Implementing Distributed SAS¤ Applications via Messaging

Cheryl Garner, SAS Institute Inc., Cary, NC
Tony Dean, SAS Institute Inc., Cary, NC
Stephanie Reinard, SAS Institute Inc., Cary, NC

Introduction

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. 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. 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 and queuing facilities that are available in the SAS System can address all of these needs by using a flexible method of data exchange through messages.

This paper will describe the direct-messaging and message-queuing concepts and introduce the messaging services that have been added to the SAS System to allow you to easily 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. You can even implement 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.

The Direct-Messaging Concept

Typically, one program communicates with another program by directly calling it. 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 acknowledgements can be returned to the sender if and when appropriate.

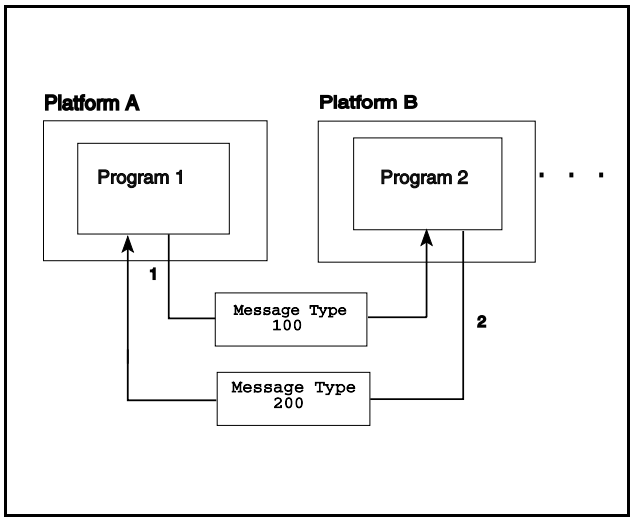

Figure 1

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. In this figure, 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 message type 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, 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.

The SAS System now 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.

The Message-Queuing Concept

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 SAS 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. This removes the distinction between client programs and server programs. Because the programs are on an equal level of communication, there is no functional difference between what may be designated as the client portion and as the server portion of an application.

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 a program 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.

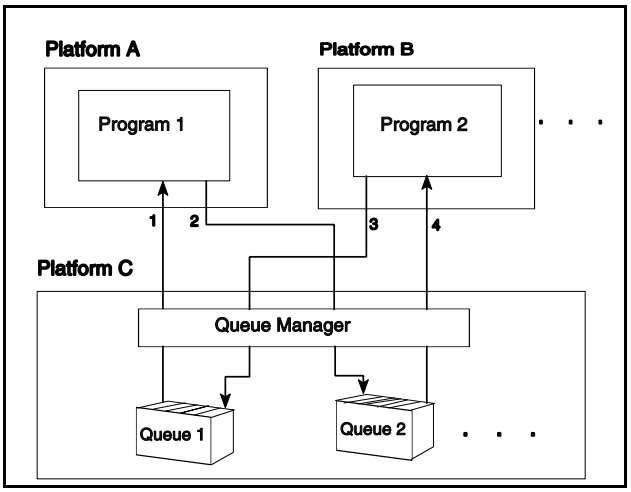

Figure 2

Figure 2 illustrates the basic structure of the SAS message-queuing facility. In this figure, 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 Figure 2 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 Figure 2 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.

The communication between programs that use the SAS message-queuing facility can be one-to-one, one-to-many, many-to-one, or any combination of these to provide you with complete flexibility in the structure of your application. These structural combinations can be used with the datagram and reply message flow modes, as previously discussed. In a one-to-one relationship, a client sends messages to a queue and the receiving program retrieves the messages in a time-frame dictated by your business needs. In a one-to-many relationship, a single client could send messages to a queue that is serviced

by the same program that runs on multiple platforms to provide load balancing. Another scenario would be for a single client to send messages to a queue that is serviced by multiple subtasks of a program that can run concurrently to provide parallel processing. In a many-to-one relationship, you could have multiple clients that send messages to a queue that is being serviced by a single server. The clients could run independently of the server's speed and would never need a direct connection between any of the clients and the server. With all of these relationships, the receiving program can optionally generate replies that are based on the messages that it retrieves.

A queue can be designated as temporary or permanent. A message is deleted from a temporary queue after it is fetched by an application, after the queue is closed, or upon network failure. 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 network failure and 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.

Message-Queuing Benefits

There are many benefits to using the SAS message queuing facility for implementing your distributed applications. The following paragraphs present several benefits and you may think of others as you visualize your distributed applications implemented with SAS message queues.

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 and a functional interface for use through a SAS data step or a SAS macro. 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 via 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 all combinations of one-to-one, one-to-many, and many-to-one application structure as well as datagram and reply modes of communication, you have total flexibility with the structure of your distributed application and, therefore, the ability to minimize the cost of your client/server computing.

A Working Example: The Application Development Manager

The following sections of code are part of a sample application called "The Application Development Manager (ADM)" that was developed with the SAS System using the messaging and queuing facilities. The ADM is designed to facilitate application development in a work group by allowing any number of people to work on an application that is contained in a single catalog. The ADM server maintains a single centralized copy of the application catalog and allows users to check-out and check-in individual entries. A message is broadcast to all members of the work group to notify them whenever an updated entry is checked back in to the central catalog or a new entry is added to the central catalog. They have the option of receiving any number of the updated entries so that they can be assured of running the most current version of the application. Because the notifications are sent to message queues, the members of the work group can also choose to request the entries at the time of the notification or at any time in the future.

The following actions are available to the users of this application:

1. register - add your identity to the work group that is defined for a specific application. A queue is also created for you as part of the registration.

2. check-out - get a copy of a specific entry from the central catalog and put it in your playpen for development

3. check-in - return an updated entry to the central catalog. The entry will be updated in the central catalog, compiled, and a message about the updated entry will be broadcast to each of the work group members' queue.

4. add - add a new entry to the central catalog. The entry will be updated in the central catalog, compiled, and a message about the updated entry will be broadcast to each of the work group members' queue.

5. query - query your message queue for any outstanding messages.

6. receive - receive one or more new or updated entries from the central catalog.

7. purge - delete any broadcast messages that you may have accumulated.

8. deregister - remove yourself from the work group that is associated with a specific application.

To implement the ADM application, each of the above actions is mapped to a message type. In order to perform an action, the client sends the ADM server a message. The message contains a message type, representing an action, as well as any information needed by the ADM server to process the requested action.

The following sections contain code segments to illustrate the implementation of the check-out, query, and check-in actions. Initialization code and error checking have been taken out of these code segments to draw attention to the messaging interface.

CHECK-OUT A CATALOG ENTRY

The check-out action involves a user making a request for a specific entry and receiving an immediate response. The response is either the requested entry or a message indicating why the entry cannot be returned. Direct-messaging was chosen for this portion of the application because the user needs an immediate response to the check-out request in order to continue working.

The following code which was taken from the client portion of the ADM application, allows a user to check out a specified catalog entry. The application was written to send a message type of 30 to the ADM server in order to request that a specific catalog entry be checked out by the user. The ADM server will respond with one of two message types. A message type of 35 has been defined to mean that the requested catalog entry is available. In this case, the entry is received and written to the location passed into this function. A message type of 39 has been defined to mean that the requested catalog entry is locked to another user in the work group. In this case, the client application prints a message and control returns to the main menu.

Four values are assigned to macro variables which are then used by this routine:

srvname - supplied to the _OPEN_ method. This variable contains the service name of the server portion of the ADM application.

uname - supplied to the _SEND_ method. This variable contains the userid of the client.

incat - supplied to the _SEND_ method. This variable contains the four-level entryname of the public catalog that is controlled by the server portion of the ADM application.

outcat - used to build the TLIST parameter that is supplied to the _ACCEPT_ATTACHMENT_ method. This variable contains the four- level entryname of the location to which the received entry will be written.

NOTE: The following code is a portion of an SCL program that is used for illustration purposes only. It is not a complete program.

```
/*******************************************/
/*                                         */
/* CHECK OUT A CATALOG ENTRY               */
/*                                         */
/*******************************************/

/*******************************************/
/* open a station for myself               */
/*******************************************/
stationid =loadclass ('sashelp.connect.station');
station = instance(stationid);
call send(station, '_OPEN_', "TESTPTP", rc);

/*******************************************/
/* connect to the ADM server               */
/*******************************************/
cnctionid =loadclass('sashelp.connect.cnction');
cobj = instance(cnctionid);
call send(cobj,'_OPEN_', station, srvname, rc);

/*******************************************/
/* send ADM server a check-out request and    */
/* include my userid and the entryname I want.*/
/*******************************************/
call send(cobj, '_SEND_', 30, 0, alist, rc,
   uname, incat);
call send(cobj, '_QUERY_', eventtype, msgtype,
   0, alist, rc);

/*******************************************/
/* a message type of 39 means that the entry  */
/* is already checked out.                 */
/*******************************************/
if msgtype = 39 then do;
   call send(cobj, '_RECV_', rc, str);
   put 'NOTE: ' str;
end; /* msgtype = 39 */

/*******************************************/
/* a message type of 35 means that the entry  */
/* is available                            */
/*******************************************/
else if msgtype = 35 then do;
   call send(cobj, '_RECV_', rc, str );

/*******************************************/
/* receive the requested entry             */
/*******************************************/
   catlist=makelist();
   catlist=getiteml(alist,1);
   outlib=scan(outcat,1,'.');
   outcat=scan(outcat,2,'.');
   rc = setnitemc(catlist, outlib ,,"OUTLIB");
   rc = setnitemc(catlist, outcat , "OUT");
   tlist=makelist();
   catlist = insertl(tlist,catlist,-1);
   call send(cobj, '_ACCEPT_ATTACHMENT_', tlist,
      rc, "COMPLETE");
   put 'NOTE: Requested entry has been
       received.';
end; /* msgtype=35 */
```

```
/*******************************************/
/* Should only receive msgtype 39 (fail) or 35*/
/* (success).  All other msgtypes are unknown.*/
/*******************************************/
else do;
   put ''ERROR:  Received unknown msgtype--'
   msgtype;
   call send(cobj, '_RECV_', rc);
end;
```

QUERY FOR BROADCAST MESSAGES

As the members of the workgroup modify their local copy of the application, they may choose to update their copy by merging any pieces that have been added or modified by others in the workgroup. Different members may have different schedules for updating their private copy. For example, developers may need adhoc updates that occur when they reach certain points in their development. On the other hand, testers may require regular updates based on their testing schedules. To meet the individual needs of each member in the workgroup, message-queuing is used for this portion of the application.

The following code which was taken from the client portion of the ADM application, allows a user to query his or her queue for any existing messages. This portion of the application has been coded to recognize a message type of 100 to mean a broadcast message about a new or modified catalog entry type. These messages would be broadcast from the ADM server portion of the application to notify users of new or updated catalog entries. The message contains a status field that indicates whether the entry was added or modified and the name of the entry that was added or modified. If the entry is a new entry, the message also contains the name of the developer who made the addition. If the queue contains one or more messages, the messages are fetched one at a time and a list of entries is constructed. The new or updated entries can then be selected from the list, fetched from the ADM server, and merged into the user's existing catalog.

One value is assigned to a macro variable which is then used by this routine:

qname - supplied to the _OPEN_method. This variable contains the name of the queue that will be queried for messages.

NOTE: The following code is a portion of an SCL program that is used for illustration purposes only. It is not a complete program.

```
/*******************************************/
/*                                         */
/*      QUERY FOR BROADCAST MESSAGES       */
/*                                         */
/*******************************************/

/*******************************************/
/*  create a station for myself            */
/*******************************************/
stationid =loadclass('sashelp.connect.station');
qstation = instance(stationid);
call send(qstation, '_OPEN_', "TESTQM", rc);

/*******************************************/
/* access my queue                         */
/*******************************************/
queueid = loadclass('sashelp.connect.queue');
queue = instance(queueid);
call send(queue, '_OPEN_', qstation, qname,
   "FETCH", qrc, "POLL");

/*******************************************/
/* the OPEN was successful; are there any     */
/*    messages for me to fetch?               */
/*******************************************/
rc=clearlist(alist);
hlist=makelist();
call send(queue, '_QUERY_', eventtype, msgtype,
   hlist, alist, qrc);

/*******************************************/
/* DELIVERY means I have messages; stay in the*/
/*   loop as long as I have messages.         */
/*******************************************/
do while(eventtype='DELIVERY');

/*******************************************/
/*  message type 100 means new/updated entries*/
/*******************************************/
    select(msgtype);
    when(100)

/*******************************************/
/*  is the status "create" or "update"?       */
/*******************************************/
    call send(queue,'_GETFIELD_',parms,qrc,
       status);
    if upcase(status) = 'CREATE' then do;
       call send(queue, '_GETFIELD_', parms,
          qrc, catname, developer);
       call display("testdata.sapp.addlist.scl",
          catname, developer);
    end;  /* if create */
    else do;
       call send(queue,'_GETFIELD_',parms,
          qrc,catname);
       call display("testdata.sapp.addlist.scl",
          catname);
    end;  /* else update */
    otherwise
       call send(queue,'_RECV_',qrc);
       put 'Unknown message type ' msgtype '.';
       put'Message discarded.';
    end; /* select */

/*******************************************/
/* fetch the next message, if any             */
/*******************************************/
   rc=clearlist(alist);
   rc=clearlist(hlist);
   call send(queue, '_QUERY_', eventtype,
      msgtype, hlist, alist,qrc);
end; /* do while */
```

CHECK-IN A CATALOG ENTRY

The check-in action involves a user sending an entry to the ADM server portion of the application to be merged into the central copy of the catalog. When an entry is checked-in, immediate confirmation of the receipt of that entry is needed. Therefore, direct-messaging is used for this portion of the application. The result of checking in an entry is for the ADM server to broadcast notification to everyone in the workgroup so that they can request the updated entries at their convenience. Message-queuing is used by the ADM server for the broadcast.

The following code which was taken from the client portion of the ADM application, allows a user to check in a specified catalog entry. The client program was written to send a message type of 40 to the ADM server program to indicate a catalog entry check-in. If the ADM server portion of the application has the catalog entry listed as checked out by this user, it will check in the entry and then broadcast a message with a message type of

100 to the users in the workgroup to notify them of the updated entry as appropriate.

Four values are assigned to macro variables which are then used by this routine:
srvname - supplied to the _OPEN_method. This variable contains the service name of the server portion of the ADM application.

uname - supplied to the _SEND_method. This variable contains the userid of the client.

incat - used to build the ALIST parameter that is supplied to the _SEND_ method. This variable contains the four-level entry name of the entry to be checked into the public catalog that is controlled by the server portion of the ADM application.

outcat - supplied to the _SEND_ method. This variable contains the four-level entryname of the public catalog controlled by the server portion of the ADM application.

NOTE: The following code is a portion of an SCL program that is used for illustration purposes only. It is not a complete program.

```
/*********************************************/
/*                                           */
/*        CHECK IN A CATALOG ENTRY           */
/*                                           */
/*********************************************/
/*********************************************/
/* open a station for myself                 */
/*********************************************/
stationid =loadclass('sashelp.connect.station');
station = instance(stationid);
call send(station, '_OPEN_', "TESTPTP", rc);

/*********************************************/
/* connect to the ADM server                 */
/*********************************************/
cnctionid =loadclass'sashelp.connect.cnction');
cobj = instance(cnctionid);
call send(cobj, '_OPEN_', station, srvname, rc);

/*********************************************/
/* prepare to send the ADM server the name of */
/* the entry I wish to check in.             */
/*********************************************/
lib = scan(incat,1,'.');
cat = scan(incat,2,'.');
ename = scan(incat,3,'.');
etype = scan(incat,4,'.');
catlist = makelist();
rc = setnitemc(catlist, "CATALOG", "TYPE");
rc = setnitemc(catlist, cat, "MEMNAME");
rc = setnitemc(catlist, lib, "LIBNAME");
entry = trim(left(ename)) || '.' ||
   trim(left(etype));
rc = setnitemc(catlist, entry, "SELECT");
alist = insertl(alist,catlist,-1);

/*********************************************/
/* send the catalog entry to the ADM server   */
/*********************************************/
call send(cobj, '_SEND_', 40, 0, alist, rc,
   uname, outcat);
if rc ne 0 then
   put 'send failed';
else do;
   rc=clearlist(alist);

/*********************************************/
/* if the ADM server did not accept my entry  */
/* successfully, find out why.               */
/*********************************************/
   call send(cobj, '_QUERY_', eventtype,
      msgtype, 0, alist, rc );
   if rc ne 0 then do;
      str = sysmsg();
      put 'QUERY for response failed' str;
   end;
   call send(cobj, '_RECV_', rc, str);
   put 'Received message = ' str;
end;

call send(cobj,'_DISCONNECT_',rc);
call send(station, '_CLOSE_', rc);
return;
```

SERVER PORTION OF THE ADM APPLICATION

The following pseudo-code is taken from the server portion of the ADM application. The server portion of this application is one main loop that both continuously receives messages with message types that have been defined to this application and performs the appropriate action based on the message type. The server portion of the application receives its messages via direct-messaging and sends messages using both direct-messaging and message-queuing.

The majority of the logic is contained in the server portion of the ADM application. This has the benefit of being able to modify the server portion of this application, either changing or adding functionality, without having to update the user portions of this application.

```
/*****************************************/
/*  Main Server Loop                     */
/*****************************************/
do while (finish='N');
   call send(station,'_QUERY_', eventtype,
      msgtype, 0, alist, cobj, rc );
   if rc eq 0 then do;
      put 'Eventtype is ' eventtype;
      if eventtype = 'DISCONNECT' then do;
         put 'Client ' cobj ' is disconnecting
            from the server.';
      end;
   else do;
      if eventtype = 'CONNECT' then
         put 'Client ' cobj ' has connected to
            server.';
      else do;
         put '  ----  client is ' cobj;
         put 'Msgtype is ' msgtype;
         put 'Receiving message from client';
         select(msgtype);
            when(999)
              /*  stop server message */
              finish='Y';
            when(10)
              /* user wants to register       */
              /* create qname based on userid */
            when(20)
              /* user wants to deregister */
            when(30)
             /* user wants to check out an    */
             /* entry, send the entry as an   */
             /* attachment                    */
            when(40)
             /* user wants to check in a      */
             /* catalog entry. accept updated */
             /* entry as an attachment and    */
             /* broadcast the updated entries */
             /* to all registered users       */
            when(50);
             /* user wants to add a new entry.*/
             /* accept the new entry as an    */
             /* attachment and broadcast the  */
             /* new entry to all registered   */
             /* users                         */
```

```
          when(60);
           /* user wants to receive new or  */
           /* updated entries, send the     */
           /* entries as attachments. Other-*/
           /* wise unknown message type     */
        end;  /* select msgtype */
     end; /* else not disconnect */
  end; /* if rc */
end; /* do while
```

Conclusion

The addition of messaging and queuing to the existing SAS System client/server toolset provides an integrated solution to your most complex client/server application needs. The logical flow of information in messages that underlies the SAS direct-messaging facility allows you to layer your distributed applications to best meet your business needs. The flexibility of the indirect communication that underlies the SAS message queuing facility allows you to implement, to deploy, to modify and to schedule the various programs that make up your applications independently and more efficiently. Used together, direct messaging and message queuing allow you to minimize the cost of implementing and running your client/server applications by allowing your data sources, hardware resources, and information goals dictate the logic structure, the platforms, and the schedules on which to run the programs that comprise your applications.

Acknowledgments

The following SAS Institute employees are responsible for the design and implementation of SAS direct-messaging and message-queuing:

Tony Dean
Steve Jenisch
Stephanie Reinard

Also, thanks to Beth Langston for her help with implementing the ADM application.

****

Author: Cheryl Garner
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
(919) 677-8000 x7941
sascgg@unx.sas.com