Paper 42

How to Use Version 7 Features to Optimize the Distributed Capabilities of SAS® Software

Cheryl Garner, SAS Institute, Inc.

Introduction

This paper introduces several new Version 7 SAS/CONNECT® capabilities designed to facilitate and secure distributed processing. The features covered in this paper include:

- asynchronous compute services
- cross-environment data access (CEDA)
- messaging services
- remote objecting services
- agent services
- encryption services

Asynchronous compute services allow you to initiate a remote submit and immediately regain control in the local SAS® session to continue processing. CEDA enables direct processing of SAS data files across multiple platforms. Messaging capabilities provide both direct and indirect messaging to facilitate the development of multi-tiered distributed applications and applications that can communicate independently of time, logic flow, and location. Remote objecting provides application developers the ability to distribute objects in their applications across hardware platforms. Agent services allow both scheduled and dynamic execution of SAS source code containing built-in conditional logic to run across your network. And, encryption services provide privacy of your data as it is sent across the network.

Asynchronous Compute Services

Compute services give you easy access to many of the remote resources on your network from a single local SAS session. SAS/CONNECT provides access to remote resources in two forms, synchronous and asynchronous. With synchronous processing, the default behavior, you wait for the results of the remote processing before you are able to continue local processing. With asynchronous processing, you immediately regain control to continue local processing while the remote job executes. Results can be obtained at a later time.

The ability to execute remote submits asynchronously allows you to continue processing on your local host while the remote submit processes in the background. Asynchronous processing provides increased time efficiency by allowing you to perform multiple tasks simultaneously. Using this technique, you can start a long running task in the background to a remote host, and immediately be able to begin another task on another remote host or continue local processing, rather than wait until the first remote task is complete before regaining control of your local SAS session. This also provides you with flexibility as to when and where tasks are performed.

By using the CWAIT=NO option on the RSUBMIT statement, you indicate that the remote process is NOT to wait until completion before returning local control to you. Therefore, you can continue local processing (this includes submitting additional processes to other remote hosts) immediately.

SAS/CONNECT stores the accumulated log and output lines from the remote process until you request the data by using the RGET command. Once the RGET command is issued, the accumulated log and output lines are retrieved and merged with the local log and output lines. However, the RGET command can only be issued one time. If the remote process has not finished executing, the remote submit continues as if it had been submitted to execute synchronously (WAIT=YES).
To avoid issuing the RGET command too early, SAS/CONNECT allows you to specify a macro variable by using the MACVAR= option on the RSUBMIT statement. This macro variable can be tested to see if the remote processing has finished. You can also use the RDISPLAY statement to view the current contents of the spooled log and output windows. The following sections explain the statements and options that enable asynchronous remote processing.

RGET Command/Statement

The RGET command and the RGET statement cause all the spooled log and output from the execution of an asynchronous remote submit to be merged into the local log and output windows. When an asynchronous remote submit executes, the log and output statements are not merged into the local log and output windows, but instead they are spooled for retrieval at a later time.

If the RGET command or RGET statement is executed while the asynchronous remote submit is still in progress, all currently spooled log and output lines are retrieved and merged into the local log and output windows, and the remote submit continues processing as if it were submitted synchronously. That is, you will NOT regain control in yor local SAS session until the remote submit has completed. If you don’t want the remote submit to become synchronous, but you want to check its progress, use the MACVAR option in the SIGNON or the RSUBMIT statement. This allows you to check the progress of an asynchronous remote submit without causing it to execute synchronously.

RDISPLAY Command /Statement

The RDISPLAY command and the RDISPLAY statement create two windows to display the spooled log and output lines that are generated by an asynchronous remote submit. One window displays the log lines and the other window displays the output lines.

When an asynchronous remote submit executes, the log and output lines are not merged into the local log and output windows; instead, they are spooled to disk until they are retrieved with the RGET statement. RDISPLAY allows you to view the spooled log and output lines created by the asynchronous remote submit without merging them into the local log and output windows. The log and output lines continue to scroll into the windows created by the RDISPLAY command as they are produced by the remote processing. The RGET command or statement must be used to actually merge the spooled lines into the local log and output windows.

RSUBMIT Command/Statement

The RSUBMIT command and the RSUBMIT statement cause SAS programming statements that are entered in the local environment to execute on a remote SAS session. Even though the statements execute in the remote environment, all responses and output are displayed in your local SAS log and output windows as they would be if you executed the program in the local SAS session.

RSUBMITs are processed in either *synchronous* or *asynchronous* mode.

Synchronous mode means that the user does not regain local control until the RSUBMIT has completed. The RSUBMIT must run to completion before the user regains control. Synchronous processing is the default processing mode.

Asynchronous mode allows the user to start an RSUBMIT in the background to a remote host and to regain local control immediately to continue with local processing or remote processing to another host.

The following RSUBMIT options enable asynchronous RSUBMITs:

```
CONNECTWAIT | CWAIT | WAIT=value
```

specifies whether this particular RSUBMIT is to be executed synchronously or asynchronously. Synchronous processing indicates that you will wait for the remote processing to complete before regaining control in the local SAS session. This is the default processing technique for RSUBMIT.

In asynchronous processing, when the RSUBMIT begins to execute in the background to the remote host, you regain control of your local SAS session to continue local processing or to use RSUBMIT to other remote sessions.

The valid values for the WAIT= option are:

YES | Y	indicates a synchronous RSUBMIT.
NO | N	indicates an asynchronous RSUBMIT.

If WAIT=NO is specified, it will also be useful to specify the MACVAR= option. This will allow you to test the status of the current asynchronous RSUBMIT by determining whether it has completed or is still in progress.

```
CMACVAR | MACVAR=value
```

where value specifies the name of the macro variable to associate with this remote session. If specified on the RSUBMIT command/statement, the MACVAR= option overrides any previous MACVAR= specifications for this remote session. The macro variable is NOT set if the RSUBMIT command fails due to incorrect syntax. Other than this one exception, the
macro variable (value) is set at the completion of the RSUBMIT command to one of the following values:

0	Indicates that the RSUBMIT is complete.
1	Indicates that the RSUBMIT failed to execute.
2	Indicates that the RSUBMIT is still in progress.

Note: If a synchronous RSUBMIT (WAIT=YES) is issued while an asynchronous RSUBMIT (WAIT=NO) is still in progress, all spooled log and output statements are merged into the local log and output windows and the RSUBMIT continues as if it were synchronous. That is, the user does not regain local control until the RSUBMIT has completed. If you don’t want this to happen, use the MACVAR= option in the SIGNON or the RSUBMIT statements so that you can check the progress of RSUBMIT without causing it to execute synchronously.

Example
In the following example, an asynchronous rsubmit is executed in order to download a large data set of sales data. The MACVAR option is used to define the REM_STATUS macro variable to be used later in the local session in order to check the completion status of the remote submit.

```
RSUBMIT WAIT=NO MACVAR=REM_STATUS;
PROC DOWNLOAD DATA=REMT.SALES OUT=LOC.SALES
STATUS = NO;
RUN;
ENDRSUBMIT;
```

Because the remote submit is processing asynchronously, control returns immediately back to the local session. In the following example, a local data set is created and then the REM_STATUS macro variable associated with the asynchronous remote submit is queried in order to determine when to execute the data step to merge the downloaded data set with the newly created local data set.

```
DATA LOC.MARCH;
DO I = 1 TO NREGIONS;
/* create local data set */
END;
RUN;

%MACRO MERGE;
%IF &REM_STATUS = 0 %THEN
DATA TOTAL;
SET LOC.SALES LOC.MARCH;
RUN;
%MEND;

%MERGE;
```

Cross Environment Data Access (CEDA)

The advantage of **CEDA** is that you can transfer your data files from one host to another, or NFS-mount a disk from another host and automatically be able to access your data without any extra steps.

Version 7 of the SAS System recognizes that diverse and distributed computers have become more common than they were when Version 6 was introduced. It is no longer unusual to have a site where multiple CPU's share access to a single disk or to a single networked file system. CEDA is the facility that allows any Version 7 SAS data file created on any directory-based host (e.g. SOLARIS, WINDOWS, HPUX, VMS, etc.) to be read by the SAS System running on any other directory-based platform. With CEDA, SAS data files (managed by the base engine) will be accessible across platforms.

In Version 6, the SAS System required you to use either data transfer services or remote library services in order to access a data file created by and residing on another host. Data transfer services is part of SAS/CONNECT and is a bulk data transfer mechanism that transfers a disk copy of the data and performs the necessary conversion of the data from one platform's representation to another's as well as any necessary conversion between SAS releases. This requires that a connection be established between two SAS sessions by using the SIGNON command and then executing either PROC UPLOAD or PROC DOWNLOAD to move the data.

Remote library services is part of both SAS/SHARE® and SAS/CONNECT and gives the user transparent access to remote data through the use of a LIBNAME statement. However, remote library services also has some prerequisites. When used with SAS/SHARE, a SHARE server must be previously invoked by a server administrator to give the local user access to the remote data. When used with SAS/CONNECT, you must establish a connection between two SAS sessions with the SIGNON command before issuing the LIBNAME statement that points to the remote data.

CEDA eliminates the need to execute any other procedure, maintain a running server, or even SIGNON to the remote host.

For example, if you have a SAS data file that resides on an HP UNIX system and you want to use it on your Windows PC, you could simply FTP the file to your PC and reference it from your SAS program as follows:

```
C:\>FTP MY.UNIX.NODE.COM
FTP>BINARY
FTP>GET UNXDATA.SAS7BDAT
FTP>QUIT

LIBNAME UNX ‘.’;
PROC PRINT DATA=UNX.UNXDATA; RUN;
```

It is important to understand that CEDA does not replace data transfer services or remote library services because of the following restrictions:

- CEDA is limited to Version 7 SAS data files - views and utility files are not supported.
- Update opens are not supported. (Input and Output opens are supported.)
- No WHERE expression optimization with an index.
- Limited to directory based platforms (bound libraries on MVS and CMS files are not part of CEDA)

If your application can operate within these restrictions, then CEDA is a simpler cross-platform data access strategy than the Version 6 data transfer services and remote library services. If your needs go beyond these restrictions, then data transfer services and remote library services are still available to provide access to remote data across all platforms and releases of SAS.

The non-MVS user cannot use CEDA to reference an MVS bound library. However, the Version 7 SAS System for MVS supports unbound (or directory-based) libraries processed with the Hierarchical File System (HFS) of UNIX System Services. As an MVS user, you can use IBM's NFS Client to get to UNIX files from an MVS SAS session. As a UNIX user, you can use IBM's NFS Server to reference MVS HFS files from a UNIX platform. CEDA performs the necessary translations so that these cross-platform references seem like local references.

By default, the SAS System creates new files using the native representation of the CPU running the SAS System. In other words, when using a PC, you create a file with ASCII characters and byte swapped integers. Two new options – **OUTREP** and **TRANTAB** – have been created to give you complete flexibility with CEDA. These options are both data set and LIBNAME options and can be used together or individually. As data set options they apply to an individual open; as LIBNAME options, they provide defaults for the entire library.

The **OUTREP** option allows you to create new files in a foreign host format and therefore is used on OUTPUT opens. This is useful when the readers of a file will be using a different CPU than the creator of the file. In the following example, an administrator running on MVS may wish to create a file on an NFS system. The readers of this file will all be running HP UNIX. The creator can force the data representation to be in the readers' format by specifying OUTREP=HP_UX. The readers will get better performance because reading the file does not require any data conversion.

```
DATA A(OUTREP=HP_UX);
```

The valid values for OUTREP in Version 7 are:

- ALPHA_OSF
- ALPHA_VMS
- HP_UX
- MVS
- OS2
- RS_6000_AIX
- SOLARIS
- VAX_VMS
- WINDOWS

The **TRANTAB** option allows you to provide a translation table that is used for character conversions. SAS searches for translate tables in SASUSER.PROFILE.CATALOG and in SASHELP.BASE.CATALOG. For example, if you had a foreign data set named FOO.A and you wanted SAS to apply the translate table MYTABLE to the data set to translate the characters from foreign encoding to local encoding, you would specify:

```
LIBNAME FOO ‘.’ TRANTAB=MYTABLE;
PROC PRINT DATA=FOO.A;
```

Messaging Services

Messaging services provide both a direct and indirect means of inter-program communication. They facilitate the development of “thin” client applications. Messaging services also allow for the exchange of information independent of location, time of execution, and speed of execution of the client and the server application.

The benefits of client/server applications are proven and many. The primary benefits include providing access to all of the resources on your network as well as maximizing the use of these resources. However, as client/server processing has been adopted and implemented by the business community, additional requirements have emerged.

In today's complex business world, tasks are best accomplished by a series of programs that work together as an application to produce a result. These programs can be spread across multiple computing environments that may or may not be homogeneous.

However, one requirement remains the same whether all of the programs that comprise an application run on a single processor or each program runs on a separate heterogeneous processor: programs must communicate with each other in order to accomplish the goal of the application. The message facility that is available in the SAS System can address all of these needs by using a flexible method of data exchange through messages.

The Direct-Messaging Concept

This section describes the direct-messaging concept and introduces the messaging services that have been added to SAS/CONNECT to allow you to write applications that can communicate with each other on a single processor or across a network. You can develop "thin" client applications that talk to "fat" servers. You can implement applications that perform parallel processing and load balancing.

Typically, one program communicates with another program by calling it directly. This can put unnatural restrictions on your applications that add complexity and hinder the flow of information. Messaging allows applications to communicate by sending each other data in messages. Any action can be taken upon receipt of a message and acknowledgments can be returned to the sender if and when appropriate.

Messaging, in its simplest form, requires that both the client and the server portions of the application be active at the same time. This is called **direct-messaging**. In other words, the client cannot send a message unless a server is listening for a message.

Figure 1 illustrates the basic structure of direct-messaging.

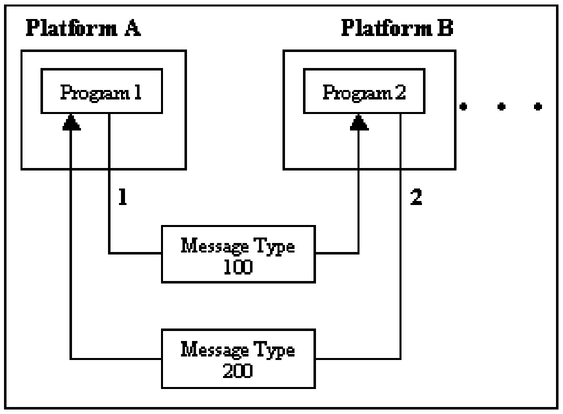

Figure 1 – Direct-Messaging

In figure 1, program 1 sends program 2 a message with a message type of 100 as shown by path #1. Program 2 receives message type 100 and generates a response with a message type of 200 that is sent back to program 1 as shown by path #2. Programs 1 and 2 are shown running on separate platforms. These programs could run on a single platform or separate platforms of the same or unique type. Also, any number of programs can be simultaneously communicating using direct-messaging.

The direct-messaging facility allows basic and flexible message construction, transmission, and notification services which span operating system and hardware boundaries across the enterprise. Messages are free-form. Their structure, which is defined by the application developer, may range from a simple collection of variables to complex hierarchies of SCL lists. Additionally, messages may include one or more attachments which can take the form of SAS data sets or filtered subsets, catalogs or catalog entries, utility files, and external files.

Each message contains a message type field. This field is used to define the set of message types that are meaningful to a particular program. When a program receives a message with a known message type, it knows the layout of the data contained in the message body and can take the appropriate action based on the values of the data.

Direct-Messaging Benefits

Messaging enables application developers to deploy multi-tiered distributed applications. This multi-tiered design allows you to separate and centralize business and data access to the server portion of the application. You can then implement a thin client application that requires little or no maintenance. Not only is it easy to segment your logic into individual programs, but these programs can execute on the host that best meets your data and resource requirements.

To illustrate these benefits consider a three-tiered implementation of a business application. The first tier could be the thin client piece which is a GUI user interface. The middle tier would then contain the business logic that is needed to manipulate data and to produce information. The third tier would perform the data access logic that is necessary to read or write the data source.

Any piece of this application could be modified without changing the other tiers of the application. For example, the data source could change from a DB2® data base to an Oracle® data base and only the third tier, the data access logic, would need to be changed.

SAS/CONNECT provides an SCL interface to direct-messaging that allows you to develop integrated AF and FRAME applications that can communicate through a basic yet flexible interface. In addition, the TCP/IP access method is the only access method that supports direct-messaging.

Application Design

When designing a direct-messaging application, the client and server section must be choreographed to not block one another. That is, you want to make sure the client and server applications are completely clear as to what to expect from each other.

A typical server application would execute the following steps:

1. Initialize environment.
2. Announce availability to the world.
3. Listen for any messages.
4. Respond to messages when required.
5. Repeat steps 3 and 4 as needed.
6. Announce unavailability to the world.

A typical client application would execute the following steps:

1. Initialize environment.
2. Announce availability to the world.
3. Establish communication with a specific server.
4. Send messages.
5. Process any responses from the server.
6. Repeat steps 4 and 5 as desired.
7. Notify server that execution is complete.
8. Announce unavailability to the world.

To accomplish the above steps, direct-messaging provides a set of SCL methods.

The Indirect-Messaging (Queuing) Concept

Often the programs that make up an application need to run on their own schedules, independent of the other programs. Also, as applications become more distributed, businesses will strive to simplify their networks and to minimize the number of direct connections that must be maintained and restarted in the event of a network failure.

This section describes the **indirect-messaging** (queuing) concept and introduces the messaging services that have been added to SAS/CONNECT to allow you to write applications that communicate asynchronously with each other. That is, one application could send messages to one or more target applications that may not be currently running and that may not run for several more hours or days. These services are extremely adaptable which can minimize the cost of restructuring your applications to meet your ever-changing business needs.

In some instances, you do not want the programs that make up your application to run at the same time or to be synchronized so that one side sends a message and waits for a reply before it can send another message. These restrictions disappear with indirect-messaging (message queuing). SAS message queuing enables programs to communicate indirectly by placing messages on queues in storage. Therefore, the pieces of your application can run independently of each other, can run at different speeds and times, and can run without a direct connection between them.

SAS message queues provide a basic and logical means of communication. Programs communicate indirectly by delivering messages to queues and by fetching from or browsing messages in queues. The message queues are administered by a queue manager. The queue manager is part of the DOMAIN server and is enabled by running with the COLLECTION option. The DOMAIN server provides a variety of intercommunication services with SAS/CONNECT.

The queue manager is a server process that is responsible for allocating the queues, maintaining access information for each of the queues, and administering the messages that belong to each queue. Queues can be designated as permanent which means that the queue manager is responsible for storing the messages sent to this type of queue and for maintaining their persistence until the messages are fetched.

Figure 2 illustrates the basic structure of the SAS message-queuing facility.

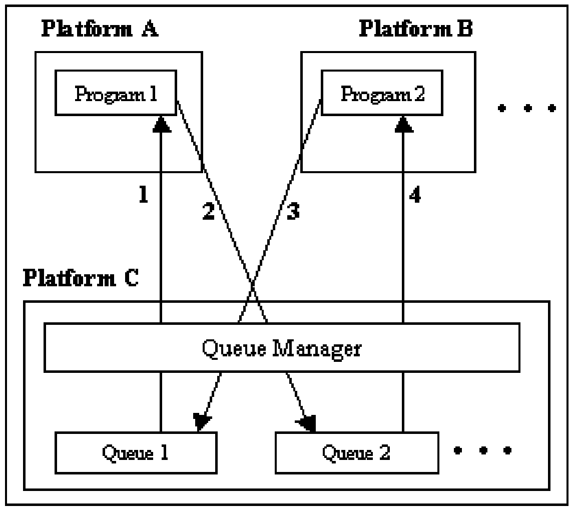

Figure 2 – Indirect-Messaging

In figure 2, program 1 receives messages from queue 1 (as shown by path #1) and writes messages to queue 2 (as shown by path #2). Likewise, program 2 receives message from queue 2 (as shown by path #4) and writes messages to queue 1 (as shown by path #3). The ellipses in the figure indicate the ability to have n number of programs communicating using n queues. Also, this figure shows the programs and queue manager each executing on a different platform. This is only one possibility; they can execute each on a different platform, all on the same platform, or any combination in between. It should also be noted that multiple programs can read or write from the same queue; you do not have to have a separate queue for each program.

Programs can be developed to communicate in either of two modes: one-way (datagram) or two-way (reply). In other words, in a datagram mode of operation, program 1 would put a message on a queue but would not expect a reply response. This is illustrated in the figure above by path #2 only. In a reply mode, program 1 would put a message on a queue and would expect a reply message to be sent to a designated reply-to-queue by program 2 after program 2 receives the original message. This is illustrated in the indirect-messaging figure by program 1 sending the initial message (path #2), program 2 fetching this message (path #4), program 2 sending the reply (path #3), and finally program 1 fetching the reply (path #1).

It is important to note that programs 1 and 2 are communicating without a direct connection between them. Therefore, they are not required to run at the same time or at the same speed. The target program could be busy when a message is put in its queue. In fact, the target program may not run for hours or days after messages for it have been put on its queue. You have total freedom to schedule the pieces of your application based on your business requirements.

A queue can be defined as permanent or temporary. Permanent queues remain until they are explicitly deleted, while temporary queues are implicitly deleted when closed. Permanent queues may either contain persistent or non-persistent messages. Persistent messages are stored on disk, and therefore guaranteed to persist through queue open and close boundaries, as well as a queue server process shutdown.

A message is deleted from a temporary queue after it is fetched by an application, after the queue is closed, or upon a queue server process shutdown. A message sent to a permanent queue is stored on disk for retrieval by any number of applications and remains intact in the event of a queue server restart.

Messages in a permanent queue are deleted only when fetched from the queue. This guarantees that a message in a permanent queue will remain there for any number of applications to browse, will persist through queue open/close boundaries, and will be removed from the queue only when it is fetched from the queue.

Indirect-Messaging (Queuing) Benefits

There are many benefits to using the SAS message queuing facility for implementing your distributed applications. As is the case with applications that are currently developed with SAS, an application developed with the message-queuing facility is completely portable. There are two interfaces available for using SAS message queues:

- an SCL interface
- a functional interface for use through a SAS data step or a SAS macro

In addition, the TCP/IP access method is the only access method that supports indirect-messaging.

Because the message-queuing facility is completely integrated with the SAS system, you continue to have the same portability that you have come to expect from your SAS applications.

A significant benefit of using SAS message-queuing for your distributed applications is that the communicating programs run independently of each other with respect to time. This indirect mode of communication has several positive results. Because the programs communicate indirectly using message queues, each program is completely removed from the interface of any other program. Therefore, an individual program could be modified to execute different logic that is based on message receipt. It could be moved to execute on a different platform, or it could be rescheduled to run at a different time and absolutely none of these things would require any changes to the other programs that make up the application. Also, individual programs could be added or deleted without any disruption to the overall application.

Another benefit of the ability to run communicating programs independently is that there is never a direct link between them. As an illustration, a program that needs to send a message to a queue connects to the queue, sends one or more messages to the queue, retrieves responses if appropriate, and then disconnects. Connections are not left idle while one program waits for a response from another. This helps to minimize the number of active connections that needs to be maintained in the network and it facilitates network restart in case of failure.

The structure of the SAS message-queuing facility insulates application developers from the details of the network. The queue manager is solely responsible for maintaining the queues and for ensuring that the messages in the queues reach their destination when requested and are not lost.

The queue manager is also responsible for establishing the information that is needed by the network protocols being used to transmit the messages to and from the queues. Because the applications programmer is not distracted by the networking details, attention can be focused solely on the business needs and the application logic necessary to meet these needs. The more time and thought that can be given to the business algorithm and flow of data the faster and better the application becomes at delivering the desired information.

A general rule of thumb for writing distributed applications is to "keep the logic as close to the data source as possible in order to minimize network traffic and the cost of client/server computing". Because the SAS message-queuing facility allows complete independence, you have total flexibility with the structure of your distributed application and, therefore, the ability to minimize the cost of your client/server computing.

Application Design

With indirect messaging, two or more applications communicate with each other indirectly using message queues. Therefore, the applications do not have to be running at the same time. The data is sent in the form of a message and saved on a message queue until the receiving application is ready to fetch it. As a precaution, messages are written to disk. Therefore even if your queue terminates, the application can retrieve its messages.

A typical server application would execute the following steps:

1. Initialize environment.
2. Announce availability to the world.
3. Locate the collection manager.
4. Establish communication with queue(s).
5. Check queue(s) for any messages.
6. Respond to messages when required.
7. Repeat steps 3 and 4 as needed.
8. Announce unavailability to the world.

A typical client application would execute the following steps:

1. Initialize environment.
2. Announce availability to the world.
3. Locate the collection manager.
4. Establish communication with queue(s).
5. Send or receive messages from an opened queue(s).

6. Repeat steps 4 and 5 as desired.
7. Release queue(s).
8. Announce unavailability to the world.

Remote Objecting Services

With **remote objecting**, the SAS/CONNECT client/server tool set is more integrated than ever with the SAS application development environment. Remote objecting is an extension of the messaging capabilities applied to the object-oriented application environment. It allows SAS/AF® developers to distribute selected partitions of their encapsulated object frameworks across remote session boundaries. In other words it is a process for remoting SAS frame objects between a local and a remote SAS session.

Enablement is through a dynamic remote object that uses methods for:

- instantiating an object in a remote SAS session
- building a method parameter list
- invoking a method on the remote instance and receiving the output results
- destroying the remote instance

Remote objects can be created and manipulated on a remote host, while the results are sent to the user's local host. Therefore, you are no longer limited to object access only within your local environment, you now have the flexibility to do the same operations remotely.

Before using remote objecting services, you must establish a connection between a local and a remote host by using the SIGNON statement.

To use remote objecting, you would execute the following steps:

1. Instantiate an object remotely by supplying a destination and class name.
2. Begin a method invocation by supplying a method name.
3. Build the argument list through typed ADD_METHOD invocations. Define value, mode, and an optional name tag.
4. Invoke the method.
5. Post-process the return_list, which is an argument list-ordered list of update and output mode arguments (names are promoted if supplied on the invocation parameter list).
6. Repeat steps 2 through 5 as needed.
7. Destroy the remote instance.
8. Repeat per step 1 as needed.

The remote objecting services are provided by the ROBJECT class. The instance methods defined to the ROBJECT class enable applications to create and act upon remote objects.

Agent Services

Agent services are used with compute services and messaging services to provide client/server-based task management for the nodes across your network. An agent is SAS source code that is used by the DOMAIN Server to control the execution of a task on a remote node. Once the task has finished executing, an agent may be designed to send a completion notification to a message queue for the client application, or the client can simply check the DOMAIN Server to see if the agent has completed execution
The source that composes an agent may be predefined and stored with the DOMAIN Server, or the source may be defined dynamically with the request for the agent to execute.

The following client/server-based agent processing is available:

- distributed agent processing
- periodic agent processing
- conditional agent processing
- parallel agent processing

These services can be used together to maximize the flexibility and functionality of a client/server application. For example, using agent services, an application can be designed to execute autonomously on a periodic basis. The application can make conditional decisions to satisfy its goals, including submitting other processes for execution on various hosts on the network.

Using compute services and distributed messaging, agents can be used to distribute work across nodes on a network that is most efficient for a client/server application. This **distributed agent processing** was designed for client/server applications which benefit from unattended, asynchronous execution.

Agents transparently utilize SAS/CONNECT to sign on a designated location with a designated user identity and then execute the agent's source code. As agents are processed, the log and output window are spooled until the agent has finished executing. Upon completion, a monitoring client may retrieve the spooled execution log and output or purge it. The agent will then signoff the designated node.

In addition, agents may be designed to use the indirect-messaging facility to communicate information to client applications using message queues. This allows agents to be constructed to generate alerts and/or dynamic reports as they execute. These conditions and information are then manifested as messages and written to queues which are monitored by interested clients.

The initiating client SAS session is not required to remain active while the agent executes, unlike a conventional local/remote SAS/CONNECT session configuration.

Periodic agent processing is a client/server-based implementation of a task scheduler. SAS has expanded the traditional task scheduling service to the client/server environment by allowing agents to run on remote hosts across a network. In general, other task management products are limited to running tasks only on the local host.

Agents can be defined so the DOMAIN Server schedules them to run at a specific date and time or on a repetitive (periodic) run schedule. Periodic scheduling can be defined to start on a specific hour and minute on either a day-of-the-week schedule or a month-and-day schedule.

Periodic report generation agents can be defined to distribute electronic reports to clients in the form of message attachment packages which are delivered to queues monitored by desktop presentation applications.

For example, throughout the day, decision support transactions can be queued for off-hours execution. At a specified time, an application server agent can be scheduled to service the transaction queue, returning results to the client which made the request. The results might be in the form of a response message with the results as attachments delivered to the designated message queue.

An agent-based application can be implemented to take advantage of conditional sequential-step logic**. Conditional agent processing** allows an agent to spawn additional agents based on conditions encountered during its execution.

For example, a filtering agent can be constructed which processes operational data stores, searching for anomalous entries. Upon locating such an anomaly, a conditional agent may deliver an alert to an audit queue which is monitored by a manager with responsibility for the operational system being filtered.

As well, other conditional logic may spawn an extraction agent to generate consolidation summaries. Once the summaries are ready, another agent may stage those summaries in the form of data set attachments to update messages delivered to a designated message queue. Replication agents might be used to monitor the queue. Once the message is received, the replication agents execute on the decision support servers which provide daytime access, completing the replication process. If problems are encountered, alerts can be delivered accordingly.

Still another alternative is **parallel agent processing**. Given that agents can spawn other agents, independent tasks can be partitioned into subordinate agents. These subordinate agents can be launched by a controlling agent to execute on various nodes within the network or as separate processes on the same node. This way each agent can be processed at the same time in parallel with one another.

Using a completion notification message queue, the controlling agent could determine when each of the subordinate agents have finished execution. The controlling agent could contain logic that synchronizes the work performed by the subordinate agents.

Client/server applications can use parallel processing to minimize overall elapsed time of a task's execution. By breaking the application into discrete encapsulated tasks, each independent task can be processed concurrently to reduce the execution time rather than running each task sequentially.

The following SCL code fragment is an example of periodic agent processing. It demonstrates how you would set up an agent to run every Monday morning at 8:30 am.

```
/* initialization */
init:
   dcl object station;
   dcl object agentObject;

   rc = 0; agentObject=0;

   stationid = loadclass('sashelp.connect.station');
   station = instance(stationid);
   agentClass = loadclass('sashelp.connect.agent');
   agentObject    = instance(agentClass);

 /* prepare to connect to Domain server */

   domainserver='//DomainNode/DomainService';
   collectionname='SugiDemo';
   agentname='dailyReport';

   station._open(collectionname, rc);


agentObject._setDomainInfo(domainserver,collectionname,
           rc,"",station);

  /* include Agent program—referenced by fileref myagent--
      into PREVIEW buffer      */

rc=preview('include','myagent');

  /* Define the agent to run every Monday morning
         at 8:30 a.m.      */

descriptor='My Morning Report';
runlocation='//SpawnerNode/SpawnerService';
securityInfo='';

  /* set up the scheduling parameters             */

schedlist=makelist();
rc=setnitemc(schedlist, 2,  "run_dow");
rc=setnitemc(schedlist, 8,  "run_hour");
rc=setnitemc(schedlist, 30,   "run_minute");

agentObject._defineAgent(agentname,rc,descriptor,
     runlocation,securityInfo,schedlist,"notifyQueue");

  /* clear out the  PREVIEW buffer                */
rc=preview('clear');

   station._close(rc);
 return;
```

Encryption Services

Encryption services protect data that is sent between hosts across a network. Encryption services use a reversible algorithm to convert plain-text data into an unintelligible form, thus protecting data from being used by unauthorized parties.

With a license for SAS/SECURE™, you can use the encryption services of RSA Data Security, Inc.'s BSAFE Toolkit and/or Microsoft's CryptoAPI. The encryption services that you use are platform dependent. Alternatively, you may use a SAS proprietary form of encryption services that are not platform dependent.

The BSAFE Toolkit is supported on the following types of UNIX, OpenVMS, and OS/2® platforms:

- AIX®
- Digital UNIX
- HP-UX
- Solaris 2
- Alpha/VMS
- VAX/VMS
- OS/2

NOTE: SAS/SECURE using RSA Data Security, Inc.'s BSAFE Toolkit for MVS is currently under development.

Microsoft's CryptoAPI is supported on the following Windows platforms:

- Windows 95 (as part of Internet Explorer 3.0+)
- Windows NT 4.0+ (as part of the operating system)

Use of Microsoft's CryptoAPI requires no additional product license. Microsoft's CryptoAPI supports both strong encryption and weak encryption. In order to have strong encryption on Windows hosts, the Microsoft Enhanced Cryptographic Service Provider must be installed. To have weak encryption, the Microsoft Base Cryptographic Service Provider must be installed.

You can use the SAS proprietary encryption services on all platforms. Encryption services provided by SAS require no additional license.

Encryption services are available with the following communications access methods on the supported hosts:

- TCP/IP
- DECnet
- NetBIOS

For example, you can use encryption services when connecting two Windows hosts using any of the access methods listed above. Also, you can use encryption services when connecting a Windows host to a UNIX host using the TCP/IP access method.

Encryption services are packaged in two forms because of export key-length restrictions:

- North American
- International

The North American version is available to North American customers and supports strong encryption -- 1024-bit RSA keys in combination with the following algorithms:

- RC2 using 128-bit keys
- RC4 using 128-bit keys
- DES using 56-bit keys
- Triple DES using 168-bit keys

The International version is available to International customers and supports weak encryption -- 512-bit RSA keys in combination with the following 40-bit key algorithms:

- RC2
- RC4

The encryption algorithms as well as the SAS Proprietary algorithm are defined as follows:

RC2 is a proprietary algorithm developed by RSA Laboratories as an alternative to DES. RC2 is a block cipher that encrypts data in blocks of 64 bits. The size of the output of the algorithm will always be a multiple of the block size. The RC2 key size can range from 8 to 256 bits.

RC4 is a proprietary algorithm developed by RSA Laboratories, RC4 is a stream cipher. A stream cipher encrypts one byte at a time. The RC4 key size can range from 8 to 2048 bits.

DES is an acronym for Data Encryption Standard and was developed by IBM. DES is a block cipher that encrypts data in blocks of 64 bits using a 56-bit key.

Triple DES executes DES three times on the data in order to get a key size that is three times that of DES. DES is a block cipher that encrypts data in blocks of 64 bits using a 56-bit key.

SAS Proprietary requires no additional product license, SAS Proprietary provides basic encryption services on all platforms.

The key sizes used depend on the encryption software that is available on your host and the value assigned to the NETENCRKEYLEN option described in the next section.

Data Encryption Options

The following SAS options are used to set the encryption services attributes:

```
NETENCRYPTALGORITHM = ("algorithm1",
"algorithm2", ...)
```

You may use the alias NETENCRALG.

To specify more than one algorithm, surround the algorithm names with parentheses and separate with commas. If there are embedded blanks in the algorithm name, enclose each algorithm with quotation marks.

You set this option at the server and optionally at the client to specify one or more encryption algorithms to use in a SAS/CONNECT session. The client and the server, however, must share an encryption algorithm in common. If you specify the option in the server session only, the client side attempts to select an algorithm that was specified at the server. If you also set the option at the client and specify an algorithm that is not specified at the server, the local host's attempt to connect to that remote host fails at signon.

Valid values for this option are:

- RC2
- RC4
- DES
- TripleDES
- SASProprietary

```
NETENCRYPT = yes | no or NETENCRYPT |
NONETENCRYPT
```

You may use the alias NETENCR.

You set this option at both the client and server. At the server, this option specifies that encryption is required for each connection from a client SAS session. At the client, it specifies that the client must connect only to a server that supports encryption.

The default for this option is that encryption is used if the NETENCRYPTALGORITHM option is set and if both the client and server sides are capable of encryption. If encryption algorithms were specified but either the client or server side is incapable of encryption, then encryption will not be performed.

Encryption may not be supported at the client or server for these reasons:

- You run a release of SAS (prior to Version 7) that does not support encryption.
- You run a release of SAS that does not have the SAS/SECURE product licensed.
- You specify encryption algorithms in the client and server SAS sessions that are incompatible.
- You do not have a cryptographic service provider installed on your Windows system.

NETENCRYPTKEYLEN = n

You may use the alias NETENCRYKEY.

You set this option in either the client or server SAS session. It specifies the key length to be used by the encryption algorithm.

Valid values for this option are:

- 128 - specifies strong encryption (1024-bit RSA and 128-bit RC2 and RC4 key algorithms)
- 40 - specifies weak encryption (512-bit RSA and 40-bit RC2 and RC4 key algorithms)
- 0 - no value is set (default)

If you require the extra security provided by strong encryption, then set the NETENCRYKEYLEN option to 128. If you prefer weak encryption in order to save CPU, then set the NETENCRKEYLEN option to 40.

If you try to connect to a host that is capable of only weak encryption with a host that is capable of both strong and weak encryption, the connection is made with weak encryption. If both hosts are capable of strong and weak encryption, then strong encryption is used. To explicitly set weak or strong encryption, set the NETENCRKEYLEN SAS option.

Example
The following statements are specified at the local host:

```
options netencralg=rc4;
options remote=unxnode comamid=tcp;
signon;
```

The NETENCRALG option specifies that the RC4 algorithm be used for encryption in the local host session.

The following statements illustrate the content of the executable file that a UNIX spawner program uses to start SAS and to specify encryption in a SAS/CONNECT remote host session:

```
#________________
# mystartup
#________________
#!/bin/ksh
. ~/.profile
sas -dmr -noterminal -no$syntaxcheck \
-comamid tcp -netencr -netencralg rc4
#_______________
```

The NETENCR option specifies that encryption is required by any local host that connects to this remote host. The NETENCRALG option specifies that the RC4 algorithm be used for encryption of all data that is exchanged with connecting local hosts.

Conclusion

This paper gives an overview of many of the new features that have been added to SAS/CONNECT for Version 7 to both facilitate and secure your distributed applications. These features were all designed to give you more flexibility and functionality to design your distributed applications. Now more than ever, you have the ability to decide where your data reside, where your processing takes place, and when you want to schedule the pieces of your applications. The bottom line of this flexibility is to minimize the cost of your distributed applications by moving the processing as close to the data source as possible in order to minimize network traffic. And the encryption services meet the constantly growing concern for securing your sensitive data as they flow over the network.

More details on the specific topics will be provided in future papers, the SAS external WEB site, and on-line Version 7 documentation. In addition, the messaging services, both direct and indirect, are production with the 6.12 maintenance release of SAS/CONNECT. More information can be found on the SAS web site.

Acknowledgments

Many people were involved in the development of the features described in this paper. They include:

Tony Dean
Barbara Foster
Cheryl Garner
Glenn Horton
Steve Jenisch
Stephanie Smith
Kanthi Yedavalli

Author

Cheryl Garner
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
(919) 677-8000
sascgg@wnt.sas.com (Cheryl Garner)