Paper 310-2008

Using SAS® BI Web Services and PROC SOAP in a Service-Oriented Architecture

Dan Jahn, SAS, Cary, NC

ABSTRACT

Many businesses are just starting to use a service-oriented architecture (SOA). In order to build an understanding of what an SOA consists of and to help create a road map for achieving the benefits of SOA, an enterprise service-oriented maturity model (ESOMM) (Oellermann, 2006) has been defined. This ESOMM divides SOA into three categories: Implementation (to provide a service), Consumption (to use a service), and Administration (to govern the use of services). SAS provides many capabilities for enhancing the maturity of an SOA. Implementation capabilities include SAS BI Web services; consumption capabilities include PROC SOAP; and administration capabilities include the use of Metadata and container features. These capabilities, as well as advice on how to increase the maturity of a service-oriented architecture, are provided.

INTRODUCTION

The primary objective of a service-oriented architecture is to increase the agility of a business. Some features of a service-oriented architecture can be supported through technology; other features are supported through policies. SAS has for many years supported many of the features of a service-oriented architecture. Lately, much of the focus around SOA has been in the area of Web services. SAS® 9.2 introduces the second generation of Web services software from SAS, and it represents a major step forward in the enterprise service-oriented maturity model where many categories of the ESOMM have been improved.

To help understand these new features of the SAS platform, we will step through a demonstration of using them in an example program. This program estimates the number of people who will be attending the Alamo at a provided date and time. This is a good example of a service since it can be useful to multiple consumers—the Alamo itself could use it to estimate how much staff they should have on hand; local business could use it to estimate how many customers they will have as a basis for their staffing and supplies; visitors could use it to determine when the best time to visit the Alamo is. The program will also demonstrate the power of combining multiple forecasts to create a new forecast. It becomes apparent that as more people realize how useful the service is, more people will start to use the service. As more consumers of your service come on board, we'll look at how can you support service level agreements and expand your service capacity as your consumer base grows. This scalability feature is not explicitly stated in the ESOMM; but we'll take a look at it since it is certainly a factor in a service implementation's maturity—a good service, as it matures, will get increased use.

Capabilities described by the ESOMM are categorized into Consumption, Implementation, and Administration; they will be italicized in this paper. The specific capabilities are:

- **Consumption:** Explicit Contracts, Testability, Explicit SLAs, Service Discoverability
- **Implementation**: Service Collaboration, Development Processes, Common Schema, Schema Bank, Versioning
- **Administration:** Deployment Management, Security Model, Monitoring, and Business Analytics

There are additional ESOMM capabilities that are not covered in this paper.

CONSUMPTION CAPABILITIES

To make a better estimate of how many people will visit the Alamo, we can include other related estimates of factors which may affect the number of visitors. One factor is the weather: the higher the chance of rain, then fewer people will visit the Alamo. So, we need to get a weather forecast. We could gather data from millions of meteorological measuring stations, and then run powerful models against that mountain of data (something that SAS is quite good at), or we could let someone else do that. The National Weather Service provides a Web service that gives weather forecasts in DWML (a documented XML schema). This format is very tricky to use so details on using that service will not be provided here. For this paper, a Web service wrapper around the National Weather Service forecast was created. This wrapper gives us several benefits: reliability, performance, and ease of use. The ESOMM describes how ease of use of a service can lead to more consumers of the service. As is the case with most programming languages that can call Web Services, SAS provides two options for calling a service: PROC SOAP and the XML LIBNAME engine. PROC SOAP allows you to put together the XML you need, and allows you to examine the

returned XML, giving you the most flexibility in which services you can call. You give it an XML document in a fileref with your request; and it returns a fileref with an XML response from the service. Here's an example:

```
%let dt=18Mar2008 11:33:10;
FILENAME request TEMP;
FILENAME response TEMP;

DATA _NULL_;
    FILE request;
    INPUT;
      _INFILE_ = resolve(_INFILE_);
    PUT _INFILE_;
    CARDS4;
    <GetWeather xmlns="http://tempuri.org/">
      <dt>&dt.</dt>
    </GetWeather>
;;;;

proc soap in=request
          out=response
          url="http://machineName/Weather/Service.asmx"
          soapaction="http://tempuri.org/GetWeather";
run;

FILENAME  SXLEMAP 'c:\sasrepository\weatherResponse.map';
LIBNAME   response xml xmlmap=SXLEMAP access=READONLY;
```

This code loads the request document into the REQUEST fileref using the CARDS statement (and take note of the use of the resolve function which forces the macro '&dt' to get resolved to its value). Then, the code calls PROC SOAP. PROC SOAP sends the request document to the endpoint (the URL). The service knows which operation to invoke based on the SOAPACTION, and generates a response. The response XML document is available in the response fileref after PROC SOAP runs. That response is then exposed as a SAS dataset through the use of a SXLEMAP. One way to create the SXLEMAP is to run the service once, writing the response document into a file (use an actual filename in the response FILENAME instead of TEMP), and then use the SAS® XML Mapper tool to create the map from that file.

For each Web service operation you call using PROC SOAP, there are four things you need to change: the request document, the endpoint (the URL option of PROC SOAP), the soap action (the SOAPACTION option of PROC SOAP), and the map you use to read the response. All four of these are obtained from the WSDL. Web Service Description Language (WSDL) is a standard that is created by service providers to help service consumers call their service. In the Web services world, the WSDL is your *explicit contract* for how to call a service. This contract really forms an agreement between a service provider and a service consumer (in words: if you send me an XML document that looks like this to this endpoint, I'll send you an XML document that looks like this as my response.) If you are not so interested in dealing with the details of a WSDL file, you can get the XML LIBNAME engine to take care of that for you. Here's an example of calling a "convention" service (the service returns the number of attendees to conventions at the San Antonio convention center on a given day.)

```
%let dt=18Mar2008 11:33:10;
FILENAME websvc 'c:\sasrepository\ConventionService.wsdl';
LIBNAME websvc xml92 xmltype=WSDL;

DATA d;
      date = put(datepart("&dt"dt), date9.) ;
run;

DATA conf;
      SET websvc.GetAttendeesResponse(parms=d);
run;
```

Note here that you don't have to specify the endpoint, SOAPACTION, the exact XML format of the request, or a map to read the response. The XML LIBNAME engine is smart enough to read the WSDL and figure that out for you. Note that there are some more complicated XML schemas that the XML LIBNAME engine can't use; but it can work with most rectangular XML documents. Your job as a SAS programmer is to figure out the correct inputs to the call. You can do this after the LIBNAME statement by calling proc datasets:

```
proc datasets dd=websvc details; run;
```

The nice thing here is that you don't have to be an expert at interpreting WSDL and schema; the XML LIBNAME engine can take care of that for you. The service is invoked as part of the SET statement. Then, you can examine what you got after the call using PROC CONTENTS:

```
proc contents data=conf varnum; run;
```

After calling each of the services (the weather service and the convention service), we can treat the resulting datasets created by the XML LIBNAME engine just like we would treat any LIBNAME. In this case, we start the next step with this code:

```
data a;
merge response.nws conf;
```

The code after that uses variables from those datasets to estimate the number of visitors.

For *testability*, one tool that is helpful is called soapUI (eviware):

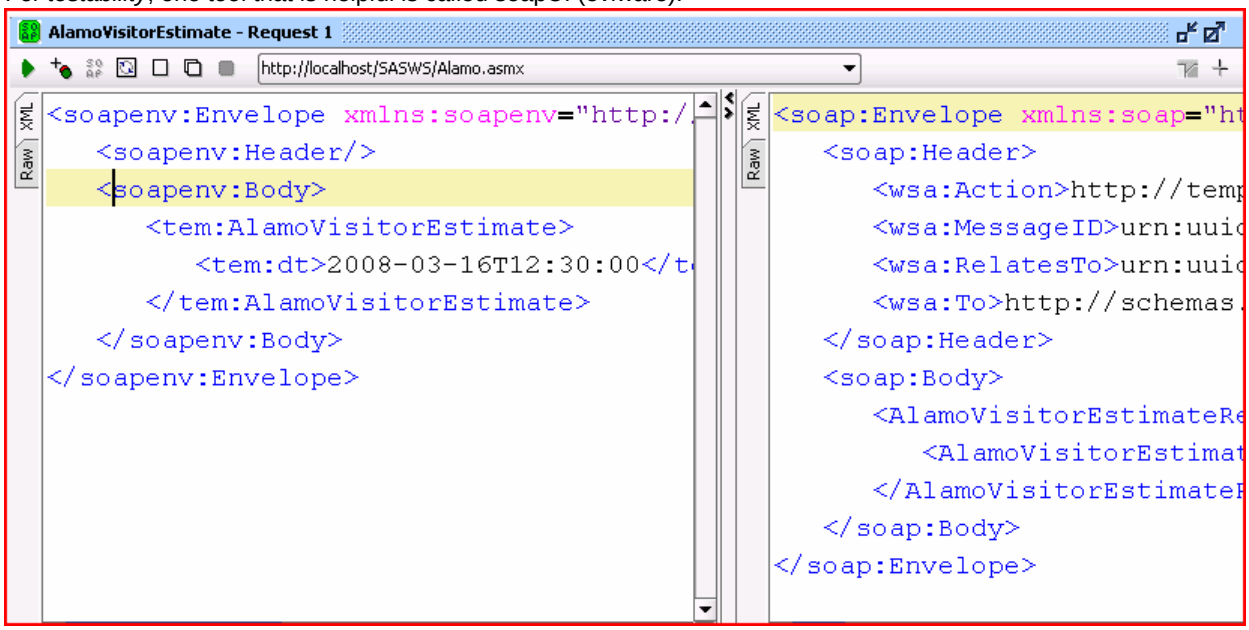

Figure 1. The soapUI Request Document on the Left, and the Response Document on the Right

You can execute the Web Service from soapUI by using the green arrow in the top-left corner of the dialog. You can also set up some performance tests:

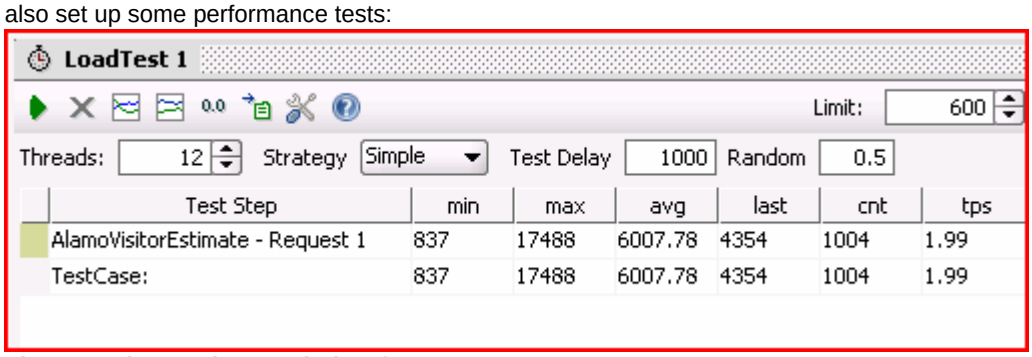

Figure 2. The Load Test Window from soapUI

If you prefer to be very familiar with the XML and use PROC SOAP, you will need to manually create the request XML document. By reading the WSDL provided from a service, soapUI is capable of generating that request document for you (see Figure 1). You can copy the text from soapUI into a cards statement as is done above for invoking the weather service. PROC SOAP gives you the option of including the SOAP envelope or not. If you don't include it, PROC SOAP will add it for you. Also, note that in the example, the "tem" namespace is defined in the toplevel XML element, so if you choose not to include the Envelope, you'll need to copy that namespace declaration to your toplevel element.

As a consumer of a service, not only do you need the explicit contract provided by the WSDL, but you also need some assurance that the service you call will be up. One technique to get this assurance is by having an *Explicit SLA* (service level agreement) with the service provider. As of this writing, there are no general standards for what is included in a SLA; but this is an agreement between the consumer and provider of a service (or set of services) that provides assurances in the areas of performance, scalability, and security. The primary reason for having an SLA is for reliability. A service is useful only if it can meet your needs over and over again, reliably.

Before you can invoke a service, you must find the service. There are many ways that you can perform *Service Discoverability:* telephone your friend (most common when starting), have a listing on a Web page (common when you start to get a few services), or a full-fledged service directory such as UDDI (Universal Description, Discovery, and Integration). A good service provider will create a useful service taxonomy; this becomes more important as more services become available to you.

There are two scenarios for using UDDI: the design-time use (which is more common) and the run-time use. When developing an application that consumes a Web service, you can search for available Web services through UDDI. There are tools that allow you to browse for services, or you can use the Web interfaces. You may want to use UDDI at run time because you don't want to hard-code specific service endpoints into your application (maybe because they can change or maybe because you want failover—but note there are better techniques for providing failover).

Here's an example of how you could call a UDDI registry using PROC SOAP, and list the available services that match a given service key. The service key was created when the service was registered in UDDI. You can have multiple bindings that provide the same service; this lists all the bindings that provide the given service:

```
filename request temp;
filename response temp;

/* Service key is the unique key for the service you want to invoke. */
%let serviceKey='2b0945b6-edc5-4ae0-a8f1-53309211d177';

data _NULL_;
      file request;
      PUT '<get_serviceDetail generic="2.0" xmlns="urn:uddi-org:api_v2">
      <serviceKey>' &serviceKey '</serviceKey></get_serviceDetail>';
run;
```

```
/**
 * Call into UDDI and retrieve the list of bindings for the service key.
 * We should get two tables, 'businessService' and 'bindingTemplate'.
**/
proc soap
      in=request
      out=response
      url='http://MYMACHINE/uddi/inquire.asmx'
      soapaction='urn:uddi-org:api_v2#get_serviceDetail'
      webusername='MYUSERNAME'
      webpassword='MYPASSWORD'
      webdomain='MYDOMAIN'
;
run;
filename sxlemap 'c:\sasrepository\servicedetail.map';
libname response sasexml xmlmap=sxlemap;

proc print data=response.Bindingtemplate; run;
```

The output from PROC PRINT is:

```
    Obs bindingKey

      1 fbb5ea03-5233-4e91-a7db-9c9d9d3124ae
      2 82067352-025f-41b9-b98f-913985faf19c
      3 a40ba52c-aba7-4580-9b03-9995a128e602
      4 043d0bbb-4add-4d01-976a-83ddec3b9cbd
      5 65057a0e-ed19-4e5d-baea-2265da19919a

    Obs description

      1 Some bogus endpoint, not a real binding.
      2 Not a real binding.
      3 May or may not work.
      4
      5 The real binding. This binding should work.

    Obs accessPoint

      1 http://www.SomeBogusEndpoint.com
      2 http://MyMachine.com/newws/test.asmx
      3 http://MyMachine.com/92SASWS/bday.asmx
      4 http://MyMachine.com/newws/bday.asmx
      5 http://MyMachine.com/newws/bdayservice.asmx

    Obs URLType

      1  http
      2  http
      3  http
      4  http
      5  http
```

IMPLEMENTATION

Service Collaboration is where services are aggregated together to create more sophisticated services. That's exactly what we're going to do with our example: combine the weather service and convention service; add some business

logic to those numbers: and create a new service that provides an estimate of the number of visitors. The Consumption Capabilities section described the SAS code which can estimate how many people will visit the Alamo. Now we can use that SAS code as the implementation of a service (which itself calls two Web services.)

The *Development Process:* To implement the service, we will define a stored process and then deploy that stored process as a Web Service. After putting the SAS code discussed above in a folder that is accessible from the machine where a stored process Server is running, we can define the metadata for that stored process. In the case of this stored process, we are keeping it very simple: It has one input (the timestamp of when the estimate is wanted), and it has one output (the number of visitors.) (Timestamp is a term from the new Prompting Framework which means a date and a time. When a timestamp is deployed as a Web Service, the generated WSDL represents the Timestamp as an xsd:DateTime. The prompting framework has replaced the stored process input parameters.)

At this point, we have our stored process, so we are ready to deploy it as a Web Service. To do that, use the folder view in SAS Management Console, and then right-click on the stored process. Select **Deploy as Web Service**. You can provide a custom namespace (Highly recommended. Part of your service governance should include standards regarding the use of namespaces.) You can also provide keywords (which may be useful for potential clients of your service in the area of *Service Discoverability.)* Once you click the Finish button in the wizards, your service is deployed and ready to be used!

Making it this simple to deploy services with SAS increases the likelihood that you will use the services. There is always a trade-off between ease of use as opposed to power and flexibility. SAS BI Web Services have the ability to not only support simple input and output parameters but also schema-based input and output documents. As your organization progresses through its SOA maturity, you are likely to start using *Common Schema* and a *Schema Bank.* The benefits of a common schema are that semantics of the XML documents you use with your service become implicit with the element names. Common schemas encourage reusable code that uses those schemas. Common schemas allow services to communicate with each other without having to go through a translation layer. By storing those schemas in a standard location (a schema bank), it makes it easy for multiple applications to reuse those schemas. A schema bank can really prove to be a *bank* whereby the value of the investments (the schema) in the bank grows through time as other people use and extend the schemas.

You can associate schemas with your input and output documents through the use of stored process DataSources (for inputs to the stored process) and DataTargets (for defining outputs from the stored process.) In order to use a schema, you must define a schema. There are many tools available on the market for creating a schema. SAS can create a schema using the XML LIBNAME engine and the XMLTYPE= schema option. Getting SAS to deal with arbitrary schemas can be a little tricky. The best option here is using the SAS XML Mapper application. XML Mapper creates SXLEMAPs, which map between SAS datasets and XML.

The implementation is responsible for maintaining the *versioning* of the service. There are two aspects to versioning: the versioning of the interface (also referred to as the messages or the schema) and versioning the implementation of the service. There are also two considerations to versioning the interface: compatible changes and incompatible changes. Compatible changes can occur when you add new operations or when you provide for extensibility through the use of the XML schema open element (xs:any). As long as any existing messages can continue to work, you have a compatible version. Some architects plan for versioning by adding the open elements to their schemas. However, using an open element causes many tools to fail to work with your service since they don't know what to put in the open element. Incompatible changes require you to create an interface version when you remove or change existing elements. The standard technique for dealing with an interface version is to add a new service, and continue to support the old service as long as any clients are using it. (Note that versioning and how long an old version is kept after a new version is introduced should be covered in a SLA.) Versioning the implementation of a service is a much easier prospect: None of your clients need to change their code. A common reason for versioning the implementation is when you find a way to improve your service; such as when you develop a new algorithm.

Specific to SAS stored processes, incompatible changes include:

- adding or removing a prompt (input parameter)
- changing a default value for a visible prompt
- changing a constraint for a prompt
- changing the name of a stored process
- moving the folder a stored process runs in
- changing a schema used in a datasource or datatarget in an incompatible way
- adding or removing a datasource or datatarget
- changing the streaming setting

- changing the package setting
- changing the exception/fault behavior of your SAS code

Compatible changes include:

- Changing the SAS code that implements a stored process that doesn't change your exception behavior.
- Adding a new stored process to the service and then regenerating the service.
- Adding schema elements to a schema in the xs:any location.
- Changing or adding documentation.
- Adding, changing, or removing a hidden prompt. Hidden prompts are not exposed in the WSDL, and callers are unaware of their existence.

It is reasonable to deduce that it is much easier to make an incompatible change than a compatible change. There are exceptions to every rule, so even though the above list of compatible changes will result in most Web service clients continuing to function, you might want to explicitly version your service when ANY change is made to it.

ADMINISTRATION

Deployment management promotes a consistent set of services. You want consistency where services are deployed and how they are registered. SAS BI Web Services uses a Web service to generate Web services. Once you deploy your full set of services, or when you migrate from development to test and production, you may want to disable the ability to generate new services. Disabling service generation prevents someone from accidentally deploying a new service. Some administrators may also feel more comfortable from a security perspective with disabling the ability to generate new services.

The *security model* includes managing access to the services, authentication, and encryption. The ability to use container security is one of the main reasons why SAS implemented the services in a native way on each container. SAS supports both host and trusted authentication mechanisms:

- Under host authentication, credentials are passed from the client to the generated service. Those credentials are then passed to the metadata server for authentication.
- Under trusted authentication, authentication is performed by the container either at the transport (HTTP) or the message (WS-Security) layer.

In both cases, the established identity is used when the stored process is invoked to check that the user is authorized to run the stored process in metadata. This creates several options for an administrator in controlling who can access a service:

- Depending on container configuration, a user must have permissions at the transport layer.
- Depending on the Web service configuration, a user must have permission at the WS-Security layer (note that use of WS-Security is optional.)
- Depending on SAS Metadata Server settings, a user must have permission at the metadata layer (users need to have Read Metadata permissions on the stored process in Metadata.

SAS has made some excellent advances in server *monitoring* and management. In SAS 9.2, SAS added many new server properties using MBeans that can be obtained through JMX. The primary benefit of JMX is that there are a wide variety of management consoles (applications) that work with JMX. This level of interoperability is typical of SAS, where SAS uses existing tools where possible. This makes it easier for IT personnel since they might not need to learn new tools and strategies for monitoring, deploying, and securing servers. This benefit extends not only to the job of administering the servers, but also to the development of policies that govern the use of servers and services.

If you notice unusual behavior through the monitoring interfaces, you can dynamically turn up the amount of logging that occurs. A new feature of SAS 9.2 is a consistent logging infrastructure across all tiers of SAS based on the log4j logging framework. SAS uses log4j in Java code; log4net in .Net code, and log4SAS in SAS code. All of these use appenders, loggers, and layouts, so that the administrator gets a consistent experience across all tiers. The benefit of changing the logging level while a server is running is that you don't have to start capturing logs until you need the information. When you turn up the logging levels, you get slower performance and more consumption of system resources (usually local disk space). So you ideally will collect logs only when you need them and isolate what you collect to where the problem is thought to be.

The ability to scale up is also a very important feature. If you create a useful service, it is likely that the service will increase in use over time. So, a system that supports growth is very important. SAS can scale up the number of servers running stored processes using load balancing.

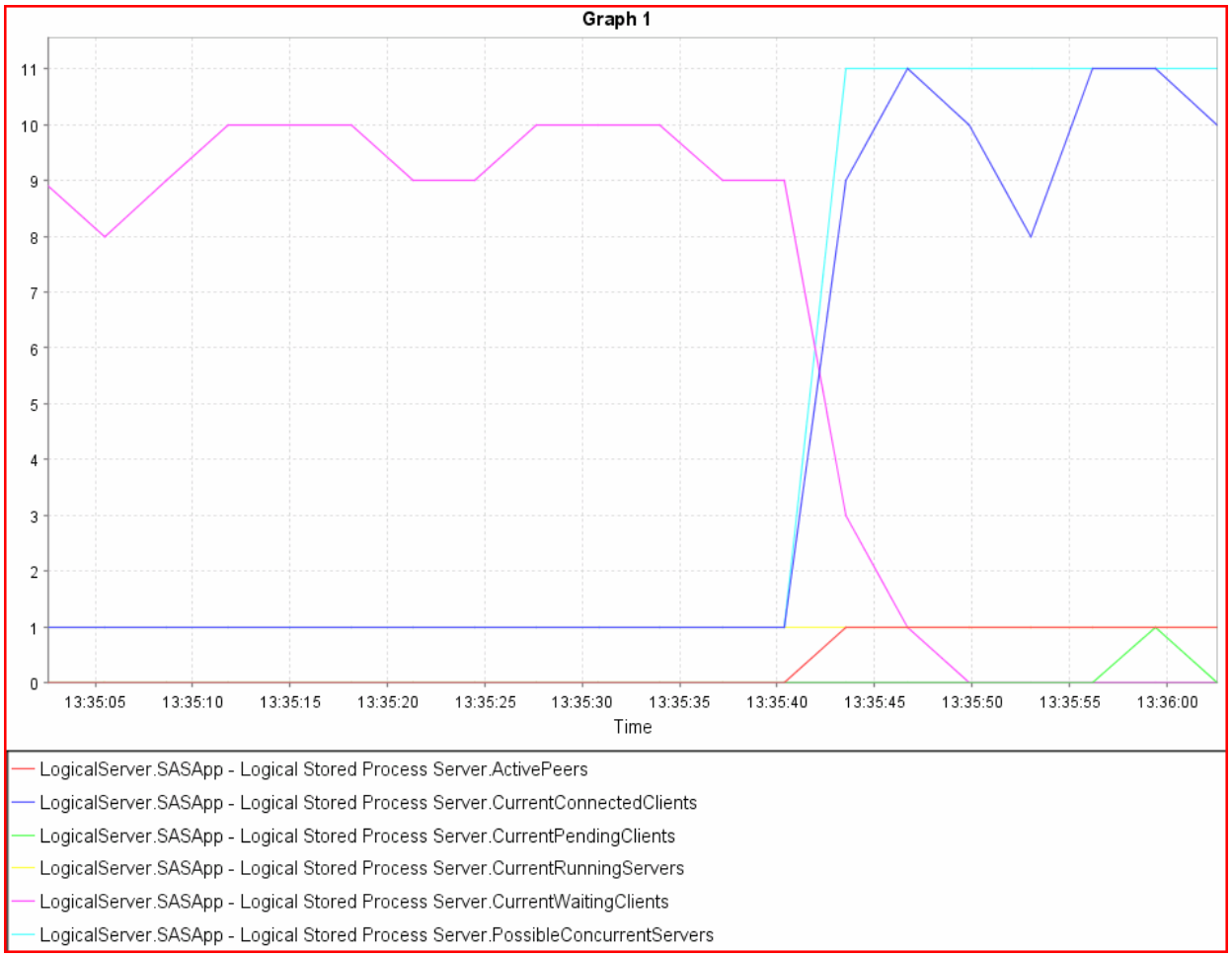

Figure 3 Monitoring counters in real time

Figure 3 was created by running our sample service under a 12 user load created using soapUI. The graph is monitoring several properties provided by the IOM spawner. It demonstrates two things:

- Use of JMX. This graph was captured from MC4J (Hinkle). The author has also used JConsole with MBeans provided by SAS. There are other management consoles that could be used.
- Load balancing. At the time 13:35:40, a spawner was started on a second machine. That spawner is automatically connected to the already running machine and started sharing the load. The properties provided by SAS illustrate at the time the second spawner was started, the average number of waiting clients (magenta line) was about 9–10. Then, moments after the second spawner started, the waiting clients dropped to 0. The cyan line shows that prior to starting the second spawner, there was only a single server that could run (this was done for demo purposes). When the second spawner started, the number of available servers jumped to 11.

While your collections of applications that use services grow, you can start to apply *Business Analytics* to the services and underlying processes. These analytics can be used to monitor service reliability, performance, and usage. You'll find that you can feed the analytics back into the processes to improve and grow your service-oriented architecture.

CONCLUSION

As your organization matures in the use of service-oriented architectures, you will start to develop business processes out of the services that you create and find. The ease and speed with which you can swap out, add, and modify services used by your processes as well as the reliability of those services are major factors in the agility of your organization.

The big underlying theme here is *interoperability*. SAS BI Web Services provide interoperability with other applications through the extensive use of Web Service standards. SAS also provides interoperability at the IT level through the use of standard containers which should make it easier to apply standard procedures and policies to services that use SAS.

Understanding the capabilities of your underlying systems is an essential prerequisite to creating your road map toward increasing the maturity of your service-oriented architecture. SAS provides a wealth of features that can help you down this road, with the ultimate payoff of increasing the agility of your organization.

REFERENCES

Eviware. (No date) *Web Service Testing*. Retrieved February 22, 2008. http://www.soapui.org/
Hinkle, Greg. 2006. *MC4J Management Console*. Retrieved February 22, 2008. http://mc4j.org
Oellermann, William. 2006. “Enabling the Service-Oriented Enterprise.” *Microsoft Architect Journal*. April, 27-32. http://msdn2.microsoft.com/en-us/library/bb245664.aspx

ACKNOWLEDGMENTS

The SAS code to use PROC SOAP to call into a UDDI registry was provided by Zachary Marshall of SAS. Reviewers of the paper include Beth Ayres and Jason Spruill from SAS.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Dan Jahn
SAS
1 SAS Campus Drive
Cary, NC 27513
E-mail: dan.jahn@sas.com
Web: http://www.sas.com/rnd/itech