 1: An Initial Metrics List with Columns to Identify and Categorize Each Metric

#	Short Name	Metric Title	Why?	Keywords (for Tagging)	How?	Need Type	High-Level Type
1	Bcalls_ Count	Blade-to-blade calls (tagged by service, by process, by user, by case/scenario/time	Limiting calls from blade to blade reduces time (due to bus use)	Blade, calls, count, service, process	Bus monitoring via processing unit against process monitor	Efficiency	Engineer
2	HDCalls _ Count	Service traffic count to Drives	Which services, applications, clients of applications are hitting the drives often. The more often RAM is used in lieu of the drives, the quicker the app will run.	User, service, raid, calls	Process-message snapshots and parse (or logging parse) for OS+bus capture (log parse)	Efficiency	Engineer

This initial Metrics List is now ready to be expanded using other sources.

3.5 Step 5: Add Required Software Performance Metrics from Other Sources

For a project that is underway, requirements and test planning documents should already exist. A subset of the metrics listed in these documents can be useful for inclusion in the software performance metrics list. In addition to these, other sources may be used as well, such as quality documents, other SEI tool related data, or Six Sigma sources.

3.5.1 Must Have List for a SOA SoS System

The example list of metrics in the appendix (page 45) is derived from other SOA SoS efforts, and provides core SWP metrics that have been useful in finding and diagnosing software performance issues. It is not an all-inclusive list, but these twenty must-have metrics should be included to further expand your list.

3.5.2 Other Sources

The SEI technical report *Quality Attribute Workshops (QAWs), Third Edition* [Barbacci 2003] describes Quality Attribute Workshops, which can be used in addition to scenarios to arrive at metrics.

3.6 Step 6: Determine All Test Events and Rate Their Maturity

In this step, the metrics list derived from previous steps is expanded by a column to show coverage to date of the metrics from previous test events. The level of coverage is loosely assessed using a series of tags for the realism and fidelity of each test's measurement of each software performance metric in the list. Once these tags are added for each event, the metrics list becomes the *metrics matrix*.

3.6.1 Dimensional Representation of SWP Factors (Test Fidelity)

Every test event has a level of fidelity, at the metric level, when compared to the final fielded SoS. This fidelity can be visualized as coordinates in a cube. For example, as shown in Figure 14, the further a test coordinate is to the far top right corner, the more representative that test event is for a metric in question. Software is presented on the y axis, hardware (processing hardware in particular) on the x axis, and the scale of the test on the z axis. In order to define the meaning of these coordinates, each level for each axis will need to be defined. Note that the cube can be extended to a hyper-cube by defining axes for each WAN.

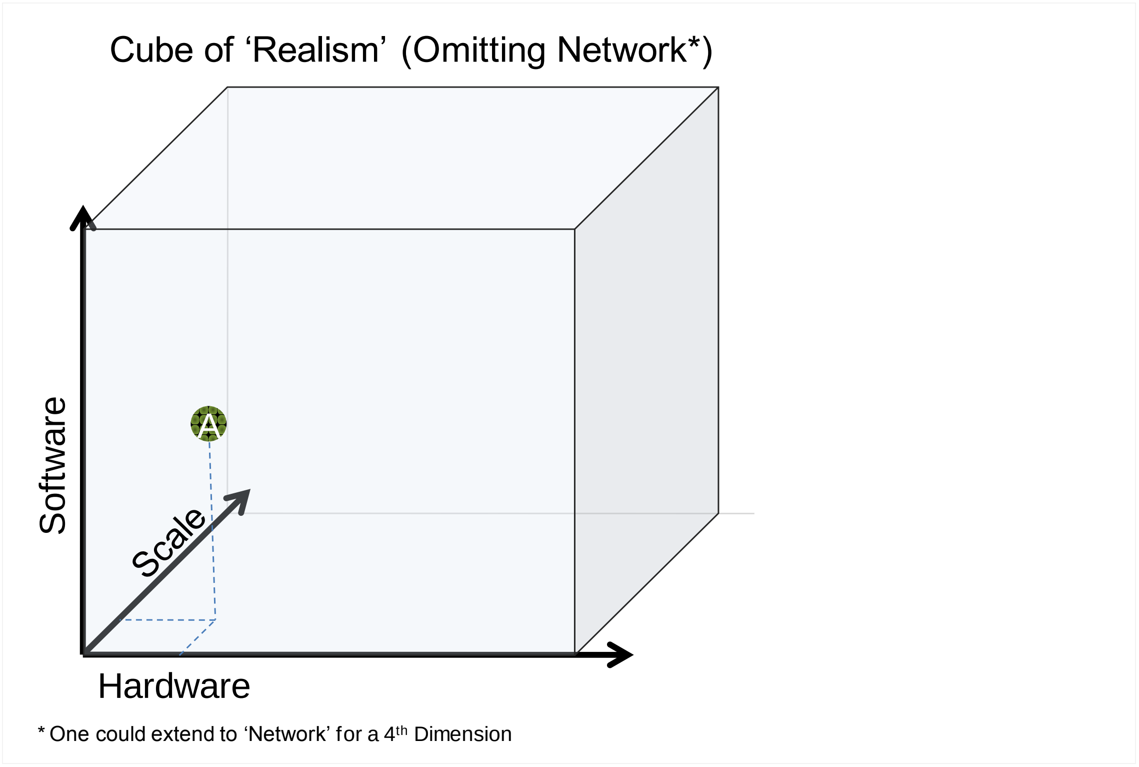

Cube of 'Realism' (Omitting Network*)
Software
A
Scale
Hardware
* One could extend to 'Network' for a 4th Dimension

Figure 14: The Cube of Test Event Realism for Software Performance

3.6.2 Notional Levels for the Software Axis in Test Events

On the software axis, notional levels might be laid out as follows:

- Mod = Modeled
  Uses software to emulate algorithms and simplified functions of most elements
- Sim = Simulated
  Uses software that acts (interfaces) as the service or package would act
- Proto = Prototype
  Uses very early software prototypes, some services coded with minimal functions, some stubbed
- EB = Early Build
  Includes nearly all services, with a major portion of key functions, in actual code that has passed early testing
- LB = Later Build
  Full functioning services in code that has passed some levels of testing, some components have been fielded
- Mat = Mature
  Complete and fielded code (services), revised and matured though use, that has all functions required for its release

3.6.3 Notional Levels for the Hardware Axis in Test Events

Likewise, the hardware (here, hardware that is related to processing) might be laid out as follows:

- Sim = Simulated
  No actual hardware; emulators used to represent all hardware
- EP = Early Prototype
  Some prototype blades, some emulated components
- LP = Late Prototype
  Prototype elements for all major hardware components, some elements have passed early test-ing
- IP = Initial Production
  LRIP production level elements for at least blades, processing units, router/radios, has passed functional testing
- FP = Full Production
  Field tested complete hardware that meets requirements for the version

Figure 15 shows an example of the lab structure for simulating the processing hardware in our notional SoS.

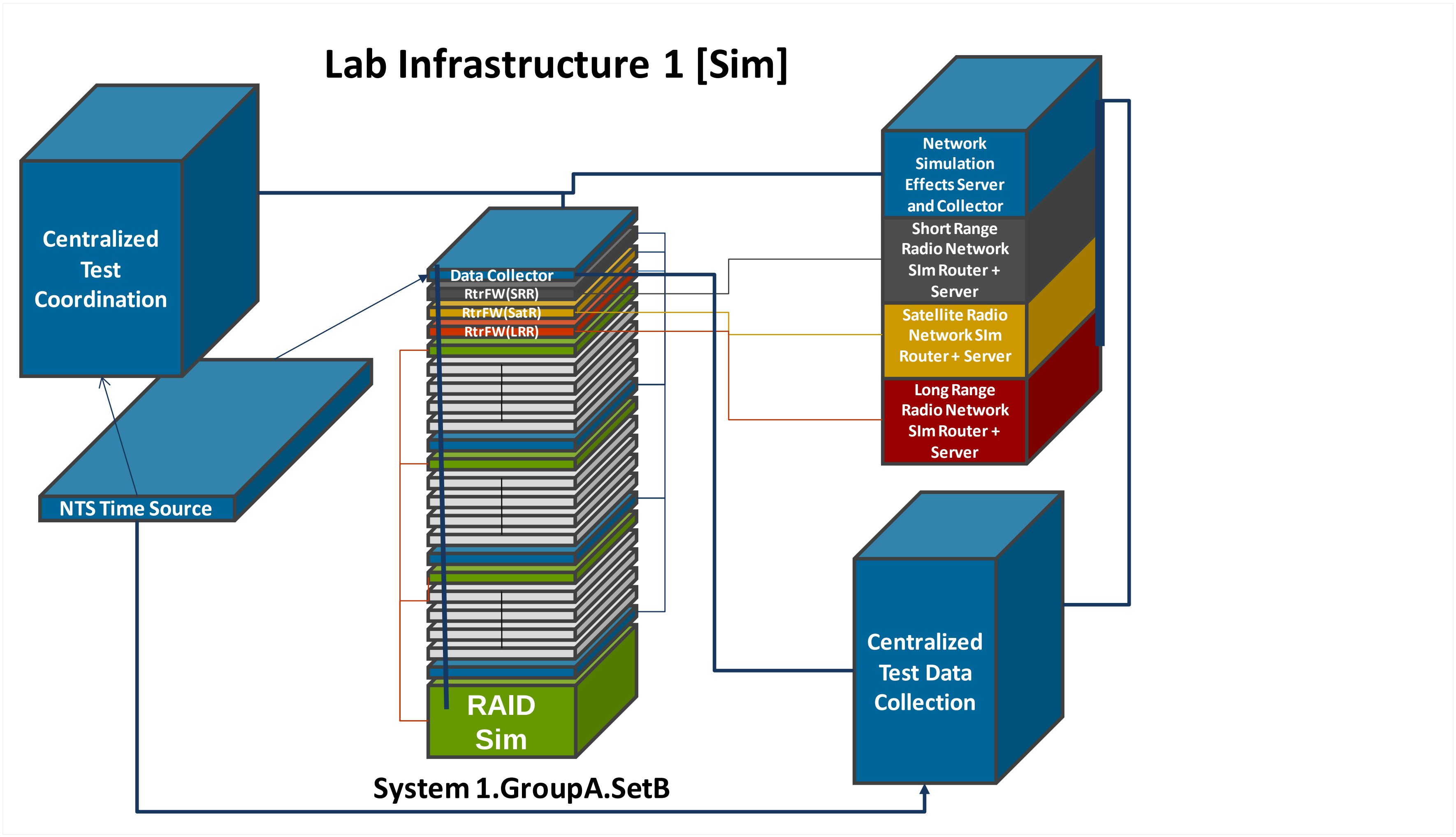

Lab Infrastructure 1 [Sim]
Centralized Test Coordination
NTS Time Source
Data Collector
RtrFW(SRR)
RtrFW(SatR)
RtrFW(LRR)
RAID Sim
System 1.GroupA.SetB
Network Simulation Effects Server and Collector
Short Range Radio Network SIm Router + Server
Satellite Radio Network SIm Router + Server
Long Range Radio Network SIm Router + Server
Centralized Test Data Collection

Figure 15: A Sample Lab Structure for Simulating the Hardware of a System and Processing Units

3.6.4 Notional Levels for the Scale Axis in Test Events

Scale becomes very important in SOA SoS due to the simple fact that as scale increases, and assuming each system provides a number of services and uses a number of provided services, then the increase in scale will cause an increase in each individual system's workload. This increase is due to a variety of factors centering on information distribution and use, offers of services, and discovery. To assess scales we then have the following (see Figure 16).

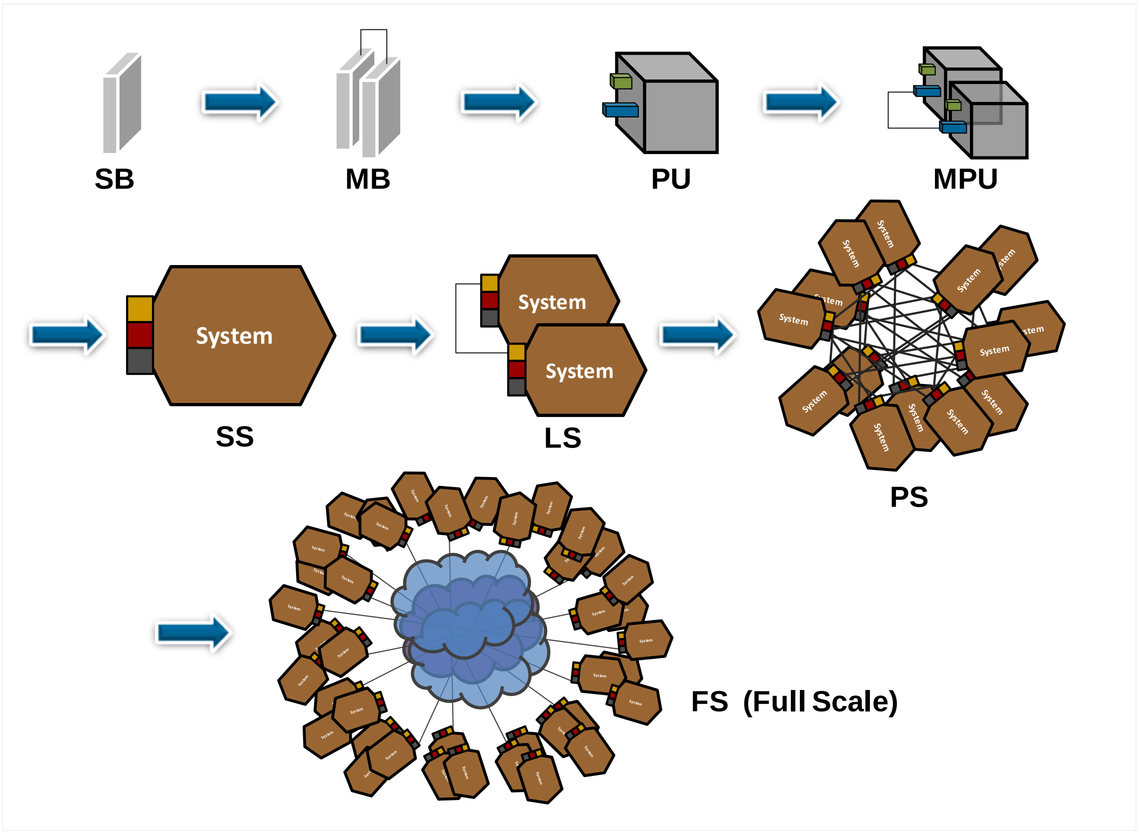


Figure 16: Notional Levels of Scale in Testing

- SB = Single Blade
  Test on a single blade or blade simulator
- MB = Multiple Blade
  Tests using multiple blades or blade simulators, up to a whole processing unit
- PU = Processing Unit
  Tests run on a single processing unit containing multiple blades or simulated blades
- MPU = Multiple Processing Unit
  Multiple processing units without some system components
- SS = Single System
  One complete system, actual or simulated, with processing units, WAN routers/radios, and LAN

- LS = Limited Multiple System
  A few systems, well below full SoS scales, with some WAN or simulated WAN
- PS = Partial (SoS) Scale
  Enough systems to form a significant fraction of the SoS and its WAN
- FS = Full (SoS) Scale
  Testing using actual, or simulated, systems and network at full SoS scales

You could also use a percentage here (i.e., relative to full scale), for example, a scaled-up simulation lab such as in Figure 17.

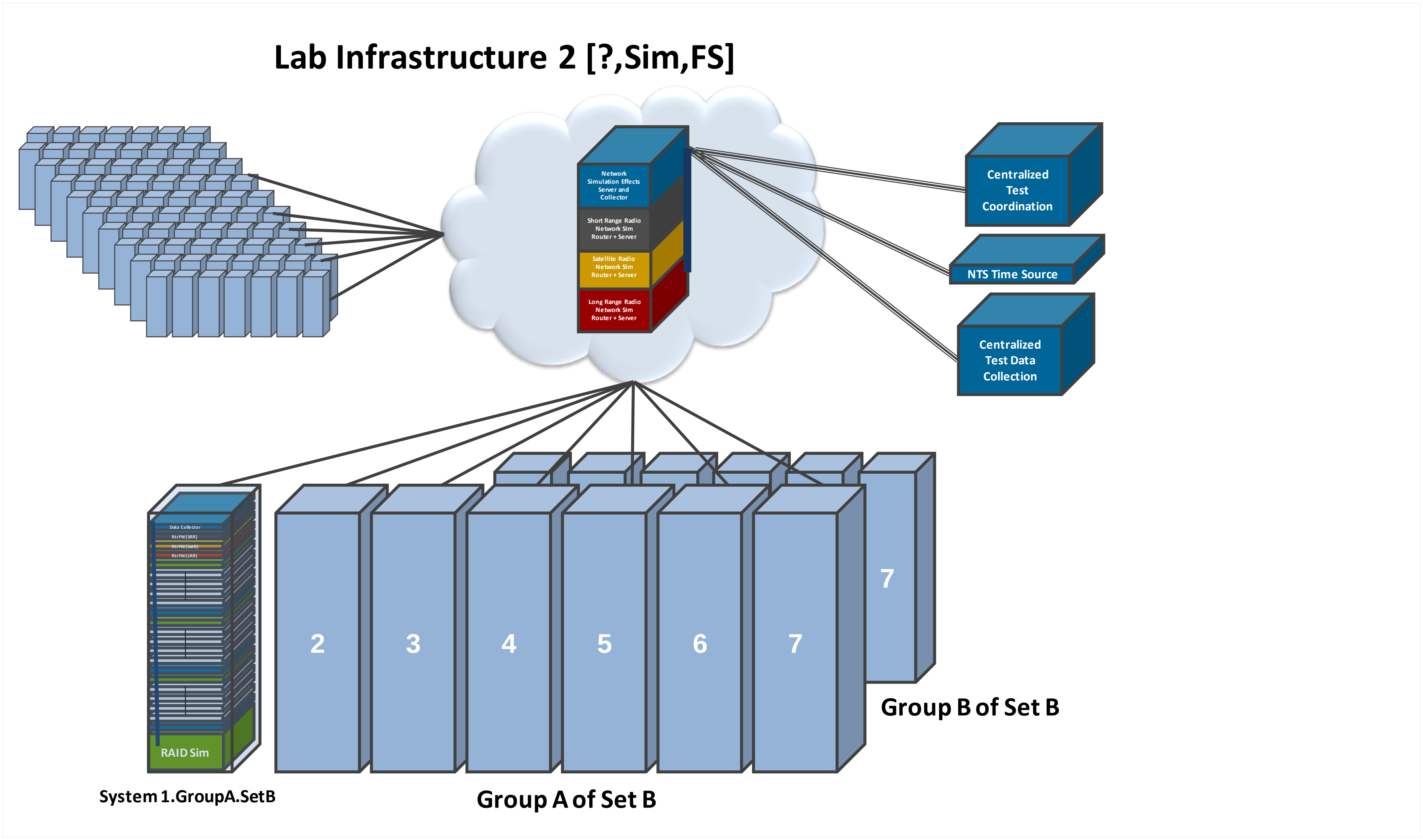

Lab Infrastructure 2 [?,Sim,FS]
Network Simulation Effects Server and Collector
Short Range Radio Network SIm Router + Server
Satellite Radio Network SIm Router + Server
Long Range Radio Network SIm Router + Server
Centralized Test Coordination
NTS Time Source
Centralized Test Data Collection
Data Collector
RtrFW(SRR)
RtrFW(SatR)
RtrFW(LRR)
RAID Sim
2
3
4
5
6
7
7
Group B of Set B
System 1.GroupA.SetB
Group A of Set B

Figure 17: A Test in a Lab at Full SoS Scale of Hundreds of Systems

3.6.5 (Optional) Notional Levels for the Network Axis in Test Events (if Used)

While the focus of this report is software performance, network fidelity for a test event can also be a factor. There are actually three networks in our notional model:

- switch backplane that connects the blades together into the processing unit
- LAN that connects the processing units together to form the system
- WAN that connects the systems into the system of systems

The first network may be assessed at the processing unit level testing on the hardware scale, its metrics align with the hardware scale of testing. Likewise the LAN may be considered in system hardware level testing and its network metrics are included at that level. In other words, add LAN utilization to the metrics list and account for its fidelity on the hardware axis.

The reason to parse the switch backplane and the LAN into the existing cube is that they are "wired" networks in this example and can be accounted for in fidelity in the hardware scale. One needs a switch backplane, real or simulated, to reach the processing unit level. One needs a system LAN to reach the system level on the hardware axis. That said, it may be useful to assign them to an axis. Extending to the more failure-prone wireless links, ideally you would need an axis to account for each wireless network, using a scale similar to this one:

- None= No connections
- BB= Black box inputs
- DR= Connections via some direct wiring/fiber between systems
- WTR=Fiber/wiring using an intermediary router/switch set to simulate spectrum subnets
- Sim= Wiring/fiber through a network simulation system to apply network effects including degradation and latency
- LF= Some radios, or early prototypes, used to link systems in a controlled environment or controlled range
- F= Full set of radios/prototypes at a controlled environment or controlled range
- FR= Full set of radios using final configurations in a near-real set of network conditions

3.6.6 Accounting for Test Quality: Notional Levels

The next step in assessing our current state is to see how each metric was collected (if it was collected) in our test events. Ideally, data should be trended for time, tagged for scenario and test conditions, and correlated to related metrics. Factors in the detail of the collection of the metric will affect how useful the data will be to evaluating the performance indicated by the metric:

- realism varies by metric inside each test event due to the available test assets and test timeframes
- tests targeted at reducing one set of risks may collect data on other related areas as a side effect
- review of full test artifacts can be mined for "off-target" collections

- off-target metric collections may be at a lower fidelity level then metrics that are included in the risk target of the test

To account for these factors we will need a quality tag to append to our event's cube coordinates for each metric derived or collected from the event.

Here are a few categories for this quality tag:

- SS=Sparsely collected, non-trended, not tagged
- TNTS=Trended for some scenarios/cases but not tagged
- TNTA=Trended for all scenarios/cases but not tagged
- TTA=Trended and tagged for all test cases/scenarios

Each metric will be annotated for each event with the three previous dimensions for the event. For example, if Event A is Proto/Sim, EP/LP, LS in our visualization cube, for one metric it may be Proto/Sim, EP/LP, LS TTA. Another metric can be represented as Proto/Sim, EP/LP, LS, SS.

3.6.7 Putting the Coordinates Together for a Test Event (for Each Metric)

While the combined tag for each test event will be largely common for all metrics, there will be differences for some metrics in the same test. For example, the middleware was completely coded in prototype, but some services are simulated only as black boxes with interfaces to the middleware. As a result the middleware metrics will have a software dimension of Prototype, but the metrics relating to the services may have only "Mod" for the software dimension. Metrics between the services and the software would then have "Mod/Proto."

3.6.8 Assembling the Metrics Matrix: Events in Columns, Metrics in Rows

Add a column for each test event for which data can be found. In the column heading, add a link to the test folder containing the test data and reports. Note the title and dates of each event. While the combined coordinates in the Cube of Realism for each test event will be largely common for most metrics, there will be some variation in the test quality tag. Determine how much of the base event applies to each metric, then determine how well the data for each metric was collected using the test quality tag. Record the combined coordinates and tag in the cell corresponding to the Event column and each metric (in rows). If known, record where, in each test event report or data set, the metric was collected and described. Table 2 depicts a partial matrix; some columns are hidden for brevity (*Why*, *How*, and *Keywords*).

Add in Links to the folder containing the actual data and test plan and report documents if possible

Table 2: An Abbreviated Example of Metrics Matrix Filled in for Test Events

##	Short Name (Used for Tagging)	Metric Title	Test Event A: April 6-10, 2088	Test Event B: May 6-20, 2088
1	BCalls_Count	Blade to blade calls (tagged by service, by process, by user, by case/scenario/time	N/A	[Sim,LP,LS] [SS] C.3.2, Report B
2	HDCalls_Count	Service traffic count to Drives	[Proto/Sim, EP/LP,LS] [TTA] Sec.2.3.2, Report A	[Sim,LP,LS] [TTA] C.3.3, Report B
3	CSWan_Count	Client calls over WAN by client, process, service, platform	[Proto/Sim, EP/LP,LS] [SS] Sec.4.3.2, Report A	[Sim,LP,LS] [TTA] C.3.4, Report B

3.7 Step 7: Determine What Metrics and Events are Missing

Having a metrics matrix from one group alone is not enough. The primary purpose of the matrix is to determine, qualitatively, the coverage of key software performance metrics. The group that arrives at this matrix, however, is often only one voice in the larger SoS organizational structure. Therefore the matrix must be vetted with other groups in the organization to

- assure that no key metric of the software performance of the SoS has been missed
- assure that no test event is missed
- begin a cross-organizational dialog on software performance

It is useful to form a continuing group to vet the matrix, communicate test ideas, assist in planning future testing to fill in gaps that were uncovered in the matrix, and provide recommend methods to improve software performance. The focus is to provide non-adversarial, peer-to-peer communication, not overt direction.

3.7.1 Forming the Software Performance Technical Interchange Group

Charter a software performance Technical Interchange Group (TIG) with select members from the middle to upper middle rungs of the architecture, engineering, and test groups as shown in Figure 18.

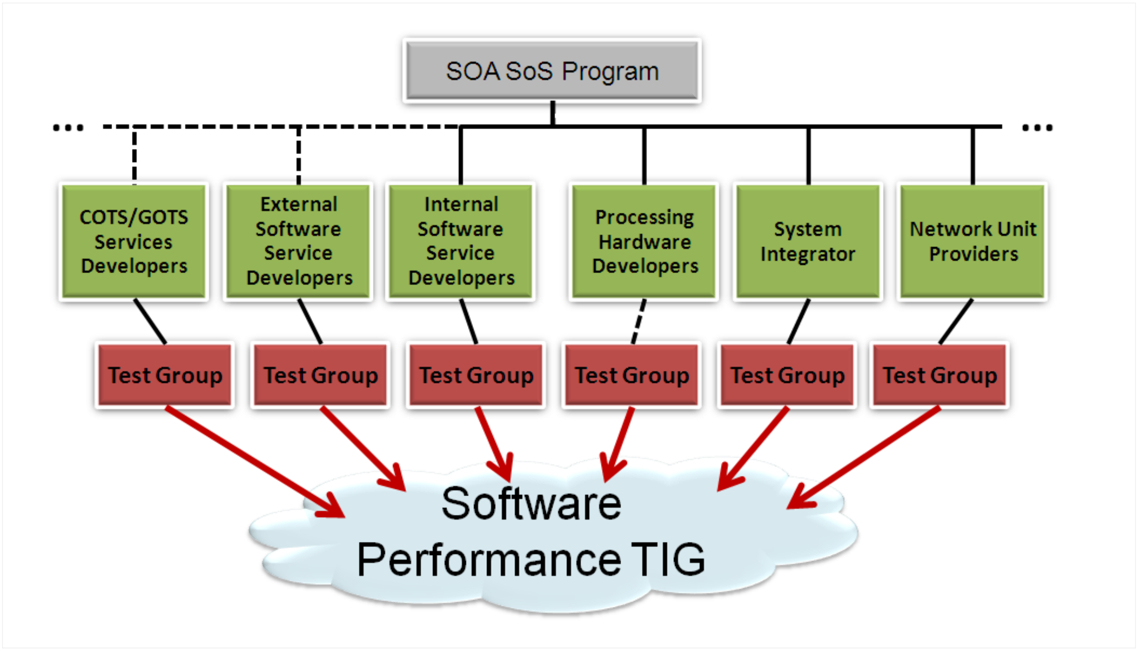


Figure 18: A Notional SWP TIM Stakeholder Group

A sample charter for this group may contain the following goals:

- Review alignment of the software measurement system to address current strategic software performance goals.

- Ratify a set of software performance measures and measurement plan to understand the state of software performance at the system and enterprise levels.
- Improve effectiveness and efficiency of measurement infrastructure to accommodate measurement plan tasks (e.g., data collection, analysis, presentation).
- Improve the understanding and use of software performance measures.
- Improve communication of software performance information to stakeholders and leadership.

Success of this group would be evidenced by the

- enhanced instantiation of software performance measures identified as leading indicators of software performance and risk
- process of managing the software performance from the technical interchange meetings is transferred to testing activities

The most important deliverable, however, is a complete, metrics matrix, continuously updated that is utilized within the design processes.

As stated in Section 1, the matrix is not a quantitative audit, but is, instead, a method to determine the current state of software performance, and a way to improve performance by improving coverage and fidelity in the matrix.

3.7.2 Other Parties

After the metrics matrix is vetted with the TIG, it is important to allow software architecture and engineering group members a chance to view the matrix. Focus on the middle rungs in each organization. Requirements groups may be able to assist with traceability (see later steps) and help with the *Why?* column. Sponsors of the effort should have an opportunity to review the matrix, since they must know the results of their patronage. These other parties can fill in gaps that may be missing and assist with later steps to improve test planning and architecture and design.

3.8 Step 8: Plan Future Tests and Mine Data from Existing Data Sets

In addition to providing a tool to plan future events, creating and vetting the metrics matrix also provides a good opportunity to look for improvements in test infrastructures and processes.

3.8.1 Test Planning: Improving Test Quality and Fidelity

Metrics that have strong architectural and requirements ties, but have few correlated test events in the metrics matrix (or that have events below desired fidelity or scale) are opportunities to improve the design of tests in planning or underway. Determine in the Cube of Realism where the desired state exists. We can improve on one axis or any number of axes to improve the understanding of the performance via the metric. For example, if the current state of a metric is [Proto/Sim, EP/LP,LS] [SS], we could include planning in an upcoming event to

- specifically collect the desired metric
- trend it for time, and tag it to source scenario (in essence improving the test quality tag as defined above)
- seek to use ether all prototypes or early build software
- use all limited production hardware, and/or keep the same limited multiple system scale or improve to a partial SoS scale

It would also be advisable to see if any test events that have no rating in this metric's row could be mined for data at the desired level. Look at the raw data from a test event that is at the [Sim,LP,PS] scale for a similar metric and determine if the desired metric can be derived by processing the raw data. To mature the metric, insert better methods of metric collection into test events as they increase in fidelity and scale.

3.8.2 Test Infrastructures

It is beneficial to use gaps in the matrix or poorly collected metrics ([SS] tagged) to improve testing infrastructure for software performance metrics. The following two areas can be useful:

Collection Apparatus and Systems

Looking at the *How* column in the matrix for metrics and looking at the commonality between these methods is one approach to review systems in use—or that need to be in use--in test labs. Look to see if a test organization consistently produces high-quality tagged metrics and then determine what methods and apparatus they used to achieve the quality collections. In many cases, the same techniques can be flowed across the diverse testing organizations to improve the infrastructure for other SoS areas. Some areas that are common to software performance metrics can provide a starting point:

- Obtain a snapshot from operating systems on blades and other hardware.

  Many operating systems have administration tools that can be captured repeatedly at set intervals (a common time stamp for all test elements and all systems), including process counts, thread counts, error counts, memory utilization, and processor utilization by process. A capture of this display to a text file that can be parsed and tagged with time, then scenarios (by step) and the “blade/PU/system,” can be mined for many software performance metrics. A common data collection server set and network time stamp system are key here.

- Obtain a snapshot from networked devices using SNMP/MIB/SYSLOG/NetFlow.

  Interface utilization for processing unit blade-to-blade traffic, LAN utilization, WAN utilization, network flows by IP address, again tagged by blade processing unit. System, time, and scenarios (by step) are valuable for determining access to slower means over faster means (i.e., staying on-blade versus going off-blade). Correlate these to running processes and memory utilization. Again, a test server and test lab network structure are required in some cases to obtain data as described.

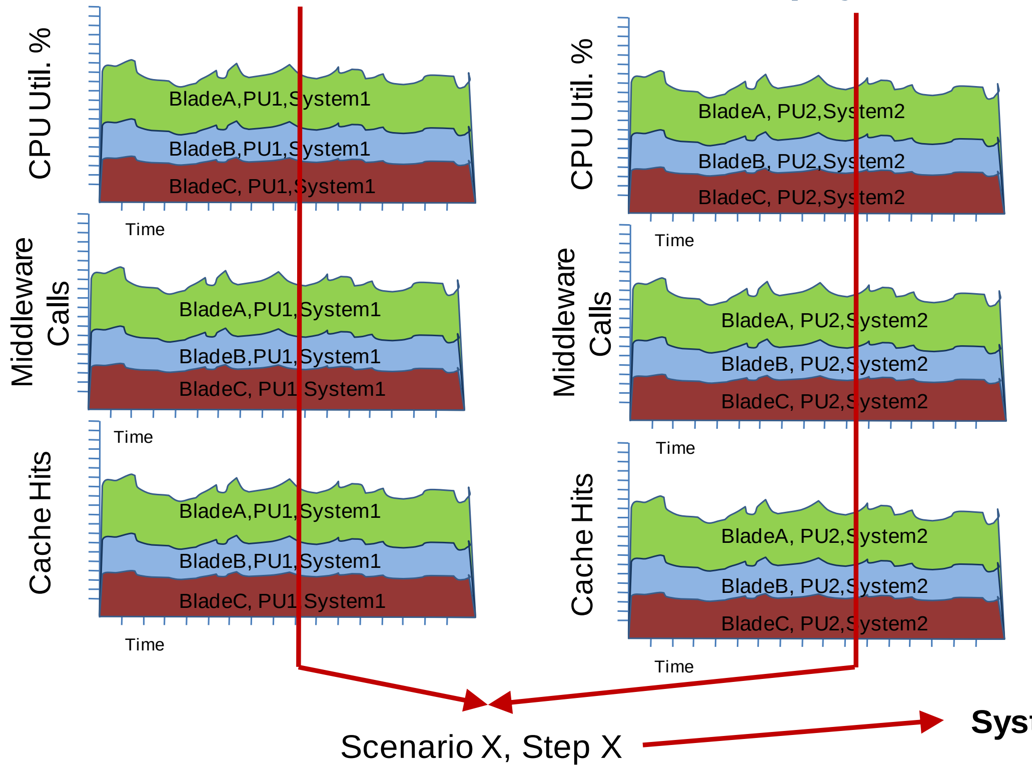
- Store raw and processed data centrally.

  Raw data and processed data should be stored in an area accessible to all testing organizations, if allowed by security. This approach allows other groups to reuse data from previously conducted tests and determine the fidelity and quality of data for their key metrics. Collaboration tools and searchable databases are helpful.

Cross Correlation and Data Mining

Improving the quality tag is a result of planning for better and later use of raw test data. Test data needs to be trended and tagged with metadata to allow later mining, especially if multiple parties or contractors are involved for each system, component, or service. Planning for this mining using the keywords from the metrics matrix can provide payoffs. In some cases, the existing metrics matrix could be fleshed out better by mining previously collected data from events. For example, mining or correlating, with time, a previous test run’s raw data to collect data for a metric in the matrix not previously collected. Tagging future data and storing it in a commonly accessible location will enable future mining of runs from planned tests. See the example in Figure 19.

Which cross correlations have a payoff?



Other correlations

- Regression comparisons?
- Gap analysis; compare w/desired performance

Tie to architecture (design, various levels)

System Architecture; Software Architecture

Figure 19: Mining Previous Test Results Using Cross Correlation

3.9 Step 9: Tie in Architecture to the Metrics

Assuming a software performance metric can be traced up the design to architecture and requirements, the resulting metric data can be used to improve the SoS's overall software performance. Architecture decisions—reflected in design, embodying specifications—determine the performance of metrics against factors; this performance must satisfy requirements at software, system, and SoS levels, as shown in Figure 20. If traceable, two methods can be used to add assessment to architecture: gap analysis (current results versus desired results) and regression comparisons (comparison of current results to previous results).[3]

3.9.1 Feedback and Traceability

With a vetted metrics matrix, it is useful to tie each metric to architecture.

- Use the ties in the *Why?* column to improve performance.
- There are likely no orphan metrics if they tie to a scenario that is reflective of the SoS design; they are just more complex to trace to the architecture.
- Repeated columns of higher fidelity and realistic events improve confidence that the metric is covered and performance is quantified.
- Architecture elements (and design elements) that are tied to performance will generate confidence with successive events.

If not indicated in the metrics matrix, using a tactic such as an Ishikawa diagram or other causality tool can link a metric, or its source scenario, to software designs and architecture decisions, and to system level requirements. Using the optional item in Step 4 above, decomposition of system requirements to software performance requirements can also aid in traceability. If no metrics for software performance yet exist in the project, reversing the steps in 2, 3, and 4 can produce recommended requirements for the SoS.

[3] See also the Requirements Management (REQM) Process Area in the CMMI and CMMI-ACQ (available online at http://www.sei.cmu.edu/cmmi).

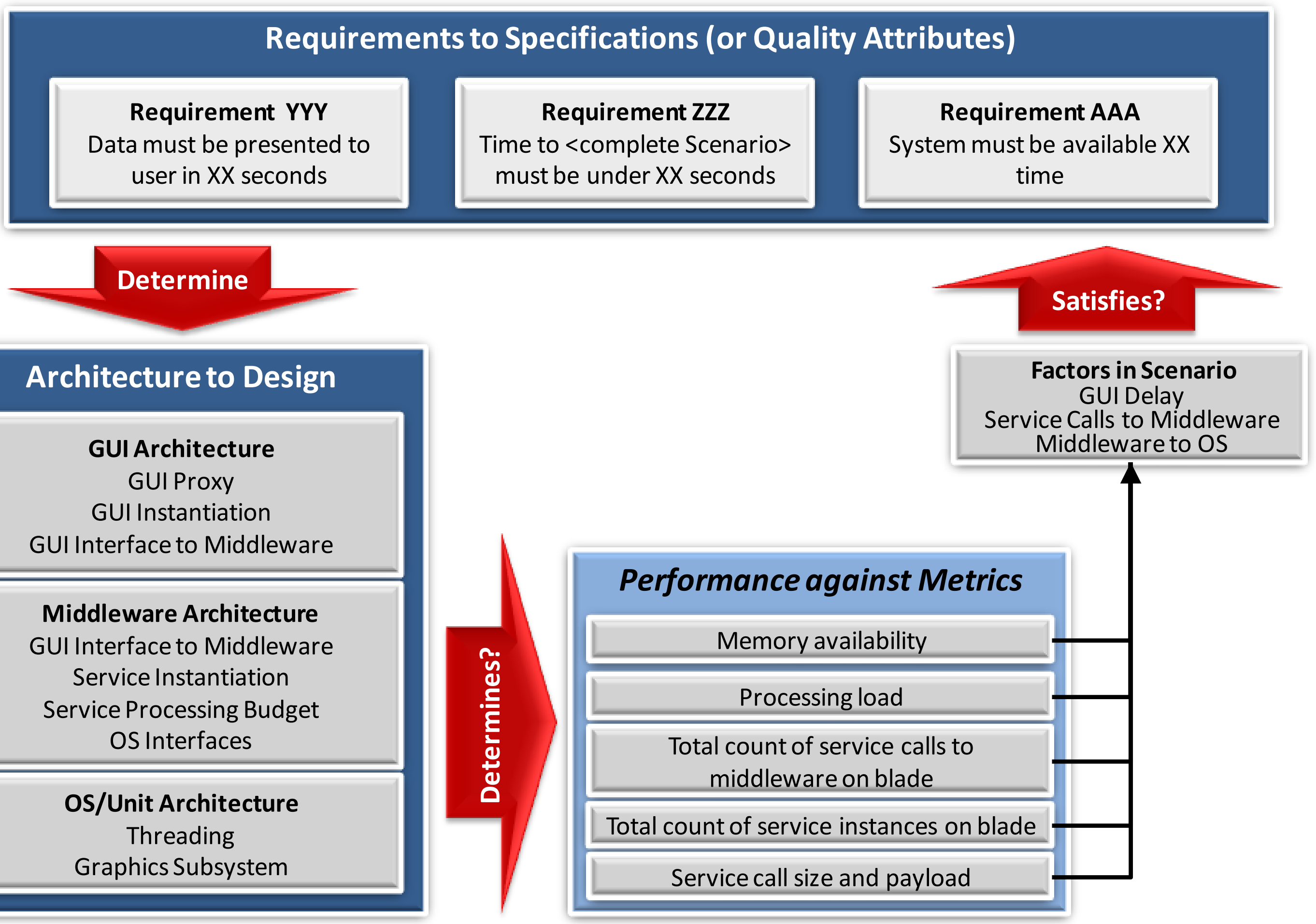

Requirements to Specifications (or Quality Attributes)
Requirement YYY
Data must be presented to user in XX seconds
Requirement ZZZ
Time to <complete Scenario> must be under XX seconds
Requirement AAA
System must be available XX time
Determine
Satisfies?
Architecture to Design
GUI Architecture
GUI Proxy
GUI Instantiation
GUI Interface to Middleware
Middleware Architecture
GUI Interface to Middleware
Service Instantiation
Service Processing Budget
OS Interfaces
OS/Unit Architecture
Threading
Graphics Subsystem
Determines?
Performance against Metrics
Memory availability
Processing load
Total count of service calls to middleware on blade
Total count of service instances on blade
Service call size and payload
Factors in Scenario
GUI Delay
Service Calls to Middleware
Middleware to OS

Figure 20: Example Metrics Tracing

3.9.2 What Do the Metrics Tell Me: Gap Analysis

With the current state and sources of metric data, with test qualities, you can look at the characteristics of the measured data itself and compare these to the desired state (traced from requirements) for each metric. For example, testing of memory availability for the most difficult scenarios, at the worst time interval in each scenario, for the most used blade, may indicate 98% utilization at a system count of 50% of full SoS scale. If the requirement for the SoS is that no blade shall exceed 75% when deployed (at full scale), then it is likely that at full scale it will exceed the 75% maximum processor utilization per blade metric. To achieve the desired state (75%) you can potentially reallocate services, determine using other metrics if orphan threads are being created and not killed, re-examine processing in service components, and so forth. Having the matrix enables you to see what other metrics were also collected in an event (“orphan count at time” being one of them), and combine results to indicate architecture or design strategies to reach desired state. Combining results across tests for the same metric in different scenarios again can illuminate strategies to improve performance.

3.9.3 What do the Metrics Tell Me? Regression Comparison

Regression comparison, which compares the same test scenarios between events of increasing fidelity and complexity or after major software integration events, can indicate if software performance is improving. Scale increases often increase utilization of many resources, increasing the values of metrics for LAN and WAN utilization, calls to services, time to discover services, processor utilization on service hosting blades, middleware use, and the like. Fidelity improvements also provide deeper insight. Inefficiencies in code will be revealed as testing increases in scale and fidelity, and the revelations will have more value with metric collection quality. It may also occur that as software increases in complexity between builds, the software will likewise acquire more sources of either errors or resource utilization. Regression comparison of performance metrics between current and previous builds will illustrate any added errors or sources of utilization. Issues can indicate the need to alter planned test events in addition to changes in design.

3.10 Step 10: Determine the Refresh Schedule

Since the first step of this process is system definition, it is only necessary to refresh the last nine steps at most. The initial effort will consume the most time, but continuous repetition using the software performance TIG as a conduit will keep the metric matrix current. Adding scenarios in each repletion, reviewing decomposition of system requirements to software performance requirements, and reviewing which groups need to be included in the software performance TIG are key to improving SoS software performance.

3.10.1 Repetition and Currency

Keep the metrics matrix current by refreshing it at least at these points (though it will remain durable once well vetted):

- program restructure or refocus
- change in SoS elements related to software performance
- added baseline elements or new requirements.
- after major test efforts (updates to columns).
- metrics added due to scenario development from QAW efforts or other architecture improvement efforts.

Populate the matrix to similar projects and efforts; it can be the kernel for another project’s software performance metrics list.

4 Summary and Conclusion

The method presented in this report is qualitative, designed to begin the effort of determining software performance for a service-oriented architecture embodied in a system of systems. The primary artifact, the metrics matrix, is a living document, coordinated across the SoS organization using technical interchange meetings and other methods. The matrix can show which events have occurred, indicate if and how well they assessed each software performance metric, and identify gaps that can be filled in further testing events. Tying requirements to the desired values of each metric can demonstrate improvement. Tracing metrics to architecture and design can improve software performance.

4.1 Summary

Understanding software performance for a SOA SoS system is complex and managers need to

- understand the system and its performance-affecting levels
- derive a metrics list from scenarios and other sources
- tie in test events to create the metrics matrix
- identify a way to circulate the matrix by understanding the organization
- distribute the matrix and metrics testing results to architecture
- keep the matrix current (otherwise status will be unknown)

These needs can be systematically remediated, at least in part, through the use of the 10-step process in this report:

1. Develop a SOA SoS Performance View.
   Develop a SOA SoS layout performance view.
2. Review Key SOA SoS Resource Limiters.
   Review key resource limiters from the layout.
3. Develop Sample Scenarios and Determine Respective SWP Impacts.
   Develop a series of scenarios, then list the performance impacts in each step and section of the scenario.
4. Create an Initial List of SWP Metrics for Your SoS
   List the metrics that affect or quantify the impacts to software performance in each scenario, and combine all the impacts into a common list of metrics.
5. Add Required Software Performance Metrics from Other Sources
   Add required software performance metrics from other sources (e.g., sub-contractors).
6. Determine All Test Events and Rate Their Maturity
   Determine all test events (including integration events) that have occurred at every level and in each organization in the SOA SoS. Rate the fidelity of each event for each metric. Add to the columns of the metrics list to form the “metrics matrix.”
7. Determine What Metrics and Events are Missing
   Circulate and vet the metrics matrix throughout all architecture and engineering test organi-

zations in the SOA SoS, asking: What metrics or test events are missing? Update the matric-es.

8. Plan Future Tests and Mine Data from Existing Data Sets
   Use the populated and vetted metrics matrix to plan future events. Identify any gaps exist in infrastructure, test methods, or test plans.
9. Tie in Architecture to the Metrics
   Using traceability, tie-in architecture to improve software performance of the SOA SoS. What elements are tied to each metric?
10. Determine the Refresh Schedule
    Determine how often the last nine of these ten steps will be repeated.

4.2 Notes and Conclusions

SOA SoS projects are often under pressure to prove how well the design of the various SoS components will perform, even in very early program stages. This 10-step process and the ensuing metrics matrix can provide the means to obtain SWP management improvement, even in the absence of full-scale testing. It is a qualitative method to better understand the state of SWP, and an indicator toward improving the design of future test events. It provides a quick link to sources, and most importantly, its recommendations can begin a cross-organizational dialog inside the SoS organization at the key working levels. SoS organizational diversity is a challenge along with architectural complexity; the use of this process and toolset can assist managers in overcoming these challenges to achieve desired technical performance.

Appendix—20 Must-Have Software Performance Metrics for a SOA SoS

#	Short Name	Metric Title	Why?	How?
1	Bcalls_Count	Blade-to-blade calls (by service, by process) by client, by system.	Limiting calls from blade-to-blade reduces time (due to backplane use)	Backplane monitoring via processing unit against process monitor
2	HDCalls_Count	Client /service/application traffic (+ count) to drives	Which services, applications, clients of applica-tions are hitting the RAID or flash often. The more often RAM is used in lieu of the drives, the quicker the application will run.	Process-message snapshots and parse (or logging parse) for OS+bus capture (log parse)
3	CSWan_Count	Client calls over WAN by client, process, service, platform	WAN has a high time cost. The fewer calls over the WAN (i.e., ad hoc network) the quicker an application will run.	Process-message snapshots and parse
4	Error_Count	Error logging and count for blade and processing unit	Which combinations of services and clients+ apps under which conditions cause issues at the system and application level. SYSLOG, SNMP, OS Capture.	Instrumentation of code w/process to service to above metrics + log parser+ statistical analysis
5	IO_count	Count of uses of off-blade, and off-processing unit resources	Used to derive proxy and other efficiencies. Can software (per application/client/proxy) consoli-date requests to the drives? Can it minimize access to off-processing unit devices? Can re-questors minimize requests to a service on a blade?	Repeated capture from OS (blade) and MIB/SNMP from processing unit LAN router

#	Short Name	Metric Title	Why?	How?
6	Inst_count	Instances/client/situation, instances/service/situation	Check for process clean up, avoid hung processes, minimize instances	Process-message snapshots and parse
7	IBPrior_Compliance	Prioritization intra-blade vs. time	Check against system (end to end) QoS. Is the software controlling itself against the processing priority.	Repeated capture from OS against process/service identification data
8	P_Count	Process count/blade /system over time	Check for process clean up, avoid hung processes, minimize instances.	Repeated capture from OS
9	Z_count	Zombie count/Instances over time	Check for process clean up, avoid hung processes which can consume resources and create instability.	Repeated capture from OS
##	Short Name	Metric title	Why?	How?
10	CPU_Util	CPU utilization (by client, service, application) over time	Prevent overutilization, prevent resource hogging/application	Repeated capture from OS
11	HDPart_Ut	Partition/disk usage over time/scenario/factor	Avoid overfilling partitions (which can slow or stop a system). Determine which situations stress disks.	Repeated capture from OS
12	LAN_Util	System LAN utilization	Prevent overuse of LAN on system, watch for processes that could be done in blade in lieu of over LAN.	SNMP MIB from Routers
13	RAM_Util	RAM utilization (by client, service, application) over time	prevent overutilization, prevent resource hogging/application.	Repeated capture from OS
14	Orphan_Thread_Count	Count of threads and orphan threads	Count of threads and orphan threads at each level trended.	Repeated capture from OS via instrumentation
15	Software Scenario_delay (and utilization by time unit)	Software level scenarios (timing, resource utilization, effectiveness)	Combines the above metrics to determine if the Software will meet requirements.	Simulation to full scale test using metrics above

#	Short Name	Metric Title	Why?	How?
16	System_Scenario_Delay (and utilization by time)	System level scenario (timing, resource utilization, effectiveness	Combines the above metrics to determine if the system will meet requirements.	Simulation to full scale operational test using metrics above.
17	MiddlewareCall_Count	Calls to middleware over interfaces vs. time vs. scenario	The more often a process can complete internally (or consolidate calls to middleware) the quicker it will run.	Process-message snapshots and parse
18	Time_Scenario	Timing (sequence or scenario based)	Ability to meet Requirements at the System and Software Level.	Task time keeping
19	SpecInT_Level (or similar processing unit metric)	SpecINT2000s vs. application vs. processing unit/processing unit	Match resource budget in performance.	SpecINT measurement tools.
20	WAN_Util	WAN utilization (note: software perspective usage, not network perspective)	Software overuse of network (minimize over WAN calls, minimize software sending large requests, etc.).	Network utilization monitors (SNMP MIB from network radios/routers)

References

URLs are valid as of the publication date of this document.

[Barbacci 2003]
Barbacci, Mario R.; Ellison, Robert J.; Lattanze, Anthony J.; Stafford, Judith A.; Weinstock, Charles B.; & Wood, William G. *Quality Attribute Workshops (QAWs), Third Edition* (CMU/SEI-2003-TR-016). Software Engineering Institute, Carnegie Mellon University, 2003. www.sei.cmu.edu/library/abstracts/reports/03tr016.cfm

[Clements 2001]
Clements, Paul; Kazman, Rick; & Klein, Mark. *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley, 2001 (ISBN: 0-201-70482-X). http://www.sei.cmu.edu/library/abstracts/books/020170482X.cfm

REPORT DOCUMENTATION PAGE		Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.		
1. **AGENCY USE ONLY** (Leave Blank)	2. **REPORT DATE** April 2010	3. **REPORT TYPE AND DATES COVERED** Final
4. **TITLE AND SUBTITLE** Characterizing Technical Software Performance Within System of Systems Acquisitions: A Step-Wise Methodology		5. **FUNDING NUMBERS** FA8721-05-C-0003
6. **AUTHOR(S)** Bryce L. Meyer & James T. Wessel		
7. **PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)** Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213		8. **PERFORMING ORGANIZATION REPORT NUMBER** CMU/SEI-2010-TR-007
9. **SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)** HQ ESC/XPK, 5 Eglin Street, Hanscom AFB, MA 01731-2116		10. **SPONSORING/MONITORING AGENCY REPORT NUMBER** ESC-TR-2010-007
11. **SUPPLEMENTARY NOTES**		
12A **DISTRIBUTION/AVAILABILITY STATEMENT** Unclassified/Unlimited, DTIC, NTIS		12B **DISTRIBUTION CODE**

13. **ABSTRACT (MAXIMUM 200 WORDS)**

The characterization of software performance (SWP) in complex, service-oriented architecture (SOA)-based system of systems (SoS) environments is an emergent study area. This report focuses on both qualitative and quantitative ways of determining the current state of SWP in terms of both test coverage (what has been tested) and confidence (degree of testing) for SOA-based SoS environments. Practical tools and methodologies are offered to aid technical and programmatic managers:

- A stepwise methodology toward SWP selection
- SWP and system architecture design considerations
- Resource limiters of SWP
- SWP and test event design considerations
- Organizational and process suggestions toward improved SWP management
- A matrix of measures including test fidelity and realism levels

These tools are not complete, but do offer a good starting point with the intent to encourage contributions to this growing body of knowledge.

This report is intended to benefit leaders within the varied acquisition communities, Program Executive Offices, and Program Management Offices. It provides detailed guidance for use by technical leadership as well.

14. **SUBJECT TERMS** Software performance, SWP, SOA, service-oriented architecture, system of systems, SoS, acquisition			15. **NUMBER OF PAGES** 64
16. **PRICE CODE**			
17. **SECURITY CLASSIFICATION OF REPORT** Unclassified	18. **SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	19. **SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	20. **LIMITATION OF ABSTRACT** UL