

[print](#) | [close](#)

APIs by Example: Data Queue APIs and CL Commands, Part 5

[System iNetwork Programming Tips Newsletter](#)

[Carsten Flensburg](#)

Carsten Flensburg

Thu, 02/22/2007 (All day)

This is the final installment of the articles about the data queue APIs and CL commands. Today, I demonstrate how to use data queues to establish a communication layer between two programs, and I discuss some of the problems and considerations involved.

This example uses local data queues, but the information about data queue setup, the programming of the transaction dialog, and the data queue API calls applies irrespective of where the physical data queues are located. From the communicating programs' view, it's merely a question of addressing the interface that the data queue APIs provide.

Using DDM data queues, the programs communicating through data queues can reside on different servers if required. They can even be on different platforms. For example, both Visual Basic and Java offer programmable interfaces to data queues. If that topic is of interest to you, follow the links at the end of this article for more information about DDM data queues as well as VB and Java data queue support.

As I mentioned in part one of this article series, data queues offer a solution to different types of design and architecture objectives. For example, it could be a need to process a workload from a web application located on a separate server accessible from the Internet, while the production data and business logic are on a back-end server behind the company firewall. Or the situation could involve a CPU-intensive application that should have its system workload moved from expensive interactive CPU cycles to less expensive batch cycles. The latter issue is potentially not only a question of cost, but also of sufficient system CPU capacity and thereby application scalability.

To build a robust and flexible data queue communication layer, the following data protocol and application design aspects are worth considering in the early phase of your development project:

Message Format: What kind of formatting should be applied to the message? For internal applications, mapped data structures might suffice, but if the data exchange involves external partners or programming languages with poor data structure facilities, XML is definitely worth considering.

If you stick with data structures, your design should allow for varying-length subfields, to avoid space constraint problems. To achieve this objective, keep a fixed-length part of the data structure, which includes offset and length fields defining the location and length of the varying-length subfields. This concept is similar to what is used by many of the APIs that return information in a receiver variable defined by a data structure.

Regardless of the format that you choose, divide the message into at least two sections:

1. A message header section containing the control data of the message, such as protocol ID, request ID and type, reply key, message format/version, list control information, offset to and length of message data, error and event codes, and other similar information, if applicable.
2. A message data section containing the actual request or response data.

Message Extensibility: How should we handle changes in or additions to the message format? For XML-formatted messages, this concern is well covered; otherwise, including message format and/or message version/release/modification level information in the message header provides the ticket to safely modify or extend a message in the future.

Message Size Limit: Some message types are by nature limitless in size. For example, when embedded lists or textual descriptions are involved, it might be impossible or senseless to define the maximum message length to cover the longest imaginable message length. Instead, you have to include a mechanism that allows a message to be split into more segments.

One way of providing such a mechanism is adding a correlation-ID and an end-of-message flag to the message header information. The correlation-ID contains the sequence number of the message segment, and the end-of-message flag is used to signal when the current message segment is the last in the chain.

For some purposes, such as displaying lists, a better approach is to have the client request the additional segments one by one as new pages are required. There's no need to build and return a full list if only the first page is ever displayed. In such a setup, a number-of-list-entries field and a list-offset field is added to the message header. The list-offset field contains a value that uniquely identifies a list entry, such as a customer number or product ID. In some cases, this could also be a composite value to ensure uniqueness.

The request message uses the first field to indicate how many list entries are needed to build one page, and it uses the second field to define where the list should start. If the second field is empty, the list begins with the first entry. The response message uses the number-of-list-entries field to indicate how many entries are returned, and it uses the list-offset field to indicate whether there are more entries to request. If this field is empty, the list is complete; otherwise, the value is to be used in the subsequent request message to retrieve the next page, thereby ensuring that this page begins at the correct list position.

Whereas these methods of preserving information between client and server job are stateless by nature, other methods involve establishing this persistence by means of a stateful message dialog, typically by assigning and exchanging a session ID or session handle. The session ID is then used to identify the storage location needed to preserve the persistent information. This could be a record in a file, a data area, a user space, a user index, or something similar.

If such a design is chosen, it is important to include a mechanism in the protocol to signal when the use of the session ID has completed, so that the related allocated storage can be released — and the session ID safely reused.

Error and Event Communication: How should error and informational messages be communicated within the data protocol, to ensure a correct and helpful dialog with the application user? Defining unique error codes and event codes and related messages helps the receiving part of the application correctly identify and properly communicate an error or event. Message files can be of great help in creating and administering error messages and event messages.

Language Support: Do you need to support more than one language within the data protocol? If so, you must include information in the message header to define the language that applies to the message dialog, so that all language-sensitive information can be presented in the correct language.

Exception Tolerance: Foreseeable, logical errors, such as a customer record not found, should be handled by the error and event communication facility that I just described. However, it is very important to include in your application design an exception trap mechanism that catches — and communicates back — unforeseen errors. These errors could be any kind of exception that occurs during program execution and that unhandled would lead to an inquiry message being sent to the QSYSOPR message queue and bring the job to a screeching halt. Over time, unhandled errors could lead to all the server jobs hanging in a message wait, and before that suffering performance as the number of message waiting jobs grows. Registering ILE condition handlers, adding RPG/IV *PSSR subroutines, and coding ditto monitor groups and (e)rror opcode extenders are all vital instruments in establishing the defensive programming style (also) required for this type of application.

Debug Facility and Application Monitoring: How do you provide for daily monitoring of the application and investigation of problems and errors? One way is to include log files to record the transaction dialog as well as the application errors and events. If necessary because of performance or storage considerations, a switch could control the transaction logging and activate it only if debugging the transaction flow or message dialog is necessary.

To sum up all these considerations, let me continue with the practical implementation in today's sample application context. In a real-life business application, the data exchange would include customer, product, pricing, availability, order, accounting, payment, and many other types of information, but to avoid the problem of generating test data, I've simply chosen to use some information already available on your system to demonstrate a data queue-driven application: The TCP/IP server start information in the system file QATOCSTART.

Your part of this demonstration is therefore to envision all the other types of information being distributed and exchanged in real business applications of similar design and architecture. In this case, I've built the Display TCP/IP Servers (DSPTCPSVR) command, which displays a list of all or a subset of the TCP/IP servers available on your system. Using the display option in the list panel, you can further display all the selected server's start attributes from the QATOCSTART file.

The following are the requirements for my data protocol design as far as request and response types are concerned:

- Request a list of TCP/IP servers, optionally subset by server type.
- Return a list of TCP/IP servers, including server file key information.
- Request TCP/IP server information for a specific server.
- Return TCP/IP server information for a specific server.

Using the QATOCSTART file layout, I then continue the design efforts by deciding which file fields to include in the TCP/IP server list. The server name, being the primary key as well, is a good start. The server type and autostart information completes the list. The remaining fields are then included in the full server attribute message.

Here's what the QATOCSTART file layout looks like:

File	:	QATOCSTART	Record format	:	QTOCSTRT
Library . . .	:	QUSRSYS	Record length	:	240
Field name	Field type	Buffer	Length	Key	Column heading
SERVERTYPE	Char	1	1		SVR TYP
SERVER	Char	2	30	1 U	Server
AUTOSTART	Char	32	4		Auto Start
LIBRARY	Char	36	10		Library of Program
PROGRAM	Char	46	10		Program to Call
EXTSTRCMD	Char	56	64		External Start CMD
EXTENDCMD	Char	120	64		External End CMD

The next step is to decide what information my request and response message headers should include and, as part of that consideration, how the server list data exchange should work. Here's what I've arrived at:

```

Data protocol: TCPSVR

Request ID: SVRLST - Inbound

Request header:
Data protocol          6   Char
Transaction ID        16   Char
Request ID             6   Char
Message format         8   Char
Message language       3   Char
Message offset value   30   Char
Number of entries req   4,0 Zoned   Valid range 1-24

Request data:
Server type            1   Char

```

```

Request ID: SVRLST - Outbound

Response header:
Data protocol          6      Char
Transaction ID        16      Char
Request ID            6      Char
Event code            4,0     Zoned
Error code            4,0     Zoned
Event message offset  4,0     Zoned
Error message offset  4,0     Zoned
Response data offset  4,0     Zoned
Message offset value  30      Char
Number of entries rtn  4,0     Zoned
Entry length          4,0     Zoned

Response data:
Server type           1      Char
Server key name       30      Char
Server name           30      Char
Auto start            4      Char
Request ID: SVRATR - Inbound

Request header:
Data protocol          6      Char
Transaction ID        16      Char
Request ID            6      Char
Message format         8      Char
Message language       3      Char

Request data:
Server key name       30      Char

Request ID: SVRATR - Outbound

Response header:
Data protocol          6      Char
Transaction ID        16      Char
Request ID            6      Char
Event code            4,0     Zoned
Error code            4,0     Zoned
Event message offset  4,0     Zoned
Error message offset  4,0     Zoned
Response data offset  4,0     Zoned

Response data:
Server type           1      Char
Server name           30      Char
Auto start            4      Char
Program name          10      Char
Program library        10      Char
Start command         64      Char
End command           64      Char

```

Given the limited number of file records and the limited amount of server information, simply returning all server information in the list would be no problem. The reason for not doing that is of course the intention of demonstrating how to build the two different types of data exchange.

To complete the picture, I have also created the following two examples of log files to capture errors and events as well as the transaction dialog:

Error and event log file example:

File	CBX1691F	Record format	CBX1691R
Library . . .	QGPL	Record length	595
Field name	Field type	Buffer	Length
LGTSTP	Timestamp	1	26
LGTYPE	Char	27	6
LGTRID	Char	33	16
LPGGMN	Char	49	12
LGFUNC	Char	61	12
LGDGCD	Zoned	73	4 0
LGDGMS	Char	77	7
LGDGDT	Char	84	512

Transaction log file example:

File	CBX1692F	Record format	CBX1692R
Library . . .	QGPL	Record length	4157
Field name	Field type	Buffer	Length
LGTSTP	Timestamp	1	26
LGTYPE	Char	27	6
LGTRID	Char	33	16
LGDTPC	Char	49	6
LGRQID	Char	55	6
LGRQTP	Char	61	1
LGDATA	Char	62	4096

Now it's time to have a look at the data queue setup. For this application, there is a separate data queue for inbound and outbound transactions, respectively. Separating the data flows makes great sense for a number of reasons, including:

- Inbound traffic is typically handled first-in-first-out, whereas outbound traffic is keyed to ensure that the server reply is received by the correct request sender. To support this requirement efficiently, the inbound data queue should therefore be created with sequence *FIFO, and the outbound data queue should be created with sequence *KEYED.
- Separating data flow means that you avoid the risk of data queue object lock conflicts between the sending and receiving processes.
- Loss of client or server side processing is easier to detect if the inbound and outbound transactions are not mixed. This setup also makes it easier to monitor the inbound workload to ensure that sufficient server jobs are available to process the incoming requests.

Here are the commands to create the two data queues needed to establish the program-to-program communication in the preceding setup:

```

CRTDTAQ DTAQ(CBX169I)
  MAXLEN(8192)
  FORCE(*NO)
  SEQ(*FIFO)
  SENDERID(*NO)
  SIZE(*MAX2GB 16)
  AUTORCL(*YES)
  TEXT('Sample application inbound data queue')

CRTDTAQ DTAQ(CBX169O)
  MAXLEN(8192)
  FORCE(*NO)

```

```

SEQ(*KEYED)
KEYLEN(16)
SENDERID(*NO)
SIZE(*MAX2GB 16)
AUTORCL(*YES)
TEXT('Sample application outbound data queue')

```

Both commands are part of the CL program to build the sample data queue application included with this article. A list of all sources involved in this APIs by Example, as well as instructions for how to create all sample application objects, is at the end of this article. After successful creation of all objects, follow these steps to perform a test run:

1. Using the Change Current Library (CHGCURLIB) command, change your job's current library to the one containing the application.
2. Using the Run Data Queue Server (RUNDTAQSVR) command, start the server process: RUNDTAQSVR LOGTRN(*YES). To avoid locking up your interactive session, submit the command to batch. Check that the submitted job has gone active before proceeding.
3. To generate some work for the server job, run the DSPTCPSVR command. The DSPTCPSVR command requests the list of TCP/IP servers, one page at a time, from the server job through the inbound data queue CBX169I and receives its reply from the outbound data queue CBX169O. Every time you page down, the next block of TCP/IP servers is requested. The list request is sent from the CBX1692L UIM List Exit Program. The list request is processed by the CBX1691 and CBX16911 data queue server programs.
4. The server job logs all transactions to the transaction log file CBX1692F. Using the Display Physical File (DSPPFM) command, you can monitor the request/response dialog in the transaction log file CBX1692F: DSPPFM FILE(CBX1692F) FROMRCD(*END).
5. From the DSPTCPSVR panel, you can select *option 5=Display server start information*. Doing so generates one request/response per selected TCP/IP server. This can of course also be monitored in the transaction log file. The server information request is sent from the CBX1692E UIM General Exit Program. The server information request is processed by the CBX1691 and CBX16912 data queue server programs.
6. To complete the test run, end the server job submitted in step 2. If you ran the data queue server job interactively, use the SysRqs (System Request) key in that job and specify option 2, which runs the End Request (ENDRQS) command.

To get a closer look at the transaction dialog and how the various parts of the sample application play together, you could also step through the programs as they run in a source debug session.

This concludes my article series about Data Queue APIs and CL Commands. I hope these articles have brought you some useful tools to work with data queues — and also the inspiration to do so.

This APIs by Example includes the following sources:

```

CBX169  -- RPGLE  -- Data Queue Sample Application - service functions
CBX169B -- SRVSRC -- Data Queue Sample Application - binder source

CBX1691 -- RPGLE  -- Run Data Queue Server - TCPSVR Protocol
CBX16911 -- RPGLE -- Run Data Queue Server - SVRLST Request
CBX16912 -- RPGLE -- Run Data Queue Server - SVRATR Request
CBX1691H -- PNLGRP -- Run Data Queue Server - Help
CBX1691X -- CMD    -- Run Data Queue Server

CBX1692 -- RPGLE  -- Display TCP/IP Servers - CPP
CBX1692E -- RPGLE -- Display TCP/IP Servers - UIM General Exit Program
CBX1692H -- PNLGRP -- Display TCP/IP Servers - Help

```

```

CBX1692L -- RPGLE -- Display TCP/IP Servers - UIM List Exit Program
CBX1692P -- PNLGRP -- Display TCP/IP Servers - Panel Group
CBX1692X -- CMD -- Display TCP/IP Servers

CBX1691F -- PF -- Error/event log file
CBX1692F -- PF -- Transaction log file
CBX169M -- CLP -- Data Queue Sample Application - build application

```

To create all these objects, copy all sources to their respective source files in your library, then compile and run CBX169M. Compilation instructions are in the source headers, as usual.

Articles and IBM documentation — Cross-Platform Data Queue Support:

Exploring the Client Access Data Queue APIs:

<http://www.systeminetwork.com/article.cfm?id=1509>

Programming with ODBC and Data Queues:

<http://www.systeminetwork.com/article.cfm?id=6584>

AS/400 Toolbox: Using Dataqueue, Record, and RecordFormat Classes:

<http://www.systeminetwork.com/article.cfm?id=7353>

Data Queues: A PC-to-iSeries Quick Link:

<http://www.systeminetwork.com/article.cfm?id=15842>

Use .NET to Develop iSeries Data Queue Applications:

<http://www.systeminetwork.com/article.cfm?id=20273>

Queue Up to Work with Data Queues in .NET Programs:

<http://www.systeminetwork.com/article.cfm?id=53385>

Using the Client Access for Microsoft Windows 95 and Windows NT OCX Control and OLE Automation Objects with Visual Basic:

http://www-912.ibm.com/s_dir/slkbase.NSF/515a7ef1f8deef8c8625680b00020380/def5574d27de318e862565c2007cbc3c?OpenDocument

CWBDQ: Q&A for the Optimized Data Queue API:

http://www-912.ibm.com/s_dir/slkbase.NSF/515a7ef1f8deef8c8625680b00020380/2202a0b08aebc41a862565c2007cb060?OpenDocument

IBM Toolbox for Java:

<http://www-03.ibm.com/servers/eserver/series/toolbox/overview.html>

Data Queue Host Server Does Not Support DDM Data Queues:

http://www-912.ibm.com/s_dir/slkbase.NSF/f5ed8d76fdf9afb88625680b00020384/8bc86c06f162066b86256d91006a6cac?OpenDocument

The previous installments of this article series:

Data Queue APIs and CL Commands, Part 1

<http://www.systeminetwork.com/article.cfm?id=53542>

Data Queue APIs and CL Commands, Part 2

<http://www.systeminetwork.com/article.cfm?id=53685>

Data Queue APIs and CL Commands, Part 3

<http://www.systeminetwork.com/article.cfm?id=53850>

Data Queue APIs and CL Commands, Part 4

<http://www.systeminetwork.com/article.cfm?id=54001>

This article series demonstrates the following data queue APIs:

Retrieve Data Queue Description (QMHQRDQD) API:

<http://publib.boulder.ibm.com/infocenter/iserics/v5r3/topic/apis/qmhqrdqd.htm>

Send Data Queue (QSNDDTAQ) API:

<http://publib.boulder.ibm.com/infocenter/iserics/v5r3/topic/apis/qsnddtaq.htm>

Receive Data Queue (QRCVDTAQ) API:

<http://publib.boulder.ibm.com/infocenter/iserics/v5r3/topic/apis/qrcvdtaq.htm>

Clear Data Queue (QCLRDTAQ) API:

<http://publib.boulder.ibm.com/infocenter/iserics/v5r3/topic/apis/qclrdtaq.htm>

Retrieve Data Queue Message (QMHRDQM) API:

<http://publib.boulder.ibm.com/infocenter/iserics/v5r3/topic/apis/qmhrdqm.htm>

All data queue APIs are documented here:

<http://publib.boulder.ibm.com/infocenter/iserics/v5r3/topic/apis/obj2.htm>

You can retrieve the source code for this API example from the following link:

http://www.pentontech.com/IBMContent/Documents/article/54098_170_DataQueue5.zip

Source URL: <http://iprodeveloper.com/rpg-programming/apis-example-data-queue-apis-and-cl-commands-part-5>