Paper 314-2010

SAS® Application Messaging: How to Integrate Disparate Processes in Your Service-Oriented Architecture

Stephen A. Vincent, SAS, Cary NC

ABSTRACT

SAS messaging interfaces can help you meet the challenges of integrating the processes in your service-oriented architecture (SOA). Messaging is a key component of SOA, and SAS application messaging allows you to leverage the power of SAS across the enterprise and beyond. SAS messaging works with various third-party message-oriented middleware (MOM) offerings, allowing loosely coupled processes to be integrated across many organizational boundaries. This paper demonstrates how to use SAS messaging interfaces and provides examples of sending and receiving messages, message queue polling, and publishing to a queue. Increase your return on investment by integrating more processes in your SOA with SAS messaging.

INTRODUCTION

Jahn (2006) claimed “the promise of SOAs is their ability to solve business integration issues.” I agree, and I believe that application messaging is the key to keeping that promise. Messaging increases the reliability and flexibility required by today’s SOAs. Many financial systems depend on messaging. Would your next credit card purchase succeed without an approval message? It probably would not. Messaging is the vital conduit providing the loose coupling between disparate applications. Loosely coupled applications and services are easier to integrate and maintain. There are many ways to integrate SAS software with other applications in your SOA. This paper focuses on SAS interfaces to third-party MOM. It provides an overview of the current SAS messaging interfaces, teaches you how to use them, and introduces several new features coming in SAS 9.3.

BACKGROUND

SOA implies cross-application integration. Architects are frequently faced with disparate applications scattered across organizational boundaries, heterogeneous operating system environments, and even different enterprises in distant geographic locations. As the size and complexity of these designs increase, so does the need for asynchronous and disconnected operation. Application messaging systems provide a platform supporting interoperability among loosely coupled applications over a message-passing bus. Figure 1 illustrates one common messaging model (SAS Institute Inc. 2009, p. 4).

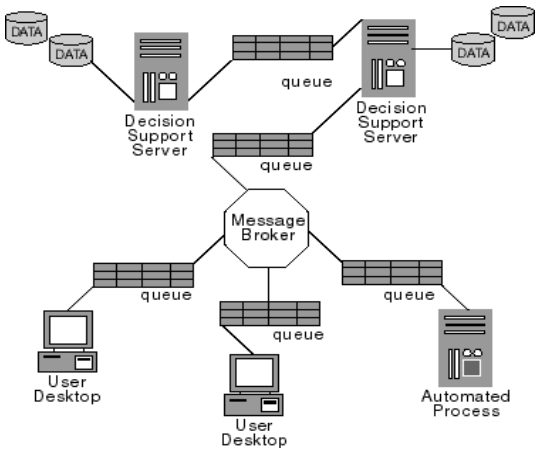

Figure 1. Store-and-Forward Messaging Model

“In a store-and-forward model, messages are sent to named queues, which are in turn hosted at specific destination network addresses. The navigation of messages from their origin occurs through a transmission network that ensures the integrity of message delivery to the destination queue and presentation to the recipient process.” (SAS Institute Inc. 2009, p. 4).

SAS LANGUAGE INTERFACES

SAS® Integration Technologies currently provides interfaces to three principal commercial messaging platforms: IBM WebSphere MQ, TIBCO Rendezvous, and Microsoft Message Queuing (MSMQ). MSMQ comes with Windows and allows you to easily begin experimenting with many topics in this paper. In SAS 9.3 we added support for Java Message Service (JMS) by way of an external file access method in addition to a SAS logging appender. The JMS interfaces expand the SAS third-party MOM support to include any vendor shipping a JMS provider. This includes the Apache ActiveMQ and other open source MOM. JMS support opens the door to many no-cost open source messaging systems that might not be as feature-rich as the larger commercial products yet satisfy the needs of many applications. SAS application messaging interfaces help you meet the challenges of integrating disparate processes and leverage the power of SAS analytics in your SOA, enterprise solution scenarios, and more.

The SAS language interfaces to third-party messaging platforms are implemented in SAS CALL routines. SAS CALL routines are similar to functions but differ in that you cannot use them in assignment statements or expressions. All SAS CALL routines are invoked with CALL statements. The name of the routine must appear after the keyword CALL in the CALL statement. CALL routines can be called within DATA step and open SAS macro code using the %SYSCALL macro statement. As with other SAS language constructs, you can specify arguments to CALL routines through DATA step variables, macro variables, quoted strings, and literals.

The current CALL routine interfaces can be grouped into the following three categories:

- interfaces specific to MSMQ
- interfaces specific to WebSphere MQ
- the Common Messaging Interface supporting MSMQ, WebSphere MQ, and TIBCO Rendezvous

The first two closely resemble the vendor's client application program interface (API). The Common Messaging Interface provides a seamless environment for writing applications that access any of these three messaging systems.

Two or more applications can use these interfaces to communicate with each other indirectly and asynchronously using message queues. The applications need not run at the same time or even in the same operating environment. An application can send a message to a queue, and a receiving application can retrieve the message when it is ready. This allows messaging to be a very valuable tool in your application-integration tool box. Segregating your business functions into separate processes that communicate through messaging allows applications and services to evolve independently.

SAS MSMQ INTERFACES

SAS MSMQ interfaces expose MSMQ in SAS language by way of SAS CALL routines. You can find the usage reference for SAS interfaces to MSMQ at http://support.sas.com/documentation/cdl/en/appmsgdg/61498/HTML/default/msmqchap.htm. The interfaces are supported on all 32-bit and 64-bit Windows platforms that SAS supports. A typical SAS program using MSMQ performs these tasks:

1. Opens one or more queues. MSMQ uses several representations to identify a queue, such as format name, pathname, instance universally unique identifier (UUID), and queue handle.
2. Composes messages. Use a map to describe the format, the number, and the type of parameters to send as part of the message. The data map is used when creating a data descriptor of the actual values of the SAS variables to be included in the message.
3. Sends and/or retrieves messages to/from opened queues. MSMQ uses the concept of a cursor to identify the location of the message within a queue.
4. Closes or deletes queues after a program has sent or retrieved all its messages. This releases the resources that were allocated when the queue was opened or created.

For a more detailed description of those tasks, see http://support.sas.com/documentation/cdl/en/appmsgdg/61498/HTML/default/msmqappl.htm.

SAS MSMQ EXAMPLE

You can run the following SAS DATA step in a Windows environment. It illustrates basic send and receive operations using MSMQ. It sends a text message to a queue, retrieves it, and puts it in the SAS log.

```
data _null_;
    length txtMsg $50;
    call msmqpathtoformat( '.\private$\testq', Qid, rc );  /* Get Q ID */
    if rc = input('03000EC0'x, ib4.) then  /*  MSMQ QUEUE_NOT_FOUND error */
        call msmqcreatequeue( Qid, rc, 'PATHNAME', '.\private$\testq' );
    call msmqopenqueue( Qid, 'SEND', 'SHARE', hQueue, rc );
    call msmqmap( hMap, rc, 'CHAR,,50' );  /* blank padded */
    call msmqsetparms( hData, hMap, rc, 'This is a test.' );
    call msmqsendmsg( hQueue, hData, 0, rc );
    call msmqclosequeue( hQueue, rc );
    call msmqopenqueue( Qid, 'RECEIVE', 'SHARE', hQueue, rc );
    call msmqreceivemsg( hQueue, 0, "RECEIVE", 0, 0, rc );
    call msmqgetparms( hMap, rc, txtmsg );  /* Parse msg into variable */
    put txtmsg=;
    call msmqfree( Qid );
    call msmqfree( hMap );
    call msmqfree( hData );
run;
```

For the sake of brevity, the above example omits many robust programming practices. Be sure to check and process nonzero return codes in your production code. For example, use the SYSMSG function to retrieve error text that is associated with a nonzero return code. You should follow any of the calls accepting a return code argument with error checking. The following code segment shows how to use the SYSMSG function, which could be used to amend the previous example:

```
length msg $256;
q = '.\private$\testq';
call msmqpathtoformat( q, Qid, rc );
if rc ^= 0 then do;
    if rc = input('03000EC0'x, ib4.) then do; /* QUEUE_NOT_FOUND */
        call msmqcreatequeue( Qid, rc, 'pathname,label', q, 'My Queue' );
        if rc ^= 0 then do;
            put 'MSMQCreateQueue failed.'; msg = sysmsg(); put msg;
        end;
    end;
    else do;
        put 'MSMQPathToFormat failed.'; msg = sysmsg(); put msg;
    end;
end;
```

The SAS MSMQ CALL routine reference (http://support.sas.com/documentation/cdl/en/appmsgdg/61498/HTML/default/msmqfncs.htm) provides details and documents many features that are not illustrated above. For example, even though SAS software uses double-precision floating-point representation for numeric values internally, messages sent and received can contain integers and floating-point numbers. The following code excerpt illustrates sending different data types:

```
call msmqmap( hMap, rc, 'short', 'long', 'double', 'char,,50' );
call msmqsetparms( hData, hMap, rc, 100, 2000000000, 3.14, 'A test' );
call msmqsendmsg( hQueue, hData, 0, rc,
                  'correlationid,label,msgid,priv_level,resp_queue',
                  '0102030405060708090A0B0C0D0E0F1011121314',
                  'Mylabel', msgid, 'private', q );
```

SAS WebSphere MQ INTERFACES

SAS WebSphere MQ interfaces expose WebSphere MQ in SAS language through SAS CALL routines. IBM WebSphere MQ provides two APIs for connecting to queue managers. You can use the first, “bindings/server mode,” only when connecting to a local queue manager. To use this API, the queue manager should be on the same machine as the client. This API avoids the overhead of network protocols. IBM provides another API, “client

mode," for clients connecting to a remote queue manager. You can use the client mode when connecting to a local queue manager, but using this mode results in an unnecessary penalty for navigating network protocols. In SAS, you use the MQMODEL macro variable to specify which API to use. The default value is SERVER, which indicates a bindings mode connection. When connecting to a remote queue manager, set MQMODEL with a value of CLIENT. You can find the usage reference for SAS interfaces to WebSphere MQ at http://support.sas.com/documentation/cdl/en/appmsgdg/61498/HTML/default/ibmmqchap.htm.

SAS interfaces to WebSphere MQ Version 6 and later are supported on these platforms:

- all supported UNIX platforms
- all supported 32-bit Windows platforms
- Windows on x64 (WebSphere MQ version 7 and later only)
- z/OS

A typical SAS program using the WebSphere MQ interfaces performs the following tasks:

- establishes a connection to a WebSphere MQ queue manager
- opens one or more queues
- puts and/or gets messages on/off the queue(s)
- releases the resources

SAS WebSphere MQ EXAMPLE

The following example shows basic send and receive operations using the SAS WebSphere MQ interfaces with a bindings mode connection. Just as in the previous MSMQ send and receive example, the following example puts a message on a queue, retrieves it, and puts the values in the log:

```
data _null_;
    length d z $50;
    a=100; b=2000000000; c=3.14; d='This is a test.';
    call mqconn( 'TEST.QMGR', hConn, cc, reason );  /* Connect to queue mgr */
    call mqod( hOD, 'gen', rc, 'objectname', 'TEST.Q' );  /* Set queue name */
    call mqopen( hConn, hOD, 'output', hObj, cc, reason );
    call mqpmo( hPMO, 'gen', rc );  /* Use default PUT msg options */
    call mqmd( hMD, 'gen', rc );  /* Use default msg descriptor */
    call mqmap( hMap, rc, 'short', 'long', 'double', 'char,,50' );
    call mqsetparms( hData, hMap, rc, a, b, c, d );
    call mqput( hConn, hObj, hMD, hPMO, hdata, cc, reason );
    call mqclose( hConn, hObj, 'none', cc, reason );
    call mqopen( hConn, hOD, 'input_shared', hObj, cc, reason );
    call mqgmo( hGMO, 'gen', rc );  /* Set get message options */
    call mqget( hConn, hObj, hMD, hGMO, msgLen, cc, reason );  /* Get msg */
    call mqgetparms( hMap, rc, w, x, y, z );  /* Parse msg into variables */
    put w= x= y= z=;
    call mqdisc( hConn, cc, reason );  /* Disconnect from queue manager */
    call mqfree( hOD );
    call mqfree( hPMO );
    call mqfree( hMD );
    call mqfree( hMap );
    call mqfree( hData );
run;
```

SAS COMMON MESSAGING INTERFACES

The SAS Common Messaging Interface provides a seamless environment for writing applications that interface with MSMQ, WebSphere MQ, and TIBCO Rendezvous. Having one interface for all three vendors allows you spend less time learning vendor-specific details. The benefits of leaving the vendor and host details in the hands of the messaging interfaces become even more evident when applications are distributed across multiple heterogeneous software and hardware environments. The TIBCO Rendezvous message delivery system differs from the other transports in some important ways. Developers must take these differences into account when using the Common Messaging Interface to support applications based on TIBCO Rendezvous. The main differences are as follows:

- TIBCO Rendezvous uses an approach called *subject-based addressing*. Both WebSphere MQ and MSMQ deliver messages to specific destination queues using queue names, but TIBCO Rendezvous broadcasts messages that have been labeled with user-defined *subject names*. Data-consumer applications *listen* for

particular subject names and receive messages only when the subject name matches a name being listened for. The communicating programs must agree in advance on the subject names to be used and the forms of messages to be exchanged.

- Because messages are broadcast to subject names instead of specific destination queues, a message can be received only by stations that are online and actively listening for the subject name associated with the message.

SAS Integration Technologies supports both the reliable and certified message delivery features of TIBCO Rendezvous 7.5.4 and later. Support for using TIBCO Rendezvous with the SAS Common Messaging Interface is available in the following operating environments:

- all supported UNIX platforms
- all supported 32-bit Windows platforms
- all supported x64 Windows platforms beginning with SAS 9.3

A typical SAS program using the Common Messaging interfaces performs the following tasks:

- initializes the type of transport and obtaining a unique identifier
- opens an existing queue by using a known transport identifier
- sends messages to a queue by using a unique queue identifier
- receives messages (and possibly attachments) from a queue
- parses the message
- gets attachments that are associated with a message (if necessary)
- copies any desired attachments to local storage
- closes all queues upon completion of the program tasks
- terminates transports that are initialized by the program

COMMON MESSAGING EXAMPLE

The following example illustrates basic send and receive operations using the SAS TIBCO Rendezvous interfaces:

```
%let qTransport=rendezvous;
%let listenQname=TEST.REND.*;
%let sendQname=TEST.REND.MESSAGE;
data _null_;
    length event y $50;
    call init( tid, "&qTransport", rc );  /* Initialize the transport */
    call openqueue( listenQ, tid, "&listenQname", 'fetch', rc );
    call openqueue( sendQ, tid, "&sendQname", 'delivery', rc );
    mapName = 'dMap1';
    map1 = 'long;char,,50';
    call setmap( mapName, 'registry', rc, map1 );
    call sendmessage( sendQ, rc, 'map', mapName, 99, 'Just a test.' );
    call receivemessage( listenQ, rc, event, attchflg, "map", mapName, x, y );
    if ( event eq 'DELIVERY' ) then put x= y=;
    call closequeue( sendQ, rc );
    call closequeue( listenQ, rc );
    call term( tid, rc );
run;
```

In the example program above, you can switch messaging platforms by redefining the three macro variable definitions at the beginning of the program. Valid values for the INIT transport argument are MQSERIES, MQSERIES-C, MSMQ, RENDEZVOUS, and RENDEZVOUS-CM. For more details, refer to the INIT CALL routine documentation (http://support.sas.com/documentation/cdl/en/appmsgdg/61498/HTML/default/init.htm).

Following is an example for MSMQ:

```
%let qTransport=msmq;
%let sendQname=.\private$\testq;
%let listenQname=&sendQname;
```

Following is an example for WebSphere MQ:

```
%let qTransport=mqseries;
%let sendQname=TEST.QMGR:TEST.Q;
%let listenQname=&sendQname;
```

If you want to use the WebSphere MQ client API, the transport would be `mqseries-c`. When using the client API, the IBM client code requires more connection information that can be obtained from the MQSERVER environment variable or a client channel definition table. In simple cases, the MQSERVER environment variable is the easiest. Its value has the form of *`channel name`*`/TCP/`*`connectionname`*, where *`connectionname`* is the node name and an optional listening port number. For example:

```
set MQSERVER=CH_1/TCP/mypc(1414)
```

For more information about WebSphere MQ client connection options, see IBM Corporation (2008).

HELPFUL HINTS

Keep in mind the following helpful hints:

- Before any messages are sent with the TIBCO Rendezvous transport, the queues that receive the messages must be running and must have a listener (that is, the queues must be opened for FETCH, FETCHX, REQUEST, or REQUESTX). Otherwise, data will be lost. Queues that are opened for REQUEST and REQUESTX automatically have their receiving (response) queues open to listen for incoming messages.
- If you are sending certified messages by using Rendezvous-CM and you plan to close the sending queue immediately after sending the message, then you might want to add a sleep() call to sleep for a couple of seconds. This delay allows the Certified Delivery Agreement to be established between the sending transport and the receiving transport. This delay can also occur when a listener is first opened to receive certified messages.
- If you intend to send attachments, use a queue that supports transactional processing. That way, all messages associated with a failed attachment can be rolled back if any part of the attachment processing fails.

OTHER SAS MESSAGING INTERFACES

The messaging interfaces discussed thus far have been geared toward surfacing third-party MOM client APIs to a SAS programmer for general use. There are other interfaces designed with more specific use cases in mind. The PACKAGE_PUBLISH CALL routine and message queue polling servers support MOM. Both of them are part of the SAS Integration Technologies product.

PUBLISHING FRAMEWORK

The Publishing Framework consists of a set of CALL routines, APIs, and graphical user interfaces that allow users to publish digital content to a variety of destinations including message queues. When publishing to a queue, all entries in the package are published to the queue by default. Queues that support transactional units of work are recommended. By using transactional queues, the queue transport prevents partial packages from remaining on the queue in cases where errors are encountered during package publishing. This example publishes a package to two different queues, and a correlation ID property is used to allow applications to selectively retrieve only packages with that correlation ID:

```
pubType = "TO_QUEUE";
firstQ = "MQSERIES://MYQMGR:MYQ";
secondQ = "MSMQ://myhost\myq";
corrValue = "12345678901234567890";
call package_publish( packageId, pubType, rc,
                      "CORRELATIONID", corrValue, firstQ, secondQ );
```

MESSAGE QUEUE POLLING SERVER

A SAS Message Queue Polling Server is a feature that allows you to monitor a message queue and start SAS programs to fulfill requests being placed on the queue. It is very useful in situations involving high-volume and time-sensitive requirements, or both. SAS 9.2 Message Queue Polling Servers support only WebSphere MQ. A polling

server is similar to a workspace server in that a SAS Object Spawner manages it. The Object Spawner creates polling server sessions as needed, based on the configuration properties of the polling server as defined in SAS metadata. The spawner monitors the depth of the queue and launches more polling server sessions if the current set of servers cannot keep pace with the volume of request messages that clients put on the queue.

Use the SAS® Management Console to configure message queue polling servers. The following is an outline of the tasks required.

1. Configure the third-party MOM.
2. Start SAS Management Console to perform the remaining tasks.
3. Define a queue manager:
   a. Select **Actions->New Server**, and select Queue Manager for WebSphere MQ.
   b. For the name, use the name of the WebSphere MQ queue manager.
   c. Enter a host name and a port number.
   d. Enter the list of available queues.
4. Define a message queue polling server:
   a. Select the queue to be monitored.
   b. Enter the command to start the SAS session and process messages.
   c. Enter the multiuser credentials for launching the sessions (typically the sassrv account).
   d. Enter the name of the machine on which the spawner is running in the server machine list.
   e. Click **Advanced Options** and enter properties such as message threshold.
5. Add the polling server to the selected **Servers** list on the **Servers** tab of the spawner properties.

For more configuration details, refer to
http://support.sas.com/documentation/cdl/en/appmsgdg/61498/HTML/default/pollingsrv.htm.

MESSAGE QUEUE POLLING SERVER DESIGN CONSIDERATIONS

Message queue polling is most useful for DATA step applications that process messages that are independent of each other. Consider the scenario where you have many Web clients driving small requests for back-end server processing. The requests can be put on a queue, and the polling servers can process the requests and put reply messages back on a reply queue from which the clients can get their response. This technique is a powerful, scalable, and flexible way to integrate disparate processes in your SOA.

When the Object Spawner launches polling server sessions, the following environment variables are automatically defined in the session.

SASQSID	specifies the Object Spawner's unique identifier.
SASQUEUE	specifies the name of the queue for the polling server.
SASQMGR	specifies the name of the queue manager hosting the queue.

When developing the SAS program used to process the messages, you can access those environment variables using the SYSGET function. Here is an example:

```
sId = sysget('SASQSID');
qMgr = sysget('SASQMGR');
qName = sysget('SASQUEUE');
```

You can also set environment variables in the server session by using the -SET option in the SAS command for the server. If the queue manager is remote and you are not using a WebSphere MQ client channel definition table, then it can be convenient to specify values for MQSERVER and MQMODEL environment variables on the SAS command. The following example illustrates the use of a remote queue manager.

```
sas -sysin "mypgm.sas" -set MQMODEL client -set MQSERVER "CHNL1/TCP/192.168.0.10(1414)"
```

MESSAGE QUEUE POLLING SERVER TERMINATION CONSIDERATIONS

When the Object Spawner is terminated, it automatically puts stop messages on the queue for the polling servers to catch and terminate themselves. Keep in mind that the developer of the polling server program should add code to catch the stop messages. To accomplish this, first be sure to set the SASQSID property in the Get Message Options object (MQGMO). Here is an example:

```
call mqgmo( hGMO, 'GEN', rc, 'SASQSID', sId );
```

Then, in the main message processing loop, check the return code from each get message call. When the SASQSID property in the hGMO has been set and matches the SASQSID sent in the stop message, the MQGET call sets its return code to –2, signifying that the spawner wants the session to terminate. Here is an example:

```
call mqget( hConn, hObj, hMD, hGMO, msgLen, cc, reason );
if cc ^= 0 then do;
   if reason = 2033 then put 'No message available';
   else do;
      if reason = -2 then do;
         /* -2 indicates that a session-specific stop message has      */
         /* been received from the object spawner queue monitor        */
         /* application.  We should clean up and shutdown immediately. */
         put "MQGET: received stop message from object spawner";
         goto Exit;
      end;
      else put 'MQGET: failed with reason= ' reason;
   end;
end;
else put 'MQGET: message received: ';
```

NEW SAS MESSAGING INTERFACES COMING SOON

If you use any messaging interfaces described above, you know that some can be challenging for beginning programmers. The interfaces provide many features and ways to customize the behavior. Having feature-rich and powerful interfaces provides many benefits but can be viewed by some as complex and difficult to use. SAS 9.3 has several new messaging interfaces that are much easier than the SAS 9.2 CALL routines. The rest of this paper is devoted to these new interfaces coming in SAS 9.3.

SAS JMS EXTERNAL FILE ACCESS METHOD

Most SAS programmers are familiar and comfortable with DATA step programming, including the use of external file access methods. SAS external file access methods associate a SAS fileref as an abstraction of an external file, device, or something that can perform record input and output. SAS 9.3 includes the SAS JMS external file access method. It will provide SAS file I/O access to any messaging platform accessible through a JMS provider. The JMS specification isolates the vendor-specific details from the client by moving the vendor implementation into a provider. The JMS specification is *not* a union of features available in all messaging platforms. Instead, it is an intersection of common features. Also, the SAS JMS external file access method does *not* support all JMS features. It provides an interface to JMS providers. The interface includes a set of Java classes that are loaded in a Java Virtual Machine (JVM) started by the SAS session. The SAS session must be started in an environment having the third-party JMS classes in the class path.

JMS applications increase their portability by retrieving administratively predefined connection factory and destination objects from a Java Naming and Directory Interface (JNDI) name space. The SAS JMS external file access method provides FILENAME statement options used to retrieve the objects when assigning a SAS fileref. The DATA step FILE and INFILE statements are used to open a destination. The PUT and INPUT statements are used for sending and receiving messages, respectively. The SAS JMS external file access method can use SAS formats and informats specified in PUT and INPUT statements, but it is important to note that the access method sends and receives JMS TextMessage objects. A TextMessage is a message whose body contains a *java.lang.String*.

ASSIGNING A FILEREF WITH THE JMS EXTERNAL FILE ACCESS METHOD

The SAS FILENAME statement is used to assign a JMS fileref. It has this syntax:

```
FILENAME fileref JMS JNDICTXTFACTORY="JNDI initial context factory"
                     JNDIPROVIDERURL="JNDI provider URL"
                     CONNFACTORY="JMS connection factory" <options>;
```

The JMS external file access method supports these options:

AUTHDOMAIN=*domain*
: names an authentication domain metadata object and is used for associating user name and password credentials with an identity. Credentials are used when creating a connection to the JMS provider. This option is valid only in the FILENAME statement, *not* in FILE and INFILE statements.

CONNFACTORY="*JMS connection factory look-up name*"
specifies the administrative object used to create a connection with a JMS provider. This option is valid only in the FILENAME statement, *not* in FILE and INFILE statements.

CONNOPTS="*additional options string*"
specifies additional options passed to the provider and used when creating a connection factory. This string takes the form of space-delimited list of name value pairs (for example, CONNOPTS="option1=value1 option2=value2"). This option is valid only in the FILENAME statement, *not* in FILE and INFILE statements.

CORRELATIONID=*correlId variable*
specifies a DATA step variable used for input/output of correlation ID values.

DELIVERYMODE=PERSISTENT | NONPERSISTENT
specifies the reliability of the sent message. The default is PERSISTENT.

DESTINATION=*destination lookup name*
specifies an administrative object used to open a destination for sending and receiving messages.

JNDICTXTFACTORY="JNDI *initial context factory*"
specifies the Java Naming and Directory Interface initial context factory. This option is valid only in the FILENAME statement, *not* in FILE and INFILE statements.

JNDIPROVIDERURL="*JNDI provider URL*"
is used to look up the JMS connection factory. This option is valid only in the FILENAME statement, *not* in FILE and INFILE statements.

LRECL=*n* | *n*k
specifies the logical record length. The default is 256.

MATCHOPTIONS=CORRELID | MESSAGEID | BOTHIDS
generates a message selector based on correlation and message ID variables.

MESSAGESID=*msgId variable*
specifies a DATA step variable used for input and output of message ID values.

MSGSELECTOR="*message filter string*"
restricts the messages delivered to the message consumer to those that match the selector. A message selector is a string. The syntax of the string is based on a subset of the SQL92 conditional expression syntax. A message selector matches a message if the selector evaluates to true when the message header fields and property values are substituted for their corresponding identifiers in the selector.

PASSWORD=*password* | _PROMPT_
specifies the password associated with the user ID given in the USER= option. This option is valid only in the FILENAME statement, *not* in FILE and INFILE statements.

PRIORITY=*n*
specifies the priority. Normal priority ranges from 0 to 4. Expedited priority ranges from 5 to 9. The default is 4.

TIMEOUT=*n*
specifies the number of milliseconds to wait for a message. The get-message call blocks until a message arrives, the time-out expires, or this message consumer is closed. The default time-out is zero, which means the get call never expires and the call blocks indefinitely until it gets a message.

TIMETOLIVE=*n*
sets the default length of time in milliseconds a message should be retained by the message system. The default is zero, which is unlimited.

USER=*userid* | _PROMPT_
specifies the identity used when creating the connection factory at the time of connection. This option is valid only in the FILENAME statement, *not* in FILE and INFILE statements.

SAS JMS EXTERNAL FILE ACCESS METHOD EXAMPLES

Like other SAS external file access methods, the JMS external file access method can be used in DATA steps to read and write records. This example illustrates basic send and receive operations using the JMS external file access method:

```
filename mymq JMS
    jndictxtfactory='com.sun.jndi.fscontext.RefFSContextFactory'
    jndiproviderurl='file:/C:/jndi' connFactory=myConFac
    destination='request';
data _null_;
    file mymq;
    a=100; b=2000000000; c=3.14; d='Just a test string.';
    put a b c d;
run;
data myDset;
    infile mymq timeout=5000;
    input a b c d $19.;
run;
proc print data=myDset;
run;
```

The SAS JMS external file access method supports aggregate file syntax, where the fileref is a connection without a destination specified. This enables one fileref to be used when opening any destination available on that connection. The following example reads the messages from the request destination and puts them on the response destination:

```
filename mymq JMS jndictxtfactory='com.sun.jndi.fscontext.RefFSContextFactory'
    jndiproviderurl='file:/C:/jndi' connFactory=connfac;
data _null_;
    infile mymq(request);
    file mymq(response);
    input;
    put _infile_;
run;
```

The following example reads only messages having a specific correlation ID from the request destination, puts them on the response destination, reads them from the response destination, and puts the values in the log for confirmation:

```
data _null_;
    length recvMsg $100 msgId $ 128 corId $ 24;
    infile mymq(request) length=len lrecl=100
           correlid=corId matchoptions=correlid;
    file mymq(response) correlid=corId;
    corId = "223456789ABCDEF012345622";
    input recvMsg $varying100. len;
    put recvMsg;
run;
data myDset;
    length recvMsg $ 1024 msgId $ 128 corId $ 24;
    infile mymq(response) correlid=corId messageid=msgId length=len;
    input recvMsg $varying1024. len;
    put corId= msgId= recvMsg=;
run;
```

The SAS 9.2 messaging CALL routines set the contents of a message into DATA step variables. DATA step variables are limited to 32kB in size. Processing messages larger than that requires parsing into multiple SAS DATA step variables. Although this might not be a problem, it can be difficult when you want only to copy the data from one queue to another or to an external file. The new SAS JMS external file access method will allow you to copy large messages easily. This will be possible with help from the new FCOPY DATA step function, also coming in SAS 9.3. Here is an example:

```
rc = fcopy( 'filref1', 'filref2' );
```

As mentioned earlier, administrators will probably define connection and destination objects in a JNDI namespace, but this is not required. Given a machine that has the Apache ActiveMQ running and the provider implementation JAR file (for example, `activemq-all-5.3.0.jar`) in the client class path, the following example requires no administrative setup. In the example, the CONNOPTS= option is used to pass options directly to the provider to be used when instantiating a connection factory. Also notice in this example, the SAS XML engine is making use of the SAS JMS external file access method to put a SAS data set on the queue in XML format, and then read it back off.

```
* Create a sample data set.;
data pointofsale;
    infile cards;
    input saleId date e8601dt22.2 amount customerId storeId $4.;
    format date e8601dt22.2 amount dollar6.2;
    cards;
0001 2009-09-15T15:51:00.51 11.91 1001 A001
0002 2009-09-15T15:52:00.52 12.92 1002 B002
0003 2009-09-15T15:53:00.53 13.93 1003 A003
;
run;
proc print data=pointofsale;
run;

* Put the data set on the queue in XML format.;
filename mymq JMS
    jndiCtxtFactory="org.apache.activemq.jndi.ActiveMQInitialContextFactory"
    jndiProviderURL="tcp://localhost:61616"
    connFactory="ConnectionFactory"
    connoptions="queue.myQ=myQ" destination=myQ lrecl=1024;
libname mymq xml92;
data mymq.pointofsale;
  set work.pointofsale;
run;

* Create a XML map.;
filename myXMLmap temp;
data _null_;
  file myXMLmap;
  infile cards;
  input;
  put _infile_;
cards;
<?xml version="1.0" encoding="windows-1252" ?>
<SXLEMAP version="1.2" name="SXLEMAP">
  <TABLE name="PointOfSale">
    <TABLE-PATH syntax="XPATH">/PointOfSale</TABLE-PATH>
    <COLUMN name="saleId">
      <PATH syntax="XPATH">/PointOfSale/SaleId</PATH>
      <TYPE>number</TYPE>
      <DATATYPE>number</DATATYPE>
      <LENGTH>8</LENGTH>
    </COLUMN>
    <COLUMN name="date">
      <PATH syntax="XPATH">/PointOfSale/Date</PATH>
      <TYPE>number</TYPE>
      <DATATYPE>datetime</DATATYPE>
```

```
          <INFORMAT width="22" ndec="2">e8601dt</INFORMAT>
          <FORMAT width="22" ndec="2">e8601dt</FORMAT>
        <LENGTH>8</LENGTH>
      </COLUMN>
      <COLUMN name="amount">
        <PATH syntax="XPATH">/PointOfSale/Amount</PATH>
        <TYPE>number</TYPE>
        <DATATYPE>number</DATATYPE>
        <LENGTH>8</LENGTH>
          <INFORMAT width="10" ndec="2">dollar</INFORMAT>
          <FORMAT width="10" ndec="2">dollar</FORMAT>
      </COLUMN>
      <COLUMN name="customerId">
        <PATH syntax="XPATH">/PointOfSale/customerId</PATH>
        <TYPE>number</TYPE>
        <DATATYPE>number</DATATYPE>
        <LENGTH>8</LENGTH>
      </COLUMN>
      <COLUMN name="storeId">
        <PATH syntax="XPATH">/PointOfSale/storeId</PATH>
        <TYPE>character</TYPE>
        <DATATYPE>string</DATATYPE>
        <LENGTH>4</LENGTH>
      </COLUMN>
    </TABLE>
</SXLEMAP>
;
run;

* Read XML off the queue and create a data set.;
data myds;
  set mymq.pointofsale;
run;
proc print data=myds;
run;
```

JMS APPENDER

The SAS Logging Facility provides a JMS appender that puts messages on a destination. You can add the following sample JMS appender and logger configuration to a SAS Logging Facility configuration file:

```
<appender name="myJMS" class="JMSAppender">
    <param name="persistent" value="true"/>
    <param name="java.naming.factory.initial"
           value="com.sun.jndi.fscontext.RefFSContextFactory"/>
    <param name="java.naming.provider.url" value="file:/c:/JNDI"/>
    <param name="factory" value="myfac"/>
    <param name="destination" value="myqueue"/>
    <layout>
        <param name="ConversionPattern" value="%d - %c -%m"/>
    </layout>
</appender>
<logger name="myLogger" additivity="false">
    <level value="INFO"/>
    <appender-ref ref="myJMS"/>
</logger>
```

Logging events sent to the `myLogger` logger are sent to the `myqueue` JMS destination. In your SAS programs, you can send events to the logger using the SAS Logging Facility language interfaces. The following example illustrates sending an information message to the queue using the above logger within a DATA step:

```
rc = log4sas_logevent('myLogger', 'INFO', 'My test message.' );
```

In open SAS code, you can use the macro interface:

```
%log4sas_info( myLogger, 'My test string, comma included.' )
```

It is easy to configure the SAS Logging Facility to produce a larger number of messages. When associating the JMS appender to a logger, be careful not to send too many messages. A good example would be sending error messages, not debug or trace messages. For details about the SAS Logging Facility, see http://support.sas.com/documentation/cdl/en/logug/61514/HTML/default/a003251511.htm.

CONCLUSION

SAS provides many messaging interfaces to help you meet the challenges of integrating applications in your SOA. The new SAS 9.3 messaging interfaces are easy to use and provide access to many more MOM vendors. Your call to action is to increase the reliability, flexibility, and scalability of your architectures by integrating more applications and services. Tap into THE POWER TO KNOW® by taking full advantage of SAS analytics by integrating more SAS in your architectures through the use of SAS application messaging. The many tools and techniques described in this paper let you leverage the power of SAS across your enterprise and beyond.

REFERENCES

IBM Corporation. 2008. *WebSphere MQ Clients*. Armonk, NY: IBM Corporation.

Jahn, Dan. 2006. “Service-Oriented Architectures – Going from Buzz to Business.” *Proceedings of the Thirty-first Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc.

SAS Institute Inc. 2006. SAS Integration Technologies available at http://support.sas.com/documentation/onlinedoc/inttech/index.html

SAS Institute Inc. 2009. *Application Messaging with SAS 9.2*. Cary, NC: SAS Institute Inc.

Sun Microsystems. 2002. *Java Message Service*. Palo Alto, CA: Sun Microsystems, Inc

ACKNOWLEDGMENTS

The author appreciates these people who graciously reviewed this paper: Darren Key, Matt Starbuck, Kat Turk, Linda Hamm, and Craig Rubendall.

CONTACT INFORMATION

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

Stephen A. Vincent
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
E-mail: Stephen.Vincent@sas.com