15th ICCRTS
"The Evolution of C2"

Assessing SOA Performance

Topic: Track 8

Coimbatore Chandersekaran
Institute for Defense Analyses
4850 Mark Center Drive
Alexandria, VA 22311

Kevin Foltz
Institute for Defense Analyses
4850 Mark Center Drive
Alexandria, VA 22311

William Simpson
Institute for Defense Analyses
4850 Mark Center Drive
Alexandria, VA 22311

POC: Kevin Foltz
Institute for Defense Analyses
4850 Mark Center Drive
Alexandria, VA 22311
kfoltz@ida.org

Assessing SOA Performance

Abstract

The U.S. Air Force is building a secure service oriented architecture (SOA) environment: its Singularly Managed Infrastructure with Enterprise Level Security (SMI-ELS). Some initial ideas for this architecture have been outlined in previous papers [1] [2]. This concept is now implemented as an initial infrastructure build (IIB), with plans to go operational. This paper discusses the current challenges in assessing this architecture. Because this architecture primarily uses message layer standards, traditional tools and methods are not adequate for assessing functionality, security, and performance. New goals and tools are required to address new challenges and limitations. Performance tests focus on assessing scalability of the architecture based not on individual links but on entire end-to-end network flows. This scalability assessment necessitates a more complex test architecture, requiring multiple simultaneous test points using tools that emulate different components in the system using different interfaces and protocols. Test plans, tools, and procedures are given for doing such an assessment. Lessons learned, a wish list for new tools, and future plans are given based on current challenges.

Introduction

In any command and control (C2) network there are some basic requirements. First, the network must transmit information in a timely manner. Commands that are delayed are less effective than commands that are sent in real-time, especially when quick and accurate actions are required. Second, the information sent must be received correctly. This requires not only error checking and correction, but cryptographic guarantees that the message received is the message that was sent, using tools such as hashing. Third, the command and control signals sent must be trusted. Included here are standard security services like authentication. Fourth, confidentiality of messages must be preserved in potentially hostile environments. Fifth, auditing is important for reviewing events after they happen. Sixth, non-repudiation is important to guarantee commands sent can be traced back to their originator. Seventh, necessary precautions must be taken to ensure actors can only take actions that they are authorized to perform. Low-level infantry should not be able to command forces in the way that high-ranking generals do.

A sample application is the Deployment Readiness Service (DRS), which is used to determine whether pilots are ready for missions. There are many factors involved in this decision, including training, credentials, health, and schedule availability. To deploy a pilot on a mission all of these issues must be considered. Doing this “manually” by checking each individually puts a huge burden on a commanding officer, since it requires knowledge of all the systems involved and the rules for access. It also requires credentials for each system and possibly training and other activities to stay current. The DRS web service is a way to capture all of this into a service accessible through a web browser, which only requires a CAC card (and the appropriate credentials) for access and use. This offers a large savings in time and effort to the commander who wishes to schedule deployments, and by using the SMI-ELS security infrastructure it also provides a more secure way to do this task.

In order to provide the functional and security requirements of today's forces, the Air Force is looking to the commercial world, where a service oriented architecture is built around simple, reusable web services rather than large, complex single-use applications [3][4][5]. The SOA concept has proven successful in the commercial world, and the Air Force would like to leverage the work that has already been done with web services and SOA design to provide similar benefits to the military.

The main problem with adoption of commercial SOA implementations is the enhanced security needs of the Air Force. While commercial entities can write off security problems against their bottom line, the military has a less tolerant attitude toward security issues. Also, given the scale of operations of the U.S. military it is worthwhile designing security into the system from the beginning rather than trying to patch an inherently insecure system. The amount of work required to change the infrastructure is so large that work done now in security, while still in the prototype stage, will pay dividends for a long time to come. The use of a formal security model allows automated testing and verification [6], which is also important for military systems that have extensive certification and accreditation requirements. A formal security model must be applied uniformly across the entire enterprise. The challenge and novelty of the Air Force's work is the application of a formal, high-assurance, enterprise-wide security model to a service oriented architecture.

The architecture of SMI-ELS includes security services and other core services. Security services are well-integrated into the architecture itself and include authentication, authorization, confidentiality, integrity, and auditing. Other core services allow discovery of assets within the architecture. These are special services because they span the entire enterprise and are managed at the enterprise level.

A standard service that operates within this architecture can be one of two types: an exposure service or an aggregation service. Each is managed by a Community of Interest (COI), which also manages data provided by the services and access rights to this data. Exposure services simply access data stores and provide results of such accesses. They expose data to a requestor. Aggregation services call other services, which can include both exposure services and other aggregation services. They aggregate the results of different services for a requestor.

Exposure and aggregation services provide the useful functionality to the end user within the architecture. The discovery service is used to find such exposure and aggregation services, and the security services are used as part of accesses to an exposure or aggregation service.

In a SOA environment it is expected that a single web service request may involve a series of other web service calls. The challenge is to measure what is happening in the system when such a call is made. Standard test tools do not understand message layer data, so WS-* standards (WS-Federation, WS-Security, WS-Trust, etc.), for example, cannot be tested easily. Performance is no longer a single response time number, but must be dissected into its individual components to assess how long each service in a call tree is taking and whether each is performing within specifications.

This paper looks at assessing performance in the SMI-ELS SOA environment. Although functionality and security are also important issues to test, performance assessment makes the strongest demands on using an enterprise-wide view, which makes it a key challenge for SOA assessment. The key challenge

and novelty of this work is doing SOA performance measurement while working within the strict security constraints of the SMI-ELS environment. The security paradigm is new, and based on new and emerging standards, protocols, and software components. The testing must be complex enough to provide good performance numbers, while being simple enough to be implemented with existing tools and flexible enough to adapt to architectural and component changes.

SOA Performance Assessment

Performance assessment of a SOA system is not a simple task. There are many different components of performance for a single service request. It is easy to measure end-to-end performance since this simply involves measuring load and response time at one or more requestors. However, end-to-end response time does not help in determining whether slow responses are due to network issues, security issues, aggregation service issues, exposure service issues, or calls to external data sources. Each component of the flow has its own service level agreement, so the end-to-end performance is a function of all of these. In order to test these service level agreements, finer granularity is needed than end-to-end response time.

To perform these finer granularity performance tests, a tool is needed that can measure response times not just at the beginning and end of a request, but also for all subsequent web service and other calls made during the request. This requires a tool that can understand web protocols and take appropriate actions based on these protocols. It must be able to select authentication certificates for use with Secure Socket Layer (SSL) sessions and correctly encrypt and sign messages as necessary. It must use Simple Object Access Protocol (SOAP) over Hypertext Transfer Protocol (HTTP) for WS-* standards-based requests.

To measure response time for each hop in the flow, which is necessary to measure all service level agreements, the test tool must be present at each hop. The test tool must be able to emulate not just a browser, but other nodes in the system, such as web applications and web services. It also must be able to correctly correlate traffic sent at one node to that received at another node in order to correctly break down end-to-end times into their individual component times.

It is tempting to simply test each hop in the flow separately by creating requests at one node and measuring response time. By doing this at different steps along the flow, different times can be computed, and the step-by-step performance can be computed by adding and subtracting times from these individual tests. However, this type of testing fails to account for system load. By running isolated tests, the effects of system load on one area are not taken into account in other areas. For example, one could test end-to-end response time for a browser request to a web application, including all network latency and security functions. Measurements could then be made for each step in this sequence in isolation. However, this fails to consider that all steps must share the same network interface, and hence the overall load is higher during an end-to-end test than during the single step test. For the web service call from the web application, this problem is exacerbated since the web application is used in two ways simultaneously: first as a service provider to a requestor, and second as a requestor to a web service.

An effective test must generate an end-to-end load on the system, while simultaneously measuring the step-by-step performance and correlating these steps together correctly, to break down the end-to-end times into their components.

In addition to simply measuring latency and throughput, performance involves looking at user loads and typical activity patterns. For example, if users will typically log in, make a service request, and then log out, this generates a very different system load than users who log in every morning, work all day using a small set of services, and then log out at the end of the day. Performance of user login and any related service calls must be tested to handle peak loads in the morning. Performance of diverse service accesses must be measured, where users must repeatedly invoke discovery and security services.

For load testing on services, a simple and effective test is to ramp up the request rate until the latency suffers significantly or errors start occurring in responses. Throughput and latency are calculated based on actual test measurements, and plots are produced to show latency vs. throughput. Test can be done using a browser interface connected to a web application, or tests can be done directly using a web service client. Browser-based requests require a longer sequence of messages since a session with a web application must first be established before making direct web service calls. Using a web service client does not have this overhead. However, a test tool must be configured to act as a web service client, which takes more setup work. In the end, both access methods should be tested if provided by the web service.

One way to test the time attributable to a service itself is to create dummy exposure and aggregation services. The dummy exposure service takes an input and immediately responds with a fixed response. The dummy aggregation service calls other dummy services and then immediately responds. The purpose is to have services with no additional latency beyond the network latency and security services for comparison to more substantial services. This provides another way to break down response time into its components. The service-specific time is simply the difference between the real service call sequence and the dummy call sequence. To determine the dummy call time, the dummy aggregation services and dummy exposure services take the place of all real services in the call tree such that the tree structure is maintained. For example, if A calls B and C, then the dummy test would have one dummy aggregation service calling two dummy exposure services. If done right, only one of each type of service needs to be implemented. Arguments passed to the first service can be used to complete the rest of the call tree recursively.

Performance goals differ for security services, core services, exposure services, and aggregation services. Security services are utilized for all initial service requests. As a result, these will see the highest frequency of use in the system. These must be scaled for enterprise-wide utilization, and latency must be minimized as much as possible. Total latency must be on the order of a couple of seconds or less. Caching can be used to improve repeated service calls, but the latency must be kept low since it is the baseline upon which all request times are added. Core services are also used by all enterprise users, so they must offer good performance under high loads.

Exposure service performance is COI-specific. It is important to note that the exposure service can only offer performance guarantees based on network performance guarantees and security service guarantees. The total end-to-end service latency and throughput will be set by the COI and will incorporate the network and security latency plus any data store calls and processing required by the service itself. Aggregation services have all the same considerations of exposure services, but must also account for the performance of other aggregation and exposure services that they call. Again, these metrics are set by the COI managing the aggregation service. However, instead of extra latency considerations happening up front, they happen before and after the service is called since other services called by an aggregation service will have their own service level agreements which must be considered.

Security tests offer the greatest opportunity for component testing. These services do not rely on other services, for the most part, so their performance can be fairly accurately gauged by isolated tests. Core services should be tested with the security services since the two will be invoked together. Caching of security information allows multiple service invocations per security invocation, so component testing of core services is also valuable, but typically it is expected that many users will use discovery services for short periods of time, which will require heavy associated security service loads.

Exposure services in general should be tested as part of the entire system, but the time from the service request to service response should be the focus of performance testing. The exact metrics will be service-specific, but the idea is that the COI can only control what happens between the time a request arrives at the service and the time a response is given, not the security or network issues that happen before or after this.

For an aggregation service, this separation is more difficult since the network and security times occur not just outside the service activity, but for all calls to other services made by the aggregation service. This is difficult because multiple calls can be made to multiple other services simultaneously, so timing issues can be difficult to diagnose. For example, if the security services are running very fast during test, end-to-end performance will look very good. However, this performance may degrade to unacceptable levels later even though the security services are still within specifications. Testing should identify the non-security and network latencies and construct a worst-case picture based on these times and the worst possible network and security times that are still within specifications. For complicated service call trees, this can be difficult, but it enables verification of whether the service is behaving properly. It also allows identification of any external services that are not meeting their service level agreements.

Beyond testing individual services and collections of services for performance, SOA testing must assess the scalability of the underlying architecture to handle increasing numbers and request loads of services. Scalability testing does not focus as much on the actual performance numbers as it does on what components are showing performance degradation. Scalability testing is concerned more with relative performance than absolute performance. The question is “If I could add one more of some component to the infrastructure, what would that component be?” Scalability assessment requires step-by-step performance numbers since end-to-end numbers only give the sum of the components and do not identify which component is the bottleneck.

Test Tools and Procedures

For performance testing, it is almost impossible to test without good tools. The ability to generate large loads and make accurate measurements that are correlated with each other is a difficult task. The basic functions of a good test tool are to generate a realistic request load and record each request's response time and other metrics. However, more is needed for SOA testing. First, the tool must be able to pause at redirect requests since these are often used in browser-based flows with federation. A desired capability here would be the option to automatically act like a web browser and complete the redirect. However, allowing the user to write code to determine what actions to take would allow testing to be more flexible. The tool should be able to pause at both HTTP GET redirects and HTTP POST redirects since either one can be used. For a review of the differences between GET and POST, see http://www.cs.tut.fi/~jkorpela/forms/methods.html#fund.

Another requirement for a good performance test tool is the ability to capture and correlate data from multiple nodes in a system. Because a service request is distributed and touches many different nodes, a distributed test tool is needed to accurately measure system performance. Simply measuring performance at each node is not enough. The tool must be able to determine which request at one node corresponds to which traffic at every other node. This allows traceability of traffic throughout the system, which is important for measuring statistics on step-by-step latency within flows and not just across all instances of a particular step.

Current SMI-ELS testing does not have a test tool that completely satisfies all requirements. It is not clear that any existing tool does. The current tool does not stop at HTTP GET redirects and cannot be distributed in the system in a way that allows correlation of data. These issues make certain tests more difficult, but they are not critical. Much of the performance testing is still possible, and future releases of the tool will address its shortcomings.

Some methods for testing performance of services are given in [7]. However, these either test in isolation, require changing the services under test by redirecting them to known test service, or rely on very coarse data, such as CPU utilization. None of these give a detailed and accurate picture of the real system under a realistic load. We propose a method that will accomplish this using a simple test tool.

Figure 1 shows the plan to make step-by-step measurements using only an end-to-end measurement capability. In this case the test is for a browser access to a web application, which calls a web service. The nominal flow is shown by the blue boxes and blue arrows. This is a method developed to address the shortcomings of the Air Force's current test tool, but this technique can be applied in general to extend a test tool's capabilities from single-hop performance measurement to hop-by-hop measurement. This can be especially useful for smaller projects where available test tools may be limited in capability.

To test the performance of the web application and exposure service, two test nodes are set up, shown in green. One emulates the browser (Test1), and the other emulates the web application (Test2). A large load is applied by Test1 to the web application. This stresses the entire system and creates a realistic system-wide load. The end-to-end response time can be measured by Test1. A smaller set of

requests is sent to the second test node, which acts like the web application and makes service calls to the exposure service. The response time of the exposure service can be measured by Test2. In practice, both test nodes could actually be implemented on the same machine using the same tool, but for simplicity they will be thought of here as separate boxes emulating specific requestors or services.

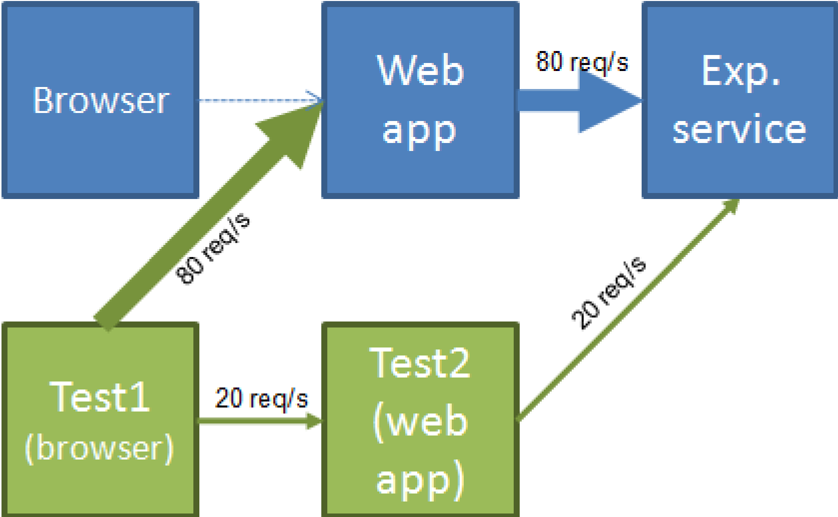

Figure 1 Capturing step-by-step performance data for a web-browser call to an exposure service.

This test setup does not correlate requests and track them through the system, so it is not an ideal setup. However, it shows how to make more detailed measurements than simple end-to-end measurements even with limited tools. The request rates are given as 80 requests per second for the primary load and 20 requests per second for the secondary load. The larger the primary load is, the less error the secondary load introduces through its skewing of the load distribution in the real system. However, the smaller the secondary load is, the fewer data points there are to compute a good average for the exposure service performance. Running long tests can give good averages with low load skewing. These concerns must be balanced for any particular test.

For an aggregation service, or a more complicated call tree with multiple aggregation and exposure services, the same basic approach can be used. Figure 2 shows a test setup to measure times for a web service client call to an aggregation service that calls an exposure service and another aggregation service, which itself calls two exposure services. As the call tree gets more complicated, more test nodes are required and more splitting of the total load is required.

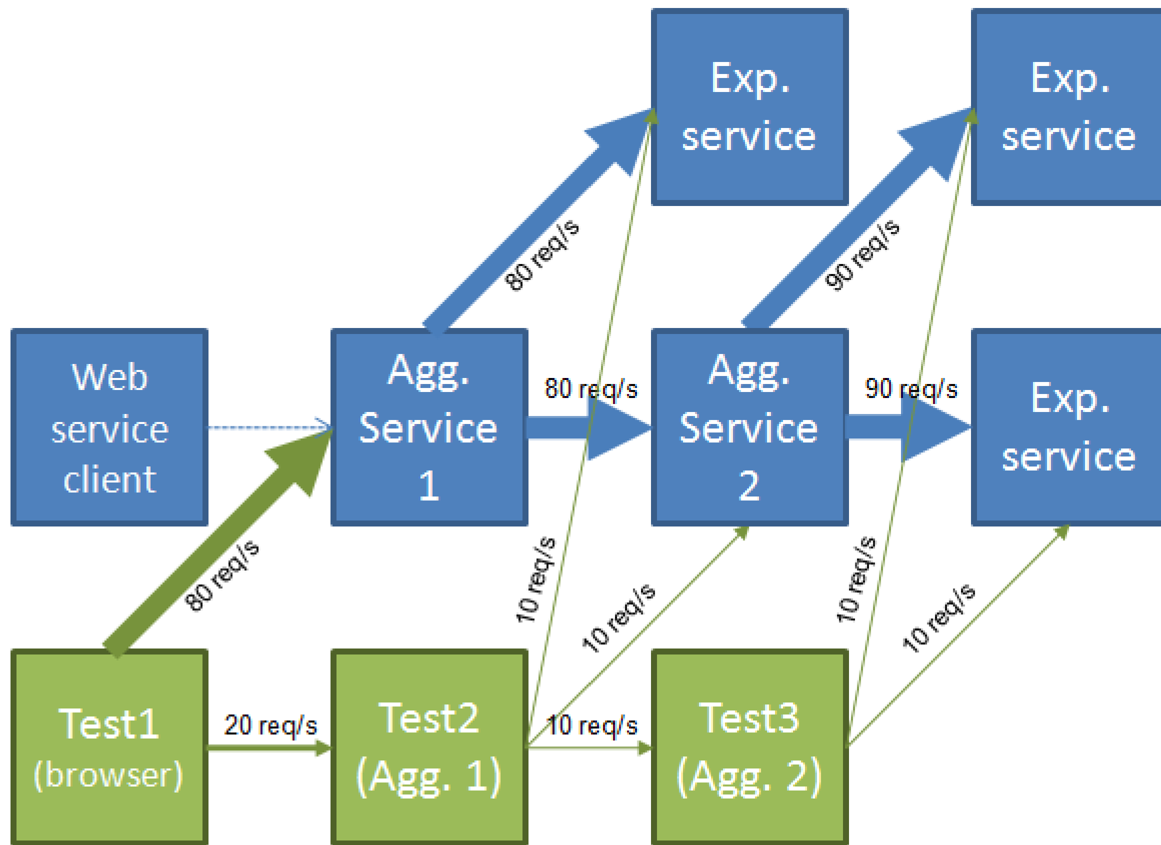

Figure 2 Capturing step-by-step performance data for a web service client call to an aggregation service.

One thing the previous test setups do not address is the network latency and security services performance. To assess whether service level agreements are being met by an aggregation service, for example, the total time spent waiting on the network and security services must be computed. For a large call tree these can be significant. The relevant network and security latencies should be subtracted

from the numbers for the services themselves to provide latency numbers for each service. One issue becomes determining which latencies are “relevant” and which are not.

Another way is to simply compute all network and security latencies and then remove them from the flows. Using the same flow as in Figure 2, Figure 3 shows sample performance data computed from a test on this network. Total response times at each node are indicated below the nodes, internal processing time is indicated within a node, and network and security times are indicated on links. In this test the total time seen by the browser is 20 seconds. This is broken into 18 seconds for aggregation service 1 to return, plus 2 seconds of network and security time in calling aggregation service 1. Of the 18 seconds at aggregation service 1, 1 second is spent internally, leaving 17 seconds for aggregation service 2. This is broken into 15 seconds for aggregation service 2 plus 2 seconds network and security time. These 15 seconds consists of 1 second internal time plus 14 seconds for exposure service 2 and exposure service 3. We see that exposure service 3 is the bottleneck, with the 14 seconds being broken into 12 seconds of network and security delay plus 2 seconds internal time.

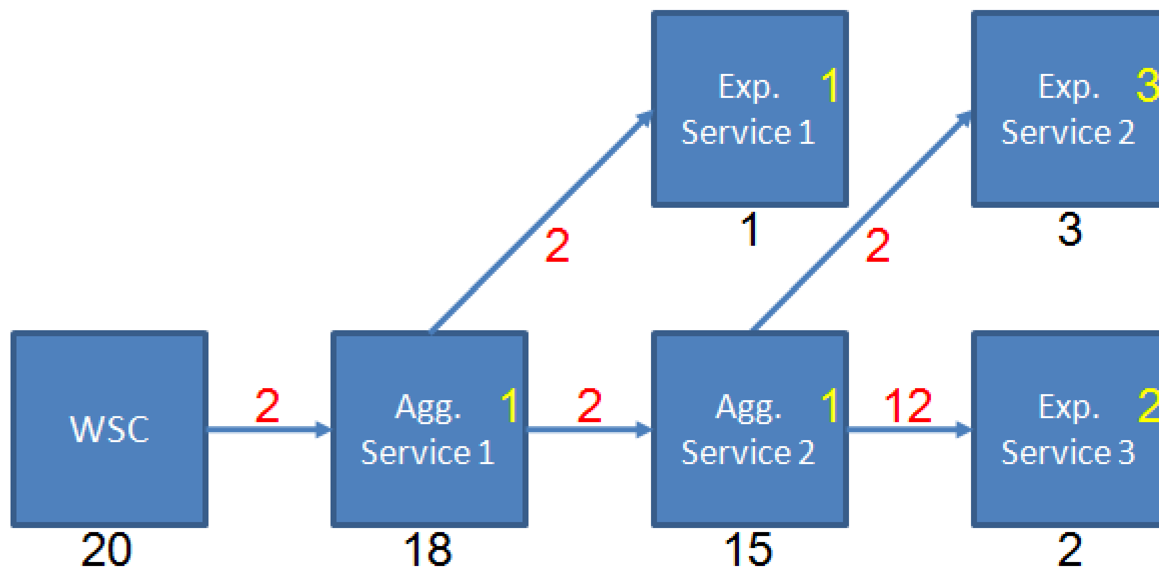

Figure 3 Sample performance test data.

If the service level agreements are set at 2 seconds for network and security time and 3 seconds for internal time at each service, we can compute the maximum end-to-end time as 2+3+2+3+2+3 = 15 seconds. It is clear that the actual end-to-end time of 20 seconds appears to be out of specifications. However, if we remove the network and security times, we get the following times:

Agg1 → Exp1: 1+1 = 2 seconds

Agg1 → Agg2 → Exp2: 1+1+3 = 5 seconds

Agg1 → Agg2 → Exp3: 1+1+2 = 4 seconds

All of these are within the overall service internal time specifications of 6 seconds for the first call sequence and 9 seconds for the second call sequence. Each individual internal time for each service is

also within the specification of 3 seconds. So, we see that even though this particular test gave poor end-to-end performance, the performance of the services under test was actually acceptable.

The opposite effect could happen if the network and security time for all links was 1 second and the service specific time was 5 seconds for exposure service 1 and exposure service 2. In this case, the overall time of 10 seconds is within specifications even though exposure services 1 and 2 are out of specification.

These examples assume that service calls are done in parallel. This is not always true. Service calls from an aggregation service could just as easily be made in series. Combinations of series and parallel calls can also exist. In this case the total time is not computed as the worst case branch of the call tree. For sequential calls the time is the sum of the subtrees. If we assume Figure 3 describes a sequential call tree, then the total times of 20, 18, and 15 for the web service client and aggregation services would change to 28, 26, and 20. The service level agreement would set a threshold of 25 seconds in this case. Again, the overall performance is out of specifications even though each service is within specifications.

For a combination of sequential and parallel calls, the appropriate combination of calculations must be done.

These examples illustrate two types of problems that can occur. First, if the test network and security functions are very fast, close to zero latency, then even poor performance by the services will look good. The end-to-end times might fall within specification even though the individual services are out of specifications since the network and security help make up the difference. When moving to an operational setting, where network and security latencies are higher but still within specifications, the service will not meet specifications.

Second, the network and security services could be especially slow, so performance would look bad even though the aggregation services and exposure services are well within specifications. In this case, the services would be held up in testing when they are actually ready to be fielded.

In order to be sure that these problems are not occurring, accurate step-by-step performance data must be computed. For the test flows described above, measurements must be taken on both sides of a transaction since this allows computation of network and security latency, which can be subtracted from service response time.

Results, Lessons Learned, and Looking Forward

Testing can show value in different areas. First, it can be a good reality check of what is actually implemented versus what was planned. The design team and the implementation team often think they agree on what is happening, but the reality is that things get misinterpreted. Testing provides a good way to examine the entire system as a whole at one time instead of fixing things piecemeal without doing an overall assessment.

Another value of testing is in performance and scalability assessment. Until the system is stressed with a given load, it is simply a matter of guesswork as to how it will perform. This is especially true in a SOA

environment, where individual components can test well in isolation, but system-wide issues can emerge when they are connected and an end-to-end test is done. Identifying bottlenecks early allows vendors to fix bugs, administrators to adjust settings, and acquisition managers to plan for hardware and software license purchases.

One lesson learned, or perhaps reinforced, from testing is that tests must be written at multiple levels of detail. Original test plans consisted of a long list of detailed tests. However, with a constantly changing architecture and implementation, these tests quickly became outdated. Only the higher-level and more generic tests survived the test of time, with their details changing as needed to adapt to the implementation. This issue was addressed by writing a more thorough high-level functional description of tests to be done while providing detailed test scripts tailored to particular products and configurations. Hence the functional tests are constant even if the actual scripts change over time. This keeps the overall intention the same as the system changes.

A lesson learned for service testing is that it is difficult to test a service in isolation from its hosting environment. The idea of creating “null” test services to generate baseline end-to-end performance numbers has proven to be difficult and less useful than anticipated. It is typically not the service itself that is the issue in load testing as much as the environment in which it is hosted, which includes processors, memory, network connections, supporting software, and configuration settings on software and hardware. To create a dummy test as a comparison for a high-availability enterprise-wide service would require creating a hosting environment that can handle such a load. In practice, dummy services were hosted on a single spare machine, and the resulting performance was surprisingly poor for such a trivial service.

A final lesson is that performance testing is complicated. When the performance test tool must understand data at many different layers, up to the message layer, configuring it and getting it to work properly requires good knowledge of the test tool, the protocols, the standards, and the system under test. For a new architecture and a test tool with limited capabilities, workarounds become the rule rather than the exception. This makes seemingly simple tests much more difficult to run. The real problem is the combination of workarounds and a changing architecture and implementation. New releases of the core infrastructure and services can require reworking the performance tests, resulting in lengthy delays when everyone just wants to get testing finished and release the latest code drop. One potential solution is to have the test team work more closely with the development team and develop tests along with the new code so that the test scripts are ready to go when the new code is released.

A wish list for testing consists of the following:

- A test tool that is powerful enough to allow fine-grained control over message traffic at different protocol layers
- A test tool that has an ability to coordinate traffic across multiple nodes
- A test tool whose output can be easily tailored to meet granularity and size restrictions
- A test tool that is easy to use.

The combination of power and ease of use is the main challenge. Offering power in one area but not another can cause a steep learning curve overall, while still not offering the functionality needed for a specific test, resulting in a failure on both fronts.

Looking forward, the SMI-ELS architecture will certainly evolve and grow. There are already plans to integrate monitoring tools, helpdesk functionality, and potentially change the message flows based on vendor upgrades. The message flow changes will require test script changes, but the monitoring tools and helpdesk functionality will require entirely new tests. These tests will most likely build on existing tests, where messages sent to monitoring and help desk components will be added to the overall aggregation and exposure service call tree, and extra test nodes will be inserted to capture performance of the new message flows.

The increasing use of virtual machines and cloud computing offers another challenge. When these issues can be worked out and implemented, they will be assessed. Virtualization creates a new challenge since the system under test is not static. Information about how many virtual machines are in use must be correlated with performance data to show how changes in requestor load and system configuration map to overall performance numbers. Cloud computing isolates the tester from the system under test even more by providing the architecture core services as a virtual service. In this case, the cloud itself must be tested prior to testing any services hosted in a cloud.

References

[1] Kevin Foltz, Coimbatore Chandersekaran, "Sharing Resources through Dynamic Communities," in Proceedings of The Tenth International Command and Control Research and Technology Symposium (ICCRTS 2005), Falls Church, VA, June 2005.

[2] Aaron Hatcher, Kevin Foltz, Coimbatore Chandersekaran, "Facilitating Flexible and Versatile Centrally Organized Community of Interest (CACOI) Organization and Control," in Proceedings of the Eleventh International Command and Control Research and Technology Symposium (ICCRTS 2006), Cambridge, England, September 2006.

[3] Channabasavaiah, Holley and Tuggle, Migrating to a service-oriented architecture, IBM *DeveloperWorks*, 16 Dec 2003.

[4] Alexander Sterff, "Analysis of Service Oriented Architectures from a Business and an IT Perspective." Masters Thesis, Technische Universität München, 15 December 2006.

[5] Amazon Web Services, http://aws.amazon.com.

[6] Igor Burdonov, Alexander Kossatchev, Alexander Petrenko, and Dmitri Galter. KVEST: Automated Generation of Test Suites from Formal Specifications. In J. Wing, J. Woodcock, and J. Davies, editors, *FM'99, Vol. I,* volume 1708 of *Lecture Notes in Computer Science*, pp. 608-621. Springer, 1999.

[7] Mamoon Yunus and Rizwan Mallal. Watch Your SOA Testing Blind Spots. Available at http://www.softwaremag.com/pdfs/whitepapers/Crosscheck_wp2.pdf.