About this document

This document is aimed at IT managers and senior developers who are very comfortable with IBM ISeries (i5/OS, AS/400, AS400) and now want to understand the issues involved in developing applications based on TCP/IP sockets. It deliberately does not contain detailed program code examples. It covers issues that you would expect to consider when looking at any new technology on your ISeries platform.

The technical fundamentals

What is a socket ?

A socket is an "end-point for communications" in the TCP/IP protocol. A common analogy is to a telephone call - the telephone handset is comparable to a socket. Both parties speak into their handset. They don't care about the underlying technology that delivers their voice i.e. it is up to the telco to sort out the underlying network bits.

So how do I get a socket ?

A socket is a programming construct, not a physical thing. Developers write programs that create sockets in their code. There is a set way of doing this - on ISeries (i5/OS, AS/400, AS400) or any other platform. Once they have created the socket they write data to the socket - like talking on the telephone - or read data from the socket - like listening on the telephone. Agreeing whose turn it is to talk (write data) or listen (read data) next is a matter of detailed application design.

TCP/IP communications ?

All communications protocols have layers - usually around 5. The exact definition of layers is not particularly important. The key thing is that the application layers sit above - and rely upon - the underlying network layers. In TCP/IP, IP (Internet Protocol) belongs in the application layers, while TCP (Transmission Control Protocol) belongs in the network layers.

The handset in the telephone analogy is a bridge from the application layer (the ability to speak to someone remotely) to the network layer (the underlying wired or wireless communications infrastructure).

Think of a socket as the thing that enables TCP/IP applications to bridge from the IP application layer through the TCP network layer.

Things are actually a bit more complicated than this i.e. there are other protocols like UDP that work in a similar way to TCP. But the principles remain the same - there are still layers.

IBM has repeatedly rebranded the original AS/400.
Several names - i5/OS, System i, ISeries, AS/400, AS400 - refer to exactly the same system.

Are sockets common ?

The Internet is a giant sockets machine. All TCP/IP application protocols use sockets under the covers. They have no choice. If you want to use low-level TCP/IP-based protocols, you must use sockets. Private TCP/IP-based networks (Intranets etc.) similarly use sockets. In short there are vast numbers of sockets being used globally every day to support TCP/IP applications.

Client-server architecture

TCP/IP protocols are fundamentally client-server. One program acts as server - like standing by the telephone waiting for an incoming telephone call. The client system talks through the socket it creates. The server system does the same. The connection between the paired sockets is through the network layers.

The simplest server of this type to understand is a Web (HTTP) server. Here the client program is typically a Web browser - Internet Explorer, Firefox etc. The HTTP client initiates a socket connection to the HTTP server which responds by creating a socket for the two sides - client and server - to talk to each other. The client Web browser requests content it understands (Web pages in whatever form) and the server dishes up that content.

Servers are multi-threaded

The client-side programming tends to be quite simple e.g. a few hundred lines of code. The server-side programming is more complex e.g. a few thousand lines of code or more. Actually the server is a bit like a telephone switchboard operator who answers the calls on an incoming telephone line but transfers the call to another person to deal with. TCP/IP servers take incoming socket connections then hand them off quickly to a dedicated worker process to handle. This means the server is often a multi-threaded application. A master server process initiates the socket connection to talk to the client system, but passes that socket to another thread of execution to do the actual read/write operations.

Connection basics

All TCP/IP connections have two elements: An IP address and a port number e.g. 217.23.164.27:80 represents port 80 on the IP address 217.23.164.27. Think of the IP address as the telephone number and the port number as a particular telephone extension. Each machine on an IP address can support multiple applications (HTTP, FTP, SMTP etc.) each on its own port number. By convention, there are so-called ‘well-known ports’. For example port 80 is almost invariably used by an HTTP application server. A client program that attaches to the wrong port is unlikely to be able to talk to the server in a protocol it understands. Client and server can run happily on the same machine.

Instead of using IP addresses, applications can use domain names. For example http://217.23.164.27:80 and http://www.pitdale.com:80 are synonymous. The mapping of domain names to IP addresses is done by domain name servers (DNS’s).

Why use sockets ?

Custom-built client and/or server

You do sockets programming only if you need a customised TCP/IP-based application.

You can, for example, write a client program that talks to an existing off-the-shelf HTTP server. Why you would want to do this is up to you. Only you can decide why you would want to customise some of the functionality already available in a standard Web browser.

You can write your own TCP/IP server. For example 217.23.164.27:4325 could be an 'adder' application. Client programs connect to this service on its port 4325 and provide two numbers. The server adds the two numbers together and returns the result.

If you write your own server you have to define your own application protocol - so a developer knows how to program the client side. This goes down to the low-level data transmission. For example the adder service could have a simple data protocol something like:

Data received - positions	Content
1	Control character - always the ASCII letter 'A'.
2-6	The first number to add. Whole number only.
7-11	The second number to add. Whole number only.

Data returned - positions	Content
1	Control character. The ASCII letter 'R' if successful addition. Otherwise ASCII 'X' if any problem means the addition could not take place.
2-8	The sum of the two numbers. Always a whole number. Zero if addition could not take place.

The rest of this document assumes you are considering a proprietary application i.e. you want to develop your own TCP/IP server.

Decision factors in considering a TCP/IP service

In deciding to develop your own TCP/IP service, you should consider the following factors:

Need

Is the application you envisage not available in another way ?. For example, FTP is a simple way to exchange data and HTTP is a general-purpose transport protocol that can deal with any content - not just Web pages. Services not based on TCP/IP are still available - for example those based on SNA. But the overwhelming dominance of TCP/IP makes any alternative solution debatable - particularly if non-IBM platforms are involved.

Speed

Sockets are about as fast as it gets. Response times end-to-end between the client program and the server program are typically 10ths of a second for simple applications like the ‘adder’ one above. Traditionally on ISeries, things like data queues and message queues are deemed fast. These however are still based on sockets. If you do your own sockets-based service there is minimal overhead - the program code is fundamentally at the lowest possible level. The main tradeoff is complexity - you basically have to write your own application protocol from scratch.

Scalability

Handling multiple client connections is built into any multi-threaded TCP/IP server. If your service peaks out, you can do replication e.g. make the same service also available on another port number. However any well-designed TCP/IP server should have a very high degree of built-in scalability. For example, an HTTP server on a busy Web site can deal with millions of hits a day.

Platform-independence

Client programs only need to know the IP server address/port and service protocol. The fact that an ISeries (i5/OS, AS/400, AS400) - or any other platform - is providing the service is completely transparent.

Deployment

Client systems that use an IP address rather than a domain name are faster because there is no DNS lookup - subject to cacheing considerations. But then if you move the service to a different machine, all client programs using the IP address need to be redirected to the new server.

You also need to decide which port number to use. Typically if you steer clear of the well-known port numbers, you are safe. Picking a port number is a 15-minute decision. Subsequently changing the port number is not as easy because this will be embedded somewhere in client systems.

Generally if client systems have these connection details in variable parameters, you can find these references and manage any transition. Typically a change of connection details is phased in over time rather than a big-bang switch e.g. the service runs in parallel on old/new connections for a few hours/days.

Development effort

This depends entirely upon your application design of course. However the core logic of a TCP/IP application server typically amounts to several thousand lines of code. Excluding application-specifics (e.g. the low-level data transmission protocols), the development effort is a few man-months. Typical design issues are explored in the next section of this document.

There are third-party software packages that wrap low-level sockets functions. These typically provide ILE/C modules that you bind with your own modules to create server-side programs. These packages should include service management functions - the ability to start/stop multiple services, logging and security management etc. Using these packages - and following the code examples they contain - means you should be able to get a simple sockets-based application up and running within a few weeks. One such package - PSPSOCK - is available from our company.

Server design considerations

A few hundred lines of code in a sockets-based server can handle core functions. However when other issues are taken into account, you typically end up with several thousand lines of code. This section covers issues you should consider when designing and coding your server.

What programming language ?

Support for sockets and threading is built into library functions in many programming languages like Java and C. Sockets came from the Unix world, where C is the systems programming language of choice. On ISeries, library functions for sockets and threading were initially available in ILE/C. Today there is much wider availability of support via APIs to RPG and other languages. ILE is still worth considering for core parts of your design i.e. the overhead of a bound procedure call will always be less than a program-to-program call.

How I start/stop the service ?

The server runs as a background job i.e. with a batch job profile. Normal ISeries (i5/OS, AS/400, AS400) work management issues like job descriptions, job queues and subsystems apply. Starting the service is a matter of submitting the job to a job queue. There is usually some built-in mechanism to ensure multiple server instances do not accidentally run at the same time e.g. ALCOBJ on a data area.

The server typically sits in a very tight loop waiting for incoming client connections. It can ‘come up for breath’ at intervals you define e.g. every few seconds. You can use that period to stop the service e.g. the server interrogates the data area on which it has a lock to see if it contains the word ‘STOP’. Or you can send a ‘STOP’ message to the server via a socket i.e. your data protocol has the STOP message in addition to other application messages. In this case your design should minimise the probability that badly-behaved client systems can accidentally bring down the entire service e.g. the STOP message must include a password that client users/systems can not easily discover.

IBM has repeatedly rebranded the original AS/400.
Several names - i5/OS, System i, ISeries, AS/400, AS400 - refer to exactly the same system.

How do I know what the server is doing ?

A server that seems to be doing nothing (i.e. no CPU usage) is typically just sitting there waiting for a client connection. You can see it in this state using the WRKTCPSTS *CNN command. If the service is hanging on another area of its operation, it can be investigated like any other ISeries job e.g. DSPJOB to examine the job log, I/O, locks, lines of code running etc.

Servers also typically provide some kind of separate log of their activity - including error messages if the service fails or degrades in a controlled manner. How the log is generated (message queue, data queue, database update etc.) is a matter of detailed design. You want to keep logging proportionate - there is not much point in a service that is initially sub-second then spending several seconds on logging activity for every client connection.

You can change the server logging level in the same way that you can stop the server i.e. include logging instructions as part of your application protocol.

How do I run development/testing/live ?

As with any other application, you need to consider how to manage code changes through the development cycle - development testing, user testing, live service etc. If these versions must run on the same machine, the answer is to deploy them on a different port number. Client systems being developed in parallel must manage which port number they talk to for each environment.

Note that new versions of the service can be handled in the data protocol itself e.g. there could be two versions of the same function supported at the same time by the server. The client system indicates in its data transmission to the server which version of the function it is targeting. This approach enables phased migration to new functionality.

Character set issues ?

Sockets programs deal with data streams. At the lowest level, streams of bytes are written to - and read from - program buffers. You may have an ASCII client (e.g. a PC program) talking to your EBCDIC-based ISeries. In this case, your application protocol needs to handle ASCII<->EBCDIC conversion.

Similar issues exist with other single-byte or double-byte character sets on the client and server sides. In an ideal world, this would be a non-issue e.g. you have a single data standard such as UNICODE end-to-end.

Data conversion here may be limited to variable application data i.e. control bytes in each message may not be converted. Encryption, scrambling and compression of data are related issues that you might want to consider as part of your application design.

How do I measure server performance ?

Measuring performance from the client side is a normal testing issue i.e. the turnaround time for the sockets-based service can be isolated from other client functions. Stress testing is comparatively easy. It is just a matter of replicating client systems during a structured test run e.g. you can run hundreds of client test harnesses that hammer away at the server. These processes can also run on the ISeries itself - as long as you understand which network communications are being factored out with that approach.

How do I manage access by client systems ?

TCP/IP network configuration controls access to IP addresses and available ports. IP address assignment and visibility is a pure network issue - the basic 'ping' test applies. Ports can be blocked or opened on devices like firewalls. On a typical network, providing a new service on ISeries constitutes no significant new threat from external attack i.e. rogue external systems should never get access to the IP address/port combination.

Typically access by internal client systems is controlled by managing deployment of the client-side application that uses the sockets-based service. It is of course possible that someone within your organisation can write rogue code that uses the service. You need to focus on what additional exposure this constitutes. For example if the service is making business data available to users that they can get in a different way anyway, then what is the additional risk ?.

You can build password protection into your application protocol. Apart from normal client-side password management considerations, this may introduce application session management issues. A simplistic server may not try to track client sessions e.g. has no knowledge that two client messages come from the same client process. The client and server can exchange session identifiers in their messages e.g. after successful password entry, the server assigns the client system a session ID that the client then returns on other messages. This then raises the issue of rogue systems using faked or intercepted session IDs, expiry of those IDs etc. In short, password protection is not as simple as it might seem at first glance.

As with any access to ISeries data and programs, there will always be a degree of faith in 'trusted clients'.

What about asynchronous processing ?

Typically all data transfer is synchronous - the client system gets all the data it needs in one hit (one socket connection) on the server. However there is nothing to stop you building asynchronous data processing into your application protocol. For example the client requests an ISeries process to start running on one socket connection then checks back later on another connection to see if the process has completed and the results are available. Session management issues apply here as well i.e. you need to ensure that the client process getting the eventual results is suitably authorised to do so.

It helps considerably if your design clearly defines terms like 'conversation', 'session', 'transaction' and 'message'. These typically have a hierarchy e.g. 'conversation' could be at the highest level in your design, while 'message' could be at the lowest level.

Major performance factors ?

In terms of designing your sockets-based application for performance, some generic considerations apply as to any performance-related ISeries (i5/OS, AS/400, AS400) work. Bad program design (poor algorithms, bad program flow etc.) must be eliminated. Database I/O and file/open close operations needs to be carefully designed. Judicious use of cacheing - for example in internal arrays - should be considered for data that changes infrequently.

Some performance issues are sockets-specific. All available choices for multi-threading need to be considered. You must consider the full lifecycle of the server master process (the one that listens on the socket) and server slave processes (the ones that do the actual read/write of data via the socket) - and how these two processes interact. You need to consider potential reusability of slave processes e.g. to avoid any overhead involved in starting them up. If the server calls on other functions to do useful work, you need to consider how those functions make themselves available. Here too you are trying to get reusability - to avoid the overhead of processes repeatedly going through a complete startup. Finally there are 'socket options' that are an integral part of the sockets protocol. A full discussion of these is beyond the scope of this document, but you can assume there about 10-20 relevant options that take a few days to investigate. You set these options once in your server program code.

There are no absolute performance rules here. If you end up with a service that is offering multi-second turnaround, then it is difficult to describe that service as 'real time'. This is a pity. Sockets applications typically shine in the real-time arena.

Relevant ISeries commands ?

There are surprisingly few commands you need to know about:

CFGTCP: Configure TCP. The top-level option for TCP/IP configuration.
CHGTCPA: Change TCP Attributes. Configure generic TCP/IP connection details. Nothing here is specific to your new service, but it is still worth confirming your general TCP/IP configuration is optimal.
WRKSVRTBLE: Work with Service Table Entries. Examine existing TCP/IP services and configure your new one.
WRKTCPSTS: Work with TCP status. Use the *CNN option to look at currently running TCP/IP connections.

Summary

Sockets are nothing new, although they came late to IBM ISeries (i5/OS, AS/400, AS400). Sockets programming is not particularly difficult and there are many publicly available code examples for ISeries and other platforms. The underlying TCP/IP protocols are extremely safe in a strategic sense.

The initial learning curve involves some fundamentally new concepts for typical ISeries developers, but design issues can easily be explored and tested. It is possible to rewrite applications in such a way that core program code is highly reusable. There are third-party libraries that wrap low-level sockets functions and provide service management features. Using these libraries allows you to focus on your own application issues.

Sockets tends to be best for very fast applications i.e. those targeted at real-time response. They are often used as a supporting background service for Web-based applications.

We are available to assist with development of your sockets-based applications. Please contact us via the URLs above.

July 2011