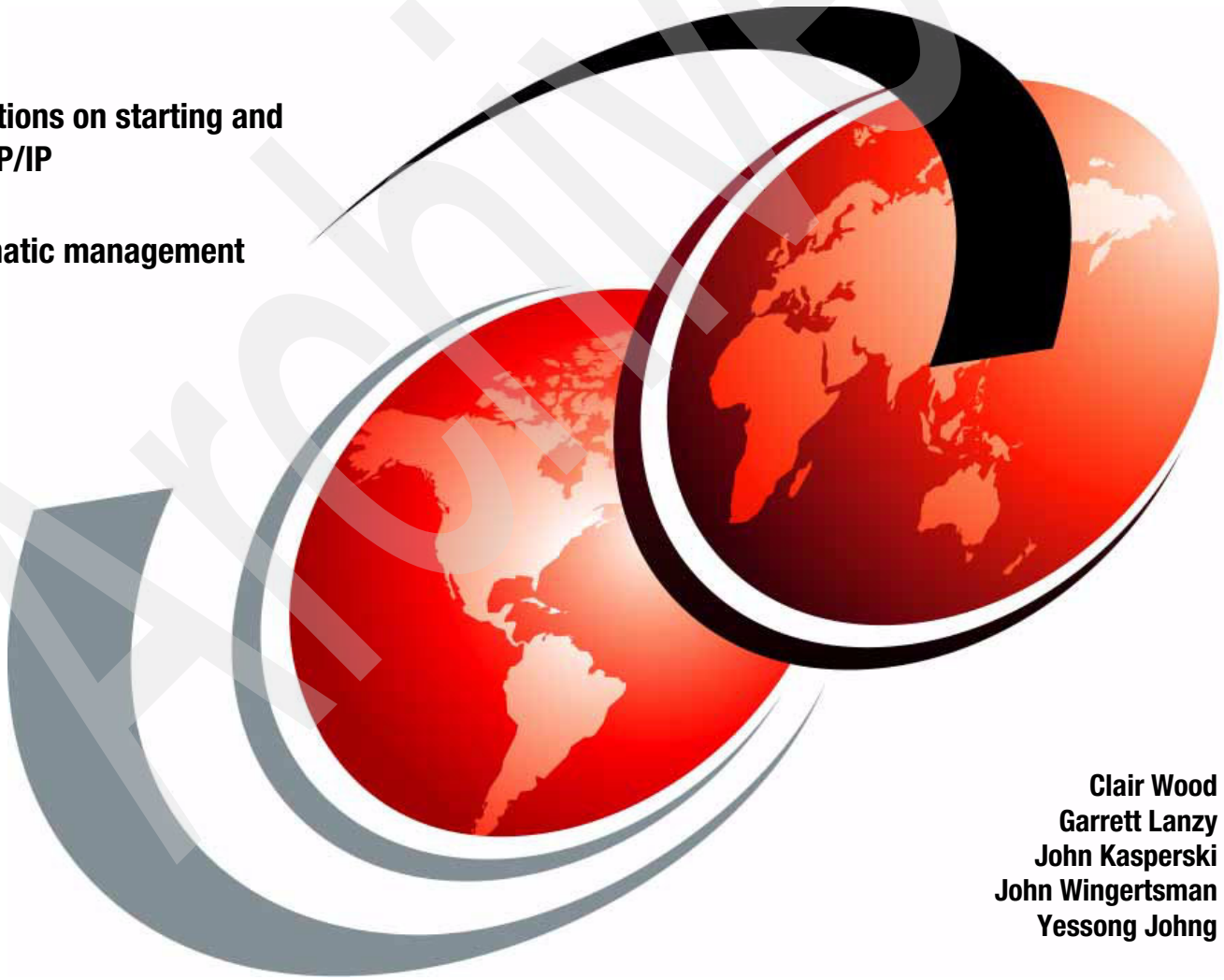
IBM

Tips and Techniques for Using TCP/IP on i5/OS

- Optimizing performance in a TCP/IP network
- Considerations on starting and ending TCP/IP
- Programmatic management of TCP/IP



Clair Wood
Garrett Lanzy
John Kasperski
John Wingertsman
Yessong Johng

Redpaper

ibm.com/redbooks

International Technical Support Organization

Tips and Techniques for Using TCP/IP on i5/OS

May 2006

Note: Before using this information and the product it supports, read the information in “Notices” on page v.

First Edition (May 2006)

This edition applies to V5R4 of IBM i5/OS (5722-SS1).

Contents

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:
This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

Redbooks (logo) ™
iSeries™
i5/OS®
Advanced 36®
AIX®
AS/400®
Domino®
Integrated Language Environment®
IBM®
Language Environment®
OS/400®
Redbooks™
System i5™

The following terms are trademarks of other companies:

Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Preface

This IBM® Redpaper provides various tips and techniques for managing TCP/IP with IBM i5/OS® in the following chapters:

- Chapter 1, “Optimizing performance in a TCP/IP network” on page 1

  Various recommendations that might improve your network configuration and enable you to obtain optimal performance.

- Chapter 2, “Considerations for starting and ending TCP/IP” on page 11

  Information about the different ways that TCP/IP can be started. It also discusses potential problems that might occur if starting TCP/IP is not carefully planned.

- Chapter 3, “Checking TCP/IP status programmatically” on page 19

  Discussion about why it is important for a TCP/IP application to check the status of TCP/IP in a proper way. It also provides an example of a program that checks TCP/IP status.

- Chapter 4, “Using alias names and setting proxy ARP and preferred interface lists programmatically” on page 23

  Procedure for assigning a name to a TCP/IP interface that can be used in place of an IP address in programs. It also provides information about a new feature called the preferred interface list to assist with handling adapter failures.

- Chapter 5, “Using exit programs” on page 29

  Notes about what you can do with the exit points that are defined for IBM-supplied TCP/IP applications.

The team that wrote this Redpaper

This Redpaper was produced by a team of specialists from around the world working at the International Technical Support Organization, Rochester Center.

Clair Wood graduated with a Bachelor of Science degree in Computer Science from Brigham Young University in 1988. After graduation, he started working at IBM as a Software Engineer on the AS/400® Device Configuration team. Other past assignments at IBM have included working on the Source/Sink and Work Management teams. His current job is team leader of the i5/OS TCP/IP Configuration Team.

Garrett Lanzy is a Senior Software Engineer leading a team that develops i5/OS TCP/IP applications. He has worked on OS/400® and i5/OS TCP/IP since 1992. He has B.S. degrees in Electrical Engineering and Computer Science from Michigan Technological University and is a member of the Association for Computing Machinery.

John Kasperski is an Advisory Software Engineer at IBM Rochester. Starting with IBM in 1991, he designed and developed projects in the Networking Software and Networking Hardware Divisions. In 1996 he joined the OS/400 Sockets/SSL development team, designing and implementing socket enhancements for the V4R2 through V5R3 releases. He became the team leader of the Sockets/SSL development team in 2001. He is team leader of the i5/OS TCP/ IP stack development team and project lead for IPv6 on i5/OS.

John Wingertsman started working for IBM in 1989. As Development Software Engineer he has been developing software in the following areas: SLIC LIOM, POSIX communications, SLIC Advanced 36® data communications, and OS/400 HTTP Server High Availability. He currently works in i5/OS TCP/IP Configuration. He has BS/MS Computer Science from New Jersey Institute of Technology where he was a member Tau Beta Pi.

Yessong Johng is an IBM Certified IT Specialist at the IBM International Technical Support Organization, Rochester Center. He started his IT career at IBM as an S/38 Systems Engineer in 1982 and has been with S/38, AS/400, iSeries™, and now IBM System i5™ for 20 years. He writes extensively and develops and teaches IBM classes worldwide on the areas of IT Optimization whose topics include Linux®, AIX®, and Windows® implementations on iSeries. His other coverage areas include TCP/IP and networking security.

Thanks to the following people for their contributions to this project:

John Mcginn
IBM Rochester

Francis Pflug
IBM Raleigh

Become a published author

Join us for a two- to six-week residency program! Help write an IBM Redbook dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You'll team with IBM technical professionals, Business Partners and/or customers.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you'll develop a network of contacts in IBM development labs, and increase your productivity and marketability.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our papers to be as helpful as possible. Send us your comments about this Redpaper or other Redbooks™ in one of the following ways:

- Use the online **Contact us** review redbook form found at:

  ibm.com/redbooks

- Send your comments in an e-mail to:

  redbook@us.ibm.com

- Mail your comments to:

  IBM Corporation, International Technical Support Organization
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400

1

Optimizing performance in a TCP/IP network

Although your TCP/IP network may be up and working fine, your network connectivity may not be optimized to deliver the best possible performance. If each of the client workstations can access all of the needed applications on the server, why would you risk altering the network configuration? For example, if it takes more than six hours to replicate a given database over the network, what if this replication time could be cut in half?

There is no guarantee that any of the items listed in this chapter could result in that significant of a performance boost, but using a combination of the items recommended here will help to improve your network configuration and enable you to obtain optimal performance.

1.1 Network / Line Description settings

When considering “optimizing throughput between two systems,” what is often overlooked is the word “between.” Many IT folks concentrate solely on the two endpoint systems and do not focus on the infrastructure between these two systems, which could very well be the cause of their throughput headaches.

The first thing to consider is the network bandwidth available for your application. You must consider all “hops” in the communications path to the remote system. At each network segment there might be different bandwidth levels available to this traffic. For instance, it probably does you no good to have 1 Gb adapters at each endpoint system when your high-throughput traffic flows over a 56 Kb link.

Also consider the networking hardware and software used between the two endpoint systems. It is possible that a firewall or other packet filtering device is affecting throughput, although most likely these devices would affect connectivity. The use of proxy servers could also introduce delays. A DNS server that is down can also cause long initial connectivity delays, appearing as throughput delays, in many cases.

1.1.1 Line Description configuration

There are two values in the configuration of an i5/OS line description that should be checked to make sure that they match the hub/switch in your network:

- Line Speed: This setting specifies the speed of the adapter in bits/second. The line speed choices vary depending on the type of adapter being used. Some adapters support multiple line speeds. Obviously the higher line speeds will result in higher performance, but the i5/OS line description line speed has to match the hub/switch to which the adapter is connected. For the fewest problems and optimal performance, do not use *AUTO if the port on the switch/hub is configured for a fixed line speed.
- Duplex: This setting specifies whether the adapter can send and receive data simultaneously. In half-duplex mode, the hardware must alternate between sending data and receiving data. In full duplex mode, one wire in the cable is dedicated to send data, and another wire in the cable is dedicated to receive data, therefore allowing data to be sent and received simultaneously. For optimal performance full duplex should be used, but this setting must match what was configured on the hub/switch.

The top communications throughput problem from an IBM service perspective in terms of frequency and severity is an incorrectly set line speed or duplex parameter (or both) in the line description. These two values should match the settings on the port of the hub/switch this network adapter plugs in to**.**

1.1.2 Maximum Frame Size and Maximum Transmission Unit (MTU)

Another value in the i5/OS line description configuration that has a large impact on performance is the Maximum Frame Size. This setting specifies the largest frame size that can be transmitted and received on this line description. The possible values for this setting are dependent on the type of adapter being used and which protocols are being used on that adapter.

The maximum frame size is typically set to the largest size that is supported by that particular adapter. However, the frame size specified for a given line description/adapter may not be consistent with the frame sizes being used by the rest of the network. To solve this issue, the i5/OS line description’s Maximum Frame Size can be overridden by specifying a lower value in the Maximum Transmission Unit (MTU) field of the TCP/IP Interface configuration. After this

is done, any traffic that uses this particular TCP/IP interface will then use this value for its MTU, regardless of what the line description was configured for. The default value for this parameter, *LIND, indicates that the MTU value should be based on the frame size that was specified in the line description.

In addition to allowing the line description's frame size value to be overridden, the TCP/IP Interface's MTU value can also be overridden. This is done by specifying a lower MTU value in the TCP/IP Route configuration. If this is done, any traffic that uses this route will use the route specified value for its MTU, regardless of what the MTU is in the TCP/IP interface or the frame size is in the line description. The default value for the route MTU parameter, *IFC, indicates that the MTU value specified in the TCP/IP Interface (that is bound to this route) should be used.

So in summary, maximum frame size on the line description can be overridden by the Maximum Transmission Unit (MTU) setting in the TCP/IP interface defined on that line description, which in turn can be overridden by the MTU setting in the TCP/IP routes that are defined. Figure 1-1 shows this relationship.

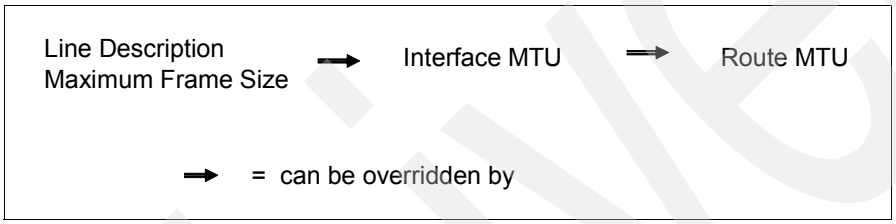


Figure 1-1 Overriding MTU values

Figure 1-2 illustrates these concepts.

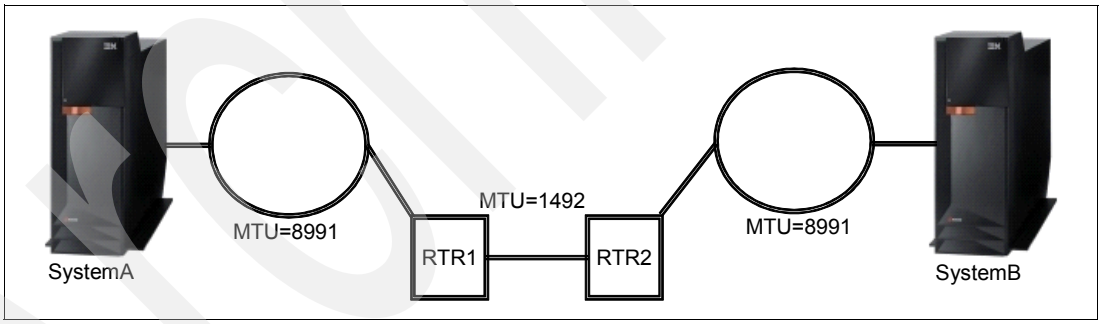


Figure 1-2 Overriding MTU values illustration

In this example, both System A and B have 1 Gb Ethernet adapters and are using Jumbo Frames. Unfortunately, there is a link between the two systems that has an MTU setting of 1492. In order to allow System A and B to communicate efficiently, a route should be defined to override the line description's maximum frame size. This route should force traffic that is sent between the two systems to use an MTU size of 1492. A route would have to be defined on each system because both have 1 Gb Ethernet adapters.

1.2 TCP/IP send and receive buffers

The TCP send and receive buffers are the buffers that reside between the application and the TCP layer in the TCP/IP stack, which is illustrated in Figure 1-3 on page 4.

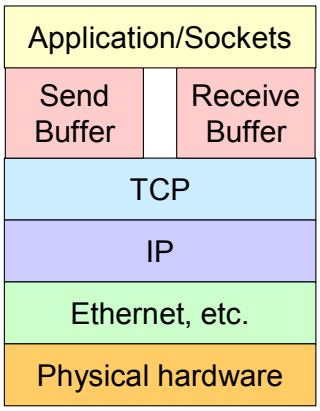


Figure 1-3 TCP/IP stack

Both of these buffers default to 8K (8192) bytes in size. This 8K buffer space for the sends and receives is taken dynamically as needed for an individual connection. If this amount is not needed for a given connection, then it is not allocated. These 8K buffer sizes are loosely enforced. i5/OS will buffer slightly more than these configured amounts during data transfers. The TCP receive window size for the connection is based on the receive buffer value.

The TCP receive buffer is the amount of buffer space that is set aside to hold incoming application data that has not yet been received by the local application. When the receive buffer space starts to fill up, TCP will "advertise" a smaller and smaller window size on the TCP/IP connection. The TCP receive buffer can be viewed in a communications trace by viewing the `Window:` portion of the trace. This value "advertises" the maximum amount of data, in bytes, that the system has room for in its receive buffer. After the TCP receive buffer has been completely filled, the remote system will be prevented from sending any additional data until the local application receives some of the data out of this buffer.

If the remote system's TCP receive buffer has been completely filled, the TCP Window size will be advertised as 0 and the sending system will then start buffering outgoing data in the TCP send buffer space. If the TCP send buffer is completely filled, the application will be prevented from sending any additional data.

The default sizes of the TCP/IP send and receive buffers are set through the CHGTCPA command as illustrated in Figure 1-4 on page 5.

```
                    Change TCP/IP Attributes (CHGTCPA)

Type choices, press Enter.

TCP keep alive . . . . . .    320          1-40320, *SAME, *DFT
TCP urgent pointer . . . . .  *BSD          *SAME, *BSD, *RFC
TCP receive buffer size  . .  8192         512-8388608, *SAME, *DFT
TCP send buffer size . . . .  8192         512-8388608, *SAME, *DFT
TCP R1 retransmission count   3            1-15, *SAME, *DFT
TCP R2 retransmission count   16           2-16, *SAME, *DFT
TCP minimum retransmit time   250          100-1000, *SAME, *DFT
TCP time-wait timeout . . . .  120         0-14400, *SAME, *DFT
TCP close connection message  *THRESHOLD   *SAME, *THRESHOLD, *ALL...
UDP checksum . . . . . .      *YES         *SAME, *YES, *NO
Path MTU discovery:
  Enablement . . . . . .      *YES         *SAME, *DFT, *NO, *YES
  Interval . . . . . .        10           5-40320, *ONCE
IP datagram forwarding . . .  *YES         *SAME, *YES, *NO
IP source routing  . . . . .  *YES         *SAME, *YES, *NO
IP reassembly time-out . . .  10           5-120, *SAME, *DFT
                                                                More...
F3=Exit   F4=Prompt   F5=Refresh   F10=Additional parameters   F12=Cancel
F13=How to use this display        F24=More keys
```

Figure 1-4 Using CHGTCPA command to set buffers

The settings in CHGTCPA are system-wide, meaning that all TCP/IP connections across the system will use these send and receive buffer size settings by default. This default 8K setting in CHGTCPA was fine for Telnet and simple interactive applications. However, 8K buffer sizes are often much too small for applications that send and receive large amounts of data.

No application that sends and receives large amounts of data should rely on the system-wide CHGTCPA send and receive settings, but rather update the buffer sizes itself on a per-connection basis. The SO_SNDBUF and SO_RCVBUF socket options enable an application to set these send and receive buffers for a given connection. All applications that are performance sensitive should be using the SO_SNDBUF and SO_RCVBUF socket options to update the buffer sizes to more appropriate values.

If 8K is too small for most applications, what should the system-wide settings be for CHGTCPA and what sizes should be used for SO_SNDBUF and SO_RCVBUF? The answer depends on several factors such as:

- How much bandwidth is available on this interface/network?
- Is the bandwidth dedicated to this system/application?
- What is the quality of the network? Packet loss? Latency?
- How much data traffic is being generated by the system/application?

If the network path has dedicated bandwidth and allows for little to no packet loss, then it is okay to have your send/receive (depending on the role your i5/OS is playing) buffers increased from the default value of 8K. However, large buffer sizes used over a poor network could worsen already poor throughput.

The only real way to determine the optimal buffer sizes for both CHGTCPA as well as the socket options is through trial-and-error. The settings that are ideal for some applications may not be appropriate for others. Some possible starting values to consider using if you are experiencing performance/throughput concerns are 64K for the CHGTCPA (system-wide) settings and 1 MB for the SO_SNDBUF and SO_RCVBUF settings for those applications that deal with bulk data throughput.

Example 1-1 shows an example of how to set the SO_SNDBUF and SO_RCVBUF socket options.

Example 1-1 How to set the SO_SNDBUF and SO_RCVBUF socket options

```
int sd, rc, bufsize;
bufsize = 64 * 1024;

rc = setsockopt(sd, SOL_SOCKET, SO_SNDBUF,
                (char *)&bufsize, sizeof(bufsize));
if (rc == -1)
{
   perror("setsockopt failed");
   exit(-1);
}

rc = setsockopt(sd, SOL_SOCKET, SO_RCVBUF,
                (char *)&bufsize, sizeof(bufsize));
if (rc == -1)
{
   perror("setsockopt failed");
   exit(-1);
}
```

1.3 Sockets programming tips and techniques

Although you may have optimally tuned your network configuration with 1 Gb Ethernet adapters/switches, Jumbo frames, and large send/receive buffers; a poorly written TCP/IP application will still perform poorly. In this section we discuss simple programming tips and techniques for optimal performance from your socket applications.

Additional information about each of the socket concepts covered in this section can be found in the V5R4 iSeries Information Center at:

http://publib.boulder.ibm.com/infocenter/iseries/v5r4/index.jsp

1.3.1 IFS vs. Sockets APIs

There are two different sets of APIs that can be used to transmit data over a network:

- read() and write() IFS APIs
- send() and recv() socket APIs

The IFS read() and write() are generic APIs that can be used to read/write data to files as well as sockets. These APIs incur additional overhead in order to allow them to work against both files and sockets.

The send() and recv() APIs, on the other hand, can only be used with sockets. These APIs have been optimized to bypass much of the generic descriptor management logic that is required when using read() and write(). This optimization was implemented in V4R5 and is unique to i5/OS.

Both the IFS and the socket APIs can be used to transfer data over the network, but the socket APIs require less CPU cost and fewer instructions to send and receive the same amount of data on both the sending and the receiving systems every time that an API call is made.

1.3.2 Nagle algorithm and TCP_NODELAY

By default, TCP/IP may introduce a small delay in the sending of data due to logic in the TCP/IP stack referred to as the Nagle algorithm (RFC 896). This algorithm works by buffering several small outgoing messages and sending them all at once. Specifically, as long as there is a sent packet for which the sender has received no acknowledgement, the sender should keep buffering its output until it has a full packet's worth of output. Part of the original design for Nagle's algorithm was to efficiently handle telnet connections. The authors did not want every keystroke on a telnet session to be sent over the network in a separate TCP/IP packet.

Most applications do not transfer data one byte at a time, but are request/response-based and do not want any delays in the processing of their requests. All TCP/IP sockets-based applications that send data (in chunks larger than 1 byte at a time) should enable the TCP_NODELAY socket option on each TCP/IP connection to prevent this send-side delay from occurring.

This Nagle algorithm also interacts badly with TCP/IP delayed acknowledgements. If both TCP/IP delayed acknowledgements and Nagle are being used, large performance delays can occur.

The TCP_NODELAY option is enabled on a given TCP/IP connection using the setsockopt() socket API as illustrated in Example 1-2.

Example 1-2 Using setsockopt() socket API

```
int sd, rc, flag = 1;

rc = setsockopt(sd, IPPROTO_TCP, TCP_NODEALY,
                (char *)&flag, sizeof(flag));
if (rc == -1)
{
   perror("setsockopt failed");
   exit(-1);
}
```

1.3.3 Sending multiple data buffers efficiently

In many cases an application may have multiple data buffers that it needs to transmit over to the remote system. For example, it may have a fixed header followed by one or more variable-length buffers. In order to send multiple blocks of data, the sockets send() API could be invoked multiple times, once for each buffer that has to be sent. Alternatively, a more efficient method to send multiple data buffers is with the sendmsg() API, which allows up to 16 data buffers to be sent at one time.

Using the sendmsg() call to send multiple buffers instead of calling the send() API multiple times will result in less processing on both the sending and receiving systems. On the sending system there will be fewer API calls being made, fewer calls down to the TCP/IP stack, fewer calls down to the adapter, and also most likely fewer TCP/IP packets sent out over the network because multiple application data buffers will be combined. On the receiving system, if there are fewer TCP/IP packets arriving (due to the remote application's send data buffers being combined), then less processing will be required to receive them. Hence the sendmsg() API will allow processing to be done more efficiently on both the sending and the receiving systems.

Like the sendmsg() API on the sending system, the recvmsg() API can be used on the receiving system to receive incoming data into multiple different data buffers. The benefit provided by recvmsg() over multiple recv() calls is much less than the combining of send() calls on the sending side.

1.3.4 Receiving data with MSG_WAITALL and SO_RCVLOWAT

When transmitting data over a TCP/IP connection, typically the receiving application can make no guarantee how much data will be returned on a given receive operation. The amount of data returned by a given recv() call will vary based on a number of conditions such as:

- Size of the buffer specified on the recv() call
- Size of the buffers being sent by the remote application
- Timing issues between when recv() is called versus when the data was transmitted
- Routers and switches on the network
- MTU size of each of the route segments between the two systems
- Amount of data already queued up to be received on the receiving system
- Amount of data this is currently flowing over the network
- Amount of data queued up to be sent on the sending system
- The values of various socket options and flags

Because the amount of incoming data returned by recv() varies (it could be as little as 1 byte), it is common for an application to call recv() over and over again until all of the incoming data has been received. Calling the recv() API repetitively in such a loop is inefficient. There are two alternative methods that an application can use to control how much data is returned on a given receive operation: the MSG_WAITALL flag and the SO_RCVLOWAT socket option.

MSG_WAITALL is a message flag that specifies that the data buffer specified on this given recv() call should be completely filled by i5/OS before control is returned to the application. If there is not enough data queued up in the TCP receive buffer to completely fill the specified data buffer, i5/OS will block waiting for additional incoming data to arrive. Support for this message flag was added in V5R2.

MSG_WAITALL is useful if the application is using fixed buffer lengths or if the application has a fixed header on the front of the data that is being transmitted. In the case of a fixed header, the application can guarantee that the entire fixed header is received at one time by using MSG_WAITALL on the recv() call. After the header has been received, all of the rest of the incoming data can then be received using a second recv() call with another MSG_WAITALL flag specified.

For example, assume a recv() with MSG_WAITALL is specified and the data buffer specified on that recv() call was 64 bytes. If remote system sends only 30 bytes, the recv() call will block waiting for another 34 bytes to arrive before it completes.

The SO_RCVLOWAT socket option is similar to the MSG_WAITALL flag except that this socket option specifies a minimum amount of data that has to be returned on all recv() calls that are done on this TCP/IP connection. Support for this socket option was added in V4R3.

If more than the minimum is sent while blocked on a recv() call, i5/OS will return the entire amount received when the minimum requirement is met. For example, if SO_RCVLOWAT was set to a value of 64 bytes and a recv() was done that specified a 512-byte buffer, when 30 bytes arrive from the remote system, the recv() call would remain blocked waiting for at least another 34 bytes to arrive. If the sending system then sent over a 200-byte block of data, then recv() would complete with 230 bytes (as long as the 200 bytes did not get fragmented by the switches and routers on the network).

1.3.5 Waiting for incoming data – SO_RCVTIMEO

Another useful socket option that is not well known is the SO_RCVTIMEO. This option allows an application to specify how long each recv() call should block waiting for incoming data to arrive from the remote system. When this option is set, all subsequent recv() calls will block only the length of time that was specified in this option. In order to use this socket option, the _XOPEN_SOURCE constant must be defined at the top of your program to a value of 520 or greater before any header file is included, as illustrated in Example 1-3.

Example 1-3 Defining _XOPEN_SOURCE constant

```
#define _XOPEN_SOURCE 520

#include <sys/socket.h>
#include <netinet/in.h>
```

The SO_RCVTIMEO socket option can be combined with the MSG_WAITALL flag or the SO_RCVLOWAT option that was mentioned earlier.

1.3.6 Inheritance of socket options from listening socket

Throughout this section we have discussed several useful socket options such as:

- SO_RCVBUF
- SO_SNDBUF
- TCP_NODELAY
- SO_RCVLOWAT
- SO_RCVTIMEO

Calling the setsockopt() API to set each of these options on a given connection takes a measurable amount of overhead and processing time. This processing time can be significant for short-lived request/response-type connections. This overhead processing time can be eliminated for server applications by setting each of these options on the listening socket before the incoming connection is accepted. Although most of these options do not make much sense on a listening socket, the settings will be saved. All socket options, including the non-blocking option, will be inherited by all new incoming connections that arrive on this listening socket.

Using inheritance of socket options from the listening socket may not seem that useful, but it can have a significant performance impact if the server application handles hundreds of connections per second.

1.3.7 Asynchronous I/O APIs on i5/OS

When a socket application has to handle multiple connections simultaneously, several design decisions should be considered. In some cases, multiple threads or jobs may be considered. The use of the select() API may also be used to allow a single thread to monitor activity on a number of sockets. If you have a socket application that has to handle multiple connections simultaneously, is multithreaded, or uses the select() API, then you should consider modifying the application to use the i5/OS Asynchronous I/O socket APIs. These APIs are:

- Asynchronous Socket APIs:
  - QsoStartAccept()
  - QsoStartRecv()
  - QsoStartSend()

- Asynchronous Secure Socket APIs:
  - gsk_secure_soc_startRecv()
  - gsk_secure_soc_startSend()
  - gsk_secure_soc_startInit()
- I/O Completion Port APIs:
  - QsoCreateIOCompletionPort()
  - QsoDestroyIOCompletionPort()
  - QsoWaitForIOCompletion()
  - QsoPortIOCompletion()

The use of these APIs allows more efficient use of system resources, improved scalability, and improved throughput and performance. Many existing i5/OS servers have already switched over to use these APIs and they have observed large increases in performance as a result. Those servers that make use of these i5/OS Asynchronous socket APIs include: HTTP Server (Powered by Apache), Domino®, FTP, and NetServer.

In order to obtain the highest possible performance for your server application, it is recommended that the Asynchronous I/O APIs listed above be used.

2

Considerations for starting and ending TCP/IP

This chapter discusses the different ways that TCP/IP may be started. It also discusses potential problems that may occur if starting TCP/IP is not carefully planned.

The discussion includes the following topics:

- Starting TCP/IP: IPL attributes versus start-up program
- Starting TCP/IP on systems with a 3494 Tape Library
- Restricted state
- Ending TCP/IP
- Starting and ending TCP/IP references

2.1 Introduction

TCP/IP may be started in a number of ways. It is important for a system administrator to understand the methods for starting TCP/IP in order to use the best possible way for a specific system. Race conditions, unnecessary start-up requests by multiple jobs, or possible start-up errors may be reduced or eliminated by understanding the concepts that are presented in this chapter.

2.2 Starting TCP/IP: IPL attributes versus start-up program

There is an IPL attribute (DSPIPLA/CHGIPLA parameter STRTCP) that can be set to control whether TCP/IP will be started as part of IPL processing or when the iSeries is brought out of restricted state. If the IPL attribute STRTCP is set to *YES then TCP/IP will be started with whatever command defaults are set on the system for an IPL or coming out of restricted state. If the IPL attribute STRTCP is set to *NO, then TCP/IP will not be started for the previously mentioned cases.

TCP/IP can be started in the following ways:

- As part of IPL processing. Set IPL attribute STRTCP(*YES). TCP/IP will be started as part of IPL processing using the STRTCP command defaults.
- As part of a “start-up” program. Set IPL attribute STRTCP(*NO). TCP/IP will not be started as part of normal IPL processing. In this case you must have a STRTCP command coded in your start-up program to have TCP/IP started.
- As part of vary on of a 3494 Tape Media Library device. Refer to 2.3, “Starting TCP/IP on systems with a 3494 Tape Library” on page 13 for information about this subject.
- As part of coming out of normal iSeries processing when coming out of restricted state. Set IPL attribute STRTCP(*YES). TCP/IP will be started as part of normal iSeries processing coming out of restricted state using the STRTCP command defaults.
- By issuing the STRTCP CL command from a command line.
- By issuing a STRTCP CL command from a program.
- Vary on of an NWSD. The vary on processing will start TCP/IP.

You should ensure that TCP/IP is started in only one place on the system. We recommend using the IPL attribute STRTCP set to *YES to start TCP/IP as part of normal IPL processing.

A few customers have encountered problems with getting all of the interfaces and servers that are configured to be autostarted with using the IPL attribute STRTCP set to *YES. Because of the workload that your system may have, during IPL you may experience delays in the vary on of communication lines that your IP addresses are configured with. In this case we recommend that you move the starting of TCP/IP to your startup program. See 2.7, “Starting and ending TCP/IP references” on page 17 for sources of additional information about this topic.

As part of i5/OS V5R4 start TCP/IP processing, the TCP/IP servers and PPP profiles that are configured to be autostarted will be started as a submit job from the QSYSWRK/QTCPIP job. The submit job to start the TCP/IP servers will be done after the interfaces that are configured to be autostarted have been processed. At this point the QTCPSTSVRS job will start the TCP/IP servers. You should keep in mind that the starting of the TCP/IP interfaces is not the same as an interface being “active.” If a TCP/IP server is dependant on an interface being active, it is normal for the server to have logic to retry for successful operations with that interface for a short period of time until the interface becomes active. There have been cases when a user-defined server is configured to be started as part of start TCP/IP processing but

does not have the logic to retry its operation with an interface and fails to start because the interface has been delayed becoming active. In this case either the server must be modified or its configuration changed to retry during its initialization or it should not be configured to be started when TCP/IP starts in order to avoid these type of failures.

You can view the logging of what happened during the starting of TCP/IP as part of the IPL process in the QSTRTCP job (user profile QPGMR). You can view the results of starting the TCP/IP servers by viewing the QTCPSTSVRS job log (user profile QTCP). After TCP/IP has started, the persistent jobs QSYSWRK/QTCPIP and QSYSWRK/QTCPMONITR will be present until TCP/IP has been ended.

The QSYSWRK/QTCPIP job processes requests to start and end TCP/IP interfaces as part of STRTCPIFC and ENDTCPIFC processing. This job is also responsible for performing the starting of interfaces that are configured to be autostarted as part of start TCP/IP processing. This job's job log contains information about the processing of starting or ending interfaces, interfaces becoming active or inactive, and important information about processing it is doing. The work done by the QSYSWRK/QTCPIP job is beyond the scope of this paper and therefore its operation will not be described here. This job is important for the operation of TCP/IP and in general processes events that are posted to it by the TCP/IP SLIC protocol stack and passes them onto the QSYSWRK/QTCPIP job for logging. Events such as interfaces becoming "active" or "inactive" are handled by the QSYSWRK/QTCPMONITR job.

When TCP/IP is started there is a remote possibility that you can receive a TCP1A1B (`Not able to determine if job NNNNNN/QTCP/QTCPIP started.`) message in response to the start TCP/IP request (STRTCP). The STRTCP command is waiting for a response to be posted back to it from the QSYSWRK/QTCPIP job. As mentioned previously, part of start-up processing for start TCP/IP is to start the QSYSWRK/QTCPIP job. If the system is busy the starting of this job may be delayed and as a result you might see this message posted to the job log. If this occurs, you should check that the QSYSWRK/QTCPIP did start. You can use the WRKACTJOB CL command to determine whether the job started.

2.3 Starting TCP/IP on systems with a 3494 Tape Library

With the release of i5/OS V5R2, support was added to allow attaching a 3494 Tape Library Device using TCP/IP. The 3494 will request that TCP/IP be started and start an IPv4 address (specified by the LCLINTNETA parameter on the CRTDEVMLB, CHGDEVMLB, or CFGDEVMLB command). The request to start TCP/IP is a result of a 3494 Tape Library device operation. This request is valid for both normal operating mode and restricted state.

When a start TCP/IP is requested by the 3494 Tape Library device driver, it is started programmatically as STRTCP STRSVR(*NO) STRIFC(*NO) STRPTPPRF(*NO) STRIP6(*NO). This means that the TCP/IP protocol stack is started, but none of the TCP/IP servers, interfaces, PPP profiles, or IPv6 will be started. However, the IPv4 address configured for the 3494 Tape Library is started.

When TCP/IP is started in the restricted state because of a request by the 3494 Tape Library device driver, TCP/IP is started with STRTCP STRSVR(*NO) STRIFC(*NO) STRPTPPRF(*NO) STRIP6(*NO), and the IPv4 address configured for the 3494 Tape Library device is started. If the iSeries is then brought out of restricted state by starting the subsystems (STRSBS command) and the IPL attribute is set to start TCP/IP (STRTCP *YES) then, assuming that the normal defaults are set for the STRTCP command, the TCP/IP servers, IP interfaces, and PPP profiles that are configured as automatically started will be started.

If TCP/IP has been ended and the iSeries is in normal operating mode and the 3494 Tape Library device operation requests that TCP/IP be started, TCP/IP will also be started with

STRTCP STRSVR(*NO) STRIFC(*NO) STRPTPPRF(*NO) STRIP6(*NO). To start the TCP/IP servers, interfaces, and PPP profiles in this scenario, you have basically two options. One is to end TCP/IP, then issue a STRTCP CL command to get the servers, interfaces, IPv6 and PPP profiles started. The other option is to issue individual CL commands to first start the TCP/IP interfaces (STRTCPIFC INTNETADR(*AUTOSTART)), then start the TCP/IP servers (STRTCPSVR SERVER(*AUTOSTART)) and the host servers (STRHOSTSVR SERVER(*ALL)) and the PPP profiles (STRTCPPTP CFGPRF(*AUTOSTART)). If you need to start the IPv6 portion of the TCP/IP protocol stack, you have to end TCP/IP and restart it as there is not an option to start IPv6 separate from start TCP/IP (STRTCP).

When TCP/IP is started as a result of a STRTCP CL command being issued assuming typical command defaults STRTCP STRSVR(*YES) STRIFC(*YES) STRPTPPRF(*YES) STRIP6(*YES), the TCP/IP server, interfaces (IPv6 and IPv6), and PPP profiles will be started. This would be the case when TCP/IP is started as a result of starting TCP/IP at IPL time by setting the IPL attribute (STRTCP *YES) to start TCP/IP. If TCP/IP is started in restricted state by a user issuing at STRTCP STRSVR(*NO) STRIFC(*NO) STRPTPPRF(*NO) STRIP6(*NO), and then the system is brought out of restricted state by starting the subsystems, the TCP/IP servers, IP interfaces, and PPP profiles will not be started. We recommend that you end TCP/IP before starting the subsystems (STRSBS).

To avoid potential problems with starting TCP/IP, with an iSeries that has a 3494 Tape Library configured to use TCP/IP, the following information should be taken into consideration.

A 3494 Tape Library can be configured to be varied on at IPL time. If the iSeries IPL attributes are configured to have TCP/IP started at IPL time (CHGIPLA STRTCP(*YES)), it is strongly recommended that the 3494 Tape Library be configured to not be varied on at IPL time.

Depending on the mix of the workload that the iSeries has during an IPL, there can be situations in which starting TCP/IP will not be completely successful. If TCP/IP is configured to start at IPL and the 3494 Tape Library is configured to vary on at IPL, then TCP/IP servers that are configured to be started automatically may not be started. The situation arises as a result of a race condition to start TCP/IP as a result of the vary on processing of the 3494 Tape Library occurring before the request to start TCP/IP as specified by the IPL attributes processing to start TCP/IP as part of IPL occurs.

To avoid potential problems associated with starting TCP/IP, it is recommended that you configure your 3494 Tape Library (CRTDEVMLB ONLINE(*NO)) to not be varied on at IPL if TCP/IP is configured to be started as part of the IPL (IPLA STRTCP *YES). The 3494 Tape Library, however, can be varied on using your system start-up program after verifying that TCP/IP is fully active. See Chapter 3, “Checking TCP/IP status programmatically” on page 19 for more information.

2.4 Restricted state

Attention: This section focuses on restricted state operation only.

The design for being able to start and end TCP/IP in restricted state is to support a limited set of capabilities that save and restore operations require. Typically an application running in the Controlling Subsystem requires that one IPv4 interface be active so it can communicate using the communications network.

Refer to 2.3, “Starting TCP/IP on systems with a 3494 Tape Library” on page 13 for details of restricted state operation and the specifics of the 3494 Tape Media Library support. There is a

specific set of operation capabilities that pertain only to the 3494 and not the general topic of TCP/IP operation in restricted state.

When the iSeries is in restricted state, it is possible to start and end TCP/IP and to start and end IPv4 interfaces. There are some iSeries limitations as to what can and cannot be done when operating in restricted state as it relates to subsystems and jobs. See 2.7, “Starting and ending TCP/IP references” on page 17 for sources of additional information.

TCP/IP can only be started with all of the part STRTCP parameters specified as *NO; specifically, STRTCP STRSVR(*NO) STRIFC(*NO) STRPTPPRF(*NO) STRIP6(*NO). Any other parameter specification is not valid. TCP/IP can be ended in restricted state by issuing the ENDTCP CL command.

IPv4 addresses can be started and ended by issuing the STRTCPIFC and ENDTCPIFC CL commands.

While you are operating in restricted state you can use the NETSTAT CL command and the options to check on the status of TCP/IP. In addition, the QTCP and QSYSOPR message queues can be displayed for additional information about such things as interfaces being “active” (TCP8A3E message) when you have requested to start them or “inactive” (TCP8A40 message) when you have requested to end them.

Starting TCP/IP servers, PPP profiles, and the IPv6 portion of the protocol stack are not permitted because the subsystems that would be required are not active.

If you start TCP/IP while in restricted state, it is recommended that you end TCP/IP before you bring the iSeries out of restricted state. You should verify that TCP/IP has ended (not active) before exiting restricted state. The results of this command inform you whether TCP/IP is active. It is also possible to obtain this information programmatically as mentioned previously.

When the iSeries is being brought into restricted state and TCP/IP is running, you should verify that TCP/IP has ended before you try to start it in restricted state. If you try to start TCP/IP before it has ended, you will receive a TCP1A04 message (TCP/IP currently active).

2.5 Ending TCP/IP

Attention: This section discusses ending TCP/IP in *normal* operating mode.

The recommended way to end TCP/IP processing is to issue the ENDTCP CL command., which is the complement of the STRTCP CL command. The processing that is performed at start TCP/IP time is taken into consideration at end TCP/IP time so as to comprehensively end the TCP/IP servers, interfaces, PPP profiles, and the protocol stack. Other mechanisms could be employed to perform end TCP/IP processing in aggregate, but we recommend that you issue the ENDTCP CL command.

You can determine whether TCP/IP has ended by issuing the NETSTAT CL command. When TCP/IP has ended, you will receive a TCP2670 message (`Not able to complete request. TCP/IP services are not available.`). You can also use the Retrieve TCP/IP Attributes (QtocRtcTCPA) API to programmatically retrieve information about the status of TCP/IP. See Chapter 3, “Checking TCP/IP status programmatically” on page 19 for more information. See 2.4, “Restricted state” on page 14 for information about ending TCP/IP in restricted state.

The following list briefly describes the processing order when you end TCP/IP:

1. End TCP/IP servers.

2. End TCP/IP interfaces and release the lock on the device (QTCPIP) for the Line Description.
3. End the QSYSWRK/QTCPIP and QSYSWRK/QTCPMONITR jobs.
4. End the SLIC TCP/IP protocol stack.
5. End TCP/IP processing.

When TCP/IP has ended you will receive a TCP1A01 message (`ENDTCP completed successfully`) message.

The processing for ending the TCP/IP servers involves making a program call to each of the configured TCP/IP server management programs. Each server management program is called in turn and is expected to do a minimum of processing with this in order to cause the end server processing in the appropriate server job or jobs. For additional considerations relating to user-defined servers, see 2.6.2, “User-defined servers” on page 17.

The processing for ending the TCP/IP interfaces is carried out by the QSYSWRK/QTCPIP job. A request is sent to the job to end each of the interfaces. The IPv6 interfaces and Stateless Address Configuration end processing is performed and then the IPv4 interfaces are ended. In general the end processing for interfaces should proceed relatively quickly. If there were to be any problems the information will be logged on the QTCPIP job log. You can view the job log after TCP/IP has been ended if you feel the need to do so.

The QSYSWRK/QTCPIP and QSYSWRK/QTCPMONITR jobs will be ended after all of the interfaces have been ended. In turn the SLIC TCP/IP protocol stack will be ended after these two jobs are ended. At this point the only work to be done is a bit of cleanup, and then the ENDTCP CL command processing program will return. End TCP/IP processing is complete.

Information about the processing that took place is logged on the QTCPIP job log and the QTCP and QSYSOPR message queues.

We recommend that you do not cancel the QSYSWRK/QTCPIP job. To end TCP/IP, you should issue the ENDTCP CL command. If you were to cancel the QSYSWRK/QTCPIP job, comprehensive end TCP/IP processing would not be performed. In this case the IP interfaces, SLIC TCP/IP protocol stack, and the QSYSWRK/QTCPMONITR jobs would end. The TCP/IP servers would not be ended. The servers might subsequently end because they encounter errors with not having the SLIC TCP/IP protocol stack active and they may not depending on what processing if any they might be performing.

2.6 Other considerations

This section covers other topics related to starting and ending TCP/IP.

2.6.1 Network servers

TCP/IP and TCP/IP interfaces may be started at the time a network server is varied on. There are several things that a user may be able to do in order to get better performance when varying on network servers after an IPL:

- Start TCP/IP prior to any vary on of network server descriptions via the IPL attributes or the start-up program, and have your network server descriptions configured so they do not automatically vary on at IPL.
- Ensure that interfaces associated with the network server descriptions do not specify AUTOSTART(*YES). This can improve the performance of starting TCP/IP because the

QTCPIP job will not be issuing vary on requests for the network servers, which in turn could cause delays in processing other TCP/IP-related requests.

- When TCP/IP has been started, vary on the network servers. The VRYCFG command allows certain objects such as network servers to be varied on simultaneously by submitting multiple batch jobs (SBMMLTJOB parameter), so you should consider using this option if you have multiple network servers.

2.6.2 User-defined servers

Starting with i5/OS V5R2, users have the ability to define their own TCP/IP servers, which can be started via the Start TCP/IP (STRTCP) and Start TCP/IP Server (STRTCPSVR) commands or ended via the End TCP/IP (ENDTCP) and End TCP/IP Server (ENDTCPSVR) commands. When processing a user-defined server, these commands will call a user-written server management program. When designing a server management program, keep in mind:

- The server management program should not perform any long-running operations. It should leave long-running operations to the server job or jobs themselves. A long-running operation or a poorly written program could prevent other servers from starting or the start or end TCP/IP operation from completing normally.
- Starting TCP/IP should not be attempted by either the server management program or the server jobs. TCP/IP status as well as the status of resources such as TCP/IP interfaces may be checked programmatically.
- Implement retry logic in the event that a needed resource such as a TCP/IP interface is not available.

2.7 Starting and ending TCP/IP references

Additional information is available in the iSeries Information Center at:

`http://publib.boulder.ibm.com/infocenter/iseries/v5r4/index.jsp`

Information about restricted state is available in the iSeries Information Center under these topics:

- **Systems management → Basic system operations → i5/OS concepts → i5/OS restricted state**
- **Systems management → Work management → Manage work → Manage subsystems → Place the system in restricted state**
- **Systems management → Work management → Experience reports → Restricted state**

Information about starting TCP/IP and system startup programs is available in the iSeries Information Center under the topics:

- **Networking → TCP/IP troubleshooting → Troubleshooting tools and techniques → Troubleshooting tips → Verify system startup considerations for networking → Timing considerations**
- **Systems management → Basic system operations → Start and stop the server → Start the server → Change the IPL startup program**

3

Checking TCP/IP status programmatically

This chapter discusses the reasons why it is important for a TCP/IP application to check the status of TCP/IP in a proper way. It also provides an example of a program that checks TCP/IP status.

3.1 Considerations for checking TCP/IP status

Many applications rely on TCP/IP, so it is important that they check the status of TCP/IP in a proper manner. In addition, it may also be necessary for an application to determine whether the system is in a restricted state. Proper checking by an application can reduce the possibility of application errors involving the lack of resources or timing situations.

If TCP/IP is in the process of starting or ending, or if the system is in a restricted state, the necessary TCP/IP support for your application may not be available. In addition, when the system is in a restricted state, other resources such as subsystems may also be unavailable. In restricted state, only the controlling subsystem and a single user job are available. TCP/IP may be active in restricted state, and it is possible to start TCP/IP interfaces.

IBM i5/OS provides an Application Programming Interface (API) called Retrieve TCP/IP Attributes (QtocRtvTCPA) that can be used to determine the status of TCP/IP and whether the system is in restricted state. Only a single call to this API is needed to determine both of these conditions, as well as a lot of other information that may be useful to an application. Status information for both IPv4 and IPv6 is provided.

3.2 CL programming example for checking TCP/IP status

The following CL program uses the Retrieve TCP/IP Attributes (QtocRtvTCPA) API to check whether TCP/IP is active and whether the system is in restricted state.

Important: This example uses new CL support, which is available in i5/OS Version 5 Release 4 Modification 0 (V5R4M0).

1. Create a source file member for the CL program. This step assumes that a library MYLIB and source file MYFILE have already been created.
   a. Start an editing session using the following command:

      ```
      STRSEU SRCFILE(MYLIB/MYFILE) SRCMBR(RTVTCPSTS) TYPE(CLLE)
      ```

      Note: The file member type must be CLLE because you are creating an Integrated Language Environment® (ILE) CL program.

   b. Enter the program source statements as shown in Example 3-1.

Example 3-1 CL program for retrieving TCP/IP status

```
PGM
            DCL         VAR(&RCVR) TYPE(*CHAR) LEN(140)
            DCL         VAR(&RCVRLEN) TYPE(*INT) LEN(4) VALUE(140)
            DCL         VAR(&FORMAT) TYPE(*CHAR) LEN(8) +
                          VALUE('TCPA0100')
            DCL         VAR(&ERRCODE) TYPE(*CHAR) LEN(8) +
                          VALUE(X'0000000000000000')
            DCL         VAR(&STSPTR) TYPE(*PTR) ADDRESS(&RCVR 8)
            DCL         VAR(&STSVAL) TYPE(*INT) STG(*BASED) LEN(4) +
                          BASPTR(&STSPTR)
            DCL         VAR(&LMTPTR) TYPE(*PTR) ADDRESS(&RCVR 136)
            DCL         VAR(&LMTVAL) TYPE(*INT) STG(*BASED) LEN(4) +
                          BASPTR(&LMTPTR)
```

```
            DCL        VAR(&ACTIVE) TYPE(*INT) LEN(4) VALUE(1)
            DCL        VAR(&SYSRSTD) TYPE(*INT) LEN(4) VALUE(1)
            /* Call the QtocRtvTCPA API to retrieve TCP/IP status.  */
            CALLPRC    PRC('QtocRtvTCPA') PARM((&RCVR) (&RCVRLEN) +
                         (&FORMAT) (&ERRCODE))
            /* Determine whether TCP/IP is fully active.            */
            IF         COND(&STSVAL *EQ &ACTIVE) THEN(SNDPGMMSG +
                         MSG('IPv4 TCP/IP is ACTIVE.'))
            ELSE       CMD(SNDPGMMSG MSG('IPv4 TCP/IP is not ACTIVE.'))
            /* Determine whether the system is in restricted state. */
            IF         COND(&LMTVAL *EQ &SYSRSTD) THEN(SNDPGMMSG +
                         MSG('The system is in restricted state.'))
            ELSE       CMD(SNDPGMMSG MSG('The system is NOT in +
                         restricted state.'))
ENDPGM
```

2. Leave the source editing session and save your changes.
3. Create the CL module object.

   ```
   CRTCLMOD MODULE(MYLIB/RTVTCPSTS) SRCFILE(MYLIB/MYFILE)
   ```

4. Create the program object.

   ```
   CRTPGM PGM(MYLIB/RTCPS) MODULE(MYLIB/RTVTCPSTS) BNDSRVPGM(QSYS/QTOCNETSTS)
   ```

   Note: This program binds to the service program QTOCNETSTS because it provides the QtocRtvTCPA API.

5. Run the program:

   ```
   CALL MYLIB/RTCPS
   ```

In this example, if IPv4 TCP/IP is fully active and if the system is not in restricted state, the following messages are sent to the job log:

```
IPv4 TCP/IP is ACTIVE.
The system is NOT in restricted state.
```

In order to check the status of IPv6 TCP/IP, the program in Example 3-1 on page 20 would have to change to specify the value TCPA1100 for the variable &FORMAT. The field used to check for restricted state is only available in format TCPA0100. Coincidentally, the fields for the status of IPv4 and IPv6 TCP/IP are at the same offset in the data returned by both the TCPA0100 and TCPA1100 formats.

3.2.1 References

Additional information is available in the iSeries Information Center at:

http://publib.boulder.ibm.com/infocenter/iseries/v5r4/index.jsp

Documentation for the Retrieve TCP/IP Attributes (QtocRtvTCPA) API is available in the iSeries Information Center under the topic **Programming** → **APIs** → **APIs by category** → **Communications** → **TCP/IP Management APIs**.

Information about CL programming is also available in the iSeries Information Center under the topic **Programming** → **CL** → **CL Programming**.

4

Using alias names and setting proxy ARP and preferred interface lists programmatically

Several new TCP/IP networking features and functions are being introduced in Version 5 Release 4 Modification 0 of i5/OS. These include the ability to assign a name to a TCP/IP interface that can be used in the place of an IP address in programs. Also, a new feature called the preferred interface list is being introduced to assist with handling adapter failures.

This chapter provides information about these new features in the following topics:

- Using interface alias names
- Proxy ARP and the preferred interface list
- Putting it all together
- References

4.1 Using interface alias names

Interface alias names provide a way to assign a name to a TCP/IP interface, which can simplify the design of programs that manage TCP/IP interfaces. Rather than hard-coding TCP/IP interface IP addresses, programs can be written to use alias names. An alias name can be assigned to an interface either through CL commands or iSeries Navigator. This includes the following set of CL commands:

- Add TCP/IP Interface (ADDTCPIFC)
- Change TCP/IP Interface (CHGTCPIFC)
- End TCP/IP Interface (ENDTCPIFC)
- Remove TCP/IP Interface (RMVTCPIFC)
- Start TCP/IP Interface (STRTCPIFC)

In addition to CL command support, the Convert Interface ID (QtocCvtIfcID) API is available to convert between alias names and IP addresses. This API supports both IPv4 and IPv6 addresses. The following example shows use of alias names and the QtocCvtIfcID API.

Important: The examples in this chapter use new CL support that is available in i5/OS Version 5 Release 4 Modification 0 (V5R4M0).

1. Define a TCP/IP interface with an alias name. This step assumes that an Ethernet line description ETHLINE already exists.

```
ADDTCPIFC INTNETADR('10.1.1.1') LIND(ETHLINE1) SUBNETMASK('255.255.255.255')
ALIASNAME(ETHLINE1)
```

2. Create a source file member for a CL module. This step assumes that a library MYLIB and source file MYFILE have already been created.
3. Start an editing session:

```
STRSEU SRCFILE(MYLIB/MYFILE) SRCMBR(NAME2IPA) TYPE(CLLE)
```

Note: The file member type must be CLLE because you are creating an Integrated Language Environment (ILE) CL program.

4. Enter the source statements.

Example 4-1 CL source for converting from an alias name to an IP address

```
PGM         PARM(&IFCNAME &IFCADDR)
              DCL         VAR(&IFCNAME) TYPE(*CHAR) LEN(50)
              DCL         VAR(&IFCADDR) TYPE(*CHAR) LEN(15)
              DCL         VAR(&RCVR) TYPE(*CHAR) LEN(72)
              DCL         VAR(&RCVRLEN) TYPE(*INT) LEN(4) VALUE(72)
              DCL         VAR(&FORMAT) TYPE(*CHAR) LEN(8) +
                            VALUE('NCII0100')
              DCL         VAR(&REQCCSID) TYPE(*INT) LEN(4) VALUE(0)
              DCL         VAR(&ERRCODE) TYPE(*CHAR) LEN(8) +
                            VALUE(X'0000000000000000')
              DCL         VAR(&IPADDRPTR) TYPE(*PTR) ADDRESS(&RCVR 8)
              DCL         VAR(&IPADDR) TYPE(*CHAR) STG(*BASED) LEN(15) +
                            BASPTR(&IPADDRPTR)
              CALLPRC     PRC('QtocCvtIfcID') PARM((&RCVR) (&RCVRLEN) +
                            (&FORMAT) (&IFCNAME) (&REQCCSID) (&ERRCODE))
```

```
            CHGVAR     VAR(&IFCADDR) VALUE(&IPADDR)

ENDPGM
```

5. Leave the editing session and save your changes.
6. Create the CL module object:

   ```
   CRTCLMOD MODULE(MYLIB/NAME2IPA) SRCFILE(MYLIB/MYFILE)
   ```

7. Create a second source file member for a CL module.
   a. Start an editing session:

      ```
      STRSEU SRCFILE(MYLIB/MYFILE) SRCMBR(CVT2IPA) TYPE(CLLE)
      ```

   b. Enter the source statements.

Example 4-2 CL source for calling the NAME2IPA module

```
PGM         PARM(&NAME)
              DCL        VAR(&NAME) TYPE(*CHAR) LEN(25)
              DCL        VAR(&IPADDR) TYPE(*CHAR) LEN(15)
              CALLPRC    PRC(NAME2IPA) PARM((&NAME) (&IPADDR))
              SNDPGMMSG  MSG(&IPADDR)

ENDPGM
```

8. Create the CL module object:

   ```
   CRTCLMOD MODULE(MYLIB/CVT2IPA) SRCFILE(MYLIB/MYFILE)
   ```

9. Create the program object:

   ```
   CRTPGM PGM(MYLIB/CVT2IPA) MODULE(MYLIB/CVT2IPA MYLIB/NAME2IPA)
   BNDSRVPGM(QSYS/QTOCNETSTS)
   ```

   Note: This program binds to the service program QTOCNETSTS because it provides the QtocCvtIfcID API.

10. Run the program:

    ```
    CALL MYLIB/CVT2IPA PARM('ETHLINE1')
    ```

In this example, the following message is sent to the job log: 10.1.1.1

4.2 Proxy ARP and the preferred interface list

Proxy ARP enables a physical interface to answer Address Resolution Protocol (ARP) requests on behalf of a virtual IP or virtual Ethernet address. The physical interface in this case is referred to as an agent interface because it answers APR requests on behalf of the virtual address. This enables the virtual address to be known to the network; otherwise, remote systems must have a route defined to the virtual IP address.

Virtual IP addresses can be used to provide both inbound and outbound load balancing, as well as fault tolerance in the event of an adapter failure. Fault tolerance capability is achieved through the specification of the same virtual IP address as the associated local interface for more than one physical interface. In releases prior to i5/OS V5R4, the operating system would select the agent based on either the highest-speed interface available (i5/OS V5R3) or the first interface activated (i5/OS V5R2).

Starting with i5/OS V5R4, you can manually select which adapters and IP addresses are to be the preferred interface for VIPA proxy ARP agent selection. You can select which interface to use by creating a preferred interface list for use in the event of an adapter failure. A preferred interface list is an ordered list of the interface addresses that will take over for the failed adapters. You can use either iSeries Navigator (via the interface properties) or the Change TCP/IP IPv4 Interface (QTOCC4IF) API to configure a preferred interface list. The preferred interface list is also configurable for both virtual Ethernet and virtual IP address interfaces.

4.3 Putting it all together

1. Define several TCP/IP interfaces with an alias names. This step assumes that Ethernet line descriptions ETHLINE1 and ETHLINE2 already exist.

```
ADDTCPIFC INTNETADR('10.1.1.1') LIND(ETHLINE1) SUBNETMASK('255.255.255.255')
ALIASNAME(ETHLINE1)
ADDTCPIFC INTNETADR('10.1.1.2') LIND(ETHLINE2) SUBNETMASK('255.255.255.255')
ALIASNAME(ETHLINE2)
ADDTCPIFC INTNETADR('10.1.1.3') LIND(*VIRTUALIP)
SUBNETMASK('255.255.255.255')
ALIASNAME(MYVIPA)
```

2. Create a source file member for a CL module. This step assumes that a library MYLIB and source file MYFILE have already been created.
   a. Start an editing session:

```
STRSEU SRCFILE(MYLIB/MYFILE) SRCMBR(CHGVIPA) TYPE(CLLE)
```

   b. Enter the source statements.

Example 4-3 CL source for setting proxy ARP and the preferred interface list

```
PGM
            /* Declare the parameters passed into this module.        */
            DCL        VAR(&VIPAIFC) TYPE(*CHAR) LEN(25) +
                         VALUE('MYVIPA                   ')
            DCL        VAR(&PRFIFC1) TYPE(*CHAR) LEN(25) +
                         VALUE('ETHLINE1                 ')
            DCL        VAR(&PRFIFC2) TYPE(*CHAR) LEN(25) +
                         VALUE('ETHLINE2                 ')
            /* Declare the variables for calling QTOCC4IF.            */
            DCL        VAR(&IFCINF) TYPE(*CHAR) LEN(196)
            DCL        VAR(&FORMAT) TYPE(*CHAR) LEN(8) +
                         VALUE('IFCH0100')
            DCL        VAR(&ERRCODE) TYPE(*CHAR) LEN(8) +
                         VALUE(X'0000000000000000')
            DCL        VAR(&IFCINFLEN) TYPE(*INT) STG(*DEFINED) +
                         LEN(4) DEFVAR(&IFCINF)
            DCL        VAR(&IFCINFIPA) TYPE(*CHAR) STG(*DEFINED) +
                         LEN(15) DEFVAR(&IFCINF 5)
            DCL        VAR(&RESERVED) TYPE(*CHAR) STG(*DEFINED) +
                         LEN(1) DEFVAR(&IFCINF 20)
            DCL        VAR(&PROXYARP) TYPE(*INT) STG(*DEFINED) +
                         LEN(4) DEFVAR(&IFCINF 21)
            DCL        VAR(&OFSPRFIFC) TYPE(*INT) STG(*DEFINED) +
                         LEN(4) DEFVAR(&IFCINF 25)
            DCL        VAR(&NUMPRFIFC) TYPE(*INT) STG(*DEFINED) +
                         LEN(4) DEFVAR(&IFCINF 29)
```

```
DCL        VAR(&LENPRFIFC) TYPE(*INT) STG(*DEFINED) +
             LEN(4) DEFVAR(&IFCINF 33)
DCL        VAR(&PRFIFCPTR) TYPE(*PTR) ADDRESS(&IFCINF 0)
DCL        VAR(&PRFIFC) TYPE(*CHAR) STG(*BASED) LEN(15) +
             BASPTR(&PRFIFCPTR)
DCL        VAR(&IFCNAME) TYPE(*CHAR) STG(*DEFINED) +
             LEN(24) DEFVAR(&IFCINF 37)
/* Initialize the interface information parameter fields  */
/* for calling the QTOCC4IF API.                          */
CHGVAR     VAR(&IFCINFLEN) VALUE(196)
CHGVAR     VAR(&RESERVED) VALUE(X'00')
CHGVAR     VAR(&PROXYARP) VALUE(1)
CHGVAR     VAR(&OFSPRFIFC) VALUE(60)
CHGVAR     VAR(&NUMPRFIFC) VALUE(2)
CHGVAR     VAR(&LENPRFIFC) VALUE(16)
CHGVAR     VAR(&IFCNAME) VALUE('*SAME                  ')
/* Convert the virtual IP address name to an IP address.  */
CALLPRC    PRC(NAME2IPA) PARM((&VIPAIFC) (&IFCINFIPA))
/* Address the beginning of the preferred interface list. */
CHGVAR     VAR(%OFS(&PRFIFCPTR)) VALUE(%OFS(&PRFIFCPTR) +
             + &OFSPRFIFC)
/* Convert the first proxy agent name to an IP address    */
/* and store it in the preferred interface list.          */
CALLPRC    PRC(NAME2IPA) PARM((&PRFIFC1) (&PRFIFC))
/* Address the next element in the preferred interface    */
/* list.                                                  */
CHGVAR     VAR(%OFS(&PRFIFCPTR)) VALUE(%OFS(&PRFIFCPTR) +
             + &LENPRFIFC)
/* Convert the second proxy agent name to an IP address   */
/* and store it in the preferred interface list.          */
CALLPRC    PRC(NAME2IPA) PARM((&PRFIFC2) (&PRFIFC))
/* Change the virtual IP interface to enable proxy ARP    */
/* and to set the preferred interface list.               */
CALL       PGM(QSYS/QTOCC4IF) PARM(&IFCINF &FORMAT +
             &ERRCODE)
ENDPGM
```

3. Create the CL module object:

```
CRTCLMOD MODULE(MYLIB/CHGVIPA) SRCFILE(MYLIB/MYFILE)
```

4. Create the program object:

```
CRTPGM PGM(MYLIB/CHGVIPA) MODULE(MYLIB/CHGVIPA MYLIB/NAME2IPA)
BNDSRVPGM(QSYS/QTOCNETSTS)
```

Note: This step assumes that the NAME2IPA module from 4.1, “Using interface alias names” on page 24 has already been created.

5. Run the program:

```
CALL MYLIB/CHGVIPA
```

Using iSeries Navigator, the preferred interface list and proxy ARP setting for interface 10.1.1.3 can now be verified by viewing the Advanced tab of the interface properties as shown in Figure 4-1 on page 28.

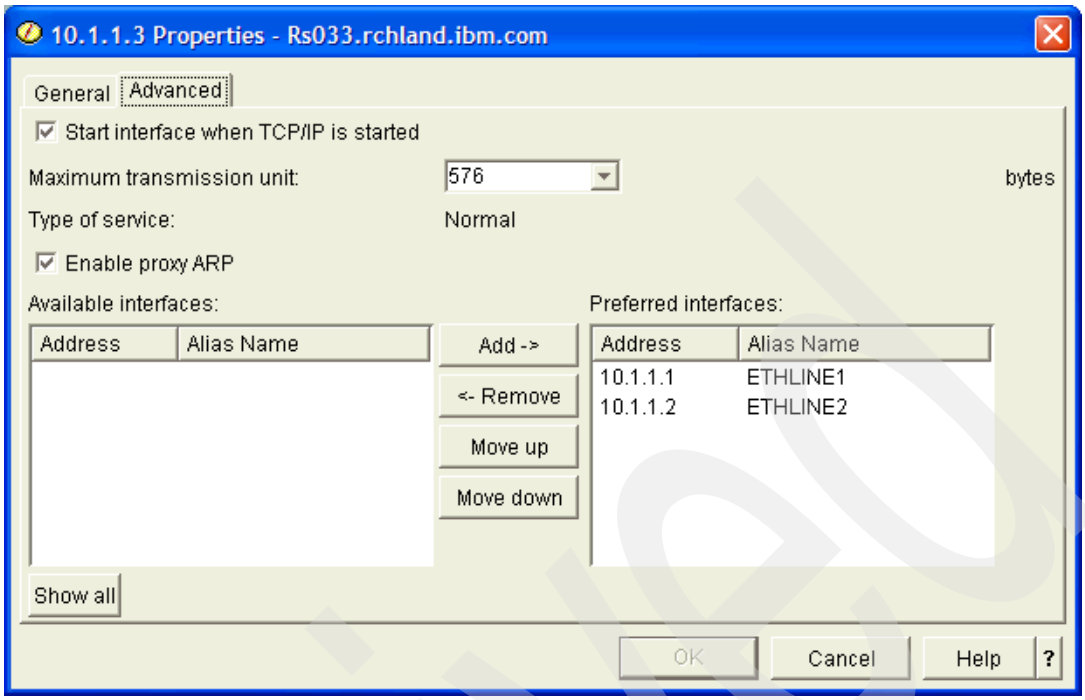


Figure 4-1 Interface properties showing preferred interface list for 10.1.1.3

4.4 References

Additional information is available in the iSeries Information Center at:

`http://publib.boulder.ibm.com/infocenter/iseries/v5r4/index.jsp`

Documentation for the Change TCP/IP IPv4 Interface (QTOCC4IF) and Convert interface ID (QtocCvtIfcID) APIs is available in the iSeries Information Center under the topic **Programming** → **APIs** → **APIs by category** → **Communications** → **TCP/IP Management APIs**.

Information about CL programming is also available in the iSeries Information Center under the topic **Programming** → **CL** → **CL Programming**.

Information about Proxy ARP and workload balancing is available in the iSeries Information Center under the topic **Networking** → **TCP/IP applications, protocols, and services** → **TCP/IP routing and workload balancing**.

5

Using exit programs

This chapter describes what you can do with the exit points that are defined for IBM-supplied TCP/IP applications.

An exit point is a defined interface that enables you to write a program that can change the behavior of i5/OS or IBM-supplied products in specific instances. The following i5/OS TCP/IP applications have exit points defined:

- Telnet
- File Transfer Protocol (FTP)
- Trivial File Transfer Protocol (TFTP) server
- Remote Execution (REXEC) server
- Dynamic Host Configuration Protocol (DHCP) server

The formal programming interface for each exit point is defined in the System i5 Information Center, and is not repeated here. Use that information to actually write any exit programs.

5.1 Basic exit program information

An exit point is a defined programming interface for writing a program called an *exit program*, which is called by IBM-supplied code in specific instances. The parameters passed to your exit program and the information that your exit program can return are defined by an *exit point* format. Each exit point has at least one associated exit point format.

Different exit points that perform similar functions may share an exit point format. This enables you to write a single exit program that you can use to process multiple exit points. For example, several of the TCP/IP applications have a “Request Validation” exit point, and all of these exit points share the same exit point format. As a result, you can write a single exit program that validates requests from any (or all) of these applications.

Some exit points have more than one exit point format. This usually happens when features are added to an application that requires a change to the parameters passed to an exit program. By defining a new exit point format, existing exit programs can still be used, but may not be able to utilize newly added features or functions. When you are designing an exit program for an exit point with more than one exit point format, make sure to look at all of the formats and choose the one that best meets your needs. (In most cases, the exit point format name with the highest numeric value is the most current and has the most features available.)

An exit program should always leave the state of the job unchanged. For example, if an exit program opens a file, it should close the file before returning. Likewise, if the exit program must change a job attribute to operate properly, it should save the value of that attribute before making the change and then restore it to the original (saved) value before ending. Remember that the exit program is called within the job where a specific exit point condition occurs, and the design of the program should not assume that calls made to it will occur in any specific order. Also note that some i5/OS TCP/IP applications “reuse” jobs for different sessions, so even the calls to your exit program within a single job may not be related. The time spent running the exit program delays processing in the TCP/IP application, so exit programs should be designed to be as efficient as possible.

After your exit program is written, you must add it to the exit point (or points) that are to call it and specify the exit point format you chose for your program. See an example of how to do this in the System i5 Information Center at:

`http://publib.boulder.ibm.com/infocenter/iseries/v5r3/topic/rzaiq/rzaiqinstep.htm`

If you do not want to write your own exit programs, you might investigate other software providers’ products. Several products are available from independent software vendors that use the TCP/IP application exit points, and one of these might meet your needs.

5.2 Request Validation exits

Several IBM-supplied TCP/IP applications implement a Request Validation exit point. This exit point enables you to increase the security of your system and gives you the opportunity to reject a specific request even if it would otherwise be allowed by normal i5/OS object security. (However, this exit point cannot be used to circumvent object security; if the user making the request does not have the required authority to access an object on the system, the Request Validation exit point *cannot* be used to make a request for that object “work.”)

As an example, XYZ Company has a call center for employees with questions about their pay and benefits. This center is staffed by people in the Human Resources department. They are given read access to the company’s master payroll file so that they can answer questions from employees about payroll issues. However, because the payroll file contains very

sensitive information, the manager of the HR department wants to prevent the call center employees from being able to send the file to another system. By writing an exit program for the Request Validation exit point, you could prevent these employees from using FTP to access the master payroll file.

The applications that implement the Request Validation exit point are:

- FTP Client
  Exit point name: QIBM_QTMF_CLIENT_REQ
- FTP Server
  Exit point name: QIBM_QTMF_SERVER_REQ
- REXEC Server
  Exit point name: QIBM_QTMX_SERVER_REQ
- TFTP Server
  Exit point name: QIBM_QTOD_SERVER_REQ

All of these exit points share a single exit point format: VLRQ0100.

5.2.1 Capabilities of a Request Validation exit program

An exit program added to one of the Request Validation exit points is called whenever that application is about to process a request made by a user. The information passed to the exit program is from the standpoint of the application calling it; for example, if the exit program is called by the FTP client application, any file or directory name is for the “local” system where the user is typing FTP subcommands (not the “remote” system where the FTP server is running).

A Request Validation exit program receives the following information when called:

- An identifier that defines which application is calling the exit program. This parameter (Application identifier) allows a single exit program to process requests from different applications.
- An identifier for the type of operation requested. This parameter (Operation identifier) tells the exit program the category of the request, such as initiating a new session, sending a file to another system, receiving a file from another system, or running a CL command).
- The user profile making the request. (For client applications, this is the user profile of the job where the client is running; for server applications, this is the user profile used to log on to the server.)
- The Internet Protocol (IP) address of the “remote” system (from the standpoint of the application calling the exit program). For client applications, this is the IP address of the server to which the client is connected; for server applications, this is the IP address of the system where the client is running. The IP address information is passed in two parameters: IP address and Length of IP address.
- Operation-specific information, which provides specific information about the request. For example, if the operation identifier specifies that a file is being sent to another system, the operation-specific information contains the full path name of the file to which is to be sent. This information is also passed in two parameters: Operation-specific information and Length of operation-specific information.

You design your exit program to make a determination about whether the request should be allowed. (However, as noted before, even a request allowed by your exit program still might fail due to i5/OS object security; for example, if the user attempts to run a CL command but does not have the authority to do so, that attempt will fail even if your exit program “allows” it.

A Request Validation exit program can only be used to further restrict an operation beyond that enforced by i5/OS object security.)

The Request Validation exit program must set an output parameter (Allow operation) that tells the TCP/IP application how the request should be handled. In the simple case, the exit program sets this parameter to either 0 (Reject operation) or 1 (Allow operation).

Two other values can be specified for the Allow operation parameter: 2 (Always allow operation) and -1 (Always reject operation). These tell the calling application that any operation with the same operation identifier requested in the same session should be treated as if the exit program allowed (2) or rejected (-1) the operation (without actually calling the exit program again). Using these values enables your exit program to prevent unnecessary overhead if the requested operation type will always result in the same response from your exit program.

Note: Setting an operation identifier to -1 (Always reject) can result in situations that initially might seem to be incorrect or confusing. For example, if an FTP server Request Validation exit program returns -1 (Always reject) in response to a CWD (change directory) FTP subcommand. the FTP server will reject any file transfer request for a file not in the “current directory” without even calling the exit program. Although this might seem to be incorrect behavior, it is not, because this operation is functionally equivalent to the user first performing a change directory subcommand followed by the request to transfer the file.

Your exit program can also perform additional functions with the information passed to it. For example, you might want to use a Request Validation exit program to keep a log of files transferred with FTP, along with the user and IP address information. Such a log might be useful for auditing, tracking intrusions into your system, and so on. Another example is that a Request Validation exit program (along with a Server Logon exit program) together can implement “anonymous” FTP; that is, the ability to make certain files on your system “publicly” available without a user needing a user profile on the FTP server system. (Anonymous FTP is covered in the FTP Security topic in the System i5 Information Center.)

5.3 Server Logon exits

The i5/OS FTP and REXEC servers implement the Server Logon exit point. The purpose of this exit point is to give you additional control over how users log on to the server. Using this exit point, you can specify the user profile for logging on a user, perform your own password and authentication processing, and change some of the initial characteristics of the session, in addition to just allowing or denying a specific logon.

The Server Logon exit points are implemented in the following TCP/IP Applications:

FTP Server Exit point name: QIBM_QTMF_SVR_LOGON

REXEC Server Exit point name: QIBM_QTMX_SVR_LOGON

5.3.1 Capabilities of a Server Logon exit program

An exit program added to one of the Server Logon exit points is called whenever that application is about to authenticate and log on a user. The Server Logon exit point has 3 defined exit point formats; the exact capabilities of your exit program depend on which exit point format you choose.

TCPL0100 format

The TCPL0100 format does *not* support the ability to return a password longer than 10 characters. If your system is running with the QPWDLVL system value set to 2 or 3 (that is,

128-character password support) and your exit program needs to return passwords to the server application, you cannot use this exit point format.

A Server Logon exit program for the TCPL0100 exit point format receives the following information when called:

- An identifier for which application is calling the exit program. This parameter (Application identifier) allows a single exit program to process requests from different applications.
- The user identifier supplied to the server, along with the length of this identifier.
- The authentication string (password) supplied to the server, along with the length of this authentication string.
- The Internet Protocol (IP) address (in dotted-decimal format) of the system where the client application is running, along with the length of this address string.

Your exit program uses this information to determine whether the logon should be allowed, and if so, how the logon should be performed. Note that the Server Logon exit program can be used to bypass the authentication that is normally performed by i5/OS during logon, so you should design your exit program carefully and test it thoroughly.

There are four output parameters that your exit program can return:

- A return code, which specifies whether the logon should be allowed at all, and if so, how the logon should be performed. Your exit program must always set a return value for this parameter. Depending on the value set for this parameter by your exit program, your exit program may also need to return values in one or more of these parameters.
- A user profile returned by your exit program. Depending on the value set by your exit program for the return code, the server will use either the user identifier passed by the client during logon processing, or a user profile name returned by your exit program in this parameter. Using this parameter, your exit program can allow someone to log on to the server who does not have a profile on your server; this capability is useful to implement anonymous FTP.
- A password string returned by your exit program. Depending on the value set by your exit program for the return code, the server will use one of the following options for supplying a password to i5/OS for authentication:
  - The authentication string received from the client application
  - The string returned in this parameter by your exit program
  - Not require a password at all (that is, your exit program can force a successful logon without requiring a password)

  Important: You should *never* store passwords in a program. This parameter is provided so that your exit program can perform algorithmic password processing (for example, secure hash functions or encryption/decryption).

- A library name returned by your program. Depending on the value set by your exit program for the return code, the server will set the current library to either the value specified by your exit program in this parameter or the current library field of the profile used to log on to the server.

TCPL0200 format

The TCPL0200 format is available only for the FTP server (it is not implemented for the REXEC server), does not support longer passwords, and has no advantages over the TCPL0300 format. It is no longer recommended for new development so is not covered here.

TCPL0300 format

The TCPL0300 exit point format uses three types of parameters:

Input	Parameters passed to your exit program
Output	Parameters returned by your exit program
Input/Output	Parameters passed to your exit program that you can modify to change how server logon options will be processed

Note: In the descriptions that follow, Input/Output parameters are marked in *italics.*

A Server Logon exit program for the TCPL0300 exit point format receives the following information when called:

- An identifier for which application is calling the exit program. This parameter (Application identifier) allows a single exit program to process requests from different applications.
- The user identifier supplied to the server, along with the length of this identifier.
- The authentication string (i.e., password) supplied to the server, along with the length of this authentication string and the CCSID (coded character set identifier) of the authentication string.
- The Internet Protocol (IP) address (in dotted-decimal format) of the system where the client application is running, along with the length of this address string.
- *The initial current library. When your exit program is called, this parameter is set to the special value *CURLIB, which means to use the current library specified in the user profile used to log on to the server. If you want a different library to be used as the initial current library, your exit program must set this parameter to the name of that library.*
- *Length of the initial home directory. When your exit program is called, this parameter is set to 0, meaning that the home directory is set to that specified in the user profile used to log on to the server. If you want a different directory to be used, your exit program must set the initial home directory parameter to the fully qualified path name of the directory, and set this parameter to the length of that fully qualified directory name.*
- *CCSID of initial home directory. When your exit program is called, this parameter is set to 0, meaning that any home directory string is returned in the CCSID of the job. If your exit program returns a home directory path name in a different CCSID, you must set this parameter to that CCSID.*
- *Application-specific information* and the length of the application-specific information. When the application identifier is 2 (REXEC server program) these parameters are not used and the length of application-specific information parameter is zero. When the application identifier is 1 (FTP server program), *the application-specific information parameter contains the following fields: (those in italics can be changed by your exit program):*
  - *Initial setting of name format*
  - *Initial current working directory (current library or home directory)*
  - *Initial file listing format (i5/OS or UNIX®)*
  - Control connection security mechanism
  - *Data connection encryption option*
  - Control connection cipher suite
  - *Data connection cipher suite*

Your exit program uses this information to determine whether the logon should be allowed, and if so, how the logon should be performed. Note that the Server Logon exit program can be used to bypass the authentication that is normally performed by i5/OS during logon, so you should design your exit program carefully and test it thoroughly.

There are six output parameters that your exit program can return (*in addition to the input/output parameters already described):*

- A return code (Allow logon) that specifies whether the logon should be allowed at all, and if so, how the logon should be performed. Your exit program must always set a return value for this parameter. Depending on the value set for this parameter by your exit program, your exit program may also need to return values in one or more of these parameters.
- A user profile returned by your exit program. Depending on the value set by your exit program for the Allow logon parameter, the server will use either the user identifier passed by the client during logon processing, or a user profile name returned by your exit program in this parameter. Using this parameter, your exit program can allow someone to log on to the server who does not have a profile on your server; this capability is useful to implement anonymous FTP.
- A password string returned by your exit program, along with parameters for the length and CCSID of this password. Depending on the value set by your exit program for the return code, the server will use one of the following options for supplying a password to i5/OS for authentication:
  - The authentication string received from the client application
  - The string returned in this parameter by your exit program
  - Not require a password at all (that is, your exit program can force a successful logon without requiring a password)

  Important: You should *never* store passwords in a program. This parameter is provided so that your exit program can perform algorithmic password processing (for example, secure hash functions or encryption/decryption).

5.4 REXEC Server Command Processing Selection exit

By default, the i5/OS REXEC server processes commands passed to it as Control Language (CL) commands. The REXEC Server Command Processing Selection exit point (exit point name: QIBM_QTMX_SVR_SELECT) enables you to specify an alternate command processor to interpret and run the command (or commands) for a specific session.

5.4.1 REXEC Server Command Processing Selection exit program capabilities

An exit program added to the REXEC Server Command Processing Selection exit point is called whenever the REXEC server is about to process a command received from an REXEC client. The exit program receives the following information when called:

- The user profile making the request.
- The Internet Protocol (IP) address of the system where the REXEC client is running. The IP address information is passed in two parameters: Remote IP address and Length of Remote IP address.
- The command string to be run and the length of the command string.

You design your exit program to make a determination about which command processor the command should be passed to and whether the REXEC server needs to perform EBCDIC - ASCII character conversion for the command and the returned data.

The exit program must set an output parameter (Command processor identifier) that tells the REXEC server how the command string should be processed (as a CL command, as a Qshell command, or as a fully qualified file name specifying an executable shell script or program). If your exit program specifies that the command is to be treated as anything other than an i5/OS CL command, your program must also set the output parameter specifying whether character conversion is performed on the data.

- Your exit program can also perform additional functions with the information passed to it. For example, you might want to use an exit program for this exit point to keep a log of all commands that are processed by the REXEC server.

5.5 Telnet exits

The i5/OS Telnet server operates by using virtual devices, which are i5/OS objects that simulate physical 5250 architecture terminals. This architecture enables applications on i5/OS to use the same programming interfaces for Telnet-connected sessions as with physical terminals. However, there are some limitations related to the way that Telnet allocates and uses virtual devices. The Telnet server provides two exit points that enable you to write exit programs that provide more flexibility in setting up and managing virtual devices.

5.5.1 Telnet Device Initialization exit point

The purpose of the Telnet Device Initialization exit point is to give you more flexibility in how a virtual device for a new Telnet session is created, and the ability to automatically sign on a user to the interactive job attached to the virtual device. The exit point name and exit point format for the Telnet Device Initialization exit point are QIBM_QTG_DEVINIT and INIT0100 (respectively).

An exit program added to the Telnet Device Initialization exit point is called after the Telnet server has received a request for a new session from a Telnet client and determined the attributes and capabilities of that client. The parameters passed between the Telnet server and the exit program fall into several categories:

- Connection information and emulated terminal capabilities. The data in these parameters are determined by the connection attributes and the programmed capabilities of the Telnet client, so the exit program cannot change these characteristics. They are provided to enable your exit program to make decisions about how the Telnet session and the associated virtual device will operate, and are contained in two multi-field parameters (and an associated length parameter).
  - The Connection description information parameter contains fields that provide the exit program information about the IP addresses of the client and server, the type of workstation requested by the client, whether a valid password (or password equivalent) was provided by the client, whether the session is encrypted using the Transport Layer Security/Secure Sockets Layer (TLS/SSL) protocols (and if so, further information about whether the client provided a valid certificate to allow client authentication as part of the TLS/SSL session initialization).
  - The Environment options parameter enables your exit program to determine the specific features that the Telnet client supports. These features are determined through a negotiation process specified by the Telnet protocol. This buffer contains the Telnet-protocol defined options negotiated by the client exactly as sent; the length of this data is specified by the Length of environment options parameter. Your exit

program can parse this buffer if you need to determine specific information about the capabilities of the Telnet client program.

- Device description information. This parameter is used to communicate the attributes about the virtual device to be created and is filled in by your exit program (if you do not want to use the Telnet defaults). Attributes you can specify include the virtual device name, keyboard identifier, code page, and character set for the virtual device.

  Note: You can choose a convention for your virtual device names and then use i5/OS workstation entries to route the jobs associated with specific virtual devices to specific subsystems. To accomplish this, choose a “root” name for all virtual devices that are to go to a specific subsystem, and create two workstation entries: one that assigns any virtual device with that root to the QINTER subsystem when the virtual device is initialized, and another to assign the virtual device to the desired target subsystem at signon. Then have your exit program assign a unique identifier for each virtual device to be routed to that subsystem and append it to the chosen root. When the user logs on to the Telnet session, the associated interactive job will be routed to the desired subsystem. By using different root values, your exit program can route Telnet sessions to different subsystems as desired.

- Automatic sign-on information. If you want to allow automatic sign-on for this session, your exit program must set a value of 1 for the Allow auto-signon output parameter. Your exit program sets values for fields in the User description information parameter to set the characteristics for the auto-signon operation; the user profile field is required to be set, and your exit program can optionally set the Current library, Initial menu, and Initial program to call fields. The Telnet server will process the signon as if the user had typed these values on the default i5/OS signon screen. (If your exit program does not set the Allow auto-signon parameter to 1, the contents of the User information parameter are completely ignored by the Telnet server.)
- The Allow connection parameter. Your exit program must return a value of 1 in this parameter for the Telnet session to be allowed; otherwise, the session will be rejected and the connection with the Telnet client will be closed.

5.5.2 Telnet Device Termination exit point

The purpose of the Telnet Device Termination exit point is to give you the ability to take some action at the end of a Telnet session. The exit point name and exit point format for the Telnet Device Termination exit point are QIBM_QTG_DEVTERM and TERM0100 (respectively).

An exit program added to the Telnet Device Termination exit point is called when the Telnet client ends the Telnet session. The Telnet server passes a single parameter to the exit program, which is the name of the virtual device that was associated with the Telnet session. There are no return parameters.

Your exit program can use this information to audit or log the end of the session, stop the virtual device, or clean up or delete the virtual device, depending on your requirements.

5.6 DHCP exits

The Dynamic Host Configuration Protocol (DHCP) server enables you to centrally administer your network and automatically assign configuration information (including IP addresses) to devices on your network. This capability avoids the need to manually configure each device.

The i5/OS DHCP server has three exit points defined to enable you to meet auditing, logging, and additional security requirements you might have.

5.6.1 DHCP Address Binding Notify exit

The purpose of the DHCP Address Binding Notify exit point is to give you the ability to take some action whenever the DHCP server successfully leases (assigns) an IP address to a specific device on your network. The exit point name and exit point format for the DHCP Address Binding Notify exit point are QIBM_QTOD_DHCP_ABND and DHCA0100 (respectively).

An exit program added to the DHCP Address Binding Notify exit point is called when the DHCP server assigns an IP address to a network device. The following parameters are passed to the exit program:

Request type Specifies whether the network device requesting an address is using the Bootstrap Protocol (BOOTP) or the DHCP protocol.

Client IP address The IP address assigned to the device by the DHCP server.

Client identifier The unique identifier of the client to which the IP address has been assigned. (This field usually contains the network interface hardware address of the client machine).

Lease duration Identifies the time period for which the client can use the IP address (in seconds), or a special value of all bits set to 1 if the lease duration is infinite.

Response packet The actual BOOTP or DHCP packet that completed the address assignment as transmitted to the requesting client machine.

The Client IP address, Client identifier, and Response packet parameters each have an associated length parameter that contains the length of the data passed to your exit program for that parameter.

There are no return parameters. An exit program for this exit point usually performs auditing and logging of assigned network addresses for problem determination, accounting, or network security monitoring.

5.6.2 DHCP Address Release Notify exit

The purpose of the DHCP Address Release Notify exit point is to give you the ability to take some action whenever the DHCP server releases (de-allocates) an IP address (which was assigned to a specific device on your network). The exit point name and exit point format for the DHCP Address Binding Notify exit point are QIBM_QTOD_DHCP_ARLS and DHCR0100 (respectively).

An exit program added to the DHCP Address Release Notify exit point is called when the DHCP server releases the IP address that has been assigned to a network device. The following parameters are passed to the exit program:

- Reason for release: specifies why the IP address was released:
  - The client machine requested release of the address.
  - The lease duration expired without the client requesting a renewal.
  - A DHCP administrator explicitly released the address.
- Client IP address: the IP address assigned to the device by the DHCP server that is now being released.

- Client identifier: the unique identifier of the client to which the IP address was assigned. (This field usually contains the network interface hardware address of the client machine.)
- The Client IP address and Client identifier parameters each have an associated length parameter that contains the length of the data passed to your exit program for that parameter.

There are no return parameters. An exit program for this exit point usually performs auditing and logging of network addresses for problem determination, accounting, or network security monitoring.

5.6.3 DHCP Request Packet Validation exit

The purpose of the DHCP Request Packet Validation exit point is to give you the ability to further restrict which packets will be processed by the DHCP server beyond the validation tests that the server already performs. The exit point name and exit point format for the DHCP Address Binding Notify exit point are QIBM_QTOD_DHCP_REQ and DHCV0100 (respectively).

An exit program added to the DHCP Request Packet Validation exit point is called each time the DHCP server receives an incoming BOOTP or DHCP request packet but before any processing of the packet has taken place. The following parameters are passed to the exit program:

Request packet An exact copy of the request packet received by the DHCP server exactly as it was received from the network.

Length of request packet The length (in bytes) of the received packet.

Your exit program should inspect the request packet and perform any tests or validation you require of the information contained in the packet. When your exit program has made a determination as to whether the packet should be processed, it sets the Allow operation output parameter to:

- 0 if the packet should be rejected; the DHCP server will not perform any additional processing of the packet.
- 1 if the request packet should be allowed to continue. The DHCP server will continue processing the packet as normal. (If the packet fails the DHCP server's validation tests, the packet will still be rejected.)

Tips and Techniques for Using TCP/IP on i5/OS

Redpaper

Optimizing performance in a TCP/IP network

Considerations on starting and ending TCP/IP

Programmatic management of TCP/IP

This IBM Redpaper provides various tips and techniques for managing TCP/IP with IBM i5/OS:

- Various recommendations that might improve your network configuration and enable you to obtain optimal performance in a TCP/IP network.

- Information about the different ways that TCP/IP can be started and ended, including potential problems that might occur if starting TCP/IP is not carefully planned.

- Discussion about why it is important for a TCP/IP application to check the status of TCP/IP in a proper way. Includes an example of a program that checks TCP/IP status.

- Procedure for assigning an alias to a TCP/IP interface that can be used in place of an IP address in programs. Information is also provided about a new feature called the preferred interface list to assist with handling adapter failures.

- Notes about using exit programs and what you can do with the exit points that are defined for IBM-supplied TCP/IP applications.

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks